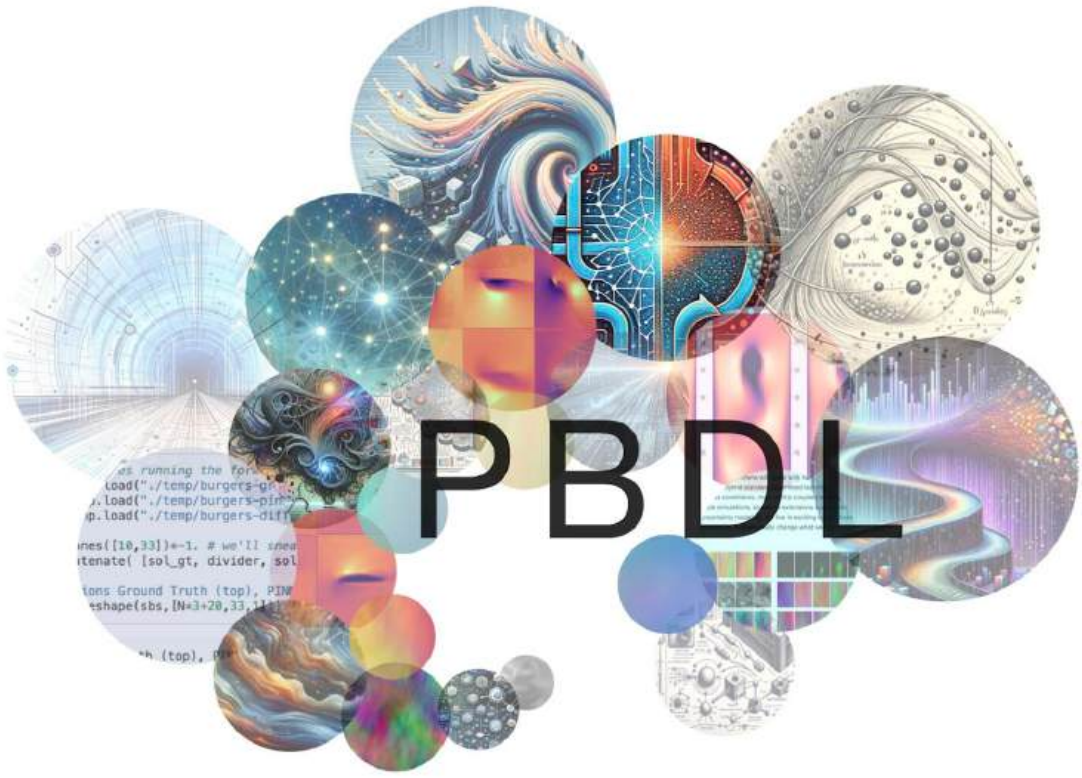
Physics-based Deep Learning

http://physicsbaseddeeplearning.org



N. Thuerey, B. Holzschuh, P. Holl, G. Kohl, M. Lino, Q. Liu, P. Schnell, F. Trost

(v0.3) *Generative AI Edition*

CONTENTS

I Introduction **5**

1 A Teaser Example **7**
1.1 Differentiable physics . . . 7
1.2 Finding the inverse function of a parabola . . . 8
1.3 A differentiable physics approach . . . 11

2 A Probabilistic Generative AI Approach **15**
2.1 Discussion . . . 19
2.2 Next steps . . . 20

3 Overview **21**
3.1 Motivation . . . 21
3.2 Categorization . . . 23
3.3 Looking ahead . . . 24
3.4 Implementations . . . 25
3.5 Models and Equations . . . 25
3.6 Simple Forward Simulation of Burgers Equation with phiflow . . . 28
3.7 Navier-Stokes Forward Simulation . . . 33
3.8 Optimization and Convergence . . . 39

II Neural Surrogates and Operators **47**

4 Supervised Training **49**
4.1 Problem setting . . . 49
4.2 Looking ahead . . . 50

5 Neural Network Architectures **51**
5.1 Spatial Arrangement . . . 51
5.2 No spatially arranged inputs . . . 52
5.3 Local vs Global . . . 52
5.4 Regular, unstructured and point-wise data . . . 53
5.5 Hierarchies . . . 54
5.6 Spectral methods . . . 55
5.7 Attention and Transformers . . . 56
5.8 Summary of Architectures . . . 57
5.9 Show me some code! . . . 57

6 Supervised training for RANS flows around airfoils **59**
6.1 Overview . . . 59
6.2 Formulation . . . 59

6.3 Code coming up… 60
6.4 RANS training data 60
6.5 Network setup 62
6.6 Training 65
6.7 Test evaluation 67
6.8 Next steps 70

7 Discussion of Supervised Approaches 71
7.1 Some things to keep in mind… 71
7.2 Supervised training in a nutshell 73

III Physical Losses 75

8 Physical Loss Terms 77
8.1 Using physical models 77
8.2 Variant 1: Residual derivatives for explicit representations 78
8.3 Variant 2: Derivatives from a neural network representation 79
8.4 Summary so far 80

9 Learning the Helmholtz-Hodge Decomposition 81
9.1 Solving Navier-Stokes 81
9.2 Setting up the Discrete PDE 82
9.3 Neural Network Training 84
9.4 Training 88
9.5 Testing with New Inputs 88
9.6 Tougher Tests: Fluid Simulations with Obstacles 90

10 Next Steps 95

11 Burgers Optimization with a PINN 97
11.1 Formulation 97
11.2 Preliminaries 98
11.3 Loss function and training 101
11.4 Evaluation 103
11.5 Next steps 108

12 Discussion of Physical Losses 109
12.1 Generalization? 109
12.2 Summary 110

IV Differentiable Physics 111

13 Introduction to Differentiable Physics 113
13.1 Differentiable operators 114
13.2 Jacobians 114
13.3 Learning via DP operators 115
13.4 A practical example 115
13.5 Implicit gradient calculations 118
13.6 Summary of differentiable physics so far 120

14 Burgers Optimization with a Differentiable Physics Gradient 121
14.1 Initialization 121
14.2 Gradients 122

14.3 Optimization . . . 124
14.4 More optimization steps . . . 126
14.5 Physics-informed vs. differentiable physics reconstruction . . . 129
14.6 Next steps . . . 131

15 So Far so Good - a First Discussion **133**
15.1 Compatibility with existing numerical methods . . . 133
15.2 Discretization . . . 133
15.3 Efficiency . . . 134
15.4 Efficiency continued . . . 134
15.5 Summary . . . 134

16 Differentiable Fluid Simulations **137**
16.1 Physical Model . . . 137
16.2 Formulation . . . 137
16.3 Starting the Implementation . . . 138
16.4 Batched simulations . . . 138
16.5 Gradients . . . 140
16.6 Optimization . . . 142
16.7 Re-simulation . . . 143
16.8 Conclusions . . . 145
16.9 Next steps . . . 146

17 Integrating DP into NN Training **147**
17.1 Switching the order . . . 148
17.2 Recurrent evaluation . . . 148
17.3 Composition of NN and solver . . . 149
17.4 In equation form . . . 150
17.5 Backpropagation through solver steps . . . 150
17.6 Alternatives: noise . . . 152
17.7 Complex examples . . . 152

18 Reducing Numerical Errors with Neural Operators **153**
18.1 Problem formulation . . . 153
18.2 Getting started with the implementation . . . 154
18.3 Simulation setup . . . 155
18.4 Network and transfer functions . . . 156
18.5 Training setup . . . 159
18.6 Interleaving simulation and NN . . . 161
18.7 Test evaluation . . . 163
18.8 Next steps . . . 168

19 Solving Inverse Problems with NNs **169**
19.1 Formulation . . . 170
19.2 Control of incompressible fluids . . . 170
19.3 Data generation . . . 171
19.4 Supervised initialization . . . 173
19.5 CFE pretraining with differentiable physics . . . 174
19.6 End-to-end training with differentiable physics . . . 174
19.7 Next steps . . . 177

20 Discussion of Differentiable Physics **179**
20.1 Integration . . . 179
20.2 Reducing data shift via interaction . . . 179
20.3 Generalization . . . 180

V Probabilistic Learning 181

21 Introduction to Probabilistic Learning 183
21.1 Uncertainty . . . 183
21.2 Forward or Backward? . . . 184
21.3 Simulation-based Inference . . . 184

22 Learning a Probability Distribution 187
22.1 Fundamentals: A Training Objective . . . 187
22.2 From Unconditional to Conditional . . . 188
22.3 Learning Distributions with Normalizing Flows . . . 188
22.4 Practical Example: Learning Gaussians . . . 189
22.5 A Simple Normalizing Flow based on Affine Couplings . . . 191
22.6 Neural ODEs: Making Normalizing Flows Continuous . . . 198
22.7 Summary of Normalizing Flows . . . 205

23 Score Matching 207
23.1 Gaussian Toy Dataset with Analytic Scores . . . 207
23.2 Learning the Score . . . 210
23.3 Langevin Dynamics . . . 214
23.4 Annealed Langevin Dynamics . . . 217
23.5 Score Summary . . . 222

24 Denoising 223
24.1 Latent Variable Models . . . 223
24.2 Full Denoising Algorithm . . . 225
24.3 Training with DDPM . . . 227

25 Flow Matching 235
25.1 Learning Flows with Velocities . . . 235
25.2 Mappings and Conditioning . . . 236
25.3 Implementing Flow Matching . . . 237
25.4 Summary . . . 242

26 Denoising and Flow Matching Side-by-side 243
26.1 Intro . . . 243
26.2 Problem statement . . . 244
26.3 Implementation and Setup . . . 244
26.4 Denoising with Diffusion Models . . . 245
26.5 Implementing DDPM . . . 246
26.6 Flow Matching . . . 249
26.7 Implementing Flow Matching . . . 250
26.8 Test Evaluation . . . 251
26.9 Quantified Results . . . 255
26.10 Next steps . . . 257

27 Incorporating Physical Constraints 259
27.1 Guiding Diffusion Models . . . 259
27.2 Physics-Guided Flow Matching . . . 260
27.3 Score Matching with Differentiable Physics . . . 264
27.4 Summary of Physics-based Diffusion Models . . . 267

28 Probabilistic Inverse Problem with Differentiable Simulations 269
28.1 Toy Problem setup . . . 269
28.2 Implementation Overview . . . 269

28.3 Physical System SDE . . . 270
28.4 Visualization of SDE paths . . . 271
28.5 Generate Training Data . . . 272
28.6 Training and Sliding Window Method . . . 277
28.7 Sampling SDE and ODE trajectories . . . 282
28.8 Next steps . . . 286

29 Diffusion-based Time Prediction 287
29.1 Conditioning . . . 288
29.2 Implementation . . . 289
29.3 Backbone Network Definition . . . 290
29.4 Variance Schedule . . . 295
29.5 Diffusion Model Definition . . . 295
29.6 Training . . . 299
29.7 Test Dataset . . . 302
29.8 Test Inference . . . 303
29.9 Accuracy of the Prediction . . . 305
29.10 Summarizing Time Predictions with Diffusion Models . . . 309

30 Unconditional Stability 311
30.1 Main Considerations for an Evaluation . . . 311
30.2 Comparing Architectures . . . 312
30.3 Stability Criteria . . . 313
30.4 Batch Size vs Rollout . . . 314
30.5 Summary . . . 315

31 Graph-based Diffusion Models 317
31.1 Diffusion Graph Net (DGN) . . . 317
31.2 Diffusion on Graphs . . . 318
31.3 Diffusion in Latent Space . . . 319
31.4 Turbulent Flows around Wings in 3D . . . 321
31.5 Distributional accuracy . . . 321
31.6 Computational Performance . . . 322

32 Distributional Accuracy of Diffusion Graph Nets 323
32.1 Implementation . . . 324
32.2 Sample-wise Accuracy . . . 327
32.3 Evaluating Distributional Accuracy . . . 331

33 Discussion of Probabilistic Learning 337

VI Reinforcement Learning 339

34 Introduction to Reinforcement Learning 341
34.1 Algorithms . . . 342
34.2 Proximal policy optimization . . . 342
34.3 Application to inverse problems . . . 343
34.4 Implementation . . . 344

35 Controlling Burgers' Equation with Reinforcement Learning 347
35.1 Overview . . . 347
35.2 Software installation . . . 347
35.3 Data generation . . . 348
35.4 Training via reinforcement learning . . . 349

35.5 RL evaluation . . . 351
35.6 Differentiable physics training . . . 352
35.7 Comparison between RL and DP . . . 356
35.8 Training progress comparison . . . 360
35.9 Next steps . . . 361

VII Improved Gradients **363**

36 Scale-Invariance and Inversion **365**
36.1 The crux of the matter . . . 365
36.2 Traditional optimization methods . . . 367
36.3 Gradient descent . . . 367
36.4 Quasi-Newton methods . . . 368
36.5 Inverse gradients . . . 370
36.6 Inverse simulators . . . 371
36.7 Summary . . . 371
36.8 Deep Dive into Inverse simulators . . . 372

37 Simple Example comparing Different Optimizers **375**
37.1 Problem formulation . . . 375
37.2 3 Spaces . . . 375
37.3 Implementation . . . 376
37.4 Gradient descent . . . 378
37.5 Newton . . . 379
37.6 Inverse simulators . . . 381
37.7 y Space . . . 384
37.8 Conclusions . . . 386
37.9 Approximate inversions . . . 387
37.10 Next steps . . . 388

38 Scale Invariant Physics Training **389**
38.1 NN training . . . 389
38.2 Loss functions . . . 390
38.3 Iterations and time dependence . . . 391
38.4 SIP training in action . . . 391
38.5 Discussion of SIP Training . . . 393

39 Learning to Invert Heat Conduction with Scale-invariant Updates **395**
39.1 Problem Statement . . . 395
39.2 Implementation . . . 395
39.3 Data generation . . . 396
39.4 Differentiable physics and gradient descent . . . 397
39.5 Stable SIP gradients . . . 397
39.6 Neural network and loss function . . . 399
39.7 Training . . . 400
39.8 Evaluation . . . 401
39.9 Next steps . . . 403

40 Half-Inverse Gradients **405**
40.1 Derivation . . . 405
40.2 Constructing the Jacobian . . . 406
40.3 Properties Illustrated via a Toy Example . . . 407
40.4 Well-conditioned . . . 407
40.5 Ill-conditioned . . . 408

40.6 Summary of Half-Inverse Gradients . . . 409

41 Coupled Oscillators with Half-Inverse Gradients 411
41.1 Inverse problem setup . . . 411
41.2 Problem statement . . . 411
41.3 Coupled linear oscillator simulation . . . 412
41.4 Training setup . . . 414
41.5 Training . . . 414
41.6 Evaluation . . . 420
41.7 Next steps . . . 422

42 Discussion of Improved Gradients 423
42.1 Addressing scaling issues . . . 423
42.2 Computational Resources . . . 423
42.3 Summary . . . 424

VIII Fast Forward Topics 425

43 Additional Topics 427

44 Model Reduction and Time Series 429
44.1 Reduced order models . . . 430
44.2 Time series . . . 430
44.3 End-to-end training . . . 431
44.4 Source code . . . 431

45 Unstructured Meshes and Meshless Methods 433
45.1 Types of computational meshes . . . 433
45.2 Unstructured meshes and graph neural networks . . . 434
45.3 Meshless and particle-based methods . . . 434
45.4 Continuous convolutions . . . 434
45.5 Learning the dynamics of liquids . . . 435
45.6 Source code . . . 436

46 Generative Adversarial Networks 437
46.1 Maximum likelihood estimation . . . 437
46.2 Adversarial training . . . 438
46.3 Regularization . . . 438
46.4 Conditional GANs . . . 439
46.5 Ambiguous solutions . . . 439
46.6 Spatio-temporal super-resolution . . . 439
46.7 Physical generative models . . . 440
46.8 Discussion . . . 440
46.9 Source code . . . 441

IX End Matter 443

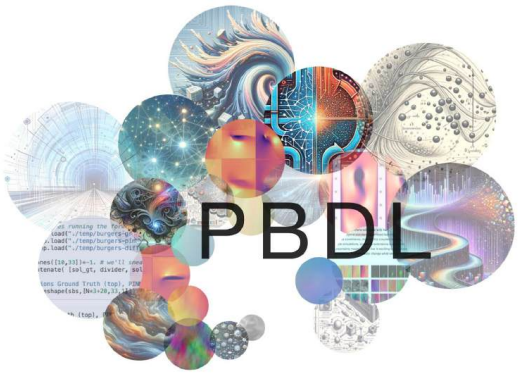
47 Outlook 445
47.1 Some specific directions . . . 445
47.2 Closing remarks . . . 446

48 References 447

49 Notation and Abbreviations **449**
49.1 Math notation: 449
49.2 Summary of the most important abbreviations: 449

Bibliography **451**

Proof Index **455**



Welcome to the *Physics-based Deep Learning Book* (v0.3, the *GenAI* edition)

TL;DR: This document is a hands-on, comprehensive guide to deep learning in the realm of physical simulations. Rather than just theory, we emphasize practical application: every concept is paired with interactive Jupyter notebooks to get you up and running quickly. Beyond traditional supervised learning, we dive into physical *loss-constraints*, *differentiable* simulations, *diffusion-based* approaches for *probabilistic generative AI*, as well as reinforcement learning and advanced neural network architectures. These foundations are paving the way for the next generation of scientific *foundation models*. We are living in an era of rapid transformation. These methods have the potential to redefine what's possible in computational science.

Note

What's new in v0.3? This latest edition takes things even further with a major new chapter on generative modeling, covering cutting-edge techniques like denoising, flow-matching, autoregressive learning, physics-integrated constraints, and diffusion-based graph networks. We've also introduced a dedicated section on neural architectures specifically designed for physics simulations. All code examples have been updated to leverage the latest frameworks.

Coming up

As a *sneak preview*, the next chapters will show:

- How to train neural networks to *predict the fluid flow around airfoils with diffusion modeling*. This gives a probabilistic *surrogate model* that replaces and outperforms traditional simulators.
- How to use model equations as residuals to train networks that *represent solutions*, and how to improve upon these residual constraints by using *differentiable simulations*.
- How to more tightly interact with a full simulator for *inverse problems*. E.g., we'll demonstrate how to circumvent the convergence problems of standard reinforcement learning techniques by leveraging *simulators in the training loop*.
- We'll also discuss the importance of *choosing the right network architecture*: whether to consider global or local interactions, continuous or discrete representations, and structured versus unstructured graph meshes.

Throughout this text, we will introduce different approaches for introducing physical models into deep learning, i.e., *physics-based deep learning* (PBDL) approaches. These algorithmic variants will be introduced in order of increasing tightness of the integration, and the pros and cons of the different approaches will be discussed. It's important to know in which scenarios each of the different techniques is particularly useful.

Executable code, right here, right now

We focus on Jupyter notebooks, a key advantage of which is that all code examples can be executed *on the spot*, from your browser. You can modify things and immediately see what happens – give it a try by [running this teaser example in your browser].

Plus, Jupyter notebooks are great because they're a form of literate programming.

Comments and suggestions

This *book*, where "book" stands for a collection of digital texts and code examples, is maintained by the Physics-based Simulation Group at TUM. Feel free to contact us if you have any comments, e.g., via old fashioned email. If you find mistakes, please also let us know! We're aware that this document is far from perfect, and we're eager to improve it. Thanks in advance ! Btw., we also maintain a link collection with recent research papers.

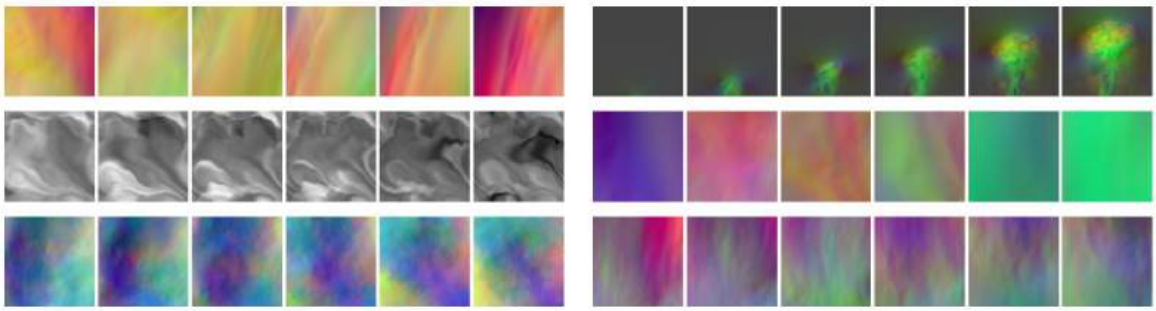


Fig. 1: Some visual examples of numerically simulated time sequences. In this book, we explain how to realize algorithms that use neural networks alongside numerical solvers.

Thanks!

This project would not have been possible without the help of the many people who contributed to it. A big thanks to everyone Here's an alphabetical list:

- Benjamin Holzschuh
- Philipp Holl
- Georg Kohl
- Mario Lino
- Qiang Liu
- Patrick Schnell
- Felix Trost
- Nils Thuerey

Additional thanks go to Li-Wei Chen, Xin Luo, Maximilian Mueller, Chloe Paillard, Kiwon Um, and all github contributors!

Citation

If you find this book useful, please cite it via:

```
@book{thuerey2021pbdl,
  title={Physics-based Deep Learning},
  author={N. Thuerey and B. Holzschuh  and P. Holl  and G. Kohl  and M. Lino  and Q.␣
↪Liu and P. Schnell  and F. Trost},
  url={https://physicsbaseddeeplearning.org},
  year={2021},
  publisher={WWW}
}
```

Time to get started

The future of simulation is being rewritten, and with the following AI and deep learning techniques, you'll be at the forefront of these developments. Let's dive in!

Part I

Introduction

CHAPTER

ONE

A TEASER EXAMPLE

Let's start with a very reduced example that highlights some of the key capabilities of physics-based learning approaches. Let's assume our physical model is a very simple equation: a parabola along the positive x-axis. We'll also directly use this example to give an outlook towards probabilistic "generative AI" approaches.

Despite being very simple, there are two solutions for every point along x, i.e. we have two *modes*, one above the other one below the x-axis, as shown on the left below. If we don't take care, a conventional learning approach will give us an approximation like the red one shown in the middle, which is completely off. With an improved learning setup, e.g., by using a discretized numerical solver, we can at least accurately represent one of the modes of the solution (shown in green on the right). Interestingly, approaches that learn the full distribution at each point, flow matching as a representative of diffusion models is used below, can capture both modes!

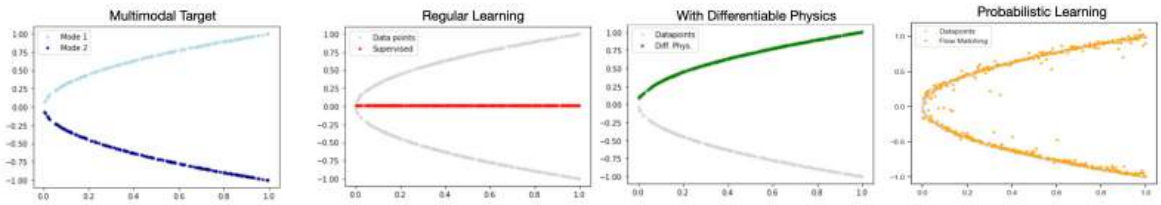


Fig. 1.1: Side by side - supervised versus differentiable physics and probabilistic training.

1.1 Differentiable physics

One of the key concepts of the following chapters is what we'll call *differentiable physics* (DP). This means that we use domain knowledge in the form of model equations, and then integrate discretized versions of these models into the training process. As implied by the name, having differentiable formulations and operators is crucial for this process to integrate with neural networks training.

Let's illustrate the properties of deep learning via DP with the following example: We'd like to find an unknown function f^* that generates solutions from a space Y, taking inputs from X, i.e. $f^* : X \to Y$. In the following, we'll often denote *idealized*, and unknown functions with a $*$ superscript, in contrast to their discretized, realizable counterparts without this superscript. Let's additionally assume we have a generic differential equation $\mathcal{P}^* : Y \to Z$ (our *model* equation), that encodes a property of the solutions, e.g. some real world behavior we'd like to match. Later on, P^* will often represent time evolutions, but it could also be a conservation law (e.g., conservation of mass, then $\mathcal{P}^*$ would measure divergence).

Using a neural network f to learn the unknown and ideal function f^*, we could turn to classic *supervised* training to obtain f by collecting data. This classical setup requires a dataset by sampling x from X and adding the corresponding solutions y from Y. We could obtain these, e.g., by classical numerical techniques. Then we train the NN f with classic methods using this dataset.

In contrast to this supervised approach, employing a differentiable physics approach takes advantage of the fact that we can often use a discretized version of the physical model $\mathcal{P}$ and employ it to guide the training of f. I.e., we want f to be aware of our *simulator* $\mathcal{P}$, and to *interact* with it. This can give fundamental improvements, as we'll illustrate below with a very simple example (more complex ones will follow later on).

Note that in order for the DP approach to work, $\mathcal{P}$ has to be *differentiable*, as implied by the name. These differentials, in the form of a gradient, are what's driving the learning process and neural network integration.

1.2 Finding the inverse function of a parabola

To illustrate the difference of supervised and DP approaches, we consider the following simplified setting: Given the function $\mathcal{P} : y \rightarrow y^2$ for y in the interval $[0, 1]$, find the unknown function f such that $\mathcal{P}(f(x)) = x$ for all x in $[0, 1]$. E.g., for $x = 0.5$, solutions would be $\pm\sqrt{0.5}$. Note: to make things a bit more interesting, we're using y^2 here for $\mathcal{P}$ instead of the more common x^2 parabola, and the *discretization* is simply given by representing the x and y via floating point numbers in the computer for this simple case.

We know that possible solutions for f are the positive or negative square root function (for completeness: piecewise combinations would also be possible). This sounds easy, so let's try to train a neural network to approximate this inverse mapping f. Doing this in the "classical" supervised manner, i.e. purely based on data, is an obvious starting point. After all, this approach was shown to be a powerful tool for a variety of other applications, e.g., in computer vision.

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

For supervised training, we can employ our solver $\mathcal{P}$ for the problem to pre-compute the solutions we need for training: We randomly choose between the positive and the negative square root. This resembles the general case, where we would gather all data beforehand, e.g., using optimization techniques to compute the solutions or even experiments. This data collection typically does not favor one particular mode from multimodal solutions.

```
# Generate data
N = 10000
X = np.random.random(N).astype(np.float32).reshape(-1, 1)

# Generation of Y-Data
sign = (- np.ones((N,))).astype(np.float32) ** np.random.randint(2, size=N)
Y = (np.sqrt(X.flatten()) * sign).reshape(-1, 1).astype(np.float32)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X)
Y_tensor = torch.tensor(Y)
```

Now we can define a network. We'll use a simple fully connected architecture with three hidden layers and ReLU activations.

```
# Define the neural network
class SimpleNN(nn.Module):
    def __init__(self, hiddendim=10):
        super(SimpleNN, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.fc1 = nn.Linear(1, hiddendim)
        self.fc2 = nn.Linear(hiddendim, hiddendim)
        self.fc3 = nn.Linear(hiddendim, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)  # Linear output
        return x
```

Next we can instantiate the model (using a hidden dimension of 128), specify a loss function (will used a simple mean squared error with PyTorch's `MSELoss()`), and the Adam optimizer. The network is trained for 50 epochs in the loop below:

```
nn_sup = SimpleNN(hiddendim=128)
criterion = nn.MSELoss()
optimizer = optim.Adam(nn_sup.parameters(), lr=0.001)

# Training loop
epochs = 50
batch_size = 5

for epoch in range(epochs):
    permutation = torch.randperm(N)
    epoch_loss = 0.0

    for i in range(0, N, batch_size):
        indices = permutation[i:i+batch_size]
        batch_x, batch_y = X_tensor[indices], Y_tensor[indices]

        optimizer.zero_grad()
        outputs = nn_sup(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    if(epoch%10==9): print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss/N:.6f}")
```

```
Epoch 10/50, Loss: 0.100193
Epoch 20/50, Loss: 0.100193
Epoch 30/50, Loss: 0.100207
Epoch 40/50, Loss: 0.100202
Epoch 50/50, Loss: 0.100203
```

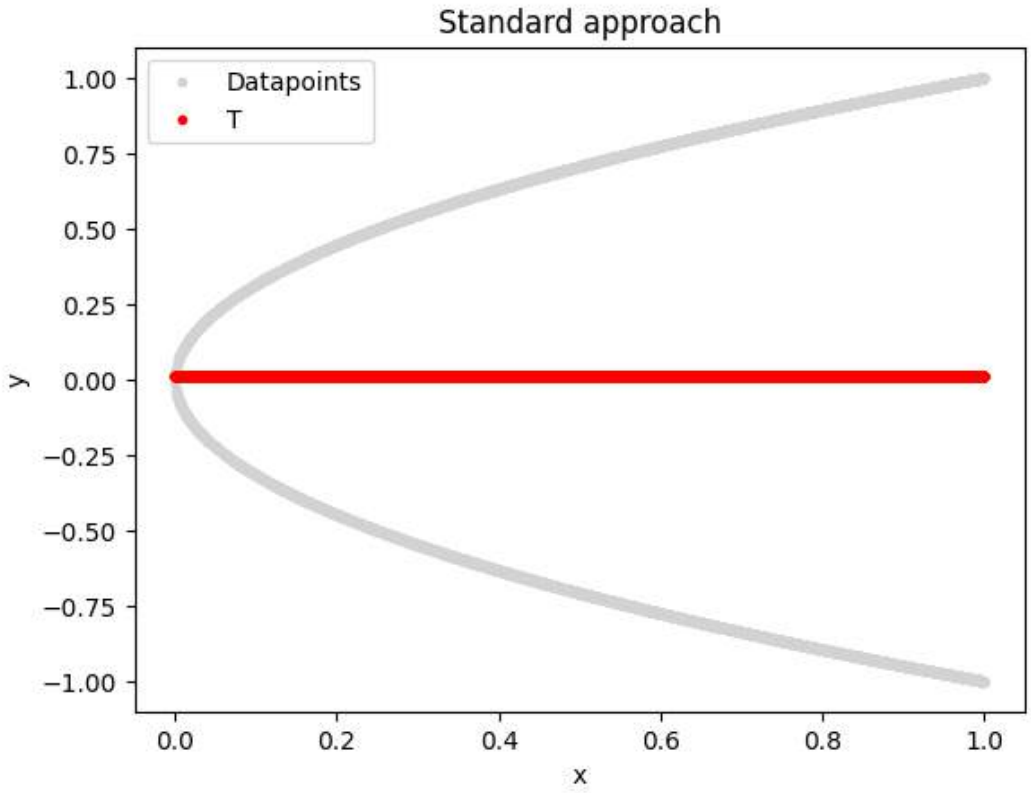
As both NN and the data set are fairly small, the training converges quickly. Let's plot the solution: the following one shows the data in light gray, and the supervised solution in red.

```
import matplotlib.pyplot as plt

plt.plot(X,Y,'.',label='Datapoints', color="lightgray")
plt.plot(X, nn_sup(torch.tensor(X)).detach(), '.',label='T', color="red")
plt.xlabel('x')
plt.ylabel('y')
```

(continues on next page)

(continued from previous page)

```
plt.title('Standard approach')
plt.legend()
plt.show()
```



This is obviously completely wrong! The red solution is nowhere near one of the two modes of our solution shown in gray. The training process has averaged between the data points on both sides of the x-axis and therefore fails to find satisfying solutions to the problem above.

Note that the red line is often not perfectly at zero, which is where the two modes of the solution should average out in the continuous setting. This is caused by the relatively coarse sampling with only 200 points in this example.

1.3 A differentiable physics approach

Now let's apply the differentiable physics idea as mentioned above to find f: we'll directly include our discretized model $\mathcal{P}$ in the training. Note that in this context, $\mathcal{P}^*$ and $\mathcal{P}$ actually provide a mapping back to the input space X, i.e. $\mathcal{P}^* : Y \rightarrow X$.

There is no real data generation step; we only need to sample from the $[0, 1]$ interval. We'll simply keep the same x locations used in the previous case, and a new instance of a NN with the same architecture as before `nn_dp`:

```
# X-Data
# X = X , we can directly re-use the X from above, nothing has changed...

# P maps Y back to X, simply by computing a square, as y is a TF tensor input, the␣
↪square operation **2 will be differentiable
def P(y):
    return torch.square(y)

# Define custom loss function using the "physics" operator P
def loss_function(y_true, y_pred):
    return criterion(y_true, P(y_pred))
```

The loss function is the crucial point for training: we directly incorporate the function to learn, f called `nn_dp`, into the loss. Keras will evaluate `nn_dp` for an inpupt from `X`, and provide the output in the second argument `y_from_nn_dp`. On this output, we'll run our "solver" `P`, and the result should match the correct answer `y_true`. In this simple case, the `loss_dp` function simply computes the square of the prediction `y_pred`.

Later on, a lot more could happen here: we could evaluate finite-difference stencils on the predicted solution, or compute a whole implicit time-integration step of a solver. Here we have a simple *mean-squared error* term of the form $|\mathcal{P}(y_{\text{pred}}) - x_{\text{true}}|^2$, which we are minimizing during training. It's not necessary to make it so simple: the more knowledge and numerical methods we can incorporate, the better we can guide the training process.

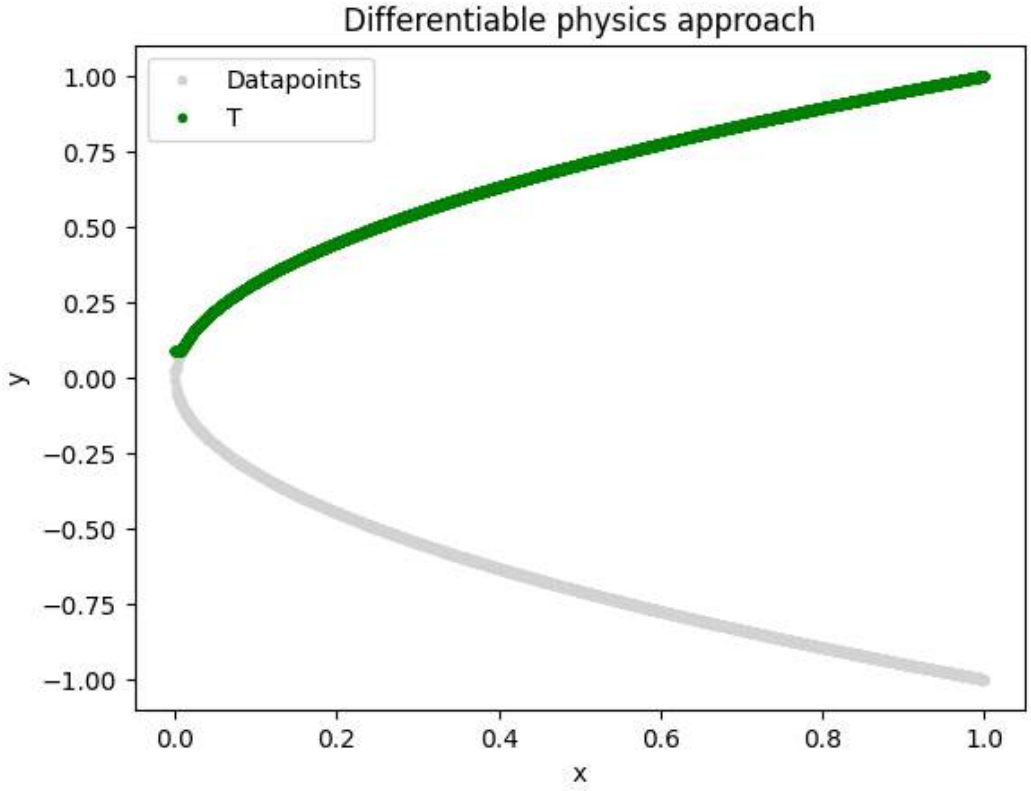
Let's instantiate the neural network again, and train the network with the *differentiable physics* loss:

```
nn_dp = SimpleNN(hiddendim=128)
optimizer = optim.Adam(nn_dp.parameters(), lr=0.001)

# Training loop
batch_size = 5

for epoch in range(epochs):
    permutation = torch.randperm(N)
    epoch_loss = 0.0

    for i in range(0, N, batch_size):
        indices = permutation[i:i+batch_size]
        batch_x = X_tensor[indices]

        optimizer.zero_grad()
        outputs = nn_dp(batch_x)
        loss = loss_function(batch_x, outputs)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    if(epoch%10==9): print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss/N:.6f}")
```

```
Epoch 10/50, Loss: 0.000004
Epoch 20/50, Loss: 0.000003
Epoch 30/50, Loss: 0.000003
Epoch 40/50, Loss: 0.000002
Epoch 50/50, Loss: 0.000002
```

Now the network actually has learned a good inverse of the parabola function! The following plot shows the solution in green.

```
# Results
plt.plot(X,Y,'.',label='Datapoints', color="lightgray")
plt.plot(X, nn_dp(torch.tensor(X)).detach(), '.',label='T', color="green")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Differentiable physics approach')
plt.legend()
plt.show()
```



This looks much better , at least if we're avoiding the origin (this part would need some extra attention).

What has happened here?

- We've prevented an undesired averaging of multiple modes in the solution by evaluating our discrete model w.r.t. current prediction of the network, rather than using a pre-computed solution. This lets us find the best mode near the network prediction, and prevents an averaging of the modes that exist in the solution manifold.
- We're still only getting one side of the curve! This is to be expected because we're representing the solutions with a deterministic function. Hence, we can only represent a single mode. Interestingly, whether it's the top or bottom

mode is determined by the random initialization of the weights in f - run the example a couple of times to see this effect in action. To capture multiple modes we'd need to extend the NN to capture the full distribution of the outputs and parametrize it with additional dimensions.

CHAPTER

TWO

A PROBABILISTIC GENERATIVE AI APPROACH

As hinted at above, we can do even better with state of the art AI techniques: we can learn the full *distribution* of the posterior, in our case the different answers for each x. Below, we'll use *flow matching* as a state of the art approach from generative, diffusion-based algorithms.

As these methods work with noisy data, we first need to specify a new dataloader, that adds different amounts of "noise" onto the y values of our data, so that the network can learn the right direction towards the two possible modes.

```
from torch.utils.data import Dataset, DataLoader
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class FlowMatchingDataset(Dataset):
    def __init__(self, data_x, data_y, n_samples=1000, sigma_min=1e-4):
        super().__init__()
        self.n_samples = n_samples
        self.sigma_min = sigma_min
        self.data_x = data_x
        self.data_y = data_y

    def __len__(self):
        return self.n_samples

    def __getitem__(self, idx):
        x0 = np.random.multivariate_normal([0.0, 0.0], np.eye(2), 1)[0]
        t = np.random.rand()  # scalar in [0,1]
        dx = self.data_x[idx] #:idx+1]
        dy_org = self.data_y[idx]# :idx+1]
        x0[0] = dx[0] # keep x value
        x1 = np.concatenate([dx,dy_org],axis=0)
        #print([self.data_x.shape,dx.shape,x1.shape])

        x_t = (1 - ( 1 - self.sigma_min) * t) * x0 + t * x1
        u_t = (x1 - x0)
        x_t = torch.tensor(x_t, dtype=torch.float32)
        t   = torch.tensor([t], dtype=torch.float32)
        u_t = torch.tensor(u_t, dtype=torch.float32)
        return x_t, t, u_t
```

The network itself is not much different from before, we only need to add an additional time input `t`:

```
class VelocityNet(nn.Module):
    def __init__(self, hidden_dim, in_dim=2, time_dim=1, out_dim=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim + time_dim, hidden_dim),
```

(continues on next page)

(continued from previous page)

```
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, out_dim)
        )

    def forward(self, x, t):
        xt = torch.cat([x, t], dim=1)
        return self.net(xt)
```

Training proceeds in line with before, we simply sample noisy samples from the dataset, and train the network to move samples towards the solutions in the dataset:

```
batch_size = 128

dataset = FlowMatchingDataset(X, Y, n_samples=N)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

nn_fm = VelocityNet(hidden_dim=128).to(device)
optimizer = optim.Adam(nn_fm.parameters(), lr=0.001)
criterion = nn.MSELoss()

for epoch in range(epochs):
    running_loss = 0.0
    for x_t, t, u_t in dataloader:
        x_t = x_t.to(device)
        t   = t.to(device)
        u_t = u_t.to(device)
        optimizer.zero_grad()
        pred_v = nn_fm(x_t, t)
        loss = criterion(pred_v, u_t)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * x_t.size(0)
    running_loss /= len(dataset)
    if(epoch%10==9): print(f"Epoch {epoch + 1}/{epochs}, Loss: {running_loss:.4f}")
```

```
Epoch 10/50, Loss: 0.3837
Epoch 20/50, Loss: 0.3773
Epoch 30/50, Loss: 0.3735
Epoch 40/50, Loss: 0.3726
Epoch 50/50, Loss: 0.3715
```

For evaluation, we now repeatedly call the neural network to improve an initial noisy sample drawn from a simple distribution, and step by step move it towards a "correct" solution. This is done in the `integrate_flow` function below.

```
def integrate_flow(nn, x0, t_span=(0.0, 1.0), n_steps=100):
    trajectory = []
    t = torch.linspace(t_span[0], t_span[1], n_steps).to(x0.device)
    dt = 1./n_steps
    x_in = x0
    for i in range(n_steps):
        x0 = x0 + dt * nn(x0, torch.tensor([i/n_steps]).expand(x0.shape[0], 1) )
        x0[:,0] = x_in[:,0] # condition on original x position
        trajectory.append(x0)
```

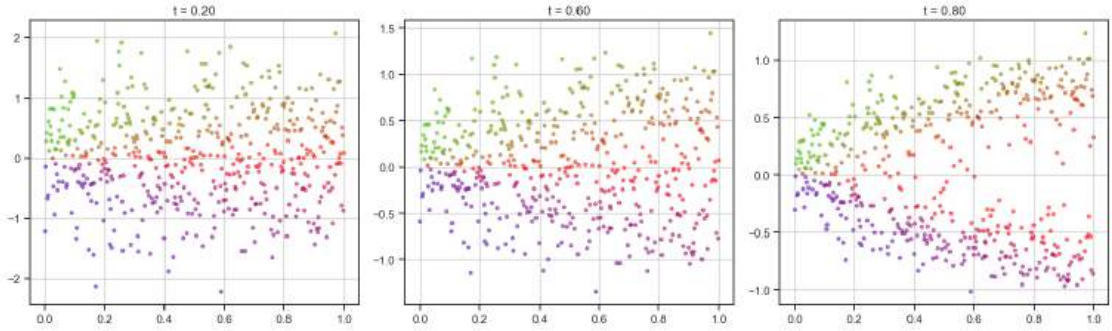
(continues on next page)

(continued from previous page)

```
    return trajectory, t

# Generate samples along x, then randomize along y
n_gen = 500
x_in = torch.linspace(0.,1., n_gen).to(device)
y_in = torch.randn(n_gen).to(device) * 0.95
x0_gen = torch.stack([x_in,y_in],axis=-1)
trajectory, time_points = integrate_flow(nn_fm, x0_gen)
```

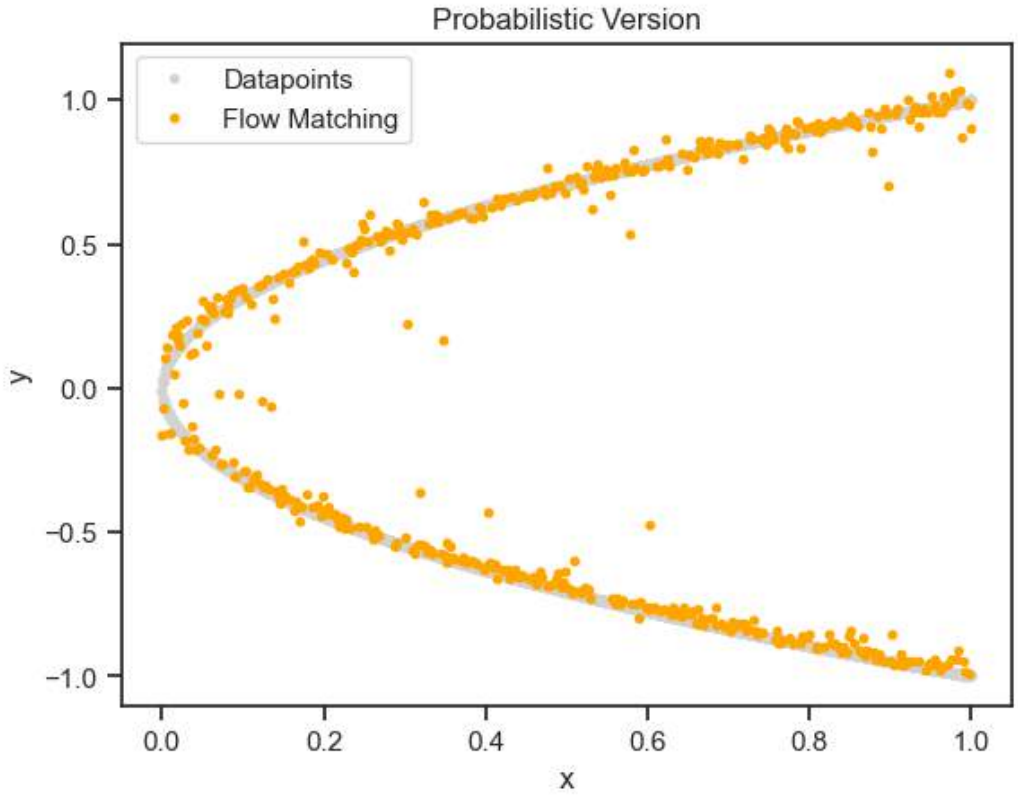
To illustrate this flow process, the next cell shows samples at different times in the flow integration. The initial random distribution slowlyl transforms into the bi-modal one for our parabola targets.

```
import seaborn as sns
sns.set_theme(style="ticks", palette="pastel")

def get_angle_colors(positions):
    angles = np.arctan2(positions[:, 1], positions[:, 0])
    angles_deg = (np.degrees(angles) + 360) % 360
    colors = np.zeros((len(positions), 3))
    for i, angle in enumerate(angles_deg):
        segment = int(angle / 120)
        local_angle = angle - segment * 120
        if segment == 0:    # 0 degrees to 120 degrees (R->G)
            colors[i] = [1 - local_angle/120, local_angle/120, 0]
        elif segment == 1:  # 120 degrees to 240 degrees (G->B)
            colors[i] = [0, 1 - local_angle/120, local_angle/120]
        else:               # 240 degrees to 360° (B->R)
            colors[i] = [local_angle/120, 0, 1 - local_angle/120]
    return colors

desired_times = [0.2, 0.6, 0.8,]
time_np = time_points.detach().cpu().numpy()
n_steps = len(time_np)
indices = [np.argmin(np.abs(time_np - t_val)) for t_val in desired_times]

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 5))
axes = axes.ravel()  # flatten the 2D array for easier indexing

xx, yy = np.mgrid[0:1:100j, -1:1:100j]
positions = np.vstack([xx.ravel(), yy.ravel()])

for i, idx in enumerate(indices):
    ax = axes[i]
    x_t = trajectory[idx].detach().cpu().numpy()
    if i == 0:
        c = get_angle_colors(x_t)
    ax.scatter(x_t[:, 0], x_t[:, 1], alpha=0.5, s=10, color=c)
    ax.set_title(f"t = {time_np[idx]:.2f}")
    ax.grid(True)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Let's also plot the solution in line with the supervised and differentiable physics variants above:

```
# Results
plt.plot(X,Y,'.',label='Datapoints', color="lightgray")
plt.plot(trajectory[-1][:,0].detach(),trajectory[-1][:,1].detach(),'.',label='Flow␣
↪Matching', color="orange")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Probabilistic Version')
plt.legend()
plt.show()
```



As promised, this approach actually resolves both "modes" of the solution in the form of points above and below the x-axis. It's still a bit noisy, but this could be alleviated by improving the learning setup, e.g., a larger network would help.

An obvious question here also is: we're back to training only with data, how about integrating the physics? That's an obvious point for improvements, and we'll address diffusion-based methods with physical constraints in more detail in a later section. As an outlook: physics-priors can help especially to drive the somewhat noisy output of a neural network towards an accurate solution.

2.1 Discussion

It's a very simple example, but it very clearly shows a failure case for supervised learning. While it might seem very artificial at first sight, many practical PDEs exhibit a variety of these modes, and it's often not clear where (and how many) exist in the solution manifold we're interested in. Using supervised learning is very dangerous in such cases. We might unknowingly get an average of these different modes.

Good and obvious examples are bifurcations in fluid flow. Smoke rising above a candle will start out straight, and then, due to tiny perturbations in its motion, start oscillating in a random direction. The images below illustrate this case via *numerical perturbations*: the perfectly symmetric setup will start turning left or right, depending on how the approximation errors build up. Averaging the two modes would give an unphysical, straight flow similar to the parabola example above.

Similarly, we have different modes in many numerical solutions, and typically it's important to recover them, rather than averaging them out. Hence, we'll show how to leverage training via *differentiable physics* in the following chapters for more practical and complex cases.

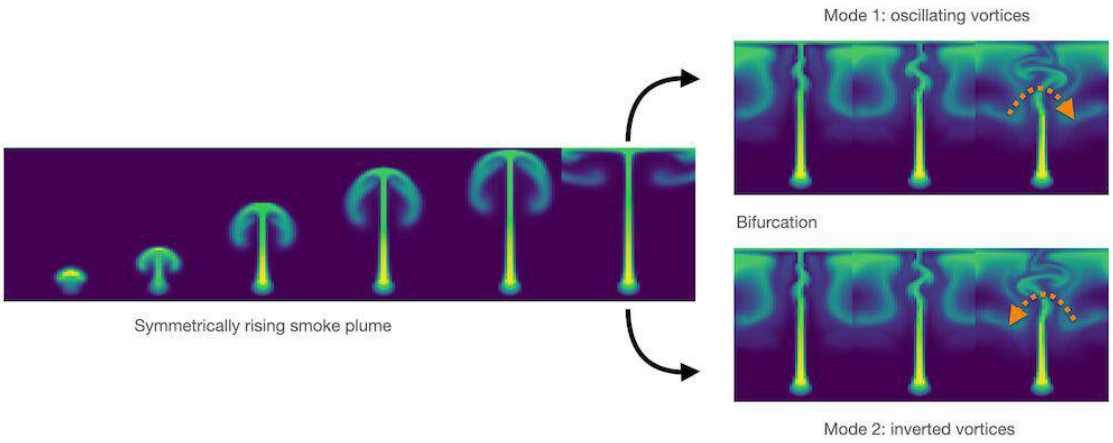


Fig. 2.1: A bifurcation in a buoyancy-driven fluid flow: the "smoke" shown in green color starts rising in a perfectly straight manner, but tiny numerical inaccuracies grow over time to lead to an instability with vortices alternating to one side (top-right), or in the opposite direction (bottom-right).

2.2 Next steps

For each of the following notebooks, there's a "next steps" section like the one below which contains recommendations about where to start modifying the code. After all, the whole point of these notebooks is to have readily executable programs as a basis for own experiments. The data set and NN sizes of the examples are often quite small to reduce the runtime of the notebooks, but they're nonetheless good starting points for potentially complex and large projects.

For the simple DP example above:

- This notebook is intentionally using a very simple setup. You can change the training setup or network architectures to improve the solutions. E.g., provide more samples around zero to improve the solution near the origin.
- Or try extending the setup to a 2D case, i.e. a paraboloid. Given the function $\mathcal{P} : (y_1, y_2) \rightarrow y_1^2 + y_2^2$, find an inverse function f such that $\mathcal{P}(f(x)) = x$ for all x in $[0, 1]$.
- If you want to experiment without installing anything, you can also [run this notebook in colab].

CHAPTER

THREE

OVERVIEW

The name of this book, *Physics-Based Deep Learning*, denotes combinations of physical modeling and **numerical simulations** with methods based on **artificial intelligence**, i.e. neural networks. The general direction of Physics-Based Deep Learning, also going under the name *Scientific Machine Learning*, represents a very active, quickly growing and exciting field of research. The following chapter will give a more thorough introduction to the topic and establish the basics for following chapters.

Fig. 3.1: Understanding our environment, and predicting how it will evolve is one of the key challenges of humankind. A key tool for achieving these goals are computer simulations, and the next generation of these simulations will likely strongly profit from integrating AI and deep learning components, in order to make even better accurate predictions about the phenomena in our environment.

3.1 Motivation

From weather and climate forecasts [Sto14] (see the picture above), over quantum physics [OMalleyBK+16], to the control of plasma fusion [MLA+19], using numerical analysis to obtain solutions for physical models has become an integral part of science.

In recent years, artificial intelligence driven by *deep neural networks*, have led to impressive achievements in a variety of fields: from image classification [KSH12] over natural language processing [RWC+19], and protein folding [Qur19], to various foundation models. The field is very vibrant and quickly developing, with the promise of vast possibilities.

3.1.1 Replacing traditional simulations?

These success stories of deep learning (DL) approaches have given rise to concerns that this technology has the potential to replace the traditional, simulation-driven approach to science. E.g., recent works show that NN-based surrogate models achieve accuracies required for real-world, industrial applications such as airfoil flows [CT22], while at the same time outperforming traditional solvers by orders of magnitude in terms of runtime.

Instead of relying on models that are carefully crafted from first principles, can sufficiently large datasets be processed instead to provide the correct answers? As we'll show in the next chapters, this concern is unfounded. Rather, it is crucial for the next generation of simulation systems to bridge both worlds: to combine *classical numerical* techniques with *A.I.* methods. In addition, the latter offer exciting new possibilities in areas that have been challenging for traditional methods, such as dealing with complex *distributions and uncertainty* in simulations.

One central reason for the importance of the combination with numerics is that DL approaches are powerful, but at the same time strongly profit from domain knowledge in the form of physical models. DL techniques and NNs are novel, sometimes difficult to apply, and it is admittedly often non-trivial to properly integrate our understanding of physical processes into the learning algorithms.

Over the last decades, highly specialized and accurate discretization schemes have been developed to solve fundamental model equations such as the Navier-Stokes, Maxwell's, or Schroedinger's equations. Seemingly trivial changes to the discretization can determine whether key phenomena are visible in the solutions or not. Rather than discarding the powerful methods that have been developed in the field of numerical mathematics, this book will show that it is highly beneficial to use them as much as possible when applying DL.

3.1.2 Black boxes?

In the past, AI and DL methods have often associated trained neural networks with *black boxes*, implying that they are something that is beyond the grasp of human understanding. However, these viewpoints typically stem from relying on hearsay and general skepticism about "hyped" topics.

The situation is a very common one in science, though: we are facing a new class of methods, and "all the gritty details" are not yet fully worked out. This is and has been pretty common for all kinds of scientific advances. Numerical methods themselves are a good example. Around 1950, numerical approximations and solvers had a tough standing. E.g., to cite H. Goldstine, numerical instabilities were considered to be a "constant source of anxiety in the future" [Gol90]. By now we have a pretty good grasp of these instabilities, and numerical methods are ubiquitous and well established. AI, neural networks follow the same path of human progress.

Thus, it is important to be aware of the fact that – in a way – there is nothing very special or otherworldly to deep learning methods. They're simply a new set of numerical tools. That being said, they're clearly very new, and right now definitely the most powerful set of tools we have for non-linear problems. That all the details aren't fully worked out and have nicely been written up shouldn't stop us from including these powerful methods in our numerical toolbox.

3.1.3 Reconciling AI and simulations

Taking a step back, the aim of this book is to build on all the powerful techniques that we have at our disposal for numerical simulations, and use them wherever we can in conjunction with deep learning. As such, a central goal is to *reconcile* the AI viewpoint with physical simulations.

Goals of this document

The key aspects that we will address in the following are:

- how to use deep learning techniques to **solve PDE** problems,
- how to combine them with **existing knowledge** of physics,

- without **discarding** numerical methods.

At the same time, it's worth noting what we won't be covering:

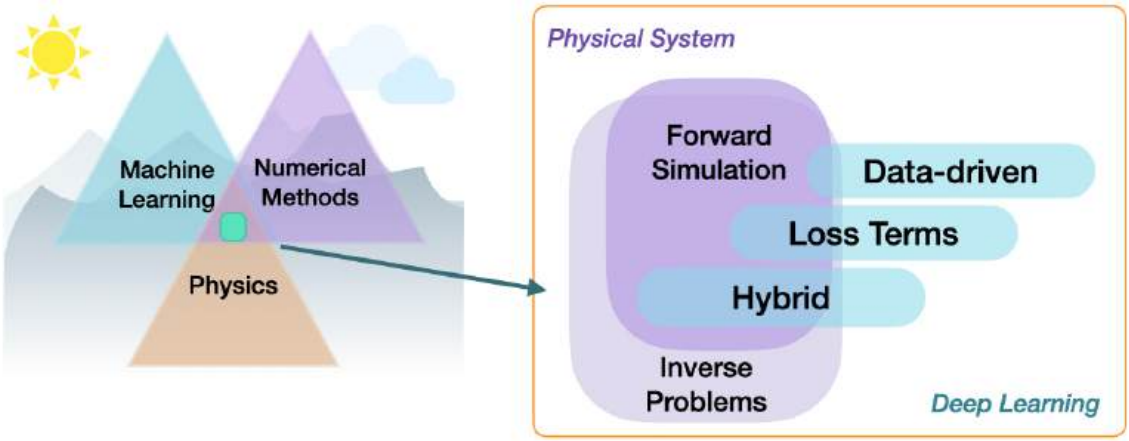
- there's no in-depth **introduction** to deep learning and numerical simulations (there are great other works already taking care of this),
- and the aim is neither a broad survey of research articles in this area.

The resulting methods have a huge potential to improve what can be done with numerical methods: in scenarios where a solver targets cases from a certain well-defined problem domain repeatedly, it can for instance make a lot of sense to once invest significant resources to train a neural network that supports the repeated solves. The development of large so-called "foundation models" is especially promising in this area. Based on the domain-specific specialization via fine-tuning with a smaller dataset, a hybrid solver could vastly outperform traditional, generic solvers. And despite the many open questions, first publications have demonstrated that this goal is a realistic one [KSA+21, UBH+20].

Another way to look at it is that all mathematical models of our nature are idealized approximations and contain errors. A lot of effort has been made to obtain very good model equations, but to make the next big step forward, AI and DL methods offer a very powerful tool to close the remaining gap towards reality [AAC+19].

3.2 Categorization

Within the area of *physics-based deep learning*, we can distinguish a variety of different approaches, e.g., targeting constraints, combined methods, optimizations and applications. More specifically, all approaches either target *forward* simulations (predicting state or temporal evolution) or *inverse* problems (e.g., obtaining a parametrization or state for a physical system from observations).



No matter whether we're considering forward or inverse problems, the most crucial differentiation for the following topics lies in the nature of the integration between DL techniques and the domain knowledge, typically in the form of model equations via partial differential equations (PDEs). The following three categories can be identified to roughly categorize *physics-based deep learning* (PBDL) techniques:

- *Supervised*: the data is produced by a physical system (real or simulated), but no further interaction exists. This is the classic machine learning approach.
- *Loss-terms*: the physical dynamics (or parts thereof) are encoded in the loss function, typically in the form of differentiable operations. The learning process can repeatedly evaluate the loss, and usually receives gradients from a PDE-based formulation. These soft constraints sometimes also go under the name "physics-informed" training.

- *Hybrid*: the full physical simulation is interleaved and combined with an output from a deep neural network; this usually requires a fully differentiable simulator. It represents the tightest coupling between the physical system and the learning process and results in a hybrid solver that combines classic techniques with AI-based ones.

Thus, methods can be categorized in terms of forward versus inverse solve, and how tightly the physical model is integrated with the neural network. Here, especially hybrid approaches that leverage *differentiable physics* allow for very tight integration of deep learning and numerical simulation methods.

3.2.1 Naming

It's worth pointing out that what we'll call "differentiable physics" in the following appears under a variety of different names in other resources and research papers. The differentiable physics name is motivated by the differentiable programming paradigm in deep learning. Here we, e.g., also have "differentiable rendering approaches", which deal with simulating how light leads forms the images we see as humans. In contrast, we'll focus on *physical* simulations from now on, hence the name.

When coming from other backgrounds, other names are more common however. E.g., the differentiable physics approach is equivalent to using the adjoint method, and coupling it with a deep learning procedure. Effectively, it is also equivalent to apply backpropagation / reverse-mode differentiation to a numerical simulation. However, as mentioned above, motivated by the deep learning viewpoint, we'll refer to all these as "differentiable physics" approaches from now on.

The hybrid solvers that result from integrating DL with a traditional solver can also be seen as a classic topic: in this context, the neural network has the task to *correct* the solver. This correction can in turn either target numerical errors, or unresolved terms in an equation. This is a fundamental problem in science that has been addressed under various names, e.g., as the *closure problem* in fluid dynamics and turbulence, as *homogenization* or *coarse-graining* in material science, and *parametrization* in climate and weather simulation. The re-invention of this goal in the different fields points to the importance of the underlying problem, and this text will illustrate the new ways that DL offers to tackle it.

3.3 Looking ahead

Physics simulations are a huge field, and we won't be able to cover all possible types of physical models and simulations.

Note

Rather, the focus of this book lies on:

- Dense *field-based simulations* (no Lagrangian methods)
- Combinations with *deep learning* (plenty of other interesting ML techniques exist, but won't be discussed here)
- Experiments are left as an *outlook* (i.e., replacing synthetic data with real-world observations)

It's also worth noting that we're starting to build the methods from some very fundamental building blocks. Here are some considerations for skipping ahead to the later chapters.

Hint: You can skip ahead if…

- you're very familiar with numerical methods and PDE solvers, and want to get started with DL topics right away. The *Supervised Training* chapter is a good starting point then.

- On the other hand, if you're already deep into NNs&Co, and you'd like to skip ahead to the research related topics, we recommend starting in the *Physical Loss Terms* chapter, which lays the foundations for the next chapters.

A brief look at our *notation* in the *Notation and Abbreviations* chapter won't hurt in both cases, though!

3.4 Implementations

This text also represents an introduction to deep learning and simulation APIs. We'll primarily use the popular deep learning API *pytorch* https://pytorch.org, but also a bit of *tensorflow* https://www.tensorflow.org, and additionally give introductions into the differentiable simulation framework *ΦFlow (phiflow)* https://github.com/tum-pbs/PhiFlow. Some examples also use *JAX* https://github.com/google/jax, which provides an interesting alternative. Thus after going through these examples, you should have a good overview of what's available in current APIs, such that the best one can be selected for new tasks.

As we're dealing with stochastic optimizations in most of the Jupyter notebooks, many of the following code examples will produce slightly different results each time they're run. This is fairly common with NN training, but it's important to keep in mind when executing the code. It also means that the numbers discussed in the text might not exactly match the numbers you'll see after re-running the examples.

3.5 Models and Equations

Below we'll give a *very* brief intro to deep learning, primarily to introduce the notation. In addition we'll discuss some *model equations* below. Note that we'll avoid using *model* to denote trained neural networks, in contrast to some other texts and APIs. These will be called "NNs" or "networks". A "model" will typically denote a set of model equations for a physical effect, usually PDEs.

3.5.1 Deep learning and neural networks

The goal in deep learning is to approximate an unknown function

$$f^*(x) = y^*, \tag{3.1}$$

where y^* denotes reference or "ground truth" solutions, and $f^*(x)$ should be approximated with an NN $f(x;\theta)$. We typically determine f with the help of some variant of a loss function $L(y, y^*)$, where $y = f(x;\theta)$ is the output of the NN. This gives a minimization problem to find $f(x;\theta)$ such that L is minimized. In the simplest case, we can use an L^2 error, giving

$$\arg\min_\theta |f(x;\theta) - y^*|_2^2. \tag{3.2}$$

We typically optimize, i.e. *train*, with a stochastic gradient descent (SGD) optimizer of choice, e.g. Adam [KB14]. We'll rely on auto-diff to compute the gradient of the *scalar* loss L w.r.t. the weights, $\partial L / \partial \theta$. It is crucial for the calculation of gradients that this function is scalar, and the loss function is often also called "error", "cost", or "objective" function.

For training we distinguish: the **training** data set drawn from some distribution, the **validation** set (from the same distribution, but different data), and **test** data sets with *some* different distribution than the training one. The latter distinction is important. For testing, we usually want *out of distribution* (OOD) data to check how well our trained model generalizes. Note that this gives a huge range of possibilities for the test data set: from tiny changes that will certainly work, up to completely different inputs that are essentially guaranteed to fail. There's no gold standard, but test data should be generated with care.

If the overview above wasn't obvious for you, we strongly recommend to read chapters 6 to 9 of the Deep Learning book, especially the sections about MLPs and "Conv-Nets", i.e. CNNs.

Note

Classification vs Regression

The classic ML distinction between *classification* and *regression* problems is not so important here: we only deal with *regression* problems in the following.

3.5.2 Partial differential equations as physical models

The following section will give a brief outlook for the model equations we'll be using later on in the DL examples. We typically target a continuous PDE operator denoted by $\mathcal{P}^*$, which maps inputs $\mathcal{U}$ to $\mathcal{V}$, where in the most general case $\mathcal{U}, \mathcal{V}$ are both infinite dimensional Banach spaces, i.e. $\mathcal{P}^* : \mathcal{U} \rightarrow \mathcal{V}$.

Learned solution operators vs traditional ones

Later on, the goal will be to learn $\mathcal{P}^*$ (or parts of it) with a neural network. A variety of different names are used in research: learned surrogates / hybrid simulators or emulators, Neural operators or solvers, autoregressive models (if timesteps are involved), to name a few.

In practice, the solution of interest lies in a spatial domain $\Omega \subset \mathbb{R}^d$ in $d \in 1, 2, 3$ dimensions. In addition, we often consider a time evolution for a finite time interval $t \in \mathbb{R}^+$. The corresponding fields are either d-dimensional vector fields, for instance $\mathbf{u} : \mathbb{R}^d \times \mathbb{R}^+ \rightarrow \mathbb{R}^d$, or scalar $\mathbf{p} : \mathbb{R}^d \times \mathbb{R}^+ \rightarrow \mathbb{R}$. The components of a vector are typically denoted by x, y, z subscripts, i.e., $\mathbf{v} = (v_x, v_y, v_z)^T$ for $d = 3$, while positions are denoted by $\mathbf{p} \in \Omega$.

To obtain unique solutions for $\mathcal{P}^*$ we need to specify suitable initial conditions, typically for all quantities of interest at $t = 0$, and boundary conditions for the boundary of Ω, denoted by Γ in the following. $\mathcal{P}^*$ denotes a continuous formulation, where we need to make mild assumptions about its continuity, we will typically assume that first and second derivatives exist.

Traditionally, we can use numerical methods to obtain approximations of a smooth function such as $\mathcal{P}^*$ via discretization. These invariably introduce discretization errors, which we'd like to keep as small as possible. These errors can be measured in terms of the deviation from the exact analytical solution, and for discrete simulations of PDEs, they are typically expressed as a function of the truncation error $O(\Delta x^k)$, where Δx denotes the spatial step size of the discretization. Likewise, we typically have a temporal discretization via a time step Δt.

Notation and abbreviations

If unsure, please check the summary of our mathematical notation and the abbreviations used in: *Notation and Abbreviations*.

With numerical simulations we solve a discretized PDE $\mathcal{P}$ by performing steps of size Δt. The solution can be expressed as a function of $\mathbf{u}$ and its derivatives: $\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathcal{P}(\mathbf{u}_x, \mathbf{u}_{xx}, ...\mathbf{u}_{xx...x})$, where $\mathbf{u}_x$ denotes the spatial derivatives $\partial \mathbf{u}(\mathbf{x}, t) / \partial \mathbf{x}$.

For all PDEs, we will assume non-dimensional parametrizations as outlined below, which could be re-scaled to real world quantities with suitable scaling factors. Next, we'll give an overview of the model equations, before getting started with actual simulations and implementation examples on the next page.

3.5.3 Some example PDEs

The following PDEs are good examples, and we'll use them later on in different settings to show how to incorporate them into DL approaches.

Burgers

We'll often consider Burgers' equation in 1D or 2D as a starting point. It represents a well-studied PDE, which (unlike Navier-Stokes) does not include any additional constraints such as conservation of mass. Hence, it leads to interesting shock formations. It contains an advection term (motion / transport) and a diffusion term (dissipation due to the second law of thermodynamics). In 2D, it is given by:

$$\begin{aligned} \frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x &= \nu \nabla \cdot \nabla u_x + g_x, \\ \frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y &= \nu \nabla \cdot \nabla u_y + g_y \ , \end{aligned} \tag{3.3}$$

where ν and $\mathbf{g}$ denote diffusion constant and external forces, respectively.

A simpler variant of Burgers' equation in 1D without forces, denoting the single 1D velocity component as $u = u_x$, is given by:

$$\frac{\partial u}{\partial t} + u \nabla u = \nu \nabla \cdot \nabla u \ . \tag{3.4}$$

Navier-Stokes

A good next step in terms of complexity is given by the Navier-Stokes equations, which are a well-established model for fluids. In addition to an equation for the conservation of momentum (similar to Burgers), they include an equation for the conservation of mass. This prevents the formation of shock waves, but introduces a new challenge for numerical methods in the form of a hard-constraint for divergence free motions.

In 2D, the Navier-Stokes equations without any external forces can be written as:

$$\begin{aligned} \frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x &= -\frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla u_x \\ \frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y &= -\frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla u_y \\ \text{subject to} \quad \nabla \cdot \mathbf{u} &= 0 \end{aligned} \tag{3.5}$$

where, like before, ν denotes a diffusion constant for viscosity.

An interesting variant is obtained by including the Boussinesq approximation for varying densities, e.g., for simple temperature changes of the fluid. With a marker field v that indicates regions of high temperature, it yields the following set of equations:

$$\begin{aligned} \frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x &= -\frac{1}{\rho} \nabla p \\ \frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y &= -\frac{1}{\rho} \nabla p + \xi v \\ \text{subject to} \quad \nabla \cdot \mathbf{u} &= 0, \\ \frac{\partial v}{\partial t} + \mathbf{u} \cdot \nabla v &= 0 \end{aligned} \tag{3.6}$$

where ξ denotes the strength of the buoyancy force.

And finally, the Navier-Stokes model in 3D give the following set of equations:

$$\begin{aligned} \frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x &= -\frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla u_x \\ \frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y &= -\frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla u_y \\ \frac{\partial u_z}{\partial t} + \mathbf{u} \cdot \nabla u_z &= -\frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla u_z \\ \text{subject to} \quad \nabla \cdot \mathbf{u} &= 0. \end{aligned} \tag{3.7}$$

3.5.4 Forward Simulations

Before we really start with learning methods, it's important to cover the most basic variant of using the above model equations: a regular "forward" simulation, that starts from a set of initial conditions, and evolves the state of the system over time with a discretized version of the model equation. We'll show how to run such forward simulations for Burgers' equation in 1D and for a 2D Navier-Stokes simulation.

3.6 Simple Forward Simulation of Burgers Equation with phiflow

This chapter will give an introduction for how to run *forward*, i.e., regular simulations starting with a given initial state and approximating a later state numerically, and introduce the ΦFlow framework (in the following "phiflow"). Phiflow provides a set of differentiable building blocks that directly interface with deep learning frameworks, and hence is a very good basis for the topics of this book. Before going for deeper and more complicated integrations, this notebook (and the next one), will show how regular simulations can be done with phiflow. Later on, we'll show that these simulations can be easily coupled with neural networks.

The main repository for phiflow is https://github.com/tum-pbs/PhiFlow, and additional API documentation and examples can be found at https://tum-pbs.github.io/PhiFlow/.

For this jupyter notebook (and all following ones), you can find a *"[run in colab]"* link at the end of the first paragraph (alternatively you can use the launch button at the top of the page). This will load the latest version from the PBDL github repo in a colab notebook that you can execute on the spot: [run in colab]

3.6.1 Model

As physical model we'll use Burgers equation. This equation is a very simple, yet non-linear and non-trivial, model equation that can lead to interesting shock formations. Hence, it's a very good starting point for experiments, and it's 1D version (from equation (3.4)) is given by:

$$\frac{\partial u}{\partial t} + u \nabla u = \nu \nabla \cdot \nabla u$$

3.6.2 Importing and loading phiflow

Let's get some preliminaries out of the way: first we'll import the phiflow library, more specifically the `numpy` operators for fluid flow simulations: `phi.flow` (differentiable versions for a DL framework *X* are loaded via `phi.X.flow` instead). This allows it to easily switch between different APIs, e.g., phiflow solvers can run in either PyTorch, Tensorflow or also JAX.

Note: Below, the first command with a "!" prefix will install the phiflow python package from GitHub via `pip` in your python environment once you uncomment it. We've assumed that phiflow isn't installed, but if you have already done so, just comment out the first line (the same will hold for all following notebooks).

```
!pip install --upgrade --quiet phiflow==3.2
from phi.flow import *
print("Using phiflow version: {}".format(phi.__version__))
```

```
Using phiflow version: 3.2.0
```

Next we can define and initialize the necessary constants (denoted by upper-case names): our simulation domain will have `N=128` cells as discretization points for the 1D velocity u in a periodic domain Ω for the interval $[-1, 1]$. We'll use 32 time `STEPS` for a time interval of 1, giving us `DT=1/32`. Additionally, we'll use a viscosity `NU` of $\nu = 0.01/\pi$.

We'll also define an initial state given by $-\sin(\pi x)$ in the numpy array `INITIAL_NUMPY`, which we'll use to initialize the velocity u in the simulation in the next cell. This initialization will produce a nice shock in the center of our domain.

Phiflow is object-oriented and centered around field data in the form of grids (internally represented by a tensor object). I.e. you assemble your simulation by constructing a number of grids, and updating them over the course of time steps.

Phiflow internally works with tensors that have named dimensions. This will be especially handy later on for 2D simulations with additional batch and channel dimensions, but for now we'll simply convert the 1D array into a phiflow tensor that has a single spatial dimension `'x'`.

```
N = 128
DX = 2./N
STEPS = 32
DT = 1./STEPS
NU = 0.01/(N*np.pi)

# initialization of velocities, cell centers of a CenteredGrid have DX/2 offsets for␣
↪linspace()
INITIAL_NUMPY = np.asarray( [-np.sin(np.pi * x) for x in np.linspace(-1+DX/2,1-DX/2,
↪N)] ) # 1D numpy array

INITIAL = math.tensor(INITIAL_NUMPY, spatial('x') ) # convert to phiflow tensor
```

Next, we initialize a 1D `velocity` grid from the `INITIAL` numpy array that was converted into a tensor. The extent of our domain Ω is specifiied via the `bounds` parameter $[-1, 1]$, and the grid uses periodic boundary conditions (`extrapolation.PERIODIC`). These two properties are the main difference between phiflow's tensor and grid objects: the latter has boundary conditions and a physical extent.

Just to illustrate, we'll also print some info about the velocity object: it's a `phi.math` tensor with a size of 128. Note that the actual grid content is contained in the `values` of the grid. Below we're printing five entries by using the `numpy()` function to convert the content of the phiflow tensor into a numpy array. For tensors with more dimensions, we'd need to specify the additional dimenions here, e.g., `'y,x,vector'` for a 2D velocity field. (For tensors with a single dimensions we could leave it out.)

```
velocity = CenteredGrid(INITIAL, extrapolation.PERIODIC, x=N, bounds=Box(x=(-1,1)))
vt = advect.semi_lagrangian(velocity, velocity, DT)
```

(continues on next page)

(continued from previous page)

```
#velocity = CenteredGrid(lambda x: -math.sin(np.pi * x), extrapolation.PERIODIC, x=N,␣
↪bounds=Box(x=(-1,1)))
#velocity = CenteredGrid(Noise(), extrapolation.PERIODIC, x=N, bounds=Box(x=(-1,1)))
↪# random init

print("Velocity tensor shape: "   + format( velocity.shape )) # == velocity.values.
↪shape
print("Velocity tensor type: "    + format( type(velocity.values) ))
print("Velocity tensor entries 10 to 14: " + format( velocity.values.numpy('x
↪')[10:15] ))
```

```
Velocity tensor shape: (x^s=128)
Velocity tensor type: <class 'phi.math._tensors.CollapsedTensor'>
Velocity tensor entries 10 to 14: [0.49289819 0.53499762 0.57580819 0.61523159 0.
↪65317284]
```

3.6.3 Running the simulation

Now we're ready to run the simulation itself. To compute the diffusion and advection components of our model equation we can simply call the existing `diffusion` and `semi_lagrangian` operators in phiflow: `diffuse.explicit(u,...)` computes an explicit diffusion step via central differences for the term $\nu\nabla \cdot \nabla u$ of our model. Next, `advect.semi_lagrangian(f,u)` is used for a stable first-order approximation of the transport of an arbitrary field `f` by a velocity `u`. In our model we have $\partial u/\partial t + u\nabla f$, hence we use the `semi_lagrangian` function to transport the velocity with itself in the implementation:

```
velocities = [velocity]
age = 0.
for i in range(STEPS):
    v1 = diffuse.explicit(velocities[-1], NU, DT)
    v2 = advect.semi_lagrangian(v1, v1, DT)
    age += DT
    velocities.append(v2)

print("New velocity content at t={}: {}".format( age, velocities[-1].values.numpy('x,
↪vector')[0:5] ))
```

```
New velocity content at t=1.0: [[0.0057228 ]
 [0.01716715]
 [0.02861034]
 [0.040052  ]
 [0.05149214]]
```

Here we're actually collecting all time steps in the list `velocities`. This is not necessary in general (and could consume lots of memory for long-running sims), but useful here to plot the evolution of the velocity states later on.

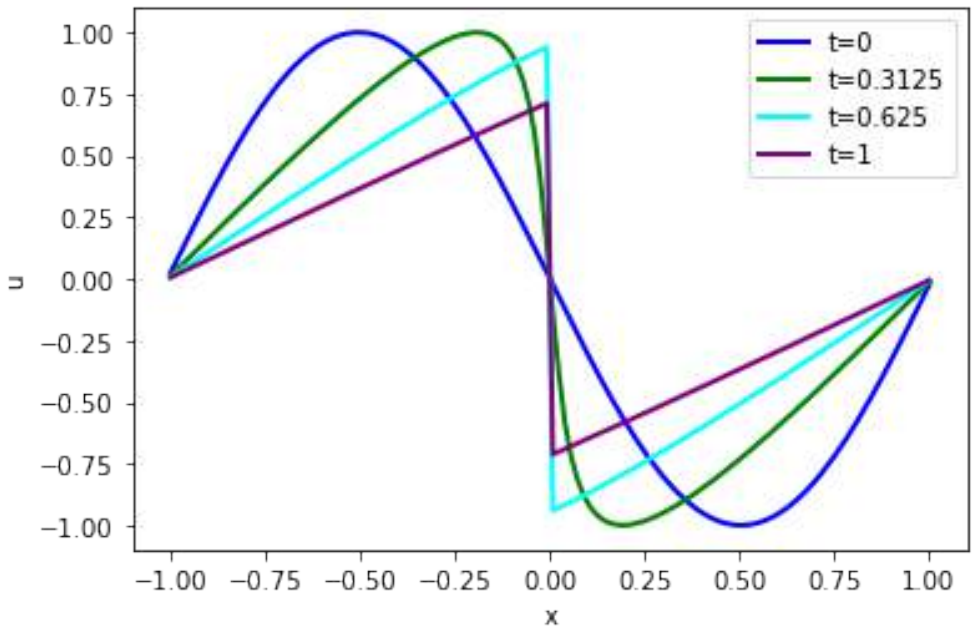
The print statements print a few of the velocity entries, and already show that something is happening in our simulation, but it's difficult to get an intuition for the behavior of the PDE just from these numbers. Hence, let's visualize the states over time to show what is happening.

3.6.4 Visualization

We can visualize this 1D case easily in a graph: the following code shows the initial state in blue, and then times $10/32, 20/32, 1$ in green, cyan and purple.

```
# get "velocity.values" from each phiflow state with a channel dimensions, i.e.
 ↪"vector"
vels = [v.values.numpy('x,vector') for v in velocities] # gives a list of 2D arrays

import pylab

fig = pylab.figure().gca()
fig.plot(np.linspace(-1,1,len(vels[ 0].flatten())), vels[ 0].flatten(), lw=2, color=
 ↪'blue',  label="t=0")
fig.plot(np.linspace(-1,1,len(vels[10].flatten())), vels[10].flatten(), lw=2, color=
 ↪'green', label="t=0.3125")
fig.plot(np.linspace(-1,1,len(vels[20].flatten())), vels[20].flatten(), lw=2, color=
 ↪'cyan',  label="t=0.625")
fig.plot(np.linspace(-1,1,len(vels[32].flatten())), vels[32].flatten(), lw=2, color=
 ↪'purple',label="t=1")
pylab.xlabel('x'); pylab.ylabel('u'); pylab.legend()
```

```
<matplotlib.legend.Legend at 0x7f9fd19fd940>
```

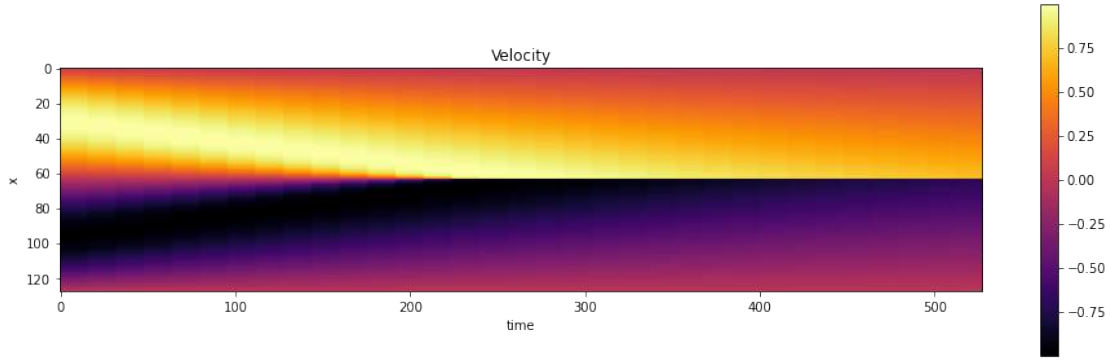


This nicely shows the shock developing in the center of our domain, which forms from the collision of the two initial velocity "bumps", the positive one on left (moving right) and the negative one right of the center (moving left).

As these lines can overlap quite a bit we'll also use a different visualization in the following chapters that shows the evolution over the course of all time steps in a 2D image. Our 1D domain will be shown along the Y-axis, and each point along X will represent one time step.

The code below converts our collection of velocity states into a 2D array, repeating individual time steps 8 times to make the image a bit wider. This is purely optional, of course, but makes it easier to see what's happening in our Burgers simulation.

```
def show_state(a, title):
    # we only have 33 time steps, blow up by a factor of 2^4 to make it easier to see
    # (could also be done with more evaluations of network)
    a=np.expand_dims(a, axis=2)
    for i in range(4):
        a = np.concatenate( [a,a] , axis=2)

    a = np.reshape( a, [a.shape[0],a.shape[1]*a.shape[2]] )
    #print("Resulting image size" +format(a.shape))

    fig, axes = pylab.subplots(1, 1, figsize=(16, 5))
    im = axes.imshow(a, origin='upper', cmap='inferno')
    pylab.colorbar(im) ; pylab.xlabel('time'); pylab.ylabel('x'); pylab.title(title)

vels_img = np.asarray( np.concatenate(vels, axis=-1), dtype=np.float32 )

# save for comparison with reconstructions later on
import os; os.makedirs("./temp",exist_ok=True)
np.savez_compressed("./temp/burgers-groundtruth-solution.npz", np.reshape(vels_img,[N,
 ↪STEPS+1])) # remove batch & channel dimension

show_state(vels_img, "Velocity")
```



This concludes a first simulation in phiflow. It's not overly complex, but because of that it's a good starting point for evaluating and comparing different physics-based deep learning approaches in the next chapter. But before that, we'll target a more complex simulation type in the next section.

3.6.5 Next steps

Some things to try based on this simulation setup:

- Feel free to experiment - the setup above is very simple, you can change the simulation parameters, or the initialization. E.g., you can use a noise field via `Noise()` to get more chaotic results (cf. the comment in the `velocity` cell above).
- A bit more complicated: extend the simulation to 2D (or higher). This will require changes throughout, but all operators above support higher dimensions. Before trying this, you probably will want to check out the next example, which covers a 2D Navier-Stokes case.

3.7 Navier-Stokes Forward Simulation

Now let's target a somewhat more complex example: a fluid simulation based on the Navier-Stokes equations. This is still very simple with ΦFlow (phiflow), as differentiable operators for all steps of the simulator are already available in phiflow. The Navier-Stokes equations (in their incompressible form) introduce an additional pressure field p, and a constraint for conservation of mass, as introduced in equation (3.6). We're also moving a marker field, denoted by d here, with the flow. It indicates regions of higher temperature, and exerts a force via a buouyancy factor ξ:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla \mathbf{u} + (0,1)^T \xi d \quad \text{s.t.} \quad \nabla \cdot \mathbf{u} = 0,$$
$$\frac{\partial d}{\partial t} + \mathbf{u} \cdot \nabla d = 0$$

Here, we're aiming for an incompressible flow (i.e., ρ = const), and use a simple buoyancy model (the Boussinesq approximation) via the term $(0,1)^T \xi d$. This approximates changes in density for incompressible solvers, without explicitly calculating ρ. We assume a gravity force that acts along the y direction via the vector $(0,1)^T$. We'll solve this PDE on a closed domain with Dirichlet boundary conditions $\mathbf{u} = 0$ for the velocity, and Neumann boundaries $\frac{\partial p}{\partial x} = 0$ for pressure, on a domain Ω with a physical size of 100×80 units. [run in colab]

3.7.1 Implementation

As in the previous section, the first command with a "!" prefix installs the phiflow python package from GitHub via `pip` in your python environment. (Skip or modify this command if necessary.)

```
!pip install --upgrade --quiet phiflow==3.1
#!pip install --upgrade --quiet git+https://github.com/tum-pbs/PhiFlow@develop

from phi.flow import *  # The Dash GUI is not supported on Google colab, ignore the␣
↪warning
import pylab
```

3.7.2 Setting up the simulation

The following code sets up a few constants, which are denoted by upper case names. We'll use 40×32 cells to discretize our domain, introduce a slight viscosity via ν, and define the time step to be $\Delta t = 1.5$.

We're creating a first `CenteredGrid` here, which is initialized by a `Sphere` geometry object. This will represent the inflow region `INFLOW` where hot smoke is generated.

```
DT = 1.5
NU = 0.01

INFLOW = CenteredGrid(Sphere(center=tensor([30,15], channel(vector='x,y')),␣
↪radius=10), extrapolation.BOUNDARY, x=32, y=40, bounds=Box(x=(0,80),y=(0,100))) * 0.
↪2
```

The inflow will be used to inject smoke into a second centered grid `smoke` that represents the marker field d from above. Note that we've defined a `Box` of size 100×80 above. This is the physical scale in terms of spatial units in our simulation, i.e., a velocity of magnitude 1 will move the smoke density by 1 unit per 1 time unit, which may be larger or smaller than a cell in the discretized grid, depending on the settings for `x,y`. You could parametrize your simulation grid to directly resemble real-world units, or keep appropriate conversion factors in mind.

The inflow sphere above is already using the "world" coordinates: it is located at $x = 30$ along the first axis, and $y = 15$ (within the 100×80 domain box).

Next, we create grids for the quantities we want to simulate. For this example, we require a velocity field and a smoke density field.

```
smoke = CenteredGrid(0, extrapolation.BOUNDARY, x=32, y=40, bounds=Box(x=(0,80),y=(0,
 ↪100)))  # sampled at cell centers
velocity = StaggeredGrid(0, extrapolation.ZERO, x=32, y=40, bounds=Box(x=(0,80),y=(0,
 ↪100)))  # sampled in staggered form at face centers
```

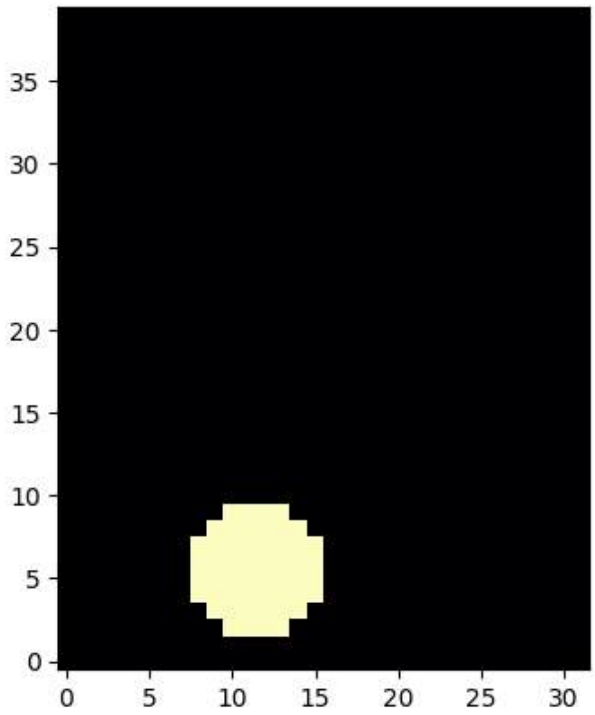
We sample the smoke field at the cell centers and the velocity in staggered form. The staggered grid internally contains 2 centered grids with different dimensions, and can be converted into centered grids (or simply numpy arrays) via the `unstack` function, as explained in the link above.

Next we define the update step of the simulation, which calls the necessary functions to advance the state of our fluid system by `dt`. The next cell computes one such step, and plots the marker density after one simulation frame.

```
def step(velocity, smoke, pressure, dt=1.0, buoyancy_factor=1.0):
    smoke = advect.semi_lagrangian(smoke, velocity, dt) + INFLOW
    buoyancy_force = (smoke * (0, buoyancy_factor)).at(velocity)  # resamples smoke␣
 ↪to velocity sample points
    velocity = advect.semi_lagrangian(velocity, velocity, dt) + dt * buoyancy_force
    velocity = diffuse.explicit(velocity, NU, dt)
    velocity, pressure = fluid.make_incompressible(velocity)
    return velocity, smoke, pressure

velocity, smoke, pressure = step(velocity, smoke, None, dt=DT)

print("Max. velocity and mean marker density: " + format( [ math.max(velocity.values)␣
 ↪, math.mean(smoke.values) ] ))

pylab.imshow(np.asarray(smoke.values.numpy('y,x')), origin='lower', cmap='magma')
```

```
Max. velocity and mean marker density: [0.1558497, 0.008125]
```

```
<matplotlib.image.AxesImage at 0x7ebc74337340>
```



A lot has happened in this `step()` call: we've advected the smoke field, added an upwards force via a Boussinesq model, advected the velocity field, and finally made it divergence free via a pressure solve.

The Boussinesq model uses a multiplication by a tuple `(0, buoyancy_factor)` to turn the smoke field into a staggered, 2 component force field, sampled at the locations of the velocity components via the `at()` function. This function makes sure the individual force components are correctly interpolated for the velocity components of the staggered velocity. Note that this also directly ensure the boundary conditions of the original grid are kept. It internally also does `StaggeredGrid(..., extrapolation.ZERO,...)` for the resulting force grid.

The pressure projection step in `make_incompressible` is typically the computationally most expensive step in the sequence above. It solves a Poisson equation for the boundary conditions of the domain, and updates the velocity field with the gradient of the computed pressure.

Just for testing, we've also printed the mean value of the velocities, and the max density after the update. As you can see in the resulting image, we have a first round region of smoke, with a slight upwards motion (which does not show here yet).

3.7.3 Datatypes and dimensions

The variables we created for the fields of the simulation here are instances of the class `Grid`. Like tensors, grids also have the `shape` attribute which lists all batch, spatial and channel dimensions. Shapes in phiflow store not only the sizes of the dimensions but also their names and types.

```
print(f"Smoke: {smoke.shape}")
print(f"Velocity: {velocity.shape}")
print(f"Inflow: {INFLOW.shape}, spatial only: {INFLOW.shape.spatial}")
```

```
Smoke: (x^s=32, y^s=40)
Velocity: (x^s=32, y^s=40, vector=x,y)
Inflow: (x^s=32, y^s=40), spatial only: (x^s=32, y^s=40)
```

Note that the phiflow output here indicates the type of a dimension, e.g., ^{S} for a spatial, and ^{V} for a vector dimension. Later on for learning, we'll also introduce batch dimensions.

The actual content of a shape object can be obtained via `.sizes`, or alternatively we can query the size of a specific dimension `dim` via `.get_size('dim')`. Here are two examples:

```
print(f"Shape content: {velocity.shape.sizes}")
print(f"Vector dimension: {velocity.shape.get_size('vector')}")
```

```
Shape content: (32, 40, 2)
Vector dimension: 2
```

The grid values can be accessed using the `values` property. This is an important difference to a phiflow tensor object, which does not have `values`, as illustrated in the code example below.

```
print("Statistics of the different simulation grids:")
print(smoke.values)
print(velocity.values)

# in contrast to a simple tensor:
test_tensor = math.tensor(numpy.zeros([3, 5, 2]), spatial('x,y'), channel(vector="x,y
↪"))
print("Reordered test tensor shape: " + format(test_tensor.numpy('vector,y,x').shape)␣
↪) # reorder to vector,y,x
#print(test_tensor.values.numpy('y,x')) # error! tensors don't return their content␣
↪via ".values"
```

```
Statistics of the different simulation grids:
(x^s=32, y^s=40) 0.008 ± 0.039 (0e+00...2e-01)
(~vector=x,y, x^s=~(x=31, y=32) int64, y^s=~(x=40, y=39) int64) -7.23e-09 ± 2.8e-02␣
↪(-1e-01...2e-01)
Reordered test tensor shape: (2, 5, 3)
```

Grids have many more properties which are documented here. Also note that the staggered grid has a non-uniform shape because the number of faces is not equal to the number of cells (in this example the x component has 31×40 cells, while y has 32×39). The `INFLOW` grid naturally has the same dimensions as the `smoke` grid.

3.7.4 Time evolution

With this setup, we can easily advance the simulation forward in time a bit more by repeatedly calling the `step` function.

```
for time_step in range(10):
    velocity, smoke, pressure = step(velocity, smoke, pressure, dt=DT)
    print('Computed frame {}, max velocity {}'.format(time_step , np.asarray(math.
↪max(velocity.values)) ))
```

```
Computed frame 0, max velocity 0.4630011022090912
Computed frame 1, max velocity 0.8966455459594727
Computed frame 2, max velocity 1.4098880290985107
Computed frame 3, max velocity 2.0411267280578613
Computed frame 4, max velocity 2.9279565811157227
```

(continues on next page)

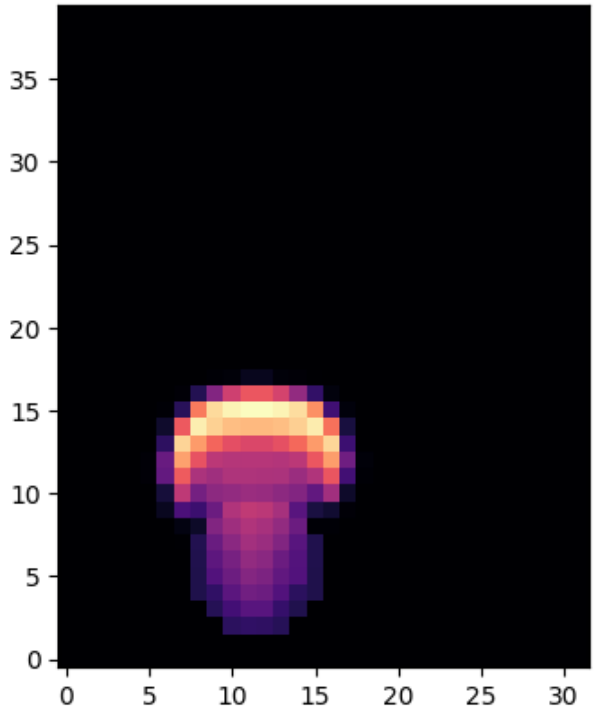
(continued from previous page)

```
Computed frame 5, max velocity 3.839479923248291
Computed frame 6, max velocity 4.526943206787109
Computed frame 7, max velocity 4.867981910705566
Computed frame 8, max velocity 5.131079196929932
Computed frame 9, max velocity 5.483874320983887
```

Now the hot plume is starting to rise:

```
pylab.imshow(smoke.values.numpy('y,x'), origin='lower', cmap='magma')
```

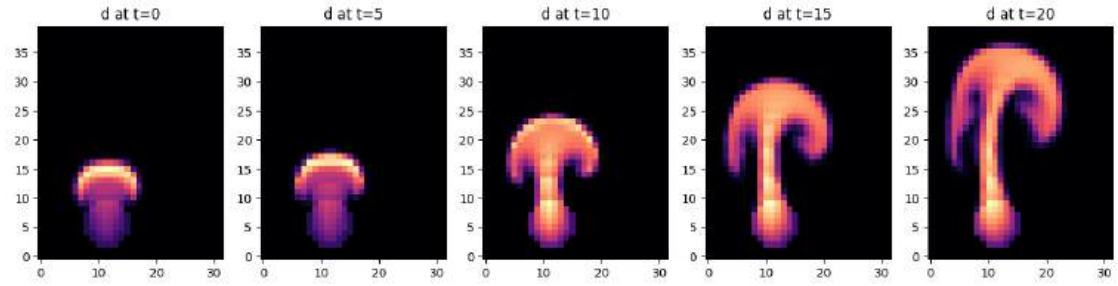
```
<matplotlib.image.AxesImage at 0x7ebc7431dd50>
```



Let's compute and show a few more steps of the simulation. Because of the inflow being located off-center to the left (with x position 30), the plume will curve towards the right when it hits the top wall of the domain.

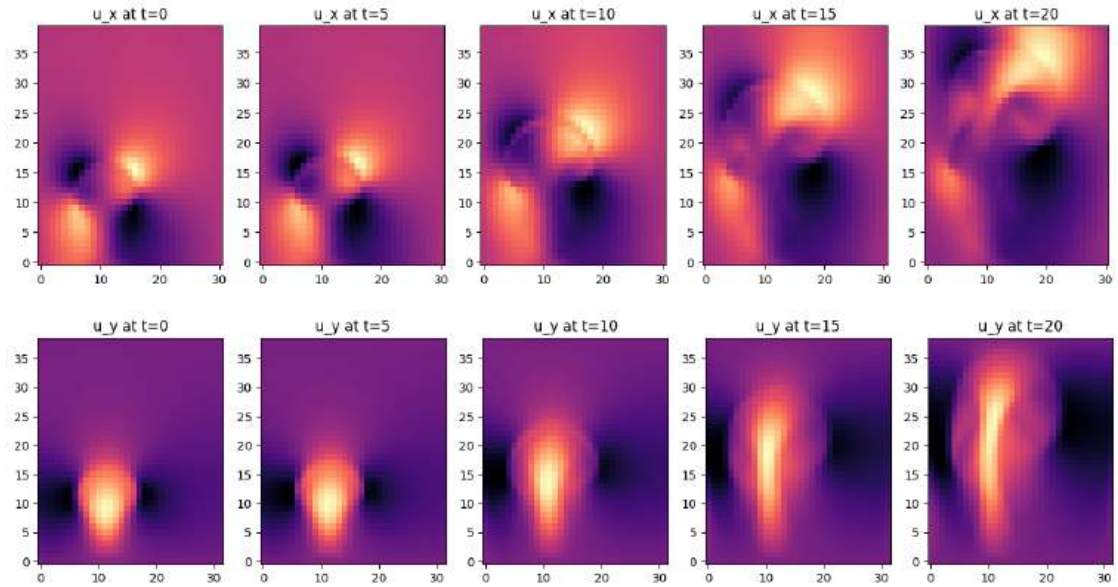
```
steps = [[ smoke.values, velocity.vector['x'], velocity.vector['y'] ]]
for time_step in range(20):
  if time_step<3 or time_step%10==0:
    print('Computing time step %d' % time_step)
  velocity, smoke, pressure = step(velocity, smoke, pressure, dt=DT)
  if time_step%5==0:
    steps.append( [smoke.values, velocity.vector['x'], velocity.vector['y']] )

fig, axes = pylab.subplots(1, len(steps), figsize=(16, 5))
for i in range(len(steps)):
    axes[i].imshow(steps[i][0].numpy('y,x'), origin='lower', cmap='magma')
    axes[i].set_title(f"d at t={i*5}")
```

```
Computing time step 0
Computing time step 1
Computing time step 2
Computing time step 10
```



We can also take a look at the velocities. The `steps` list above already stores `vector[0]` and `vector[1]` components of the velocities as numpy arrays, which we can show next.

```
fig, axes = pylab.subplots(1, len(steps), figsize=(16, 5))
for i in range(len(steps)):
    axes[i].imshow(steps[i][1].numpy('y,x'), origin='lower', cmap='magma')
    axes[i].set_title(f"u_x at t={i*5}")

fig, axes = pylab.subplots(1, len(steps), figsize=(16, 5))
for i in range(len(steps)):
    axes[i].imshow(steps[i][2].numpy('y,x'), origin='lower', cmap='magma')
    axes[i].set_title(f"u_y at t={i*5}")
```



It looks simple here, but this simulation setup is a powerful tool. The simulation could easily be extended to more complex cases or 3D, and it is already fully compatible with backpropagation pipelines of deep learning frameworks.

In the next chapters we'll show how to use these simulations for training NNs, and how to steer and modify them via trained NNs. This will illustrate how much we can improve the training process by having a solver in the loop, especially when the solver is *differentiable*. Before moving to these more complex training processes, we will cover a simpler supervised

approach in the next chapter. This is very fundamental: even when aiming for advanced physics-based learning setups, a working supervised training is always the first step.

3.7.5 Next steps

You could create a variety of nice fluid simulations based on this setup. E.g., try changing the spatial resolution, the buoyancy factors, and the overall length of the simulation run.

3.8 Optimization and Convergence

This chapter will give an overview of the derivations for different optimization algorithms. In contrast to other texts, we'll start with *the most classic* optimization algorithm, Newton's method, derive several widely used variants from it, before coming back full circle to deep learning (DL) optimizers. The main goal is the put DL into the context of these classical methods. While we'll focus on DL, we will also revisit the classical algorithms for improved learning algorithms later on in this book. Physics simulations exaggerate the difficulties caused by neural networks, which is why the topics below have a particular relevance for physics-based learning tasks.

Note

Deep-dive Chapter: This chapter is a deep dive for those interested in the theory of different optimizers. It will skip evaluations as well as source code, and instead focus on theory. The chapter is highly recommend as a basis for the chapters of *Scale-Invariance and Inversion*. However, it is not "mandatory" for getting started with topics like training via *differentiable physics*. If you'd rather quickly get started with practical aspects, feel free to skip ahead to *Supervised Training*.

3.8.1 Notation

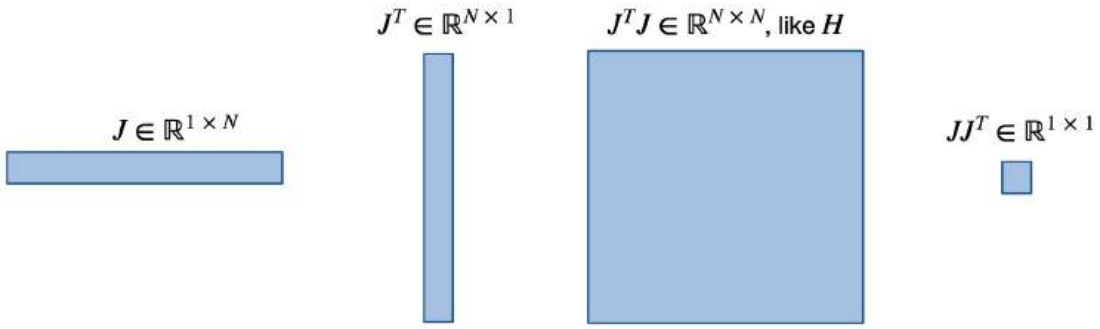
This chapter uses a custom notation that was carefully chosen to give a clear and brief representation of all methods under consideration. We have a scalar loss function $L(x) : \mathbb{R}^N \to \mathbb{R}$, the optimum (the minimum of L) is at location x^*, and Δ denotes a step in x. Different intermediate update steps in x are denoted by a subscript, e.g., as x_n or x_k.

In the following, we often need inversions, i.e. a division by a certain quantity. For matrices A and B, we define $\frac{A}{B} \equiv B^{-1}A$. When a and b are vectors, the result is a matrix obtained with one of the two formulations below. We'll specify which one to use:

$$\frac{a}{b} \equiv \frac{aa^T}{a^Tb} \text{ or } \frac{a}{b} \equiv \frac{ab^T}{b^Tb} \tag{3.8}$$

Applying $\partial/\partial x$ once to L yields the Jacobian $J(x)$. As L is scalar, J is a row vector, and the gradient (column vector) ∇L is given by J^T. Applying $\partial/\partial x$ again gives the Hessian matrix $H(x)$, and another application of $\partial/\partial x$ gives the third derivative tensor denoted by $K(x)$. We luckily never need to compute K as a full tensor, but it is needed for some of the derivations below. To shorten the notation below, we'll typically drop the (x) when a function or derivative is evaluated at location x, e.g., J will denote $J(x)$.

The following image gives an overview of the resulting matrix shapes for some of the commonly used quantities. We don't really need it afterwards, but for this figure N denotes the dimension of x, i.e. $x \in \mathbb{R}^N$.



3.8.2 Preliminaries

We'll need a few tools for the derivations below, which are summarized here for reference.

Not surprisingly, we'll need some Taylor-series expansions. With the notation above it reads:

$$L(x + \Delta) = L + J\Delta + \frac{1}{2} H \Delta^2 + \cdots$$

Then we also need the *Lagrange form*, which yields an exact solution for a ξ from the interval $[x, x + \Delta]$:

$$L(x + \Delta) = L + J\Delta + \frac{1}{2} H(\xi) \Delta^2$$

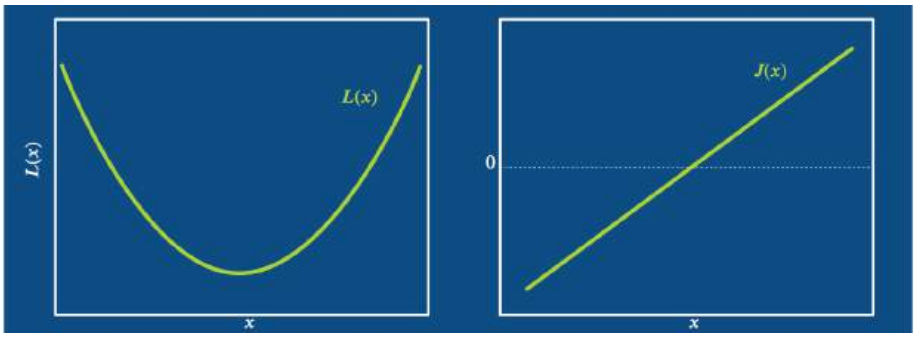
In several instances we'll make use of the fundamental theorem of calculus, repeated here for completeness:

$$f(x + \Delta) = f(x) + \int_0^1 \mathrm{d}s \; f'(x + s\Delta)\Delta \; .$$

In addition, we'll make use of Lipschitz-continuity with constant $\mathcal{L}$: $|f(x + \Delta) + f(x)| \leq \mathcal{L}\Delta$, and the well-known Cauchy-Schwartz inequality: $u^T v \leq |u| \cdot |v|$.

3.8.3 Newton's method

Now we can start with arguably the most classic algorithm for optimization: *Newton's method*. It is derived by approximating the function we're interested in as a parabola. This can be motivated by the fact that pretty much every minimum looks like a parabola close up.

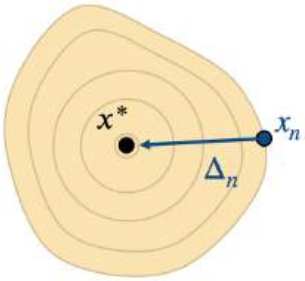


So we can represent L near an optimum x^* by a parabola of the form $L(x) = \frac{1}{2} H (x - x^*)^2 + c$, where c denotes a constant offset. At location x we observe H and

$J^T = H \cdot (x_k - x^*)$. Re-arranging this directly yields an equation to compute the minimum: $x^* = x_k - \frac{J^T}{H}$. Newton's method by default computes x^* in a single step, and hence the update in x of Newton's method is given by:

$$\Delta = -\frac{J^T}{H} \tag{3.9}$$

Let's look at the order of convergence of Newton's method. For an optimum x^* of L, let $\Delta_n^* = x^* - x_n$ denote the step from a current x_n to the optimum, as illustrated below.



Assuming differentiability of J, we can perform the Lagrange expansion of J^T at x^*:

$$\begin{aligned} 0 = J^T(x^*) &= J^T(x_n) + H(x_n)\Delta_n^* + \frac{1}{2}K(\xi_n){\Delta_n^*}^2 \\ \frac{J^T}{H} &= -\frac{K}{2H}{\Delta_n^*}^2 - \Delta_n^* \end{aligned}$$

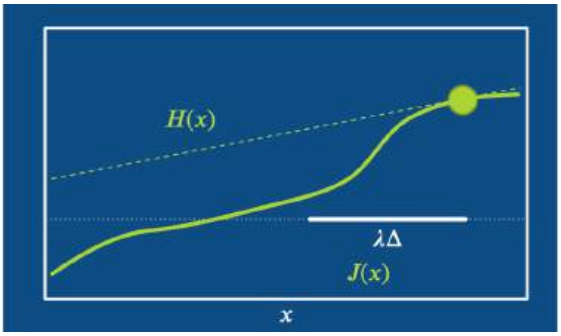
In the second line, we've already divided by H, and dropped (x_n) and (ξ_n) to shorten the notation. When we insert this into Δ_n^* we get:

$$\begin{aligned} \Delta_{n+1}^* &= x^* - x_{n+1} \\ &= x^* - (x_n - \frac{J^T}{H}) \\ &= \Delta_n^* - \frac{K}{2H}{\Delta_n^*}^2 - \Delta_n^* \\ &= -\frac{K}{2H}{\Delta_n^*}^2 \end{aligned}$$

Thus, the distance to the optimum changes by ${\Delta_n^*}^2$, which means once we're close enough we have quadratic convergence. This is great, of course, but it still depends on the pre-factor $\frac{K}{2H}$, and will diverge if its > 1. Note that this is an exact expression, there's no truncation thanks to the Lagrange expansion. And so far we have quadratic convergence, but the convergence to the optimum is not guaranteed. For this we have to allow for a variable step size.

3.8.4 Adaptive step size

Thus, as a next step for Newton's method we introduce a variable step size λ which gives the iteration $x_{n+1} = x_n + \lambda\Delta = x_n - \lambda\frac{J^T}{H}$. As illustrated in the picture below, this is especially helpful if L is not exactly a parabola, and a small H might overshoot in undesirable ways. The far left in this example:



To make statements about convergence, we need some fundamental assumptions: convexity and smoothness of our loss function. Then we'll focus on showing that the loss decreases, and that we move along a sequence of smaller sets $\forall x\ L(x) < L(x_n)$ with lower loss values.

First, we apply the fundamental theorem to L

$$L(x+\lambda\Delta) = L + \int_0^1 \text{ds } J(x+s\lambda\Delta)\lambda\Delta \ ,$$

and likewise express J around this location with it:

$$J(x+s\lambda\Delta) = J + s\lambda\Delta^T \int_0^1 \text{dt } H(x+st\lambda\Delta)$$

Inserting this J^T into L yields:

$$\begin{aligned}
L(x+\lambda\Delta) - L(x) &= \int_0^1 \text{d}s \Big[J + s\lambda\Delta^T \int_0^1 \text{d}t\ H(x+st\lambda\Delta) \Big]^T \lambda\Delta \\
&= \int_0^1 \text{d}s \Big[-H\Delta + \int_0^1 \text{d}t H(x+st\lambda\Delta)\ s\lambda\Delta \Big]^T \lambda\Delta \\
&= \int_0^1 \text{d}s \int_0^1 \text{d}t \big[-H\Delta + H(x+st\lambda\Delta)\ s\lambda\Delta \big]^T \lambda\Delta \\
&= \int_0^1 \text{d}s \int_0^1 \text{d}t \big[-H\Delta(1+\lambda s) + [H(x+st\lambda\Delta) - H]\ s\lambda\Delta \big]^T \lambda\Delta \\
&= -(H\Delta)^T (1+\frac{\lambda}{2})\lambda\Delta + \lambda^2 \int_0^1 \text{d}s\ s \int_0^1 \text{d}t \big[[H(x+st\lambda\Delta) - H]\ \Delta \big]^T \Delta
\end{aligned} \tag{3.10}$$

Here we've first used $\Delta = -J^T/H$, and moved it into the integral in the third line, alongside the $s\lambda\Delta$ term. Line four factors out $H\Delta$. This allows us to evaluate the integral for the first term $-H\Delta(1+\lambda s)$ in the last line of equation (3.10) above.

In the next sequence of steps, we will first employ $(H\Delta)^T\Delta = \|H^{\frac{1}{2}}\Delta\|^2$. This term will be shortened to $\epsilon \equiv \|H^{\frac{1}{2}}\Delta\|^2$ in the second line below. Due to the properties of H, this ϵ "just" represents a small positive factor that will stick around till the end. Afterwards, in line 3 and 4 below, we can start finding an upper bound for the change of the loss. We'll first use a Cauchy-Schwartz inequality, and then make use of a special Lipschitz condition for affine conjugate matrices. For H, it takes the form $\|H(x)^{-\frac{1}{2}}(H(y)-H(x))\| \leq \mathcal{L}\|H(x)^{\frac{1}{2}}(y-x)\|^2$. This requires H to by symmetric, positive-definite,

which isn't too unreasonable in practice. Continuing from above, we get:

$$
\begin{aligned}
\cdots &= -(\lambda + \frac{\lambda^2}{2}) \| H^{\frac{1}{2}} \Delta \|^2 + \lambda^2 \int_0^1 \mathrm{d}s \; s \int_0^1 \mathrm{d}t \big[[H(x + st\lambda\Delta) - H] \; \Delta \big]^T H^{-\frac{1}{2}} H^{\frac{1}{2}} \Delta \\
&= -\lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \lambda^2 \int_0^1 \mathrm{d}s \; s \int_0^1 \mathrm{d}t \big[H^{-\frac{1}{2}} [H(x + st\lambda\Delta) - H] \; \Delta \big]^T H^{\frac{1}{2}} \Delta \\
&\leq -\lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \lambda^2 \int_0^1 \mathrm{d}s \; s \int_0^1 \mathrm{d}t \big\| H^{-\frac{1}{2}} [H(x + st\lambda\Delta) - H] \; \Delta \big\| \; \| H^{\frac{1}{2}} \Delta \| \\
&\leq -\lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \lambda^3 \int_0^1 \mathrm{d}s \; s^2 \int_0^1 \mathrm{d}t \; t \; \mathcal{L} \| H^{\frac{1}{2}} \Delta \|^2 \; \| H^{\frac{1}{2}} \Delta \| \\
&= -\lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \mathcal{L}\lambda^3 \int_0^1 \mathrm{d}s \; s^2 \int_0^1 \mathrm{d}t \; t \; \| H^{\frac{1}{2}} \Delta \|^3 \\
&= -\lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \mathcal{L}\lambda^3 \int_0^1 \mathrm{d}s \; s^2 \int_0^1 \mathrm{d}t \; t \; \epsilon^{\frac{3}{2}} \\
&= -\lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \frac{\lambda^3 \mathcal{L} \; \epsilon^{\frac{3}{2}}}{6}
\end{aligned}
\tag{3.11}
$$

Due to $H^T = H$, we've moved $H^{-\frac{1}{2}}$ inside the integral in line 2. In line 4, we've pulled s, t and λ out of the integrals as much as possible, in addition to applying the special Lipschitz condition. The last three lines just simplify the terms, express the occurrences of H in terms of ϵ, and evaluate the integrals. This leaves us with a cubic form in terms of λ, the step size. Most importantly, the first, linear term is negative, and hence will dominate for small λ. With this we've shown that the step will be negative for sufficiently small λ:

$$
L(x + \lambda\Delta) = L(x) - \lambda\epsilon + \frac{\lambda^2 \epsilon}{2} + \frac{\lambda^3 \mathcal{L} \; \epsilon^{\frac{3}{2}}}{6} \tag{3.12}
$$

However, this inherently requires us to freely choose λ, hence this proof is not applicable for the fixed step size above. A nice property of it is that we've "only" required Lipschitz continuity for H, not for J or even L.

To conclude, we've shown that Newton's method with an adaptive step size provably converges, which is great. However, it requires the Hessian H as a central ingredient. Unfortunately, H very difficult to obtain in practice. This is a real show stopper, and motivates the following methods. They keep the basic step of Newton's method, but approximate H.

3.8.5 Approximating the Hessian

Next, we will revisit three popular algorithms derived from Newton's method.

3.8.6 Broyden's method

The first approach to approximate H is to make a very rough guess, namely to start with the identity matrix $H_0 = \mathbf{1}$, and then iterate to update H_n via a finite difference approximation. For Broyden's method, we use the vector division $\frac{a}{b} = \frac{ab^T}{b^T b}$.

For simplifying the finite difference, we'll additionally assume that we already have reached $J(x_n) = 0$ at the current position x_n. This is of course not necessarily true, but yields the following, nicely reduced expression to modify H over the course of the optimization:

$$
H_{n+1} = H_n + \frac{J(x_{n+1})^T}{\Delta}
$$

As before, we use a step of $\Delta = -\frac{J^T}{H}$ for x, and the denominator comes from the finite difference $\frac{J(x_{n+1})^T - J(x_n)^T}{\Delta}$ with the assumption that the current Jacobian is zero. Keep in mind, that Δ is a vector here (see the vector division above), so the finite difference gives a matrix of size $N \times N$ that can be added to H_n.

Broyden's method has the advantage that we never have to compute a full Hessian, and the update of H can be evaluated efficiently. However, the two assumptions above make it a very rough approximation, and hence the normalization of the update step via the inverse Hessian in $\frac{J^T}{H}$ is correspondingly unreliable.

3.8.7 BFGS

This leads to the BFGS algorithm (named after *Broyden-Fletcher-Goldfarb-Shanno*), which introduces some important improvements: it does not assume J to be zero immediately, and compensates for redundant parts of the updates. This is necessary, as the finite difference $(J(x_{n+1}) - J(x_n))/\Delta_n$ gives a full approximation of H. We could try to perform some kind of averaging procedure, but this would strongly deteriorate the existing content in H_n. Hence, we subtract only the existing entries in H_n along the current step Δ. This makes sense, as the finite difference approximation yields exactly the estimate along Δ. In combination, using the vector division $\frac{a}{b} \equiv \frac{aa^T}{a^Tb}$, these changes give an update step for H of:

$$H_{n+1} = H_n + \frac{J(x_{n+1})^T - J(x_n)^T}{\Delta_n} - \frac{H_n \Delta_n}{\Delta_n}$$

In practice, BFGS also makes use of a line search to determine the step size λ. Due to the large size of H, commonly employed variants of BFGS also make use of reduced representations of H to save memory. Nonetheless, the core step of updating the Hessian matrix via a finite difference approximation along the search direction lies at the core of BFGS, and allows it to at least partially compensate for scaling effects of the loss landscape. Currently, BFGS-variants are the most widely used algorithm for solving classical non-linear optimization problems.

3.8.8 Gauss-Newton

Another attractive variant of Newton's method can be derived by restricting L to be a classical L^2 loss. This gives the *Gauss-Newton* (GN) algorithm. Thus, we still use $\Delta = -\frac{J^T}{H}$, but rely on a squared loss of the form $L = |f|^2$ for an arbitrary $f(x)$. The derivatives of f are denoted by J_f, H_f, in contrast to the generic J, H for L, as before. Due to the chain rule, we have $J = 2\ f^T J_f$.

The second derivative yields the following expression. For GN, we simplify it by omitting the second-order terms in the second line below:

$$\begin{aligned} H &= 2J_f^T J_f + 2f^T H_f \\ &\approx\ 2J_f^T J_f \\ &= 2\frac{J^T J}{4|f|^2} \\ &= \frac{J^T J}{2L} \end{aligned} \tag{3.13}$$

Here the remaining $J_f^T J_f$ term of the first order approximation can be simplified thanks to our focus on an L^2 loss: $J_f = J/(2f^T)$ and $|f|^2 = L$.

The last line of equation (3.13) means we are basically approximating the Hessian with J squared. This is reasonable in many scenarios, and inserting it into our update step above gives the Gauss-Newton update $\Delta_{\text{GN}} \approx -\frac{J^T}{J^T J}$.

Looking at this update, it essentially employs a step of the form
$\Delta_{\text{GN}} \approx -\frac{1}{J}$, i.e., the update is based on an approximate inverse of the Jacobian of L. This inverse gives an approximately equal step size in all parameters, and as such provides an interesting building block that we will revisit in later chapters.

In the form above, it still means we have to invert a large matrix, which is costly, and the matrix itself may not even be invertible.

3.8.9 Back to Deep Learning

We've shown above that Newton's method lies at the heart of many popular algorithms for non-linear optimization. Gauss-Newton now finally provides us with a stepping stone back towards Deep Learning algorithms, specifically, to the Adam optimizer.

3.8.10 Adam

As usual, we start with a Newton step $\Delta = -\lambda \frac{J^T}{H}$, but even the simplest approximation of $H \approx J^T J$ from Gauss-Newton requires inverting a potentially huge matrix. This is not feasible for the weights of neural networks, and hence a valid question is, how can we further simplify this step? For Adam, the answer is: with a diagonal approximation. Specifically, Adam uses:

$$H \approx \sqrt{\text{diag}(J^T J)} \tag{3.14}$$

This is a very rough approximation of the true Hessian. We're simply using the squared, first derivatives here, and in general, of course, $\left(\frac{\partial f}{\partial x}\right)^2 \neq \frac{\partial^2 f}{\partial x^2}$. This only holds for the first-order approximation from Gauss-Newton, i.e., the first term of equation (3.13). Now Adam goes a step further, and only keeps the diagonal of $J^T J$. This quantity is readily available in deep learning in the form of the gradient of the weights, and makes the inversion of H trivial. As a result, it at least provides some estimate of the curvature of the individual weights, but neglects their correlations.

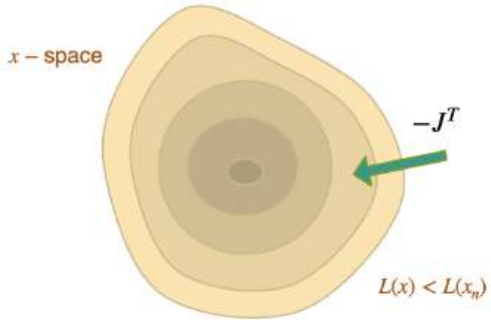
Interestingly, Adam does not perform a full inversion via $\text{diag}(J^T J)$, but uses the component-wise square root. This effectively yields $\sqrt{\text{diag}(J^T J)} \approx \sqrt{\text{diag}(J^2)} \approx \text{diag}(J)$. Thus, Adam moves along $\Delta \approx \text{sign}(-J)$, approximately performing a step of fixed size along all dimensions.

In practice, Adam introduces a few more tricks by computing the gradient J^T as well as the squared gradient with *momentum*, an averaging of both quantities over the course of the iterations of the optimization. This makes the estimates more robust, which is crucial: a normalization with an erroneously small entry of the gradient could otherwise lead to an explosion. Adam additionally adds a small constant when dividing, and the square-root likewise helps to mitigate overshoot.

To summarize: Adam makes use of a first-order update with a diagonal Gauss-Newton approximation of the Hessian for normalization. It additionally employs momentum for stabilization.

3.8.11 Gradient Descent

To arrive at gradient descent (GD) optimization, we now take the final step to assume $H = 1$ in $-\lambda \frac{J^T}{H}$. This leaves us with an update consisting of a scaled gradient $\Delta = -\lambda J^T$.



Interestingly, without any form of inversion, J^T by itself has the right shape to be added to x, but it "lives" in the wrong space $1/x$, rather than x itself. This is illustrated by the fact that J^{-1}, as employed by all "classic" schemes above in some form, is inherently different from J^T. This problem of a lacking inversion step is especially pronounced for problems involving physics, and hence one of the crucial take-away messages of this chapter. We will revisit it in much more detail later on in *Scale-Invariance and Inversion*. For now, it's recommended to keep the relationships between the different algorithms in mind, even when "just" using Adam or GD.

As a final step, let's look at the convergence of GD. We again assume convexity and differentiability of L. Expanding a step of Δ with a Taylor series, and bounding the second order term with a Lipschitz condition $J(x+\Delta) - J(x) \leq \mathcal{L}\Delta$, we get:

$$\begin{aligned} L(x+\Delta) - L(x) &\leq J(x)\Delta + \frac{\mathcal{L}}{2}|\Delta|^2 \\ &= J(-\lambda J^T) + \frac{\mathcal{L}}{2}|-\lambda J^T|^2 \\ &= -\lambda J J^T + \lambda^2 \frac{\mathcal{L}}{2}|J|^2 \\ &= -\lambda |J|^2 + \frac{\lambda^2 \mathcal{L}}{2}|J|^2 \end{aligned}$$

Like above for Newton's method in equation (3.11) we have a negative linear term that dominates the loss for small enough λ. In combination, we have the following upper bound due to the Lipschitz condition in the first line $L(x+\Delta) \leq L(x) - \lambda|J|^2 + \frac{\lambda^2\mathcal{L}}{2}|J|^2$. By choosing $\lambda \leq \frac{1}{\mathcal{L}}$, we can simplify these terms further, and can an upper bound that depends on J squared: $L(x+\Delta) \leq L(x) - \frac{\lambda}{2}|J|^2$ and thus ensures convergence.

This result unfortunately does not help us much in practice, as for all common usage of GD in deep learning $\mathcal{L}$ is not known. It is still good to know that a Lipschitz constant for the gradient would theoretically provide us with convergence guarantees for GD.

With this we conclude our tour of classical optimizers and their relation to deep learning methods. It's worth noting that we've focused on non-stochastic algorithms here for clarity, as the proofs would become more involved for stochastic algorithms.

With all this background knowledge, it a good time to start looking at some practical examples that start as-simple-as-possible, with fully supervised approaches for training.

Part II

Neural Surrogates and Operators

CHAPTER

FOUR

SUPERVISED TRAINING

We will first target a "purely" data-driven approach, in line with classic machine learning. We'll refer to this as a *supervised* approach in the following, to indicate that the network is fully supervised by data, and to distinguish it from using physics-based losses. One of the central advantages of the supervised approach is that we obtain a *surrogate model* (also called "emulator", or "Neural operator"), i.e., a new function that mimics the behavior of the original $\mathcal{P}$.

The purely data-driven, *supervised training* is the central starting point for all projects in the context of deep learning. While it can yield suboptimal results compared to approaches that more tightly couple with physics, it can be the only choice in certain application scenarios where no good model equations exist. In this chapter, we'll also go over the basics of different neural network *architectures*. Next to training methodology, this is an imporant choice.

4.1 Problem setting

For supervised training, we're faced with an unknown function $f^*(x) = y^*$, collect lots of pairs of data $[x_0, y_0^*], ...[x_n, y_n^*]$ (the training data set) and directly train an NN to represent an approximation of f^* denoted as f.

The f we can obtain in this way is typically not exact, but instead we obtain it via a minimization problem: by adjusting the weights θ of our NN representation of f such that we minimize the error over all data points in the training set

$$\arg\min_\theta \sum_i \left(f(x_i;\theta) - y_i^* \right)^2 . \tag{4.1}$$

This will give us θ such that $f(x;\theta) = y \approx y^*$ as accurately as possible given our choice of f and the hyperparameters chosen for training. Note that above we've assumed the simplest case of an L^2 loss. A more general version would use an error metric $e(x, y)$ in the loss L to be minimized via $\arg\min_\theta \sum_i e(f(x_i;\theta), y_i^*))$. The choice of a suitable metric is a topic we will get back to later on. The minimization above constitutes the actual "learning" process, and is non-trivial because f is usually a non-linear function.

The training data typically needs to be of substantial size, and hence it is attractive to use numerical simulations solving a physical model $\mathcal{P}$ to produce a large number of reliable input-output pairs for training. This means that the training process uses a set of model equations, and approximates them numerically, in order to fit the NN representation f. This has quite a few advantages, e.g., we don't have the measurement noise of real-world devices and we don't need manual labour to annotate a large number of samples to get training data.

On the other hand, this approach inherits the common challenges of replacing experiments with simulations: first, we need to ensure the chosen model has enough power to predict the behavior of the simulated phenomena that we're interested in. In addition, the numerical approximations have *numerical errors* which need to be kept small enough for a chosen application (otherwise even the best NN has no chance to be provide a useful answer later on). As these topics are studied in depth for classical simulations, and the existing knowledge can likewise be leveraged to set up DL training tasks.

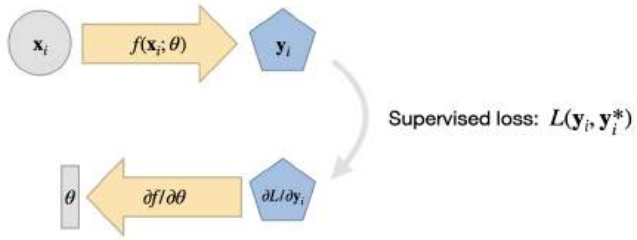


Fig. 4.1: A visual overview of supervised training. It's simple, and a good starting point in comparison to the more complex variants we'll encounter later on.

4.2 Looking ahead

The numerical approximations of PDE models for real world phenomena are often very expensive to compute. A trained NN on the other hand incurs a constant cost per evaluation, and is typically trivial to evaluate on specialized hardware such as GPUs or NN compute units.

Despite this, it's important to be careful: NNs can quickly generate huge numbers of in between results. Consider a CNN layer with 128 features. If we apply it to an input of 128^2, i.e. ca. 16k cells, we get 128^3 intermediate values. That's more than 2 million. All these values at least need to be momentarily stored in memory, and processed by the next layer. Nonetheless, replacing complex and expensive solvers with fast, learned approximations is a very attractive and interesting direction.

An important decision to make at this stage is which neural network architecture to choose.

CHAPTER

FIVE

NEURAL NETWORK ARCHITECTURES

The connectivity of the individual "neurons" in a neural network has a substantial influence on the capabilities of the network. Typical ones consist of a large number of these connected "neuron" units. Over the course of many years, several key architectures have emerged as particularly useful choices, and in the following we'll go over the main considerations for choosing an architecture. Our focus is to introduce ways of incorporating PDE-based models (the "physics"), rather than the subtleties of NN architectures.

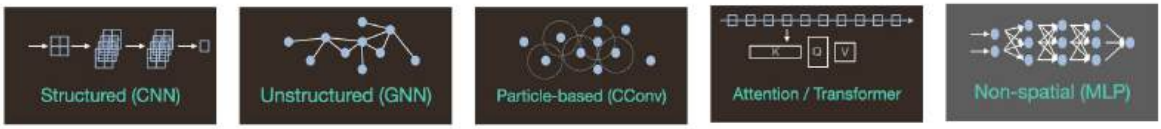


Fig. 5.1: We'll discuss a range of architecture, from regular convolutions over graph- and particle-based convolutions to newer attention-based variants.

5.1 Spatial Arrangement

A first, fundamental aspect to consider for choosing an architecture in the context of physics simulations (and for ruling out unsuitable options) is the spatial arrangement of the data samples. We can distinguish four main cases:

1. No spatial arrangement,
2. A regular spacing on a grid (*structured*),
3. An irregular arrangement (*unstructured*), and
4. Irregular positions without connectivity (*particle* / *point*-based).

For certain problems, there is no spatial information or arrangement (case 1). E.g., predicting the temporal evolution of temperature and pressure of a single measurement probe over time does not involve any spatial dimension. The opposite case are probes placed on a nicely aligned grid (case 2), or at arbitrary locations (3). The last variant (case 4) is a slightly special case of the third one, where no clear or persistent links between sample points are known. This can, e.g., be the case for particle-based representations of turbulent fluids, where neighborhood change quickly over time.

5.2 No spatially arranged inputs

The first case is a somewhat special one: without any information about spatial arrangements, only *dense* ("fully connected" / MLP) neural networks are applicable.

If you decide to use a **neural fields** approach where the network receives the position as input, this has the same effect: the NN will not have any direct means of querying neighbors via architectural tricks ("inductive biases"). In this case, the building blocks below won't be applicable, and it's worth considering whether you can introduce more structure via a discretization.

Note that *physics-informed neural networks* (PINNs) also fall into this category. We'll go into more detail here later on (*Introduction to Differentiable Physics*), but generally it's advisable to consider switching to an approach that employs prior knowledge in the form of a discretization. This usually substantially improves inference accuracy and improves convergence. That PINNs can't solve real-world problems despite many years of research points to the fundamental problems of this approach.

Focusing on dense layers still leaves a few choices concerning the number of layers, their size, and activations. The other three cases have the same choices, and these hyperparameters of the architectures are typically determined over the course of training runs. General recommendations are that *ReLU* and smoother variants like *GELU* are good choices, and that the number of layers should scale together with their size. Next, we'll focus on the remaining three cases with spatial information in the following, as differences can have a profound impact here. So, below we target cases where we have a "computational domain" specifying the region of interest in which the samples are located.

5.3 Local vs Global

Probably the most important aspect of different architectures then is the question of their *receptive field*: this means for any single sample in our domain, which neighborhood of other sample points can influence the solution at this point. This is similar to classic considerations for PDE solving, where denoting a PDE as *hyperbolic* indicates its local, wave-like behavior in contrast to an *elliptic* one with global behavior. Certain NN architectures such as the classic convolutional neural networks (CNNs) support only local influences and receptive fields, while hierarchies with pooling expand these receptive field to effectively global ones. An interesting variant here are spectral architectures like FNOs, which provide global receptive fields at the expense of other aspects. In addition Transformers (with attention mechanisms), provide a more complicated but scalable alternative.

Thus, a fundamental distinction can be made in terms of spatially local vs global architectures, and for the latter, how they realize the global receptive field. The following table provides a first overview, and below we'll discuss the pros and cons of each variant.

	Grid	Unstructured	Points	Non-spatial
Local	CNN , ResNet	GNN	CConv	-
Global				MLP
- Hierarchy	U-Net, Dilation	Multi-scale GNN	Multi-scale CConv	-
- Spectral	FNO	Spectral GNN	(-)	-
- Sequence	Transformer	Graph Transformer	Point Trafo.	-

Note

Knowledge about the dependencies in your data, i.e., whether the dependencies are local or global, is important knowledge that should be leveraged.

If your data has primarily **local** influences, choosing an architecture with support for global receptive fields will most likely have negative effects on accuracy: the network will "waste" resources trying to capture global effects, or worst case approximate local effects with smoothed out global modes.

Vice versa, trying to approximate a **global** influence with a limited receptive field will be an unsolvable task, and most likely introduce substantial errors.

5.4 Regular, unstructured and point-wise data

The most natural start for making use of spatially arranged data is to employ a regular grid. Note that it doesn't have to be a Cartesian grid, but could be e deformed and adaptive grid [CT22]. The only requirement is a grid-like connectivity of the samples, even if they have an irregular spacing.

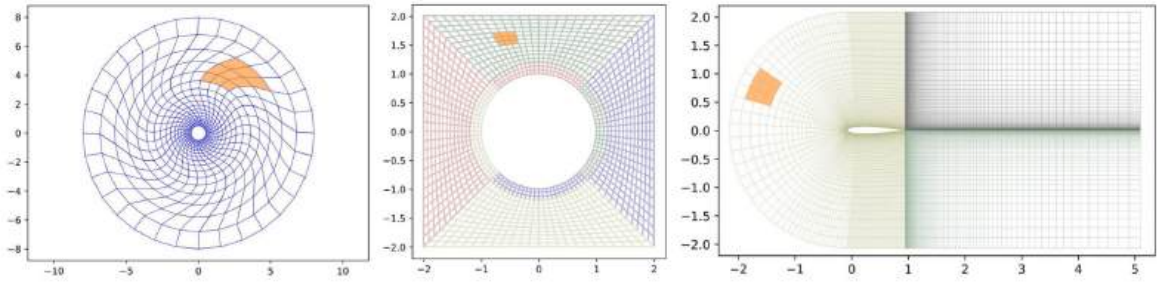


Fig. 5.2: A 3x3 convolution (orange) shown for differently deformed regular multi-block grids.

For unstructured data, graph-based neural networks (GNNs) are a good choice. While they're often discussed in terms of *message-passing* operations, they share a lot of similarities with structured grids: the basic operation of a message-passing step on a GNN is equivalent to a convolution on a grid [SGGP+20]. Hierarchies can likewise be constructed by graph coarsening [LPT25]. Hence, while we'll primarily discuss grids below, keep in mind that the approaches carry over to GNNs. As dealing with graph structures makes the implementation more complicated, we won't go into details until later.

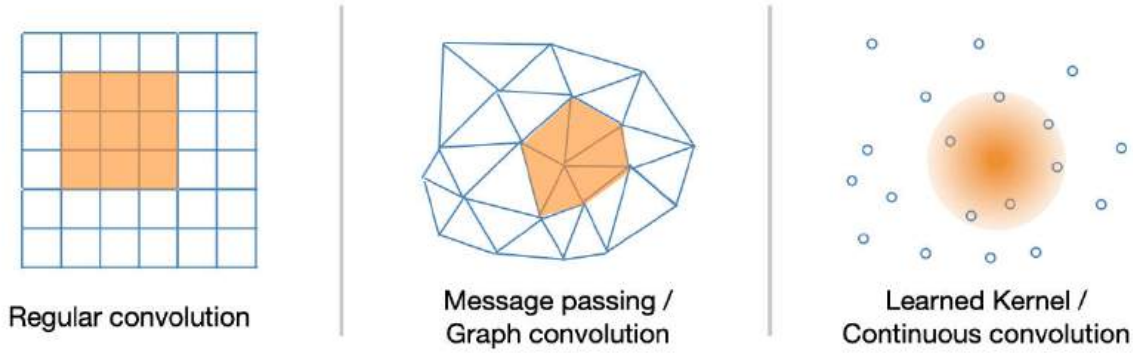


Fig. 5.3: Convolutions (and hierarchies) work very similarly irrespective of the structure of the data. Convolutions apply to grids, graphs and point-samples, as shown above. Likewise, the concepts discussed for grid-based algorithms in this book carry over to graphs and point collections.

Finally, point-wise (Lagrangian) samples can be seen as unstructured grids without connectivity. However, it can be

worth explicitly treating them in this way for improved learning and inference performance. Nonetheless, the two main ideas of convolutions and hierarchies carry over to Lagrangian data: continuous convolution kernels are a suitable tool, and neighborhood based coarsening yields hierarchies [PUKT22].

5.5 Hierarchies

A powerful and natural tool to work with **local** dependencies are convolutional layers on regular grids. The corresponding neural networks (CNNs) are a classic building block of deep learning, and very well researched and supported throughout. They are comparatively easy to train, and usually very efficiently implemented in APIs. They also provide a natural connection to classical numerics: discretizations of differential operators such as gradient and Laplacians are often thought of in terms of "stencils", which are an equivalent of a convolutional layer with a set of specific weights. E.g., consider the classic stencil for a normalized Laplacian ∇^2 in 1D: $[1, -2, 1]$. It can directly be mapped to a 1D convolution with kernel size 3 and a single input and output channel. The non trainable weights of the kernel can be set to the coefficients of the Laplacian stencil above.

Using convolutional layers is quite straight forward, but the question of how to incorporate **global** dependencies into CNNs is an interesting one. Over time, two fundamental approaches have been established here in the field: *hierarchical* networks via pooling (U-Nets [RFB15]), and sparse, point wise samples with enlarged spacing (Dilation [YK15]). They both reach the goal of establishing a global receptive field, but have a few interesting differences under the hood.

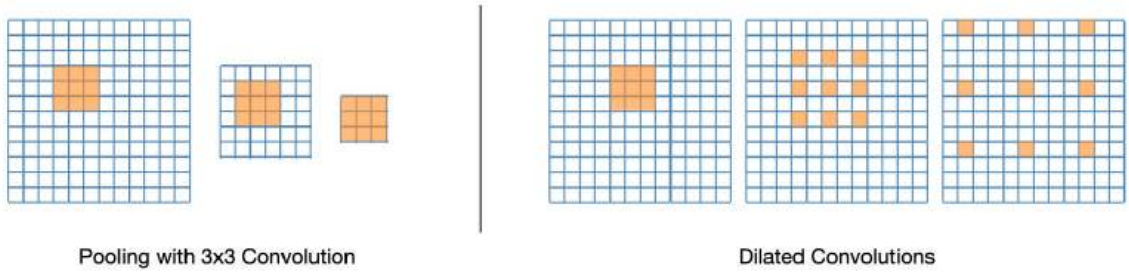


Fig. 5.4: A 3x3 convolution shown for a pooling-based hierarchy (left), and a dilation-based convolution (right). Not that in both cases the convolutions cover larger ranges of the input data. However, the hierarchy processes $O(logN)$ less data, while the dilation processes the full input with larger strides. Hence the latter has an increased cost due to the larger number of sample points, and the less regular data access.

- U-Nets are based on *pooling* operations. Akin to a geometric multigrid hierarchy, the spatial samples are down-sampled to coarser and coarser grids, and upsampled in the later half of the network. This means that even if we keep the kernel size of a convolution fixed, the convolution will be able to "look" further in terms of physical space due to the previous downsampling operation. The number of sample points decreases logarithmically, making convolutions on lower hierarchy levels very efficient. While the different re-sampling methods (mean, average, point-wise …) have a minor effect, a crucial ingredient for U-Net are *skip connection*. They connect the earlier layers of the first half directly with the second half via feature concatenation. This turns out to be crucial to avoid a loss of information. Typically, the deepest "bottle-neck" layer with the coarsest representation has trouble storing all details of the finest one. Providing this information explicitly via a skip-connection is crucial for improving accuracy.
- Dilation in the form of *dilated convolutions* places the sampling points for convolutions further apart. Hence instead of, e.g., looking at a 3x3 neighborhood, a convolution considers a 5x5 neighborhood but only includes 3x3 samples when calculating the convolution. The other samples in-between the used points are typically simply ignored. In contrast to a hierarchy, the number of sample points remains constant.

While both approaches reach the goal, and can perform very well, there's an interesting tradeoff: U-Nets take a bit more effort to implement, but can be much faster. The reason for the performance boost is the sub-optimal memory access of

the dilated convolutions: they skip through memory with a large stride, which gives a slower performance. The U-Nets, on the other had, basically precompute a compressed memory representation in the form of a coarse grid. Convolutions on this coarse grid are then highly efficient to compute. However, this requires slightly more effort to implement in the form of adding appropriate pooling layers (dilated convolutions can be as easy to implement as replacing the call to the regular convolution with a dilated one). The implementation effort of a U-Net can pay off significantly in the long run, when a trained network should be deployed in an application.

As mentioned above hierarchies are likewise important for graph nets. However, the question whether to "dilate or not" is not present for graph nets: here the memory access is always irregular, and dilation is unpopular as the strides would be costly to compute on general graphs. Hence, regular hierarchies in the form of multi-scale GNNs are highly recommended if global dependencies exist in the data.

5.6 Spectral methods

A fundamentally different avenue for establishing global receptive fields is provided by spectral methods, typically making use of Fourier transforms to transfer spatial data to the frequency domain. The most popular approach from this class of methods are *Fourier Neural Operators* (FNOs) [LKA+21]. An interesting aspect is the promise of a continuous representation via the functional representation, where a word of caution is appropriate: the function spaces are typically truncated, so it is often questionable whether the frequency representation really yields suitable solutions beyond the resolution of the training data.

In the following, however, we'll focus on the aspect of receptive fields in conjunction with performance aspects. Here, FNO-like methods have an interesting behavior: they modify frequency information with a dense layer. As the frequency signal after a Fourier transform would have the same size as the input, the dense layer works on a set of the M largest frequencies. For a two dimensional input that means M^2 modes, and the corresponding dense layer thus requires M^4 parameters.

An inherent advantage and consequence of the frequency domain is that all basis functions have global support. That means despite only working with a subset of all frequencies, FNOs can process (and modify) all parts of an input signal. This natural treatment of **global dependencies** is a inherent advantage of spectral methods.

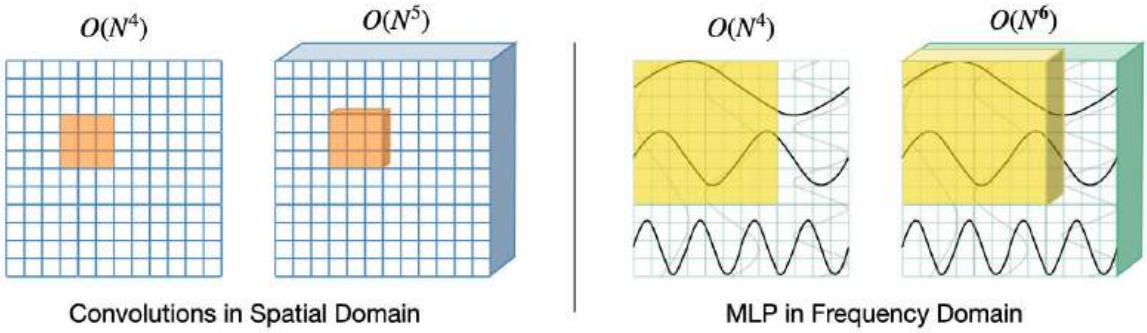
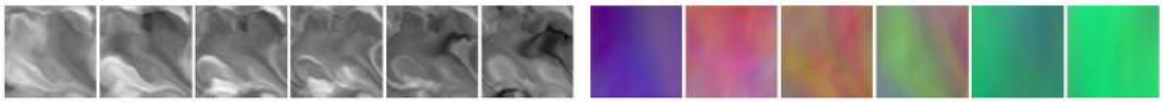


Fig. 5.5: Spatial convolutions (left, kernel in orange) and frequency processing in FNOs (right, coverage of dense layer in yellow). Not only do FNOs scale less well in 3D (**6th** instead of 5th power), their scaling constant is also proportional to the domain size, and hence typically larger.

Unfortunately, they're not well suited for higher dimensional problems: Moving from two to three dimensions increases the size of the frequencies to be handled to M^3. For the dense layer, this means M^6 parameters, a cubic increase. For convolutions, there's no huge difference in 2D: a regular convolution with kernel size K requires K^2 weights in 2D, and induces another $O(K^2)$ scaling for processing features, in total $O(K^4)$. However, in 3D regular convolutions scale much better: in 3D only the kernel size increases to K^3, giving an overall complexity of $O(K^5)$ in 3D. Thus, the exponent is 5 instead of 6.

To make things worse, the frequency coverage M of FNOs needs to scale with the size of the spatial domain, hence typically $M > K$ and $M^6 \gg K^5$. Thus, FNOs would require intractable amounts of parameters, and are thus not recommendable for 3D (or higher dimensional) problems. Architectures like CNNs require much fewer weights, and in conjunction with hierarchies can still handle global dependencies efficiently.



5.7 Attention and Transformers

A newer and exciting development in the deep learning field are attention mechanisms. They've been hugely successful in the form of *Transformers* for processing language and natural images, and bear promise for physics-related problems. However, it's still open, whether they're really generally preferable over more "classic" architectures. The following section will give an overview of the main pros and cons.

Transformers generally work in two steps: the input is encoded into *tokens* with an encoder-decoder network. This step can take many forms, and usually primarily serves to reduce the number of inputs, e.g., to work with pieces of an image rather than individual pixels. The attention mechanism then computes a weighting for a collection if incoming tokens. This is a floating point number for each token, traditionally interpreted as indicating which parts of the input are important, and which aren't. In modern architectures, the floating point weighting of the attention are directly used to modify an input. In *self-attention*, the weighting is computed from each input towards all other input tokens. This is a mechanism to handle **global dependencies**, and hence directly fits into the discussion above. In practice, the attention is computed via three matrices: the query Q, the key matrix K, and a value matrix V. For N tokens, the outer product QK^T produces an $N \times N$ matrix, and runs through a Softmax layer, after which it is multiplied with V (containing a linear projection of the input tokens) to produce the attention output vector.

In a Transformer architecture, the attention output is used as component of a building block: the attention is calculated and used as a residual (added to the input), stabilized with a layer normalization, and then processed in a two-layer *feed forward* network (FFN). The latter is simply a combination of two dense layers with an activation in between. This *Transformer block*, summarized below visually, is applied multiple times before the final output is decoded into the original space.

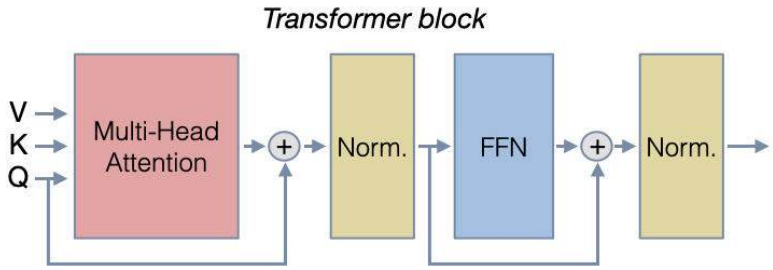


Fig. 5.6: Visual summary of a single transformer block. A full network repeats this structure several times to infer the result.

This Transformer architecture was shown to scale extremely well to networks with huge numbers of parameters, one of the key advantages of Transformers. Note that a large part of the weights typically ends up in the matrices of the attention, and not just in the dense layers. At the same time, attention offers a powerful way for working with global dependencies in inputs. This comes at the cost of a more complicated architecture. An inherent problem of the self-attention mechanism above is that it's quadratic in the number of tokens N. This naturally puts a limit on the size and resolution of inputs. Under the hood, it's also surprisingly simple: the attention algorithm computes an $N \times N$ matrix, which is not too far from applying a simple dense layer (this would likewise come with an $N \times N$ weight matrix) to resolve global influences.

This bottleneck can be addressed with *linear attention*: it changes the algorithm above to multiply Q and $(K^T V)$ instead, applying a non-linearity (e.g., an exponential) to both parts beforehand. This avoids the $N \times N$ matrix and scales linearly in N. However, this improvement comes at the cost of a more approximate attention vector.

An interesting aspect of Transformer architectures is also that they've been applied to structured as well as unstructured inputs. I.e., they've been used for graphs, points as well as grid-based data. In all cases the differences primarily lie in how inputs are mapped to the tokens. The attention is typically still "dense" in the token space. This is a clear limitation: for problems with a known spatial structure, discarding this information will inevitably need to be compensated for, e.g., with a larger weight count or lower inference accuracy. Nonetheless, Transformers are an extremely active field within DL, and clearly a potential contender for future NN algorithms.

5.8 Summary of Architectures

The paragraphs above have given an overview over several fundamental considerations when choosing a neural network architecture for a physics-related problem. To re-cap, the main consideration when choosing an architecture is knowledge about **local** or **global** dependencies in the data. Tailoring an architecture to this difference can have a big impact. And while the spatial structure of the data seems to dictate choices, it can be worth considering to transfer the data to another data structure. E.g., to project unstructured points onto a (deformed) regular grid to potentially improve accuracy and performance.

Also, it should be mentioned that hybrids of the *canonical* architectures mentioned above exist: e.g., classic U-Nets with skip connections have been equipped with components of Transformer architectures (like attention and normalization) to yield an improved performance. An implementation of such a "modernized" U-Net can be found in *Diffusion-based Time Prediction*.

5.9 Show me some code!

Let's finally look at a code example that trains a neural network: we'll replace a full solver for *turbulent flows around airfoils* with a surrogate model from [TWPH20] using a U-Net with a global receptive field as operator.

CHAPTER

SIX

SUPERVISED TRAINING FOR RANS FLOWS AROUND AIRFOILS

6.1 Overview

For this example of supervised training we target turbulent airflows around wing profiles: the learned operator should provide the average motion and pressure distribution around a given airfoil geometry for different Reynolds numbers and angles of attack. Thus, inputs to the neural network are airfoil shape, Reynolds numbers, and angle of attack, and it should compute a time averaged velocity field with 2 components, and the pressure field around the airfoil.

This is classically approximated with *Reynolds-Averaged Navier Stokes* (RANS) models, and this setting is still one of the most widely used applications of Navier-Stokes solvers in industry. However, instead of relying on traditional numerical methods to solve the RANS equations, we now aim for training a surrogate model via a neural network that completely bypasses the numerical solver. [run in colab]

6.2 Formulation

With the supervised formulation from *Supervised Training*, our learning task is pretty straight-forward, and can be written as

$$\arg\min_\theta \sum_i (f(x_i;\theta) - y_i^*)^2,$$

where x and y^* each consist of a set of physical fields, and the index i evaluates the difference across all discretization points in our data sets.

The goal is to infer velocity $\mathbf{u} = u_x, u_y$ and a pressure field p in a computational domain Ω around the airfoil in the center of Ω. u_x, u_y and p each have a dimension of 128^2. As inputs we have the Reynolds number $\mathrm{Re} \in \mathbb{R}$, the angle of attack $\alpha \in \mathbb{R}$, and the airfoil shape $\mathbf{s}$ encoded as a rasterized grid with 128^2. Re and α are provided in terms of the freestream flow velocity $\mathbf{f}$, the x and y components of which are represented as constant fields of the same size, and contain zeros in the airfoil region. Thus, put together, both input and output have the same dimensions: $x, y^* \in \mathbb{R}^{3\times128\times128}$. The inputs contain $[f_x, f_y, \mathrm{mask}]$, while the outputs store the channels $[p, u_x, u_y]$. This is exactly what we'll specify as input and output dimensions for the NN below.

A point to keep in mind here is that our quantities of interest in y^* contain three different physical fields. While the two velocity components are quite similar in spirit, the pressure field typically has a different behavior with an approximately squared scaling with respect to the velocity (cf. Bernoulli). This implies that we need to be careful with simple summations (as in the minimization problem above), and that we should take care to normalize the data. If we don't take care, one of the components can dominate and the aggregation in terms of mean will lead the NN to spend more resources to learn the large component rather than the other ones causing smaller errors.

6.3 Code coming up...

Let's get started with the implementation. Note that we'll skip the data generation process here. The code below is adapted from [TWPH20] and this codebase, which you can check out for details. Below, we'll simply use a small set of training data generated with a Spalart-Almaras RANS simulation in OpenFOAM. First, let's import the required module, and install the dataloader from git.

```
import os, sys, random
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

import torch
import torch.nn as nn
import torch.optim as optim

!pip install --upgrade --quiet git+https://github.com/tum-pbs/pbdl-dataset
from pbdl.torch.loader import Dataloader
```

The next cell will download the training data from HuggingFace, which can take a moment... The PBDL dataloader call below directly splits it into 320 samples for training, and 80 samples for validation. These validation samples are using the same airfoil shapes as the training samples, but different conditions (later on we'll download new shapes for testing).

```
BATCH_SIZE = 10

loader_train, loader_val = Dataloader.new_split(
    [320, 80],
    "airfoils",
    batch_size=BATCH_SIZE, normalize_data=None,
)
```

```
Warning: `airfoils` is stored in single-file format. The download might take some␣
 ↪time.
Success: Loaded airfoils with 400 simulations and 1 samples each.
```

6.4 RANS training data

Now we have the training and validation data. In general it's very important to understand the data we're working with as much as possible (for any ML task the *garbage-in-gargabe-out* principle definitely holds). We should at least understand the data in terms of dimensions and rough statistics, but ideally also in terms of content. Otherwise we'll have a very hard time interpreting the results of a training run. And despite all the *AI magic*: if you can't make out any patterns in your data, NNs most likely also won't find any useful ones.

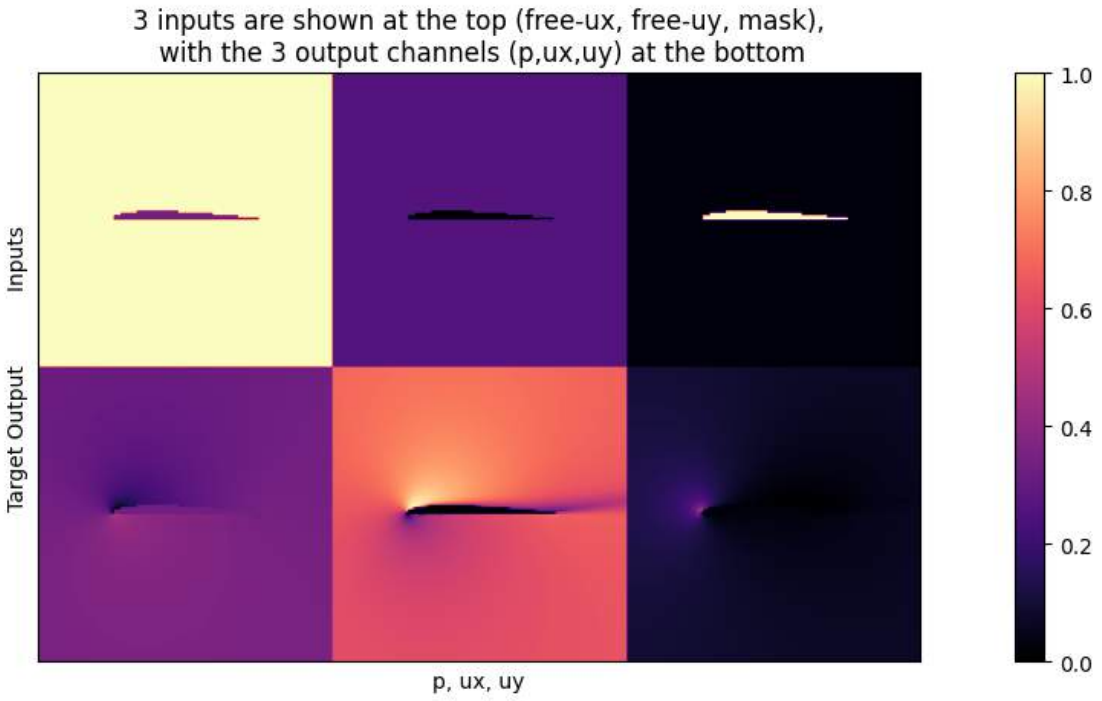
Hence, let's look at one of the training samples. The following is just some helper code to show images side by side.

```
def plot(a1, a2, mask=None, stats=False, bottom="NN Output", top="Reference",␣
 ↪title=None):
    c = []
    if mask is not None: mask = np.asarray(mask)
    for i in range(3):
        a2i = np.asarray(a2[i])
        if mask is not None: a2i = a2i - mask*a2i # optionally mask out inner region
        b = np.flipud(np.concatenate((a2i, a1[i]), axis=1).transpose())
        min, mean, max = np.min(b), np.mean(b), np.max(b)
```

(continues on next page)

(continued from previous page)

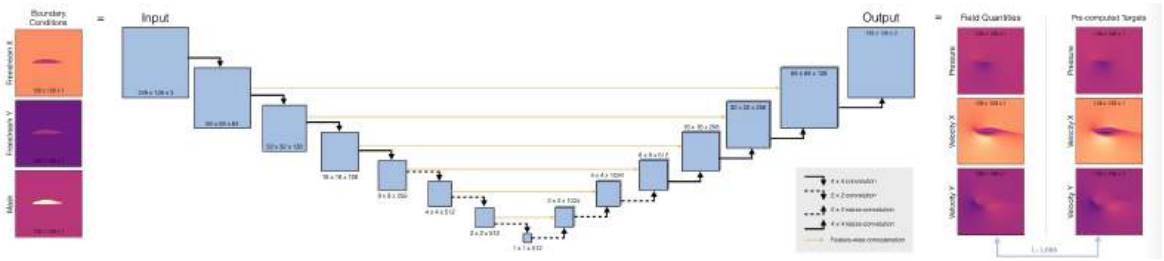
```
        if stats:
            print("Stats %d: " % i + format([min, mean, max]))
        b -= min
        b /= max - min
        c.append(b)
    fig, axes = plt.subplots(1, 1, figsize=(16, 5))
    axes.set_xticks([]) ; axes.set_yticks([])
    im = axes.imshow(np.concatenate(c, axis=1), origin="upper", cmap="magma")
    fig.colorbar(im, ax=axes)
    axes.set_xlabel("p, ux, uy")
    axes.set_ylabel("%s            %s" % (bottom, top))
    if title is not None: plt.title(title)
    plt.show()

inputs, targets = next(iter(loader_train))
plot(inputs[0], targets[0], stats=False, bottom="Target Output", top="Inputs", title=
 ↪"3 inputs are shown at the top (free-ux, free-uy, mask),\n with the 3 output␣
 ↪channels (p,ux,uy) at the bottom")
```



6.5 Network setup

Now we can set up the architecture of our neural network, we'll use a fully convolutional U-net. This is a widely used architecture that uses a stack of convolutions across different spatial resolutions. The main deviation from a regular convnet is the hierarchy (for a global receptive field), and to introduce *skip connections* from the encoder to the decoder part. This ensures that no information is lost during feature extraction. (Note that this only works if the network is to be used as a whole. It doesn't work in situations where we'd, e.g., want to use the decoder as a standalone component.)

Here's a overview of the architecure:



First, we'll define a helper to set up a convolutional block in the network, `blockUNet`. Note, we don't use any pooling! Instead we use strides and transpose convolutions (these need to be symmetric for the decoder part, i.e. have an uneven kernel size), following best practices. The full pytroch neural network is managed via the `DfpNet` class.

```
def blockUNet( in_c, out_c, name, size=4, pad=1, transposed=False, bn=True,␣
↪activation=True, relu=True, dropout=0.0 ):

    block = nn.Sequential()

    if not transposed:
        block.add_module(
            "%s_conv" % name,
            nn.Conv2d(in_c, out_c, kernel_size=size, stride=2, padding=pad,␣
↪bias=True),
        )
    else:
        block.add_module(
            "%s_upsam" % name, nn.Upsample(scale_factor=2, mode="bilinear")
        )
        # reduce kernel size by one for the upsampling (ie decoder part)
        block.add_module(
            "%s_tconv" % name,
            nn.Conv2d( in_c, out_c, kernel_size=(size - 1), stride=1, padding=pad,␣
↪bias=True ),
        )

    if bn:
        block.add_module("%s_bn" % name, nn.BatchNorm2d(out_c))
    if dropout > 0.0:
        block.add_module("%s_dropout" % name, nn.Dropout2d(dropout, inplace=True))

    if activation:
        if relu:
            block.add_module("%s_relu" % name, nn.ReLU(inplace=True))
        else:
            block.add_module("%s_leakyrelu" % name, nn.LeakyReLU(0.2, inplace=True))
```

(continues on next page)

(continued from previous page)

```
    return block


class DfpNet(nn.Module):
    def __init__(self, channelExponent=6, dropout=0.0):
        super(DfpNet, self).__init__()
        channels = int(2**channelExponent + 0.5)

        self.layer1 = blockUNet( 3,            channels * 1, "enc_layer1",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, )
        self.layer2 = blockUNet( channels,     channels * 2, "enc_layer2",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, )
        self.layer3 = blockUNet( channels * 2, channels * 2, "enc_layer3",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, )
        self.layer4 = blockUNet( channels * 2, channels * 4, "enc_layer4",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, )
        self.layer5 = blockUNet( channels * 4, channels * 8, "enc_layer5",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, )
        self.layer6 = blockUNet( channels * 8, channels * 8, "enc_layer6",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, size=2, pad=0, )
        self.layer7 = blockUNet( channels * 8, channels * 8, "enc_layer7",␣
↪transposed=False, bn=True, relu=False, dropout=dropout, size=2, pad=0, )

        # note, kernel size is internally reduced by one for the decoder part
        self.dlayer7 = blockUNet( channels * 8, channels * 8,  "dec_layer7",␣
↪transposed=True, bn=True, relu=True, dropout=dropout, size=2, pad=0, )
        self.dlayer6 = blockUNet( channels * 16, channels * 8, "dec_layer6",␣
↪transposed=True, bn=True, relu=True, dropout=dropout, size=2, pad=0, )
        self.dlayer5 = blockUNet( channels * 16, channels * 4, "dec_layer5",␣
↪transposed=True, bn=True, relu=True, dropout=dropout, )
        self.dlayer4 = blockUNet( channels * 8, channels * 2,  "dec_layer4",␣
↪transposed=True, bn=True, relu=True, dropout=dropout, )
        self.dlayer3 = blockUNet( channels * 4, channels * 2,  "dec_layer3",␣
↪transposed=True, bn=True, relu=True, dropout=dropout, )
        self.dlayer2 = blockUNet( channels * 4, channels,      "dec_layer2",␣
↪transposed=True, bn=True, relu=True, dropout=dropout, )
        self.dlayer1 = blockUNet( channels * 2, 3,             "dec_layer1",␣
↪transposed=True, bn=False, activation=False, dropout=dropout, )

    def forward(self, input):
        # note, this Unet stack could be allocated with a loop, of course...
        out1 = self.layer1(input)
        out2 = self.layer2(out1)
        out3 = self.layer3(out2)
        out4 = self.layer4(out3)
        out5 = self.layer5(out4)
        out6 = self.layer6(out5)
        out7 = self.layer7(out6)
        # ... bottleneck ...
        dout6 = self.dlayer7(out7)
        dout6_out6 = torch.cat([dout6, out6], 1)
        dout6 = self.dlayer6(dout6_out6)
        dout6_out5 = torch.cat([dout6, out5], 1)
        dout5 = self.dlayer5(dout6_out5)
        dout5_out4 = torch.cat([dout5, out4], 1)
        dout4 = self.dlayer4(dout5_out4)
        dout4_out3 = torch.cat([dout4, out3], 1)
```

(continues on next page)

(continued from previous page)

```
        dout3 = self.dlayer3(dout4_out3)
        dout3_out2 = torch.cat([dout3, out2], 1)
        dout2 = self.dlayer2(dout3_out2)
        dout2_out1 = torch.cat([dout2, out1], 1)
        dout1 = self.dlayer1(dout2_out1)
        return dout1

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find("BatchNorm") != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)
```

Next, we can initialize an instance of the `DfpNet`.

Below, the `EXPO` parameter here controls the exponent for the feature maps of our Unet: this directly scales the network size (an exponent of 4 gives a network with ca. 585k parameters). This is a medium sized network for a generative NN with $3 \times 128^2 =$ ca. $49k$ outputs, and still yields fast training times. Hence it's a good starting point. The `weights_init` function initializes the conv net to a reasonable initial value range, so that we can directly train with a fixed learning rate (otherwise learning rate schedules are highly recommended).

```
# channel exponent to control network size
EXPO = 4

torch.set_default_device("cuda:0")
device = torch.get_default_device()

net = DfpNet(channelExponent=EXPO)
net.apply(weights_init)

# crucial parameter to keep in view: how many parameters do we have?
nn_parameters = filter(lambda p: p.requires_grad, net.parameters())
print("Trainable params: {}   -> crucial! always keep in view... ".format( sum([np.
 ↪prod(p.size()) for p in nn_parameters]) ))

LR    = 0.0002          # learning rate

loss = nn.L1Loss()
optimizer = optim.Adam(net.parameters(), lr=LR, betas=(0.5, 0.999), weight_decay=0.0)
```

```
Trainable params: 585027   -> crucial! always keep in view...
```

As the subtle hint in the print statement indicates, the parameter count is a crucial number to have in view when training NNs. It's easy to change settings, and get a network that has millions of parameters, and as a result can cause wasting resources at training time (and potentially training instabilities). The number of parameters definitely has to be matched with the amount of training data, and should also scale with the depth of the network. How exactly these three relate to each other is problem dependent, though.

6.6 Training

Finally, we can train the NN. This step can take a while, as the training runs over all 320 samples 100 times, and continually evaluates the validation samples to keep track of how well the current state of the NN is doing.

```
EPOCHS = 200    # number of training epochs

loss_hist = []
loss_hist_val = []

if os.path.isfile("dfpnet"): # NT_DEBUG
    print("Found existing network, loading & skipping training")
    net.load_state_dict(torch.load("dfpnet"))

else:
    print("Training from scratch...")
    pbar = tqdm(initial=0, total=EPOCHS, ncols=96)
    for epoch in range(EPOCHS):

        # training
        net.train()
        loss_acc = 0
        for i, (inputs, targets) in enumerate(loader_train):
            inputs = inputs.float()
            targets = targets.float()

            net.zero_grad()
            outputs = net(inputs)
            lossL1 = loss(outputs, targets)
            lossL1.backward()
            optimizer.step()
            loss_acc += lossL1.item()

        loss_hist.append(loss_acc / len(loader_train))

        # evaluate validation samples
        net.eval()
        loss_acc_v = 0
        with torch.no_grad():
            for i, (inputs, targets) in enumerate(loader_val):
                inputs = inputs.float()
                targets = targets.float()

                outputs = net(inputs)
                loss_acc_v += loss(outputs, targets).item()

        loss_hist_val.append(loss_acc_v / len(loader_val))
        pbar.set_description("loss train: {:7.5f}, loss val: {:7.5f}".format( loss_
↪hist[-1], loss_hist_val[-1] ) , refresh=False); pbar.update(1)

    torch.save(net.state_dict(), "dfpnet")
    print("training done, saved network weights")

loss_hist = np.asarray(loss_hist)
loss_hist_val = np.asarray(loss_hist_val)
```

```
Training from scratch...
```

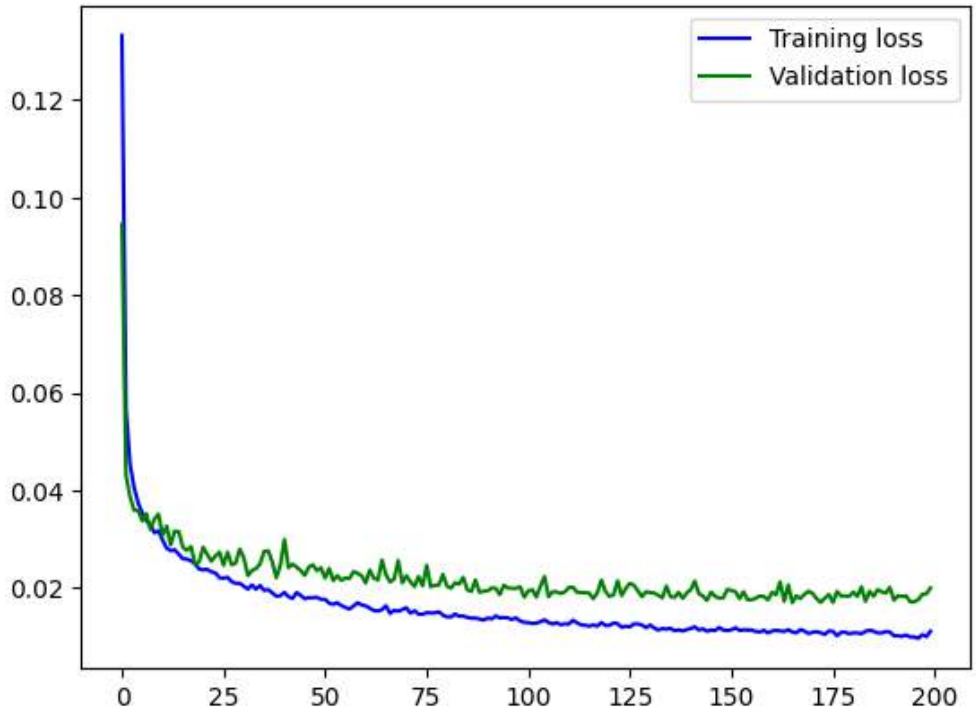
```
loss train: 0.01113, loss val: 0.01996: 100%|████████████████| 200/200 [05:40
↪<00:00,  1.61s/it]
```

```
training done, saved network weights
```

The NN is trained, the losses should have gone down in terms of absolute values: With the standard settings from an initial value of around 0.2 for the validation loss, to ca. 0.02 after training.

Let's look at the graphs to get some intuition for how the training progressed over time. This is typically important to identify longer-term trends in the training. In practice it's tricky to spot whether the overall trend of 100 or so noisy numbers in a command line log is going slightly up or down - this is much easier to spot in a visualization.

```
plt.plot(np.arange(loss_hist.shape[0]), loss_hist, "b", label="Training loss")
plt.plot(np.arange(loss_hist_val.shape[0]), loss_hist_val, "g", label="Validation loss
↪")
plt.legend()
plt.show()
```



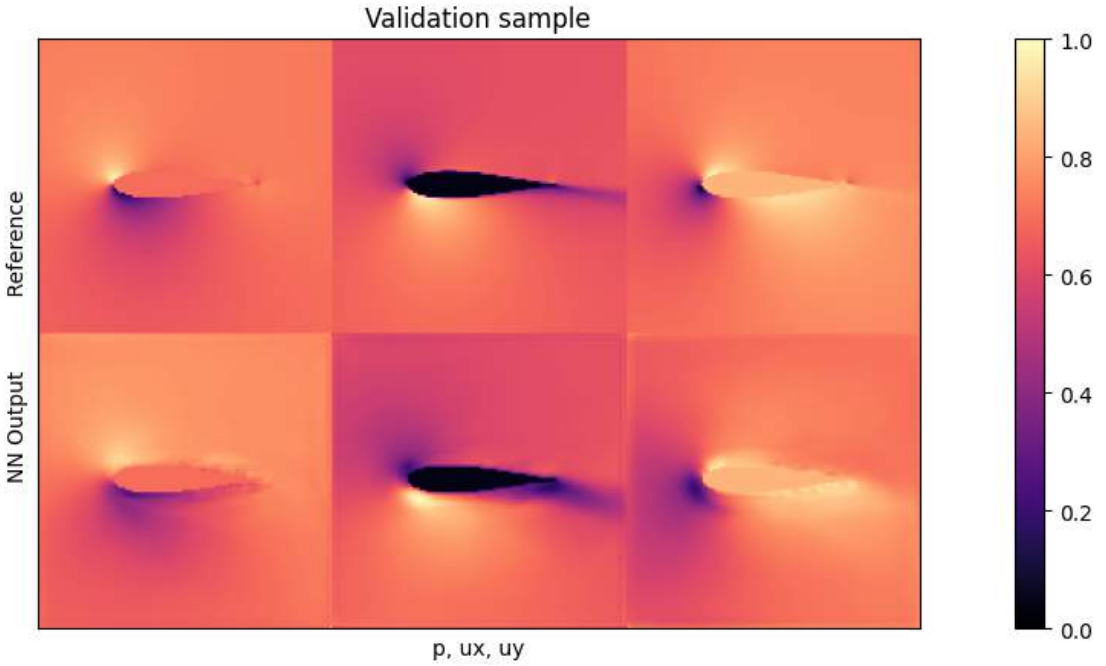
You should see a curve that goes down for ca. 40 epochs, and then starts to flatten out. In the last part, it's still slowly decreasing, and most importantly, the validation loss is not increasing. This would be a certain sign of overfitting, and something that we should avoid. (Try decreasing the amount of training data artificially, then you should be able to intentionally cause overfitting.)

Note that the validation loss is generally higher above, as the dataset here is relatively small. At some point, the network will not be able to get new information from it that transfers to the validation samples.

A general word of caution here: never evaluate your network with training data. That won't tell you much because overfitting is a very common problem. At least use data the network hasn't seen before, i.e. validation data, and if that

looks good, try some more different (at least slightly out-of-distribution) inputs, i.e., *test data*. The next cell runs the trained network on a batch of samples from the validation data, and displays one with the `plot` function.

```
net.eval()
inputs, targets = next(iter(loader_val))
inputs = inputs.float()
targets = targets.float()

outputs = net(inputs)

outputs = outputs.data.cpu().numpy()
inputs = inputs.cpu()
targets = targets.cpu()
plot(targets[0], outputs[0], mask=inputs[0][2], title="Validation sample")
```



This shows a good resemblance here between input out network output. The region around the airfoil is typically still a bit noisy (this is caused by the Dirichlet boundary, and could be alleviated with a modified loss and larger networks). The pressure values are typically the most difficult ones to learn. We'll save the more detailed evaluation for the test data, though.

6.7 Test evaluation

Now let's look at actual test samples: In this case we'll use new airfoil shapes as out-of-distribution (OOD) data. These are shapes that the network has not seen in any training samples, and hence it tells us a bit about how well the NN generalizes to unseen inputs (the validation data wouldn't suffice to draw conclusions about generalization).

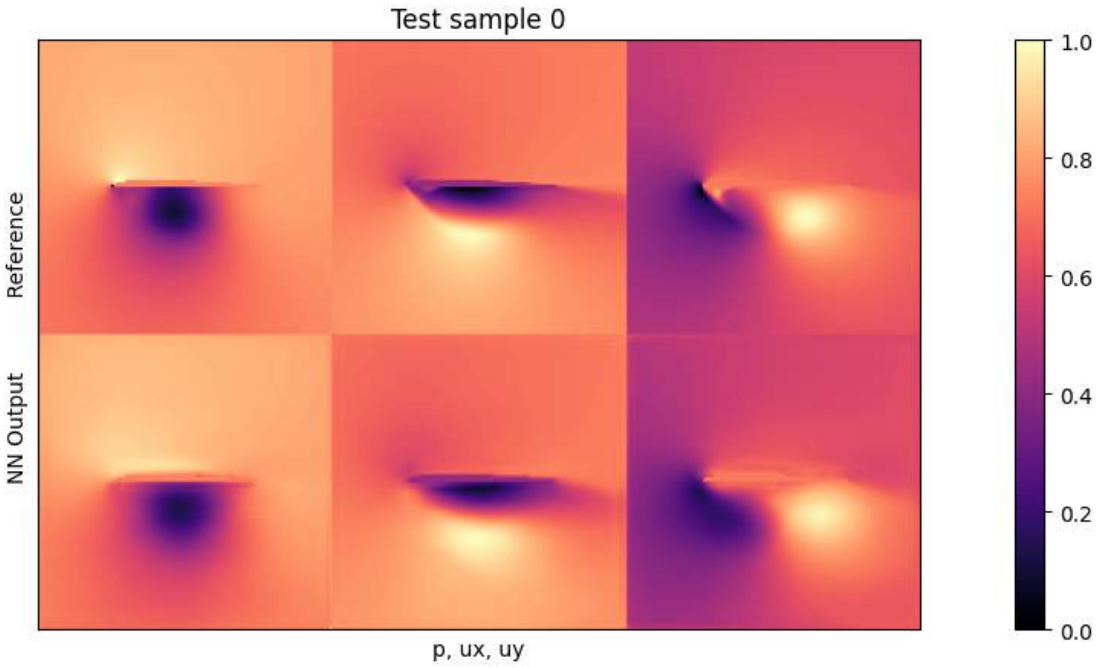
We'll use the same visualization as before, and as indicated by the Bernoulli equation, especially the *pressure* in the first column is a challenging quantity for the network. Due to it's cubic scaling w.r.t. the input freestream velocity and localized peaks, it is the toughest quantity to infer for the network.

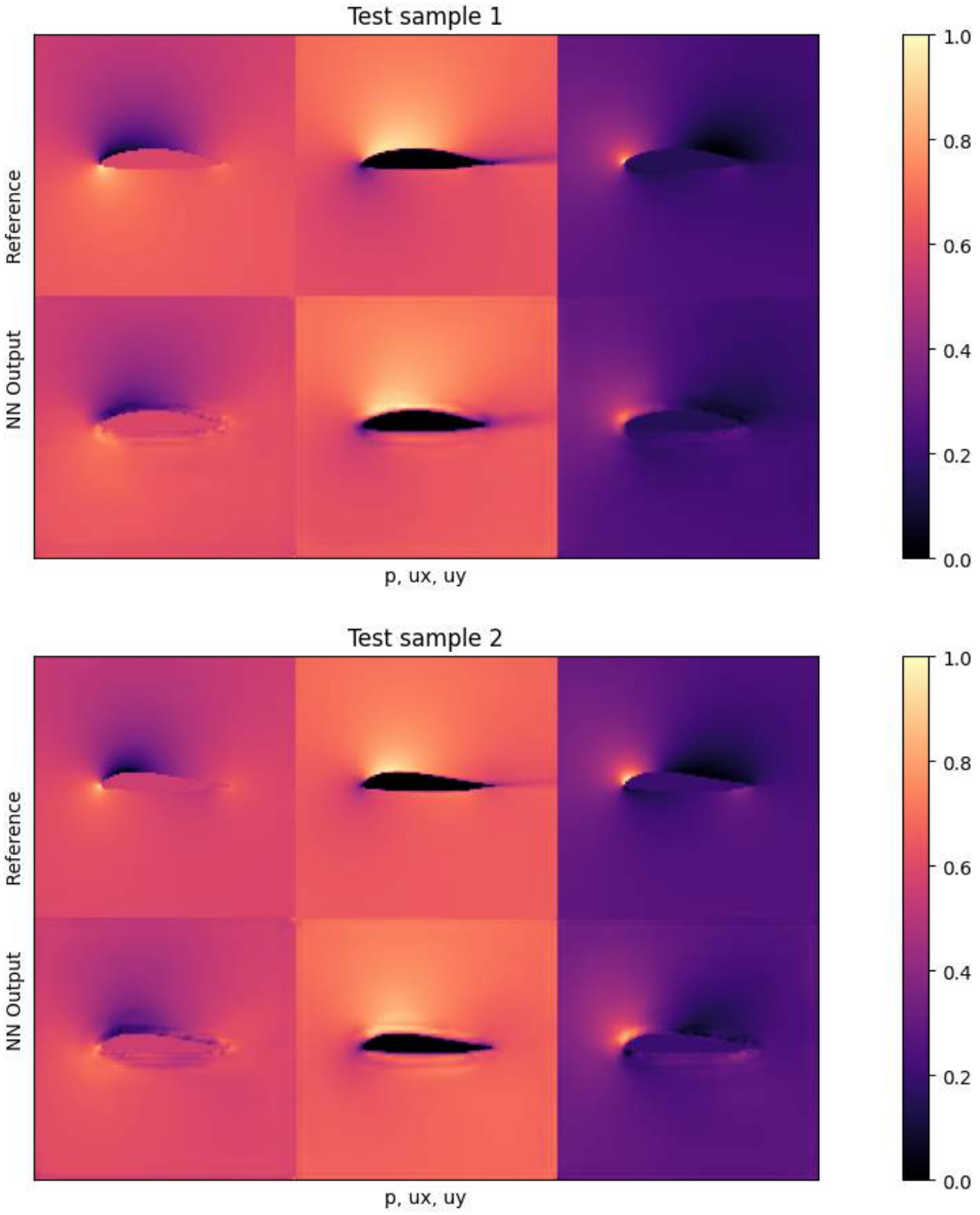
The cell below first downloads a smaller archive with these test data samples, and then runs them through the network.

The evaluation loop also computes the accumulated L1 error such that we can quantify how well the network does on the test samples.

```
loader_test = Dataloader( "airfoils-test", batch_size=1, normalize_data=None,␣
↪shuffle=False )
loss = nn.L1Loss()

net.eval()
L1t_accum = 0.
for i, testdata in enumerate(loader_test, 0):
    inputs_curr, targets_curr = testdata
    inputs = inputs_curr.float()
    targets = targets_curr.float()

    outputs = net(inputs)

    outputs_curr = outputs.data.cpu().numpy()
    inputs_curr = inputs_curr.cpu()
    targets_curr = targets_curr.cpu()

    L1t_accum += loss(outputs, targets).item()
    if i<3: plot(targets_curr[0] , outputs_curr[0], mask=inputs_curr[0][2], title=
↪"Test sample %d"%(i))

print("\nAverage relative test error: {}".format( L1t_accum/len(loader_test) ))
```

```
Warning: `airfoils-test` is stored in single-file format. The download might take␣
↪some time.
Success: Loaded airfoils-test with 10 simulations and 1 samples each.
```






```
Average relative test error: 0.026288176793605088
```

The average test error with the default settings should be close to 0.025. As the inputs are normalized, this means the average relative error across all three fields is around 2.5% w.r.t. the maxima of each quantity. This is not too bad for new shapes, but clearly leaves room for improvement.

Looking at the visualizations, you'll notice that especially high-pressure peaks and pockets of larger y-velocities are missing

in the outputs. This is primarily caused by the small network, which does not have enough resources to reconstruct details. The L^2 also has an averaging behavior, and favours larger structures (the surroundings) over localized peaks.

Nonetheless, we have successfully replaced a fairly sophisticated RANS solver with a small and fast neural network architecture. It has GPU support "out-of-the-box" (via pytorch), is differentiable, and introduces an error of only a few per-cent. With additional changes and more data, this setup can be made highly accurate [CT22].

6.8 Next steps

There are many obvious things to try here (see the suggestions below), e.g. longer training, larger data sets, larger networks etc.

- Experiment with learning rate, dropout, and network size to reduce the error on the test set. How small can you make it with the given training data?
- The setup above uses normalized data. Instead you can recover the original fields by undoing the normalization to check how well the network does w.r.t. the original quantities.
- As you'll see, it's a bit limited here what you can get out of this dataset, head over to the main github repo of this project to download larger data sets, or generate own data.

CHAPTER

SEVEN

DISCUSSION OF SUPERVISED APPROACHES

The previous example illustrates that we supervised training serves as a basis that can solve non-trivial tasks. The main workload is collecting a large enough data set of examples. Once that exists, we can train a network to approximate the solution manifold represented by these solutions, and the trained network can give us predictions very quickly. There are a few important points to keep in mind when using supervised training.

7.1 Some things to keep in mind...

7.1.1 Natural starting point

Supervised training is the natural starting point for **any** DL project. It really **always** makes sense to start with a fully supervised test using as little data as possible. This will be a pure overfitting test, but if your network can't quickly converge and give a very good performance on a single example, then there's something fundamentally wrong with your code or data. Thus, there's no reason to move on to more complex setups that will make finding these fundamental problems more difficult.

Best practices

To summarize the scattered comments of the previous sections, here's a set of "golden rules" for setting up a DL project.

- Always start with a 1-sample overfitting test.
- Check how many trainable parameters your network has, and that your data is normalized properly.
- Make sure the NN converges.
- Then slowly increase the amount of training data (and potentially network parameters and depth).
- Adjust hyperparameters (especially the learning rate).
- Finally, introduce other components such as differentiable solvers or diffusion training.

7.1.2 Stability

A nice property of the supervised training is also that it's very stable. Things won't get any better when we include more complex physical models, or look at more complicated NN architectures.

Thus, again, make sure you can see a nice exponential falloff in your training loss when starting with the simple overfitting tests. This is a good setup to figure out an upper bound and reasonable range for the learning rate as the most central hyperparameter. You'll probably need to reduce it later on, but you should at least get a rough estimate of suitable values for η.

7.1.3 Know your data

All data-driven methods obey the *garbage-in-garbage-out* principle. Because of this it's important to work on getting to know the data you are dealing with. While there's no one-size-fits-all approach for how to best achieve this, we can strongly recommend to track a broad range of statistics of your data set. A good starting point are per quantity mean, standard deviation, min and max values. If some of these contain unusual values, this is a first indicator of bad samples in the data set.

These values can also be easily visualized in terms of histograms, to track down unwanted outliers. A small number of such outliers can easily skew a data set in undesirable ways.

Finally, checking the relationships between different quantities is often a good idea to get some intuition for what's contained in the data set. The next figure gives an example for this step.

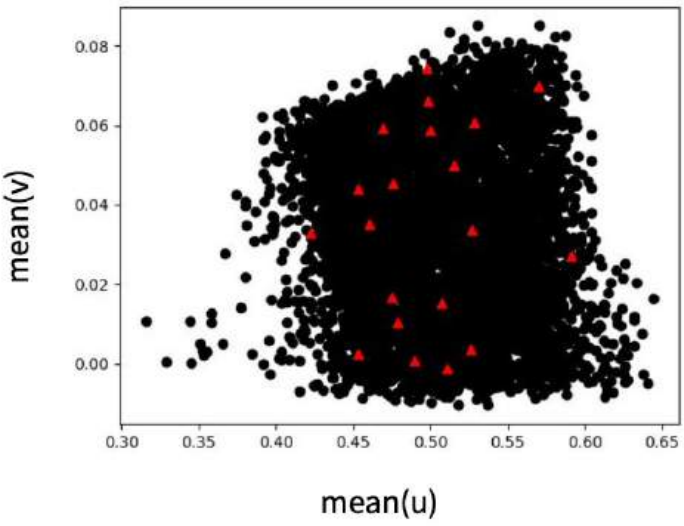


Fig. 7.1: An example from the airfoil case of the previous section: a visualization of a training data set in terms of mean u and v velocity of 2D flow fields. It nicely shows that there are no extreme outliers, but there are a few entries with relatively low mean u velocity on the left side. A second, smaller test data set is overlayed with red triangles, showing that its samples cover the range of mean motions well.

7.1.4 Where's the magic?

A comment that you'll often hear when talking about DL approaches, and especially when using relatively simple training methodologies is: "Isn't it just interpolating the data?"

Well, **yes** it is! And that's exactly what the NN should do. In a way - there isn't anything else to do. This is what *all* DL approaches are about. They give us smooth representations of the data seen at training time. Even if we'll use fancy physical models at training time later on, the NNs just adjust their weights to represent the signals they receive, and reproduce it.

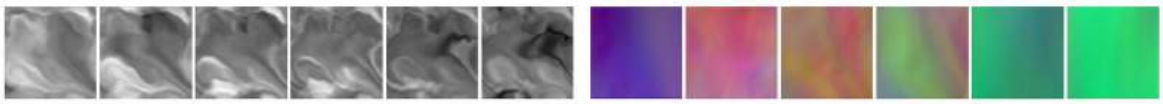
Due to the hype and numerous success stories, people not familiar with DL often have the impression that DL works like a human mind, and is able to extract fundamental and general principles from data sets ("messages from god" anyone?). That's not what happens with the current state of the art. Nonetheless, it's the most powerful tool we have to approximate complex, non-linear functions. It is a great tool, but it's important to keep in mind, that once we set up the training correctly, all we'll get out of it is an approximation of the function the NN was trained for - no magic involved.

An implication of this is that you shouldn't expect the network to work on data it has never seen. In a way, the NNs are so good exactly because they can accurately adapt to the signals they receive at training time, but in contrast to other learned representations, they're actually not very good at extrapolation. So we can't expect an NN to magically work with new inputs. Rather, we need to make sure that we can properly shape the input space, e.g., by normalization and by focusing on invariants.

To give a more specific example: if you always train your networks for inputs in the range $[0 \dots 1]$, don't expect it to work with inputs of $[27 \dots 39]$. In certain cases it's valid to normalize inputs and outputs by subtracting the mean, and normalizing via the standard deviation or a suitable quantile (make sure this doesn't destroy important correlations in your data). Looking ahead, a fast solver might be sufficient to handl the large offset of around 27, so that the NN can focus on a restricted input range in terms of a normalized residual.

As a rule of thumb: make sure you actually train the NN on the inputs that are as similar as possible to those you want to use at inference time.

This is important to keep in mind during the next chapters: e.g., if we want an NN to work in conjunction with a certain simulation environment, it's important to actually include the simulator in the training process. Otherwise, the network might specialize on pre-computed data that differs from what is produced when combining the NN with the solver, i.e it will suffer from *distribution shift*.



7.2 Supervised training in a nutshell

To summarize, supervised training has the following properties.

✓ Pros:

- Very fast training.
- Stable and simple.
- Great starting point.

✗ Con:

- Lots of data needed (loading can become a bottleneck).
- Potentially sub-optimal performance in terms of accuracy and generalization.
- Interactions with external "processes" (such as embedding into a solver) are difficult.

The next chapters will explain how to alleviate these shortcomings of supervised training. First, we'll look at bringing model equations into the picture via soft constraints, and afterwards we'll revisit the challenges of bringing together numerical simulations and learned approaches. Finally, we'll extend the basic approach for generative modeling with diffusion models and flow matching.

Part III

Physical Losses

CHAPTER

EIGHT

PHYSICAL LOSS TERMS

The supervised setting of the previous sections can quickly yield approximate solutions with a simple and stable training process. However, it's unfortunate that we only use physical models and numerical methods as an "external" tool to produce lots of data [?].

We as humans have a lot of knowledge about how to describe physical processes mathematically. As the following chapters will show, we can improve the training process by guiding it with our human knowledge of physics.

8.1 Using physical models

Given a PDE for $\mathbf{u}(\mathbf{x}, t)$ with a time evolution, we can typically express it in terms of a function $\mathcal{F}$ of the derivatives of $\mathbf{u}$ via

$$\mathbf{u}_t = \mathcal{F}(\mathbf{u}_x, \mathbf{u}_{xx}, ...\mathbf{u}_{xx...x}),$$

where the $_\mathbf{x}$ subscripts denote spatial derivatives with respect to the spatial dimensions (this could of course also include mixed derivatives with respect to different axes). $\mathbf{u}_t$ denotes the changes over time. Given a solution $\mathbf{u}$, we can compute the residual R, which naturally should be equal to zero for a correct solution:

$$R = \mathbf{u}_t - \mathcal{F}(\mathbf{u}_x, \mathbf{u}_{xx}, ...\mathbf{u}_{xx...x}) = 0.$$

In this context, we can approximate the unknown $\mathbf{u}$ itself with a neural network. If the approximation is accurate, the PDE residual should likewise be zero.

This nicely integrates with the objective for training a neural network: we can train for minimizing this residual in combination with direct loss terms. In addition to relying on the residual, we can use pre-computed solutions $[x_0, y_0], ...[x_n, y_n]$ for $\mathbf{u}$ with $\mathbf{u}(\mathbf{x}) = y$ as targets. This is typically important, as most practical PDEs do not have unique solutions unless initial and boundary conditions are specified. Hence, if we only consider R we might get solutions with random offset or other undesirable components. The supervised sample points therefore help to *pin down* the solution in certain places. Now our training objective becomes

$$\arg\min_\theta \sum_i \alpha_0 \left(f(x_i;\theta) - y_i^*\right)^2 + \alpha_1 R(x_i), \tag{8.1}$$

where $\alpha_{0,1}$ denote hyperparameters that scale the contribution of the supervised term and the residual term, respectively. We could of course add additional residual terms with suitable scaling factors here.

It is instructive to note what the two different terms in equation (8.1) mean: The first term is a conventional, supervised L2-loss. If we were to optimize only this loss, our network would learn to approximate the training samples well, but might

average multiple modes in the solutions, and do poorly in regions in between the sample points. If we, instead, were to optimize only the second term (the physical residual), our neural network might be able to locally satisfy the PDE, but could have large difficulties find a solution that fits globally. This can happen due to "null spaces" in the solutions, i.e., different solutions that all satisfy the residuals. Then local points can converge to different solutions, in combination yielding a very suboptimal one. Therefore, we optimize both objectives simultaneously such that, in the best case, the network learns to approximate the specific solutions of the training data while still capturing knowledge about the underlying PDE.

Note that, similar to the data samples used for supervised training, we have no guarantees that the residual terms R will actually reach zero during training. The non-linear optimization of the training process will minimize the supervised and residual terms as much as possible, but there is no guarantee. Large, non-zero residual contributions can remain. We'll look at this in more detail in the upcoming code example, for now it's important to keep in mind that the physical constraints formulated this way only represent *soft constraints*, without guarantees of minimizing these constraints.

The previous overview did not really make clear how an NN produces $\mathbf{u}$. We can distinguish two different approaches here: via a chosen explicit representation of the target function (v1 in the following), or with a *Neural field* based on fully-connected neural networks to represent the solution (v2). E.g., for v1 we could set up a *spatial* grid (or graph, or a set of sample points), while in the second case no explicit representation exists, and the NN instead receives the *spatial coordinate* to produce the solution at a query position. We'll outline these two variants in more detail the following.

8.2 Variant 1: Residual derivatives for explicit representations

For variant 1, we choose the discretization and set up a computational mesh that covers our target domain. Without loss of generality, let's assume this is a Cartesian grid that samples the space with positions $\mathbf{p}$. Now, an NN is trained to produce the solution on the grid: $\mathbf{u}(\mathbf{p}) = f(\mathbf{x};\theta)$. For a regular grid, a CNN would be a good choice for f, while for triangle meshes we could use a graph-network, or a network with point-convolutions for particles.

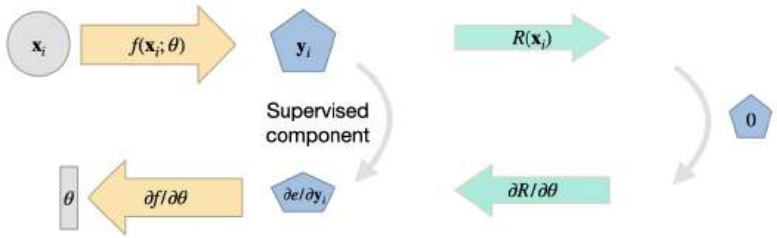


Fig. 8.1: Variant 1: the solution is represented by a chosen computational mesh, and produced by an NN. The residual is discretized there, and can be combined with supervised terms.

Now, we can discretize the equations of R on our computational mesh, and compute derivatives with our method of choice. Only caveat: to incorporate the residual into training, we have to formulate the evaluation such that a deep learning framework can backpropagate through the calculations. As our network $f()$ produces the solution $\mathbf{u}$, and the residual depends on it ($R(\mathbf{u})$), we at least need $\partial R/\partial \mathbf{u}$, such that the gradient can be backpropagated for the weights θ. Luckily, if we formulate R in terms of operations of a DL framework, this will be taken care of by the backpropagation functionality of the framework.

This variant has a fairly long "tradition" in DL, and was, e.g., proposed by Tompson et al. [TSSP17] early on to learn divergence free motions. To give a specific example: if our goal is to learn velocities $\mathbf{u}(t)$ which are divergence free

$\nabla \cdot \mathbf{u} = 0$, we can employ this training approach to train an NN without having to pre-compute divergence free velocity fields as training data. For brevity, we will drop the spatial index ($\mathbf{p}$) here, and focus on t, which we can likewise simplify: divergence-freeness has to hold at all times, and hence we can consider a single step from $t = 0$ with $\Delta t = 1$, i.e., a normalized step from a divergent $\mathbf{u}(0)$ to a divergence-free $\mathbf{u}(1)$. For a normal solver, we'd have to compute a pressure $p = \nabla^{-2}\mathbf{u}(0)$, such that $\mathbf{u}(1) = \mathbf{u}(0) - \nabla p$. This is the famous fundamental theorem of vector calculus, or Helmholtz decomposition, splitting a vector field into a *solenoidal* (divergence-free) and irrotational part (the pressure gradient).

To learn this decomposition, we can approximate p with a CNN on our computational mesh: $p = f(\mathbf{u}(0); \theta)$. The learning objective becomes minimizing the divergence of $\mathbf{u}(0)$, which means minimizing $\nabla \cdot (\mathbf{u}(0) - \nabla f(\mathbf{u}(0); \theta))$. To implement this residual, all we need to do is provide the divergence operator $(\nabla \cdot)$ of $\mathbf{u}$ on our computational mesh. This is typically easy to do via a convolutional layer in the DL framework that contains the finite difference weights for the divergence. Nicely enough, in this case we don't even need additional supervised samples, and can typically purely train with this residual formulation. Also, in contrast to variant 2 below, we can directly handle fairly large spaces of solutions here (we're not restricted to learning single solutions). An example implementation can be found in this code repository.

Overall, this variant 1 has a lot in common with *differentiable physics* training (it's basically a subset) that will be covered with a lot more detail in *Introduction to Differentiable Physics*. Hence, we'll focus a bit more on direct NN representations (variant 2) in this chapter.

8.3 Variant 2: Derivatives from a neural network representation

The second variant of employing physical residuals as soft constraints instead uses fully connected NNs to represent $\mathbf{u}$. This *physics-informed* (PINN) approach was popularized by Raissi et al. [RPK19], and has some interesting pros and cons that we'll outline in the following. By now, this approach can be seen as part of the *Neural field* representations that e.g. also include NeRFs and learned signed distance functions.

The central idea with Neural fields is that the aforementioned general function f that we're after can also be used to obtain a representation of a physical field, e.g., a field $\mathbf{u}$ that satisfies $R = 0$. This means $\mathbf{u}(\mathbf{x})$ will be turned into $\mathbf{u}(\mathbf{x}, \theta)$ where we choose the NN parameters θ such that a desired $\mathbf{u}$ is represented as precisely as possible, and $\mathbf{u}$ simply returns the right value at spatial location $\mathbf{x}$.

One nice side effect of this viewpoint is that NN representations inherently support the calculation of derivatives w.r.t. inputs. The derivative $\partial f / \partial \theta$ was a key building block for learning via gradient descent, as explained in *Overview*. Now, we can use the same tools to compute spatial derivatives such as $\partial \mathbf{u} / \partial x = \partial f / \partial x$, Note that above for R we've written this derivative in the shortened notation as $\mathbf{u}_x$. For functions over time this of course also works by adding t as input to compute $\partial \mathbf{u} / \partial t$, i.e. $\mathbf{u}_t$ in the notation above.

Thus, for some generic R, made up of $\mathbf{u}_t$ and $\mathbf{u}_x$ terms, we can rely on the backpropagation algorithm of DL frameworks to compute these derivatives once we have a NN that represents $\mathbf{u}$. Essentially, this gives us a function (the NN) that receives space and time coordinates to produce a solution for $\mathbf{u}$. Hence, the input is typically quite low-dimensional, e.g., 3+1 values for a 3D case over time, and often produces a scalar value or a spatial vector. Due to the lack of explicit spatial sampling points, an MLP, i.e., fully-connected NN is the architecture of choice here.

To pick a simple example, Burgers equation in 1D, $\frac{\partial u}{\partial t} + u \nabla u = \nu \nabla \cdot \nabla u$, we can directly formulate a loss term $R = \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \nu \frac{\partial^2 u}{\partial x^2}$ that should be minimized as much as possible at training time. For each of the terms, e.g. $\frac{\partial u}{\partial x}$, we can simply query the DL framework that realizes u to obtain the corresponding derivative. For higher order derivatives, such as $\frac{\partial^2 u}{\partial x^2}$, we can query the derivative function of the framework multiple times. In the following section, we'll give a specific example of how that works in tensorflow.

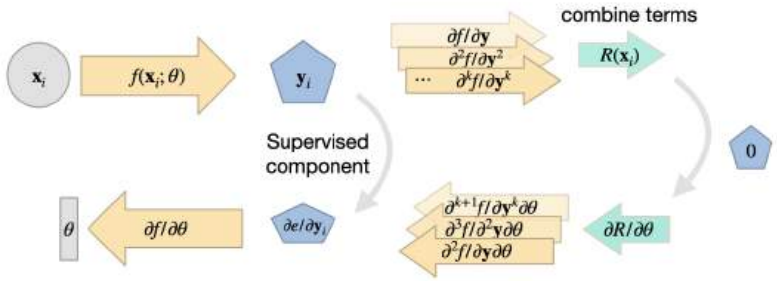


Fig. 8.2: Variant 2: the solution is produced by a fully-connected network, typically requiring a supervised loss with a combination of derivatives from the neural network for the residual. These NN derivatives have their own share of advantages and disadvantages.

8.4 Summary so far

The approach above gives us a method to include physical equations into DL learning as a soft constraint: the residual loss. While v1 relies on an inductive bias in the form of a discretization, v2 relies on derivatives computed by via autodiff. Typically, v2 is especially suitable for *inverse problems*, where we have certain measurements or observations for which we want to find a PDE solution. Because of the ill-posedness of the optimization and learning problem, and the high cost of the reconstruction (to be demonstrated in the following), the solution manifold shouldn't be overly complex for these PINN approaches. E.g., it is typically very difficult to capture time dependence or a wide range of solutions, such as with the previous supervised airfoil example.

Next, we'll demonstrate these concepts with code: first, we'll show how learning the Helmholtz decomposition works out in practice with a **v1**-approach. Afterwards, we'll illustrate the **v2** PINN-approaches with a practical example.

CHAPTER

NINE

LEARNING THE HELMHOLTZ-HODGE DECOMPOSITION

In the following notebook we'll following the aforementioned paper by Tompson et al. [TSSP17] and train a neural network that to perform a Helmholtz-Hodge decomposition. This is a very classic and time consuming part of many numerical solvers, and enables splitting an arbitrary vector field into a solenoidal (divergence-free) and irrotational part (the pressure gradient). Because this is traditionally very time consuming, it's an interesting goal for a learned approach. As a stepping stone towards integrating full solvers, we'll formulate a physics-based loss via a discretized PDE-constraint (the approach denoted as physical loss training *v1* in the previous section).

9.1 Solving Navier-Stokes

To motivate the topic, let's briefly revisit how the incompressible Navier-Stokes equations are often solved: a fundamental variant employs operator splitting to separately solve advection and pressure correction (giving first-order accuracy in time). Thus, given an arbitrary flow field $\mathbf{u}$, we compute a self advection step solving $\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = 0$. This step is usually efficient, with a complexity linear in the number of discretization points, and gives an advected flow field $\tilde{\mathbf{u}}$.

The pressure-projection (or *Chorin* projection) step using the aforementioned Helmholtz-Hodge decomposition requires solving a Poisson problem $\nabla^2 p = \nabla \cdot \tilde{\mathbf{u}}$, an elliptic PDE that is non-trivial to solve in the general case. In the Navier-Stokes setting it gives an instantaneous scalar pressure field p, the gradient of which happens to capture the divergent parts of the velocity field. Hence, once we subtract ∇p we obtain a divergence free field that satisfies $\nabla \cdot \mathbf{u} = 0$.

Our goal below will be to obtain p for new velocity inputs $\mathbf{u}$, and the neat property of this problem is that we can perform this training without a supervised setup. I.e., we don't have to precompute a large number of $(\mathbf{u}, p)$ pairs to train this, but rather we'll rely on the discretized PDE-constraint. This could be called an *unsupervised* training, but this labeling is misleading, as the training target in all "unsupervised" cases is simply computed on the fly.

The more important conceptual aspect of the approach is that we have a target PDE $\nabla \cdot \mathbf{u} = 0$, that we can discretize on our computational domain with a suitable finite-difference operator. Once we do this in a differentiable way, we can directly use the FD operator to train our neural network: as we aim for minimizing $\nabla \cdot \tilde{\tilde{\mathbf{u}}}$, this effectively gives us an L_2 loss with a target of 0. From this loss, we can back-propagate through the discretized divergence operator to obtain a gradient for $\mathbf{u}$, which is computed (as outlined above), as $\tilde{\tilde{\mathbf{u}}} = \tilde{\mathbf{u}} - \nabla p$, where p is the output of our neural network f with parameters θ. As we know that the divergence is all that is required to uniquely determine the pressure field, we can pass the divergence to f to simplify the inference task. Thus $f(\nabla \cdot \tilde{\mathbf{u}}; \theta)$ will receive a gradient backpropagated from the loss through all steps to update its state θ such that the velocity field above will end up divergence free.

Putting these steps together, we aim for solving the minimization problem \$arg $\min_\theta \sum_i \left(\nabla \cdot (\tilde{\mathbf{u}} - \nabla f(\nabla \cdot \tilde{\mathbf{u}}; \theta)) \right)^2$,$where the i$\$ subscript denotes an arbitrary number of different inputs, e.g., from a minibatch at training time.

9.2 Setting up the Discrete PDE

As before, we'll use the `PhiFlow` framework. The cell below installs it via `pip`, and imports the PyTorch backend together with `tqdm` for tracking the progress. Note that this notebook can be switched to TensorFlow quite easily by changing the import below.

```
!pip install --quiet phiflow==3.3 tqdm
from tqdm import tqdm
from phiml import nn

from phi.torch.flow import *
# this notebook largely works identically with tensorflow - try replacing the line␣
 ↪above with this one
#from phi.tf.flow import *
```

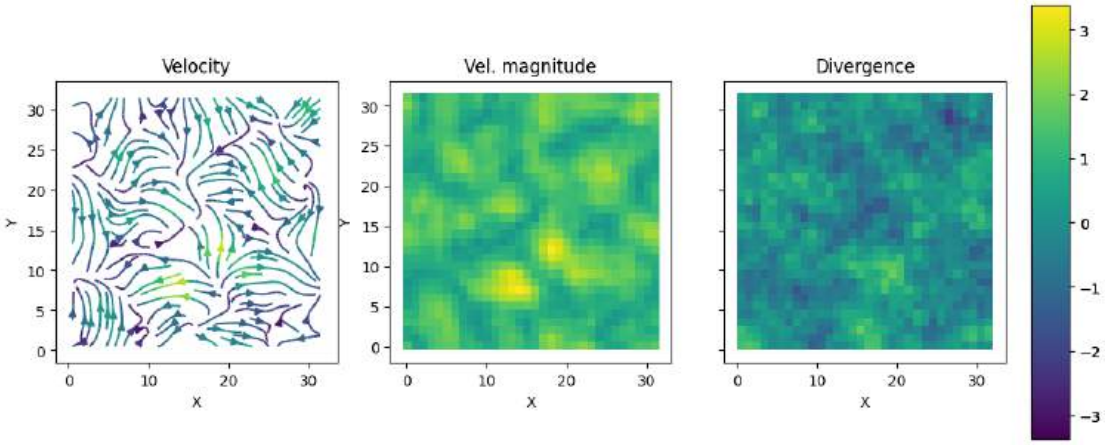
Let's define some basic constants, the grid resolution (which we'll reuse as *physical* dimensions in PhiFlow), the number of different flow fields we'll consider for training, and the batch size.

```
RES_X = 32
RES_Y = 32

NUM_SAMPLES_TRAIN = 100
BATCH_SIZE = 10
```

We will use random, periodic flow fields generated via PhiFlow's frequency-based synthesis. The next cell will pre-compute all these flow fields (100 in total), and visualize an example. Note, we're simply taking random, divergent flow fields here. We don't know what the correct divergence-free counterparts are.

```
vel = field.StaggeredGrid(
    values=field.Noise(batch(batch=NUM_SAMPLES_TRAIN)),
    extrapolation=math.extrapolation.PERIODIC,
    bounds=geom.Box(x=RES_X, y=RES_Y),
    resolution=math.spatial(x=RES_X, y=RES_Y),
)

# plot the velocity field
plot = vis.plot({ "velocity": vel.batch[0], "vel. magnitude": math.vec_length(vel.
 ↪batch[0].at_centers().values), "divergence": field.divergence(vel.batch[0]), })

# for initial assessments, let's print the typical divergence of an input
print( f"Initial L2 of divergence: {math.l2_loss(field.divergence(vel.batch[0]))},␣
 ↪max={field.divergence(vel.batch[0]).values.max}" )
```

```
Initial L2 of divergence: 195.28241, max=1.8969224691390991
```



Here you can see the flowlines together with velocity magnitudes and the divergence per cell. The latter is exactly what we're want to remove. This visualization shows that the divergence is smaller than the actual magnitude of the velocities, with an average of around 0.4, as indicated by the L2 output right above the images.

Next, we will define a Navier-Stokes simulation step. Given our reduced setup without external forces or obstacles, it's very simple: a call to an advection function in PhiFlow, followed by `fluid.make_incompressible()` to invoke the Poisson solver. Here we need to pass a custom solver to treat the rank deficiency in the periodic solve (it's ambiguous with respect to constant offsets). This is not neceessary later on for situations with a unique pressure solution. We'll also directly annotate this function and the following ones for JIT compilation with `@jit_compile`. This is important for good performance on GPUs, but it makes debugging much harder. So when changing the code, it's highly recommended to remove them. The code will work just as well without, just slower. Once everything's running as it should, re-activate JIT compilation for the *real* training runs.

```
@jit_compile
def step(v, dt = 1.0):
    v = advect.mac_cormack(v, v, dt)
    v, p = fluid.make_incompressible(v, [], solve=Solve(rank_deficiency=0))
    return v, p

v,p = step(vel)
```

```
 warnings.warn("Possible rank deficiency detected. Matrix might be singular which␣
↪can lead to convergence problems. ")
```

This cell directly runs the solver on our random velocity field from before. If you're curious, you can plot it via `vis.plot()`. However, as we've demonstrated in earlier chapters that PhiFlow's NS solver works, we'll focus on the NN training. Our goal below is to train the network without having to precompute many pressure examples with a function like `make_incompressible()` above. Instead, we'll use the differentiable divergence operator to set up our training.

9.3 Neural Network Training

Were facing an elliptic PDE problem here, and hence a NN architecture with global communication is important, cf. *Neural Network Architectures*. Below, we initialize a U-Net, but feel free to try the ResNet variant, which works less well due to its local receptive field. (Given that property, it's doing surprisingly well.) As we're dealing with a periodic domain for simplicity, the NN likewise needs to be configured for periodic processing via `periodic=True`. It's input is a single channel (the divergence), and the output a very different content, the pressure. However, for the network this is likewise simply a single, scalar channel. The `filters=24` determine the total number of parameters. Feel free to increase this to improve accuracy (and reduce computational performance of the NN inference). This is the classic accuracy vs performance trade-off that NNs share with all classic numerical methods.

```
from phiml import nn

network = nn.u_net(in_channels=1, out_channels=1, levels=4, periodic=True, batch_
 ↪norm=False, in_spatial=(RES_X, RES_Y), filters=24)

# uncomment this for an alternative architecture that could generalize to different␣
 ↪resolutions
#network = nn.res_net(in_channels=1,out_channels=1,layers=[32,32,32])

# print network and parameter summary
from phiml.backend import BACKENDS
if any([b.name == 'torch' for b in BACKENDS]):
    print(network)
    print("Total number of trainable parameters: "+ str( sum(p.numel() for p in␣
 ↪network.parameters()) ))
elif any([b.name == 'tf' for b in BACKENDS]):
    network.summary()
```

```
UNet(
  (inc): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(1, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
 ↪padding_mode=circular)
      (1): Identity()
      (2): ReLU()
      (3): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
 ↪padding_mode=circular)
      (4): Identity()
      (5): ReLU(inplace=True)
    )
  )
  (down1): Down(
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
 ↪mode=False)
    (conv): ResNetBlock(
      (sample_input): Identity()
      (bn_sample): Identity()
      (bn1): Identity()
      (conv1): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): Identity()
      (conv2): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
  )
  (up1): Up(
    (up): Upsample(scale_factor=2.0, mode='bilinear')
```

(continues on next page)

(continued from previous page)

```
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(48, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪padding_mode=circular)
        (1): Identity()
        (2): ReLU()
        (3): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪padding_mode=circular)
        (4): Identity()
        (5): ReLU(inplace=True)
      )
    )
  )
  (down2): Down(
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    (conv): ResNetBlock(
      (sample_input): Identity()
      (bn_sample): Identity()
      (bn1): Identity()
      (conv1): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): Identity()
      (conv2): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
  )
  (up2): Up(
    (up): Upsample(scale_factor=2.0, mode='bilinear')
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(48, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪padding_mode=circular)
        (1): Identity()
        (2): ReLU()
        (3): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪padding_mode=circular)
        (4): Identity()
        (5): ReLU(inplace=True)
      )
    )
  )
  (down3): Down(
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    (conv): ResNetBlock(
      (sample_input): Identity()
      (bn_sample): Identity()
      (bn1): Identity()
      (conv1): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): Identity()
      (conv2): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
  )
  (up3): Up(
    (up): Upsample(scale_factor=2.0, mode='bilinear')
    (conv): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(48, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
```

(continues on next page)

(continued from previous page)

```
 ↪padding_mode=circular)
        (1): Identity()
        (2): ReLU()
        (3): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
 ↪padding_mode=circular)
        (4): Identity()
        (5): ReLU(inplace=True)
      )
    )
  )
  (outc): Conv2d(24, 1, kernel_size=(1, 1), stride=(1, 1))
)
Total number of trainable parameters: 83521
```

As can be seen here, the NN has around 83k trainable parameters.

Next we can setup the training process: the `eval_nn()` function below takes a batch of flow fields, computes their divergence, and calls the NN with these inputs. The resulting scalar field is converted to a PhiFlow tensor and interpreted as a pressure field. Its gradient is subtracted from the input flow following Helmholtz-Hodge. Our goal is to make the resulting flow field `v` incompressible, i.e., it's divergence should be minimal.

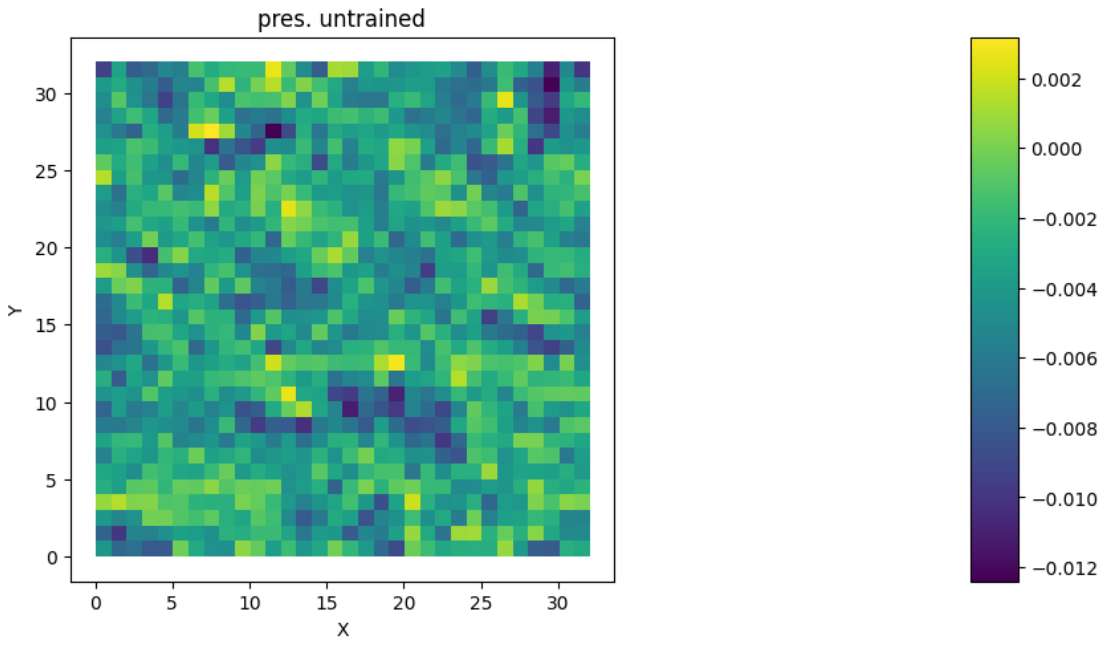
To achieve this goal, we'll define a loss function that computes the L^2 of the per-cell divergence `loss_div()`. For convenience, we'll also define a helper functions `loss_func` that will be used by PhiFlow. This function evaluates the NN for an input, and computes the resulting loss. PhiFlow will call this function and then make sure the gradient backpropagates from the single loss value through the discrete divergence operator to the weights of the NN. Note that in the incompressible setting we always remove the divergence present in the flow, and hence the time step does not play a role.

```
@jit_compile
def eval_nn(v):
    nn_input = field.divergence(field=v, order=2)
    p = math.native_call(network, nn_input.values)
    p = field.CenteredGrid(
        values=p,
        extrapolation=math.extrapolation.PERIODIC,
        bounds=geom.Box(x=RES_X, y=RES_Y),
        resolution=spatial(x=RES_X, y=RES_Y),
    )
    grad_pres = field.spatial_gradient(p, boundary=math.extrapolation.PERIODIC, at=v.
 ↪sampled_at)
    v = v - grad_pres
    return v, p

# loss functions

@jit_compile
def loss_div(v):
    div = field.divergence(field=v,order=2)
    div_sum = 2 * math.l2_loss(div)
    return div_sum, div

@jit_compile
def loss_func(v):
    v, p = eval_nn(v)
    loss, div = loss_div(v)
    return math.mean(loss,dim=batch), v, p
```

Before training the network, let's look at what a randomly initialized network produces. Below we evaluate the untrained network on the first flow field input:

```
vel_untrained, pres_untrained = eval_nn(vel.batch[0])

# optionally, visualize the outputs: this doesn't look much different from before as␣
↪the NN is untrained
#plot = vis.plot( {"vel untrained": vel_untrained, "vel len. untrained": math.vec_
↪length(vel_untrained.at_centers().values), "div. untrained": field.divergence(vel_
↪untrained), })

# print the loss and divergence sum of the corrected velocity from untrained NN
loss, div_untrained = loss_div(vel_untrained)
print(f"Loss for untrained network: {loss}")

# also, we visualize the pressure field
plot = vis.plot(pres_untrained,title="pres. untrained")
```

```
Loss for untrained network: 390.21857
```



Not surprisingly, the pressure field looks quite random, and the loss in terms of divergence is large.

9.4 Training

With our setup so far, training is very simple: we simply choose a few of the random flow fields, and evaluate the network, and change its weights so that the divergence loss becomes smaller. We simply loop over our precomputed flow fields without randomization below for a fixed number of epochs. Our network converges surprisingly fast. After a few epochs the loss should have decreased by more than two orders of magnitude.

```
optimizer = nn.adam(network, learning_rate=5e-3)
for epoch in tqdm(range(15)):
    for b in range(int(NUM_SAMPLES_TRAIN / BATCH_SIZE)):
        vel_input = vel.batch[b*BATCH_SIZE:b*BATCH_SIZE + BATCH_SIZE]
        loss, pred_v, pred_p = nn.update_weights(
            network, optimizer, loss_func, vel_input
        )

loss, _,_ = loss_func(vel)
print(f"Final loss={loss.numpy()}")
```

```
  0%|          | 0/15 [00:00<?, ?it/s]
100%|██████████| 15/15 [00:16<00:00,  1.12s/it]
```

```
Final loss=0.6785882711410522
```

It's good to see the loss going down, but of course the big question now is: how does this network fare with new inputs. I.e., how well does it generalize to different, arbitrary flows given that it was only trained on the synthetic, randomly sampled flows.

9.5 Testing with New Inputs

We can check this by producing a few new inputs. Below, we'll likewise use PhiFlow's noise generation to get new fields, but to make things interesting we're increasing the scale by a factor of $2\times$. Hence, the network will receive divergence inputs with magnitudes it hasn't seen before. These samples are effectively *out of the distribution* of the training inputs.

```
NUM_SAMPLES_TEST = 10

vel_test = field.StaggeredGrid(
    values=field.Noise(math.batch(batch=NUM_SAMPLES_TEST),scale=2.),
    extrapolation=math.extrapolation.PERIODIC,
    bounds=geom.Box(x=RES_X, y=RES_Y),
    resolution=math.spatial(x=RES_X, y=RES_Y),
)
```

Now we run our trained network and the solver (for comparison) on these inputs:

```
vel_nn, p_nn          = eval_nn(vel_test)
vel_solver, p_solver  = fluid.make_incompressible(vel_test, [])

loss, div = loss_div(vel_test)
print(f"Original, mean divergence={loss.mean:.3f}, div. max={math.max(div.values).
 ↪mean:.3f}")

loss, div = loss_div(vel_nn)
print(f"NN, mean divergence={loss.mean:.3f}, div. max={math.max(div.values).mean:.3f}
```

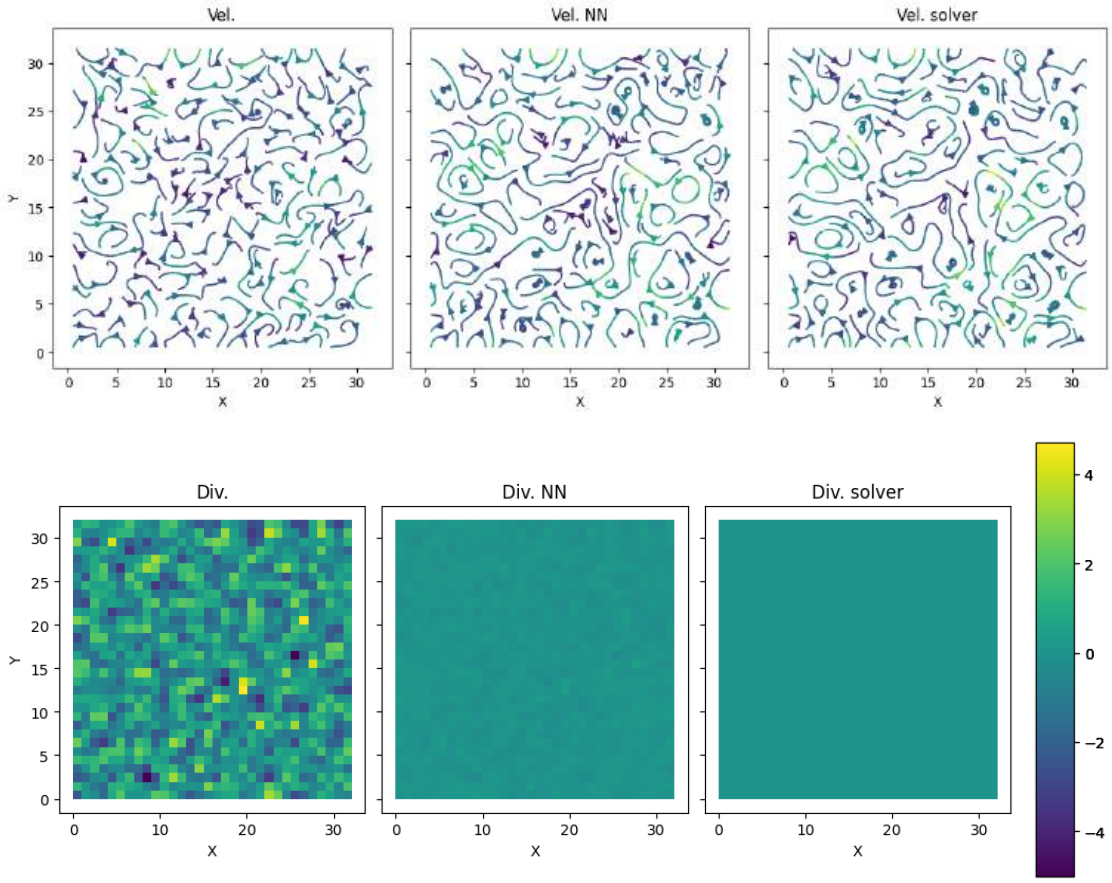
(continues on next page)

(continued from previous page)

```
 ↪")

loss, div = loss_div(vel_solver)
print(f"Solver, mean divergence={loss.mean:.3f}, div. max={math.max(div.values).mean:.
 ↪3f}")

plot = vis.plot({ "vel.": vel_test.batch[0],  "vel. NN": vel_nn.batch[0],  "vel.␣
 ↪solver": vel_solver.batch[0] })
plot = vis.plot({ "div.": field.divergence(vel_test.batch[0]), "div. NN": field.
 ↪divergence(vel_nn.batch[0]), "div. solver": field.divergence(vel_solver.batch[0]), }
 ↪)
```

```
anaconda3/envs/torch24/lib/python3.12/site-packages/phiml/math/_optimize.py:631:␣
 ↪UserWarning: Possible rank deficiency detected. Matrix might be singular which␣
 ↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
 ↪).
  warnings.warn("Possible rank deficiency detected. Matrix might be singular which␣
 ↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
 ↪).")
```

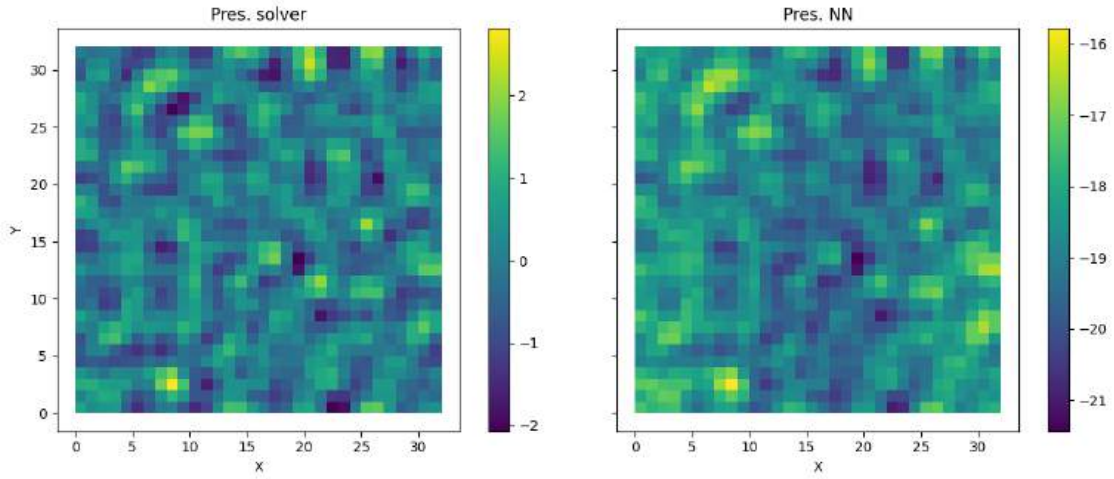
```
Original, mean divergence=2059.143, div. max=4.946
NN, mean divergence=11.510, div. max=0.417
Solver, mean divergence=0.000, div. max=0.000
```



The quantitative evaluation confirms that our network has learned to reduce the divergence. The remaining one is larger than the one from PhiFlow's solver, but nonetheless orders of magnitude smaller than the original one. The images at the bottom nicely visualize this.

Next, let's check what pressure field the network has actually learned to produce.

```
# plot the pressure on the training data, compared with the pressure given by␣
 ↪traditional solver.
plot = vis.plot({"pres. solver": p_solver.batch[0], "pres. NN": p_nn.batch[0]}, same_
 ↪scale=False)
```



Interestingly, these fields share some similarities in terms of their structures, but are clearly not identical. Also, they typically have strongly varying magnitudes (the NN training is stochastic, so the differences can vary). Often, the NN learns to generate values that are 10 times larger, while PhiFlows pressure solvers is constrained to produce values with a zero mean.

Nonetheless, as measured above, the pressure field of the network successfully removes most of the divergence, and mathematically, the Poisson problem here has a null-space in the form of a constant offset: the offset does not play a role once we take the derivatives, and hence all offsets are perfectly valid solutions. Our differentiable loss used for training does not have any constraint on absolute values, and hence the network is free to use any offset it happens to produce from its random initialization.

9.6 Tougher Tests: Fluid Simulations with Obstacles

Our initial goal were Navier-Stokes solvers, and hence it's import to check whether our network can actually cope with inputs in the form of flow fields produced by a fluid solver. After all, it was only trained on the synthetic, randomized fields that are quite different from the motions of fluids. To make things more interesting, we'll directly include a buoyant smoke plume to drive the simulation, and an obstacle.

The next cell sets up a simple domain with a smoke source and a square obstacle right above it.

```
smoke = CenteredGrid(0, extrapolation.PERIODIC, x=RES_X, y=RES_Y, bounds=Box(x=RES_X,␣
 ↪y=RES_Y))
vel_smk = StaggeredGrid(0, extrapolation.PERIODIC, x=RES_X, y=RES_Y, bounds=Box(x=RES_
 ↪X, y=RES_Y))

RECTANGLE = Obstacle(Box(x=(12, 20), y=(18, 20)))
```

(continues on next page)

(continued from previous page)

```
INFLOW_LOCATION = tensor((16, 5), channel(vector='x,y'))
INFLOW = 0.6 * CenteredGrid(Sphere(center=INFLOW_LOCATION, radius=3), extrapolation.
 ↪PERIODIC, x=RES_X, y=RES_Y, bounds=Box(x=RES_X, y=RES_Y))
CYLINDER = Obstacle(geom.infinite_cylinder(x=16, y=20, radius=3, inf_dim=None))
```

In our NN-supported solver, we now have some options regarding how to impose the zero-velocity boundary conditions for the obstacle. The safest way to handle them is to set velocities both before and after invoking the NN-based pressure correction step. The former ensure the network sees the solid obstacle, and the second one ensures the boundaries are fulfilled even if the network has made slight errors at the obstacle boundary. Hence, we'll call PhiFlow's `apply_boundary_conditions()` two times in the `step_nn()` function below.

In parallel, we'll also define a `step_obs()` function that runs a classic flow solver without NN for comparison.

```
dt = 0.1

@jit_compile
def step_nn(v, dt, f, obstacle):
    v = v + f * dt
    v = advect.mac_cormack(v, v, dt)
    v = fluid.apply_boundary_conditions(v, obstacle)
    nn_input = field.divergence(field=v, order=2)
    p = math.native_call(network, nn_input.values)

    p = field.CenteredGrid( values=p,
        extrapolation=math.extrapolation.PERIODIC,
        bounds=geom.Box(x=RES_X, y=RES_Y),
        resolution=spatial(x=RES_X, y=RES_Y),
    )
    grad_pres = field.spatial_gradient(p, at=v.sampled_at)
    v = v - grad_pres
    v = fluid.apply_boundary_conditions(v, obstacle)
    return v, p

@jit_compile
def step_obs(v, dt, f, obstacle):
    v = v + f * dt
    v = advect.mac_cormack(v, v, dt)
    v, p = fluid.make_incompressible(v, obstacle, solve=Solve(rank_deficiency=0))
    return v, p
```

Now we're ready to run and compare the simulations. The code below does this for `STEPS` simulation steps, running both solvers one after the other. The advected marker densities are tracked in the `traj_smk_X` lists, and allow for an intuitive, qualitative check regarding the results. The marker density is both driving and following the flow velocity, and thus nicely highlights changes in the motion of the simulated fluid.

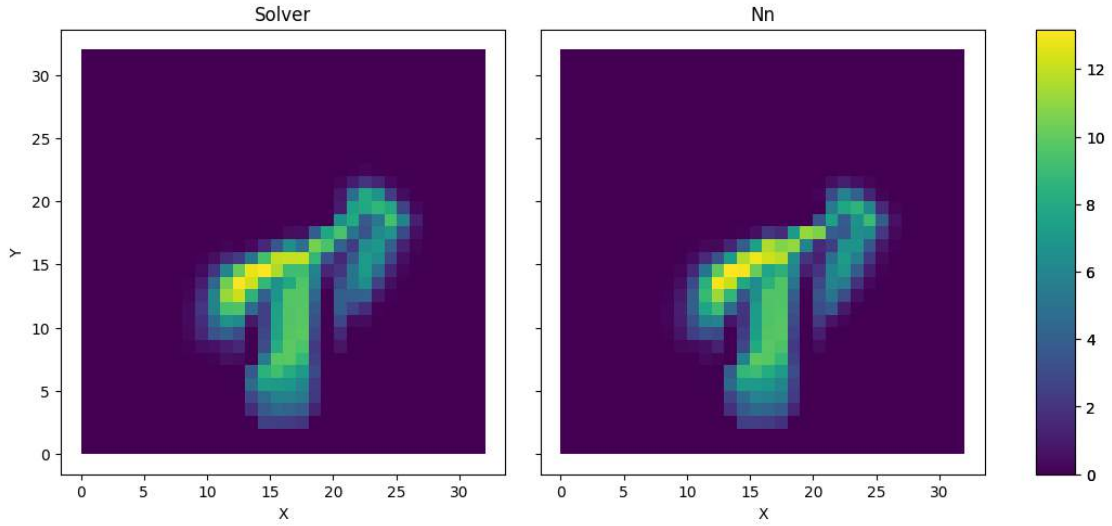
```
STEPS = 50
smk_solv = smoke
smk_nn = smoke
vel_smk_solv = vel_smk
vel_smk_nn = vel_smk
traj_smk_solv = [smk_solv]
traj_smk_nn = [smk_nn]

# flexible backends require a small workaround for PyTorch's no_grad context here; by␣
 ↪default (eg for TF) use a null-context
from contextlib import nullcontext
```

(continues on next page)

(continued from previous page)

```
context = nullcontext()
if any([b.name == 'torch' for b in BACKENDS]):
    context = torch.no_grad()

with context:
    for i in tqdm(range(STEPS)):
        smk_solv = advect.mac_cormack(smk_solv, vel_smk_solv, dt=dt) + INFLOW
        smk_nn = advect.mac_cormack(smk_nn, vel_smk_nn, dt=dt) + INFLOW

        buoyancy_force_solv = smk_solv * (0, 1.0) @ vel_smk_solv
        buoyancy_force_nn = smk_nn * (0, 1.0) @ vel_smk_nn

        vel_smk_solv, _ = step_obs(vel_smk_solv, dt, buoyancy_force_solv, [RECTANGLE])
        vel_smk_nn, _ = step_nn(vel_smk_nn, dt, buoyancy_force_nn, [RECTANGLE])

        traj_smk_solv.append(smk_solv)
        traj_smk_nn.append(smk_nn)

traj_smk_solv = field.stack(traj_smk_solv, batch('time'))
traj_smk_nn = field.stack(traj_smk_nn, batch('time'))

plot = vis.plot({"Solver": traj_smk_solv.time[STEPS-1], "NN": traj_smk_nn.time[STEPS-
↪1]}) # show last frame
```

```
100%|██████████| 50/50 [01:37<00:00,  1.96s/it]
```



The plot above shows the last frame of the simulations, and they're remarkably similar. This results shows that the NN has successfully learned to reduce the divergence of arbitrary flows, despite having only seen the synthetic noisy inputs at training time.

Moreover, it highlights the success of the differentiable discrete operator used for training. The training converges very quickly, and the trained network generalizes in a *zero-shot* fashion to completely new inputs. The network hasn't even seen any obstacles at training time, and can still infer (mostly correct) pressure fields to handle them. This aspect is nonetheless an interesting point for improvements of this implementation. The NN could receive the obstacle geometry as an additional input, and could be trained to pay special attention to the boundary region via increased loss values.

Lastly, if you're interested in watching the full evolution of the simulated trajectories, you can comment out and run

the following cell. It generates a movie that can be watched in the browser via PhiFlow's `plot(...,animate=X)` function.

```
#vis.plot({"solver": traj_smk_solv, "nn": traj_smk_nn}, animate='time') # uncomment␣
↪to view animation
```

CHAPTER

TEN

NEXT STEPS

- Improve the architecture of the neural network to boost its performance; obvious aspects to improve on are depth and features per layer, but other architectures are likewise worth a try.
- Especially fully convolutional architectures like the `res_net` above are worth a try: evaluate how well a trained NN with these architectures can handle new and larger input grids.
- Add support for obstacles with an additional input and an updated training data set.

CHAPTER

ELEVEN

BURGERS OPTIMIZATION WITH A PINN

To illustrate how the physics-informed losses work for variant 2, let's consider a reconstruction task as an inverse problem example. We'll use Burgers equation $\frac{\partial u}{\partial t} + u \nabla u = \nu \nabla \cdot \nabla u$ as a simple yet non-linear equation in 1D, for which we have a series of *observations* at time $t = 0.5$. The solution should fulfill the residual formulation for Burgers equation and match the observations. In addition, let's impose Dirichlet boundary conditions $u = 0$ at the sides of our computational domain, and define the solution in the time interval $t \in [0, 1]$.

Note that similar to the previous forward simulation example, we will still be sampling the solution with 128 points ($n = 128$), but now we have a discretization via the NN. So we could also sample points in between without having to explicitly choose a basis function for interpolation. The discretization via the NN now internally determines how to use its degrees of freedom to arrange the activation functions as basis functions. So we have no direct control over the reconstruction. [run in colab]

11.1 Formulation

In terms of the x, y^* notation from *Models and Equations* and the previous section, this reconstruction problem means we are solving

$$\arg\min_\theta \sum_i \left(f(x_i; \theta) - y_i^* \right)^2 + R(x_i),$$

where now x_i denotes a space-time point $x_i = [p_i, t_i]$, the reference solutions are $y_i^* = y^*(x_i)$, and the index i indicates different sampling points for our data set. Both f and y^* represent the solution of u at different locations in space and time, and as we're dealing with a 1D velocity, $f, y^* : \mathbb{R}^2 \to \mathbb{R}$. In this example, y^* denotes a reference u for $\mathcal{P}$ being Burgers equation, which f should approximate as closely as possible at all chosen space-time points $x_i = [p_i, t_i]$.

While the first term above is the "supervised" data term, the second one denotes the residual function R. It collects additional evaluations of $f(;\theta)$ and its derivatives to formulate the residual for $\mathcal{P}$. This approach – using derivatives of a neural network to compute a PDE residual – is typically called a *physics-informed* approach, yielding a *physics-informed neural network* (PINN) [RPK19] to represent a solution for the inverse reconstruction problem.

Thus, in the formulation above, R should simply converge to zero above. We've omitted scaling factors in the objective function for simplicity. Note that, effectively, we're only dealing with individual point samples of a single solution u for $\mathcal{P}$ here.

11.2 Preliminaries

This notebook is a bit older, and hence requires an older tensorflow version. The next cell installs/downgrades TF to a compatible version. This can lead to "errors" on colab due to pip dependencies, which you can safely ignore:

```
!pip3 install --upgrade --quiet tensorflow==2.15.0 tensorflow-probability==0.23.0
```

Next, we'll load phiflow (using a legacy version 1.5.1 from a custom repository) below. We'll use it with the tensorflow backend and initialize the random sampling.

```
!pip install --upgrade --quiet git+https://github.com/thunil/PhiFlow.git

from phi.tf.flow import *
import numpy as np

#rnd = TF_BACKEND  # for phiflow: sample different points in the domain each iteration
rnd = math.choose_backend(1)  # use same random points for all iterations
```

We're importing phiflow here, but we won't use it to compute a solution to the PDE as in *Simple Forward Simulation of Burgers Equation with phiflow*. Instead, we'll use the derivatives of an NN (as explained in the previous section) to set up a loss formulation for training.

Next, we set up a simple NN with 8 fully connected layers and `tanh` activations with 20 units each.

We'll also define the `boundary_tx` function which gives an array of constraints for the solution (all for $t = 0.5$ in this example), and the `open_boundary` function which stores constraints for $x = \pm 1$ being 0.

```
def network(x, t):
    """ Dense neural network with 8 hidden layers and 3021 parameters in total.
        Parameters will only be allocated once (auto reuse).
    """
    y = math.stack([x, t], axis=-1)
    for i in range(8):
        y = tf.layers.dense(y, 20, activation=tf.math.tanh, name='layer%d' % i,␣
 ↪reuse=tf.AUTO_REUSE)
    return tf.layers.dense(y, 1, activation=None, name='layer_out', reuse=tf.AUTO_
 ↪REUSE)

def boundary_tx(N):
    x = np.linspace(-1,1,128)
    # precomputed solution from forward simulation:
    u = np.asarray( [0.008612174447657694, 0.02584669669548606, 0.043136357266407785,␣
 ↪0.060491074685516746, 0.07793926183951633, 0.0954779141740818, 0.11311894389663882,␣
 ↪0.1308497114054023, 0.14867023658641343, 0.1665634396808965, 0.18452263429574314, 0.
 ↪20253084411376132, 0.22057828799835133, 0.23865132431365316, 0.25673879161339097, 0.
 ↪27483167307082423, 0.2929182325574904, 0.3109944766354339, 0.3290477753208284, 0.
 ↪34707880794585116, 0.36507311960102307, 0.38303584302507954, 0.40094962955534186, 0.
 ↪4188235294008765, 0.4366357052408043, 0.45439856841363885, 0.4720845505219581, 0.
 ↪4897081943759776, 0.5072391070000235, 0.5247011051514834, 0.542067187709797, 0.
 ↪5593576751669057, 0.5765465453632126, 0.5936507311857876, 0.6106452944663003, 0.
 ↪6275435911624945, 0.6443221318186165, 0.6609900633731869, 0.67752574922899, 0.
 ↪6939334022562877, 0.7101938106059631, 0.7263049537163667, 0.7422506131457406, 0.
 ↪7580207366534812, 0.7736033721649875, 0.7889776974379873, 0.8041371279965555, 0.
 ↪8190465276590387, 0.8337064887158392, 0.8480617965162781, 0.8621229412131242, 0.
 ↪8758057344502199, 0.8891341984763013, 0.9019806505391214, 0.9143881632159129, 0.
 ↪9261597966464793, 0.9373647624856912, 0.9476871303793314, 0.9572273019669029, 0.
 ↪9654367940878237, 0.9724097482283165, 0.9767381835635638, 0.9669484658390122, 0.
```

(continues on next page)

(continued from previous page)

```
↪659083299684951, -0.659083180712816, -0.9669485121167052, -0.9767382069792288, -0.
↪9724097635533602, -0.9654367970450167, -0.9572273263645859, -0.9476871280825523, -0.
↪9373647681120841, -0.9261598056102645, -0.9143881718456056, -0.9019807055316369, -0.
↪8891341634240081, -0.8758057205293912, -0.8621229450911845, -0.8480618138204272, -0.
↪833706571569058, -0.8190466131476127, -0.8041372124868691, -0.7889777195422356, -0.
↪7736033858767385, -0.758020740007683, -0.7422507481169578, -0.7263049162371344, -0.
↪7101938950789042, -0.6939334061553678, -0.677525822052029, -0.6609901538934517, -0.
↪6443222327338847, -0.6275436932970322, -0.6106454472814152, -0.5936507836778451, -0.
↪5765466491708988, -0.5593578078967361, -0.5420672759411125, -0.5247011730988912, -0.
↪5072391580614087, -0.4897082914472909, -0.47208460952428394, -0.4543985995006753, -
↪0.4366355580500639, -0.41882350871539187, -0.40094955631843376, -0.
↪38303594105786365, -0.36507302109186685, -0.3470786936847069, -0.3290476440540586, -
↪0.31099441589505206, -0.2929180880304103, -0.27483158663081614, -0.2567388003912687,
↪ -0.2386513127155433, -0.22057831776499126, -0.20253089403524566, -0.
↪18452269630486776, -0.1665634500729787, -0.14867027528284874, -0.13084990929476334,␣
↪-0.1131191325854089, -0.09547794429803691, -0.07793928430794522, -0.
↪06049114408297565, -0.0431364527809777, -0.025846763281087953, -0.
↪00861212501518312] );
    t = np.asarray(rnd.ones_like(x)) * 0.5
    perm = np.random.permutation(128)
    return (x[perm])[0:N], (t[perm])[0:N], (u[perm])[0:N]

def _ALT_t0(N): # alternative, impose original initial state at t=0
    x = rnd.random_uniform([N], -1, 1)
    t = rnd.zeros_like(x)
    u = - math.sin(np.pi * x)
    return x, t, u

def open_boundary(N):
    t = rnd.random_uniform([N], 0, 1)
    x = math.concat([math.zeros([N//2]) + 1, math.zeros([N//2]) - 1], axis=0)
    u = math.zeros([N])
    return x, t, u
```

Most importantly, we can now also construct the residual loss function `f` that we'd like to minimize in order to guide the NN to retrieve a solution for our model equation. As can be seen in the equation at the top, we need derivatives w.r.t. t, x and a second derivative for x. The first three lines of `f` below do just that.

Afterwards, we simply combine the derivates to form Burgers equation. Here we make use of phiflow's `gradient` function:

```
def f(u, x, t):
    """ Physics-based loss function with Burgers equation """
    u_t = gradients(u, t)
    u_x = gradients(u, x)
    u_xx = gradients(u_x, x)
    return u_t + u*u_x - (0.01 / np.pi) * u_xx
```

Next, let's set up the sampling points in the inner domain, such that we can compare the solution with the previous forward simulation in phiflow.

The next cell allocates two tensors: `grid_x` will cover the size of our domain, i.e., the -1 to 1 range, with 128 cells, while `grid_t` will sample the time interval $[0, 1]$ with 33 time stamps.

The last `math.expand_dims()` call simply adds another `batch` dimension, so that the resulting tensor is compatible with the following examples.

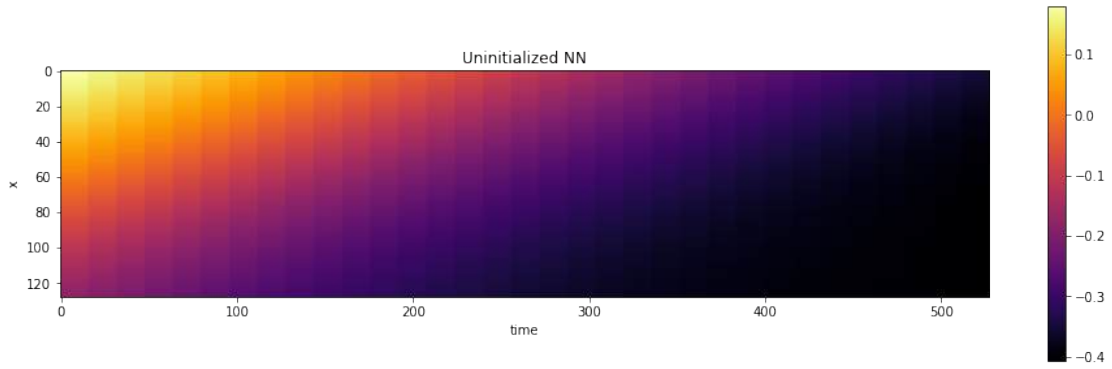
```
N=128
grids_xt = np.meshgrid(np.linspace(-1, 1, N), np.linspace(0, 1, 33), indexing='ij')
grid_x, grid_t = [tf.convert_to_tensor(t, tf.float32) for t in grids_xt]

# create 4D tensor with batch and channel dimensions in addition to space and time
# in this case gives shape=(1, N, 33, 1)
grid_u = math.expand_dims(network(grid_x, grid_t))
```

Now, `grid_u` contains a full graph to evaluate our NN at 128×33 positions, and returns the results in a $[1, 128, 33, 1]$ array once we run it through `session.run`. Let's give this a try: we can initialize a TF session, evaluate `grid_u` and show it in an image, just like the phiflow solution we computed previously.

(Note, we'll use the `show_state` as in *Simple Forward Simulation of Burgers Equation with phiflow*. Hence, the x axis does not show actual simulation time, but is showing 32 steps "blown" up by a factor of 16 to make the changes over time easier to see in the image.)

```
import pylab as plt
print("Size of grid_u: "+format(grid_u.shape))

session = Session(None)
session.initialize_variables()

def show_state(a, title):
    for i in range(4): a = np.concatenate( [a,a] , axis=3)
    a = np.reshape( a, [a.shape[1],a.shape[2]*a.shape[3]] )
    fig, axes = plt.subplots(1, 1, figsize=(16, 5))
    im = axes.imshow(a, origin='upper', cmap='inferno')
    plt.colorbar(im) ; plt.xlabel('time'); plt.ylabel('x'); plt.title(title)

print("Randomly initialized network state:")
show_state(session.run(grid_u),"Uninitialized NN")
```

```
Size of grid_u: (1, 128, 33, 1)
Randomly initialized network state:
```



This visualization already shows a smooth transition over space and time. So far, this is purely the random initialization of the NN that we're sampling here. So it has nothing to do with a solution of our PDE-based model up to now.

The next steps will actually evaluate the constraints in terms of data (from the `boundary` functions), and the model constraints from `f` to retrieve an actual solution to the PDE.

11.3 Loss function and training

As objective for the learning process we can now combine the *direct* constraints, i.e., the solution at $t = 0.5$ and the Dirichlet $u = 0$ boundary conditions with the loss from the PDE residuals. For both boundary constraints we'll use 100 points below, and then sample the solution in the inner region with an additional 1000 points.

The direct constraints are evaluated via `network(x, t)[:, 0] - u`, where `x` and `t` are the space-time location where we'd like to sample the solution, and `u` provides the corresponding ground truth value.

For the physical loss points, we have no ground truth solutions, but we'll only evaluate the PDE residual via the NN derivatives, to see whether the solution satisfies the PDE model. If not, this directly gives us an error to be reduced via an update step in the optimization. The corresponding expression is of the form `f(network(x, t)[:, 0], x, t)` below. Note that for both data and physics terms the `network()[:, 0]` expressions don't remove any data from the L^2 evaluation, they simply discard the last size-1 dimension of the $(n, 1)$ tensor returned by the network.

```
# Boundary loss
N_SAMPLE_POINTS_BND = 100
x_bc, t_bc, u_bc = [math.concat([v_t0, v_x], axis=0) for v_t0, v_x in zip(boundary_
 ↪tx(N_SAMPLE_POINTS_BND), open_boundary(N_SAMPLE_POINTS_BND))]
x_bc, t_bc, u_bc = np.asarray(x_bc,dtype=np.float32), np.asarray(t_bc,dtype=np.
 ↪float32) ,np.asarray(u_bc,dtype=np.float32)
#with app.model_scope():
loss_u = math.l2_loss(network(x_bc, t_bc)[:, 0] - u_bc)  # normalizes by first␣
 ↪dimension, N_bc

# Physics loss inside of domain
N_SAMPLE_POINTS_INNER = 1000
x_ph, t_ph = tf.convert_to_tensor(rnd.random_uniform([N_SAMPLE_POINTS_INNER], -1, 1)),
 ↪ tf.convert_to_tensor(rnd.random_uniform([N_SAMPLE_POINTS_INNER], 0, 1))
loss_ph = math.l2_loss(f(network(x_ph, t_ph)[:, 0], x_ph, t_ph))  # normalizes by␣
 ↪first dimension, N_ph

# Combine
ph_factor = 1.
loss = loss_u + ph_factor * loss_ph # allows us to control the relative influence of␣
 ↪loss_ph

optim = tf.train.GradientDescentOptimizer(learning_rate=0.02).minimize(loss)
#optim = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss) # alternative,␣
 ↪but not much benefit here
```

The code above just initializes the evaluation of the loss, we still didn't do any optimization steps, but we're finally in a good position to get started with this.

Despite the simple equation, the convergence is typically very slow. The iterations themselves are fast to compute, but this setup needs a *lot* of iterations. To keep the runtime in a reasonable range, we only do 10k iterations by default below (`ITERS`). You can increase this value to get better results.

```
session.initialize_variables()

import time
start = time.time()

ITERS = 10000
for optim_step in range(ITERS+1):
  _, loss_value = session.run([optim, loss])
  if optim_step<3 or optim_step%1000==0:
```

(continues on next page)

(continued from previous page)

```
        print('Step %d, loss: %f' % (optim_step,loss_value))
        #show_state(grid_u)

end = time.time()
print("Runtime {:.2f}s".format(end-start))
```

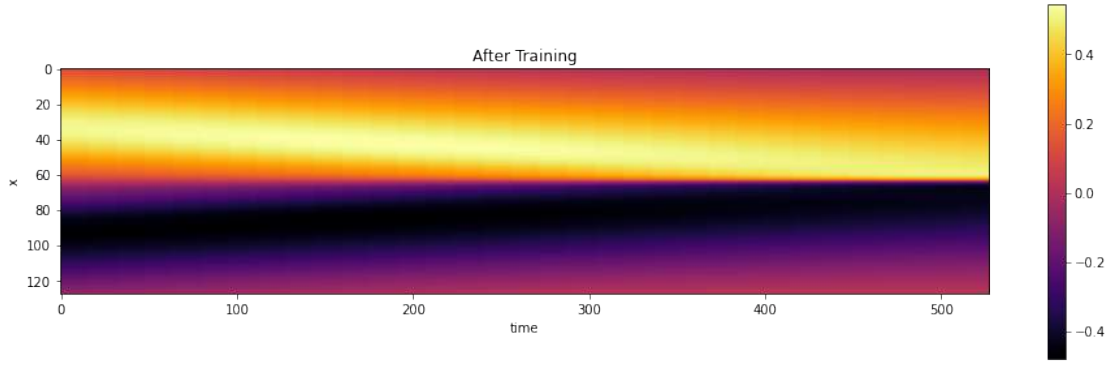
```
Step 0, loss: 0.100037
Step 1, loss: 0.097836
Step 2, loss: 0.096151
Step 1000, loss: 0.054487
Step 2000, loss: 0.049147
Step 3000, loss: 0.045515
Step 4000, loss: 0.042890
Step 5000, loss: 0.039945
Step 6000, loss: 0.037641
Step 7000, loss: 0.035227
Step 8000, loss: 0.033604
Step 9000, loss: 0.031556
Step 10000, loss: 0.029434
Runtime 101.02s
```

As the training uses well established building blocks (the dense layers) it should only take around 2 minutes on a typical notebook, and the error should go down significantly (roughly from around 0.2 to ca. 0.03), and the network seems to successfully converge towards a solution.

Let's show the reconstruction of the network, by evaluating the network at the centers of a regular grid, so that we can show the solution as an image. Note that this is actually fairly expensive, as we have to run through the whole network with a few thousand weights for all of the 128×32 sampling points in the grid.

It looks pretty good on first sight: There's been a very noticeable change compared to the random initialization shown above:

```
show_state(session.run(grid_u),"After Training")
```

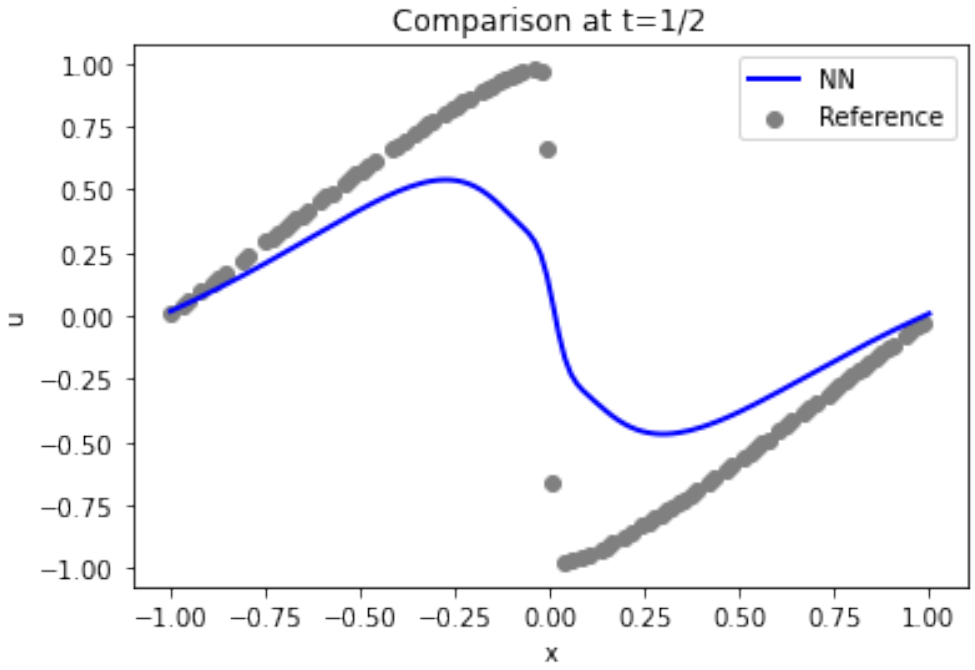


11.4 Evaluation

Let's compare solution in a bit more detail. Here are the actual sample points used for constraining the solution (at time step 16, $t = 1/2$) shown in gray, versus the reconstructed solution in blue:

```
u = session.run(grid_u)

# solution is imposed at t=1/2 , which is 16 in the array
BC_TX = 16
uT = u[0,:,BC_TX,0]

fig = plt.figure().gca()
fig.plot(np.linspace(-1,1,len(uT)), uT, lw=2, color='blue', label="NN")
fig.scatter(x_bc[0:100], u_bc[0:100], color='gray', label="Reference")
plt.title("Comparison at t=1/2")
plt.xlabel('x'); plt.ylabel('u'); plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f8eca2a7a00>
```



Not too bad at the sides of the domain (the Dirichlet boundary conditions $u = 0$ are fulfilled), but the shock in the center (at $x = 0$) is not well represented.

Let's check how well the initial state at $t = 0$ was reconstructed. That's the most interesting, and toughest part of the problem (the rest basically follows from the model equation and boundary conditions, given the first state).

It turns out that the accuracy of the initial state is actually not that good: the blue curve from the PINN is quite far away from the constraints via the reference data (shown in gray)… The solution will get better with larger number of iterations, but it requires a surprisingly large number of iterations for this fairly simple case.

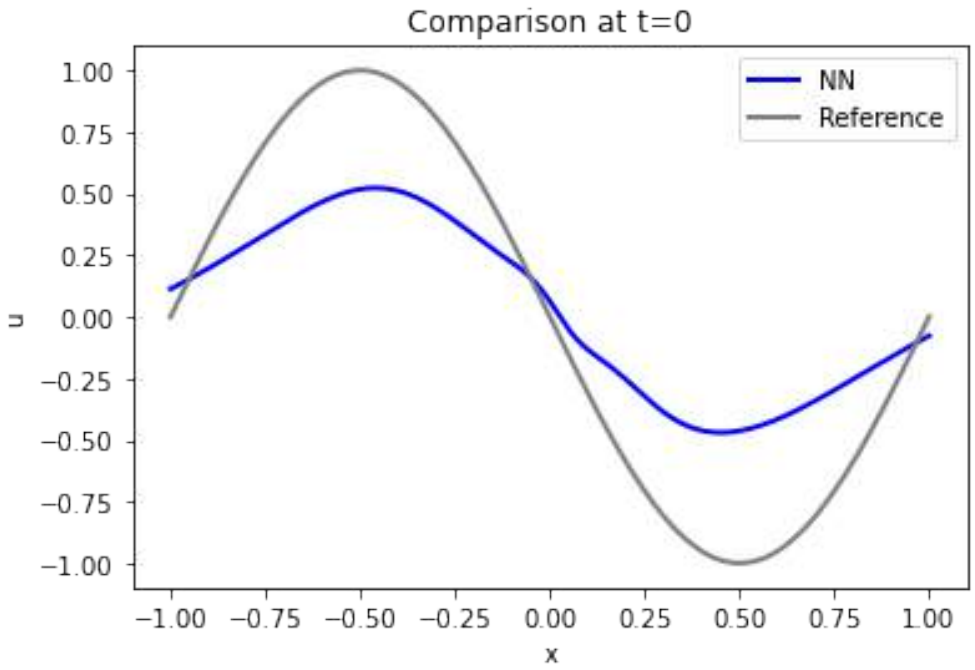
```
# ground truth solution at t0
t0gt = np.asarray( [ [-math.sin(np.pi * x) * 1.] for x in np.linspace(-1,1,N)] )
```

(continues on next page)

(continued from previous page)

```
velP0 = u[0,:,0,0]

fig = plt.figure().gca()
fig.plot(np.linspace(-1,1,len(velP0)), velP0, lw=2, color='blue', label="NN")
fig.plot(np.linspace(-1,1,len(t0gt)), t0gt, lw=2, color='gray', label="Reference")
plt.title("Comparison at t=0")
plt.xlabel('x'); plt.ylabel('u'); plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f8eca1b48e0>
```



Especially the maximum / minimum at $x = \pm 1/2$ are far off, and the boundaries at $x = \pm 1$ are not fulfilled: the solution is not at zero.

We have the forward simulator for this simulation, so we can use the $t = 0$ solution of the network to evaluate how well the temporal evaluation was reconstructed. This measures how well the temporal evolution of the model equation was captured via the soft constraints of the PINN loss.

The graph below shows the initial state in blue, and two evolved states at $t = 8/32$ and $t = 15/32$. Note that this is all from the simulated version, we'll show the learned version next.

(Note: The code segments below also have some optional code to show the states at `[STEPS//4]`. It's commented out by default, you can uncomment or add additional ones to visualize more of the time evolution if you like.)

```
# re-simulate with phiflow from solution at t=0
DT = 1./32.
STEPS = 32-BC_TX # depends on where BCs were imposed
INITIAL = u[...,BC_TX:(BC_TX+1),0] # np.reshape(u0, [1,len(u0),1])
print(INITIAL.shape)

DOMAIN = Domain([N], boundaries=PERIODIC, box=box[-1:1])
state = [BurgersVelocity(DOMAIN, velocity=INITIAL, viscosity=0.01/np.pi)]
```

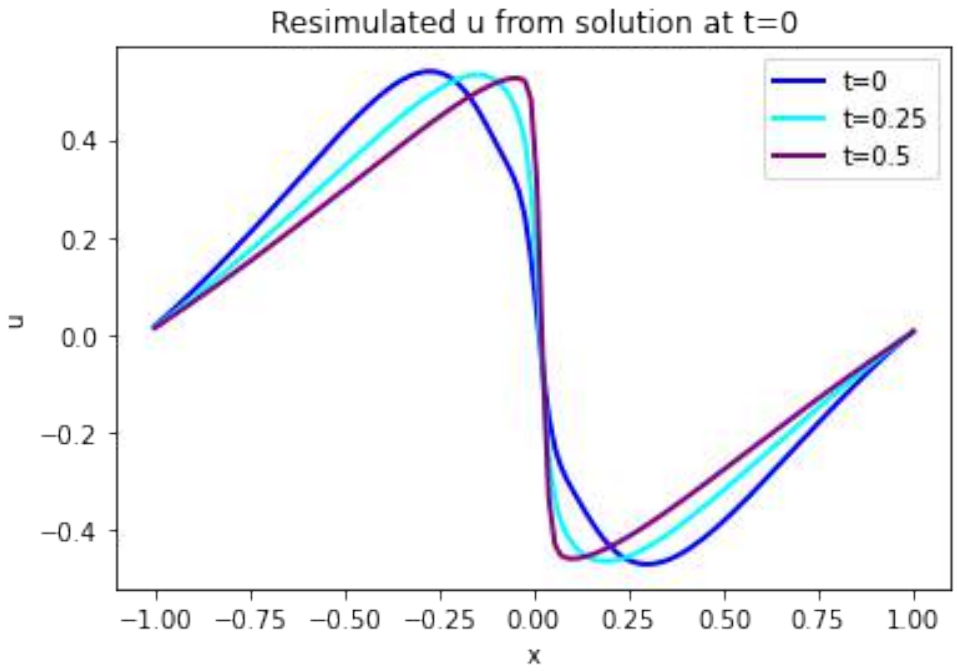
(continues on next page)

(continued from previous page)

```
physics = Burgers()

for i in range(STEPS):
    state.append( physics.step(state[-1],dt=DT) )

# we only need "velocity.data" from each phiflow state
vel_resim = [x.velocity.data for x in state]

fig = plt.figure().gca()
pltx = np.linspace(-1,1,len(vel_resim[0].flatten()))
fig.plot(pltx, vel_resim[ 0].flatten(),       lw=2, color='blue',  label="t=0")
#fig.plot(pltx, vel_resim[STEPS//4].flatten(), lw=2, color='green', label="t=0.125")
fig.plot(pltx, vel_resim[STEPS//2].flatten(), lw=2, color='cyan',  label="t=0.25")
fig.plot(pltx, vel_resim[STEPS-1].flatten(),  lw=2, color='purple',label="t=0.5")
#fig.plot(pltx, t0gt, lw=2, color='gray', label="t=0 Reference") # optionally show GT,
 ↪ compare to blue
plt.title("Resimulated u from solution at t=0")
plt.xlabel('x'); plt.ylabel('u'); plt.legend()
```

```
(1, 128, 1)
```

```
<matplotlib.legend.Legend at 0x7f8eca556820>
```



And here is the PINN output from `u` at the same time steps:

```
velP = [u[0,:,x,0] for x in range(33)]
print(velP[0].shape)

fig = plt.figure().gca()
fig.plot(pltx, velP[BC_TX+ 0].flatten(),       lw=2, color='blue',  label="t=0")
```

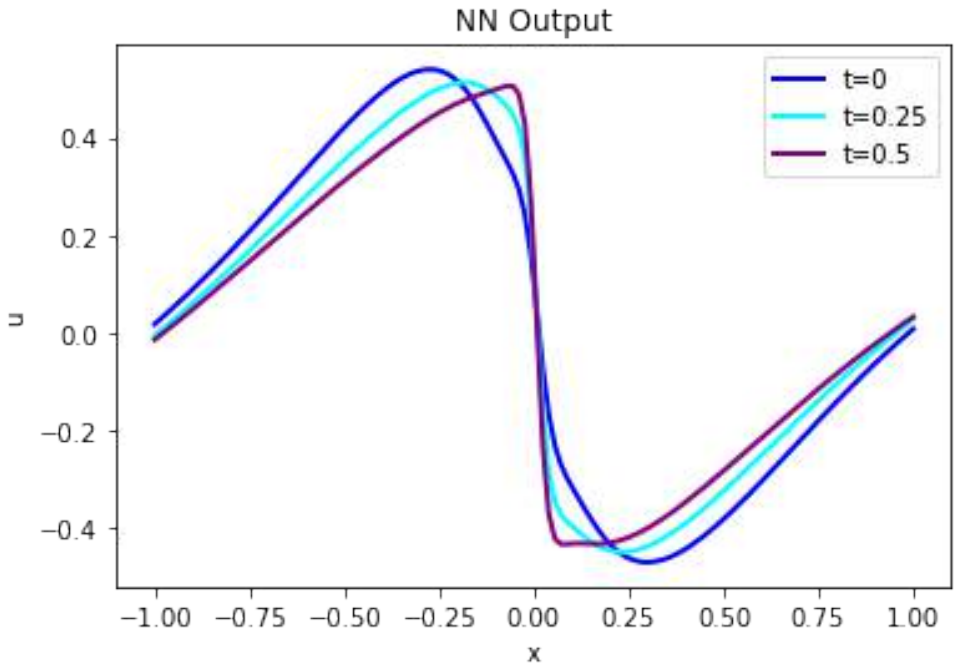
(continues on next page)

(continued from previous page)

```
#fig.plot(pltx, velP[BC_TX+STEPS//4].flatten(), lw=2, color='green', label="t=0.125")
fig.plot(pltx, velP[BC_TX+STEPS//2].flatten(), lw=2, color='cyan',  label="t=0.25")
fig.plot(pltx, velP[BC_TX+STEPS-1].flatten(),  lw=2, color='purple',label="t=0.5")
plt.title("NN Output")
plt.xlabel('x'); plt.ylabel('u'); plt.legend()
```

```
(128,)
```

```
<matplotlib.legend.Legend at 0x7f8ec93a4310>
```

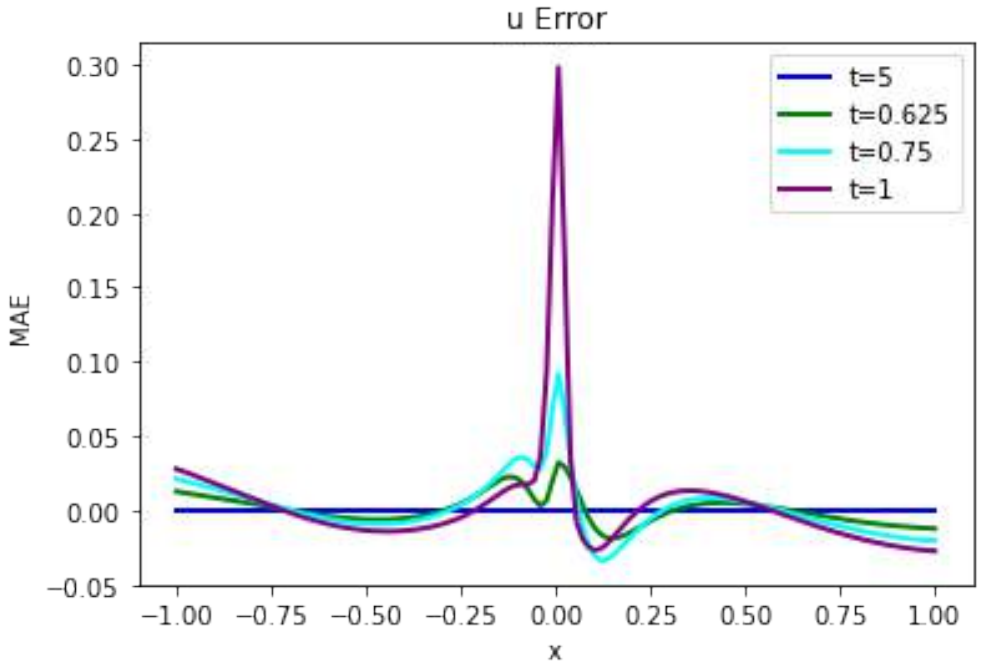


Judging via eyeball norm, these two versions of u look quite similar, but not surprisingly the errors grow over time and there are significant differences. Especially the steepening of the solution near the shock at $x = 0$ is not "captured" well. It's a bit difficult to see in these two graphs, though, let's quantify the error and show the actual difference:

```
error = np.sum( np.abs( np.asarray(vel_resim[0:16]).flatten() - np.asarray(velP[BC_
 ↪TX:BC_TX+STEPS]).flatten() )) / (STEPS*N)
print("Mean absolute error for re-simulation across {} steps: {:7.5f}".format(STEPS,
 ↪error))

fig = plt.figure().gca()
fig.plot(pltx, (vel_resim[0       ].flatten()-velP[BC_TX         ].flatten()), lw=2,␣
 ↪color='blue',  label="t=5")
fig.plot(pltx, (vel_resim[STEPS//4].flatten()-velP[BC_TX+STEPS//4].flatten()), lw=2,␣
 ↪color='green', label="t=0.625")
fig.plot(pltx, (vel_resim[STEPS//2].flatten()-velP[BC_TX+STEPS//2].flatten()), lw=2,␣
 ↪color='cyan',  label="t=0.75")
fig.plot(pltx, (vel_resim[STEPS-1 ].flatten()-velP[BC_TX+STEPS-1 ].flatten()), lw=2,␣
 ↪color='purple',label="t=1")
plt.title("u Error")
plt.xlabel('x'); plt.ylabel('MAE'); plt.legend()
```

```
Mean absolute error for re-simulation across 16 steps: 0.01136
```

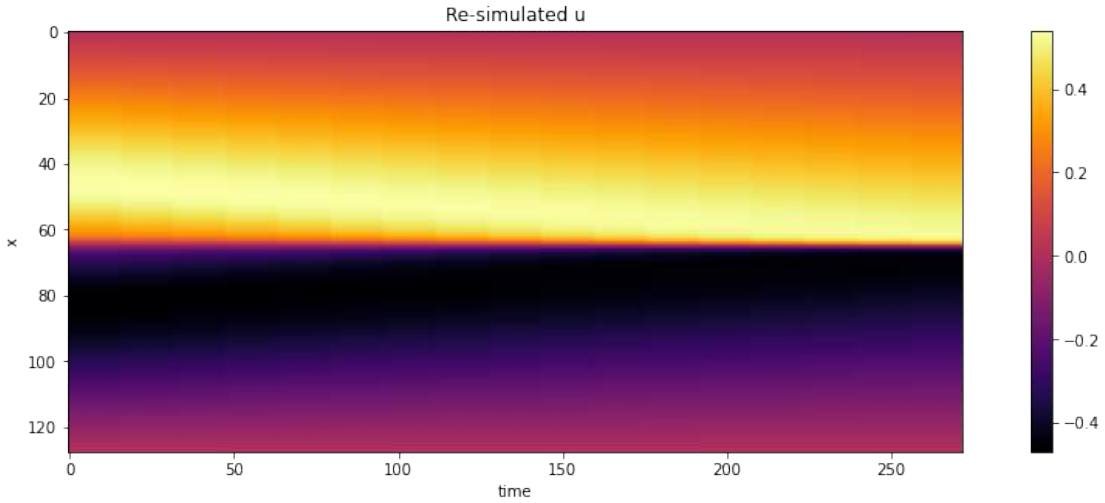
```
<matplotlib.legend.Legend at 0x7f8eaa76ed90>
```



The code above will compute a mean absolute error of ca. $1.5 \cdot 10^{-2}$ between ground truth re-simulation and the PINN evolution, which is significant for the value range of the simulation.

And for comparison with the forward simulation and following cases, here are also all steps over time with a color map.

```
# show re-simulated solution again as full image over time
sn = np.concatenate(vel_resim, axis=-1)
sn = np.reshape(sn, list(sn.shape)+[1] ) # print(sn.shape)
show_state(sn,"Re-simulated u")
```



Next, we'll store the full solution over the course of the $t = 0 \dots 1$ time interval, so that we can compare it later on to the full solution from a regular forward solve and compare it to the differential physics solution.

Thus, stay tuned for the full evaluation and the comparison. This will follow in *Burgers Optimization with a Differentiable Physics Gradient*, after we've discussed the details of how to run the differential physics optimization.

```
vels = session.run(grid_u) # special for showing NN results, run through TF
vels = np.reshape( vels, [vels.shape[1],vels.shape[2]] )

# save for comparison with other methods
import os; os.makedirs("./temp",exist_ok=True)
np.savez_compressed("./temp/burgers-pinn-solution.npz",vels) ; print("Vels array␣
↪shape: "+format(vels.shape))
```

```
Vels array shape: (128, 33)
```

11.5 Next steps

This setup is just a starting point for PINNs and physical soft-constraints, of course. The parameters of the setup were chosen to run relatively quickly. As we'll show in the next sections, the behavior of such an inverse solve can be improved substantially by a tighter integration of solver and learning.

The solution of the PINN setup above can also directly be improved, however. E.g., try to:

- Adjust parameters of the training to further decrease the error without making the solution diverge.
- Adapt the NN architecture for further improvements (keep track of the weight count, though).
- Activate a different optimizer, and observe the change in behavior (this typically requires adjusting the learning rate). Note that the more complex optimizers don't necessarily do better in this relatively simple example.
- Or modify the setup to make the test case more interesting: e.g., move the boundary conditions to a later point in simulation time, to give the reconstruction a larger time interval to reconstruct.

CHAPTER

TWELVE

DISCUSSION OF PHYSICAL LOSSES

The good news so far is - we have a DL method that can include physical laws in the form of soft constraints by minimizing residuals. However, as the very simple previous example illustrates, this causes new difficulties, and is just a conceptual starting point.

On the positive side, we can leverage DL frameworks with backpropagation to compute the derivatives of the model. At the same time, this makes the loss landscape more complicated, relies on the learned representation regarding the reliability of the derivatives. Also, each derivative requires backpropagation through the full network. This can be very expensive, especially for higher-order derivatives.

And while the setup is relatively simple, it is generally difficult to control. The NN has flexibility to refine the solution by itself, but at the same time, tricks are necessary when it doesn't focus on the right regions of the solution.

12.1 Generalization?

One aspect to note with the previous PINN optimization is that the positions where we test and constrain the solution are the final positions we are interested in. As such, from a classic ML standpoint, there is no real distinction between training, validation and test sets. Computing the solution for a known and given set of samples is much more akin to classical optimization, where inverse problems like the previous Burgers example stem from.

For machine learning, we typically work under the assumption that the final performance of our model will be evaluated on a different, potentially unknown set of inputs. The *test data* should usually capture such *out of distribution* (OOD) behavior, so that we can make estimates about how well our model will generalize to "real-world" cases that we will encounter when we deploy it in an application. The v1 version, using a prescribed discretization actually had this property, and could generalized to new inputs.

In contrast, for the PINN training as described here, we reconstruct a single solution in a known and given space-time region. As such, any samples from this domain follow the same distribution and hence don't really represent test or OOD samples. As the NN directly encodes the solution, there is also little hope that it will yield different solutions, or perform well outside of the training range. If we're interested in a different solution, we have to start training the NN from scratch.

12.2 Summary

Thus, the physical soft constraints allow us to encode solutions to PDEs with the tools of NNs. As they're more widely used, we'll focus on PINNs (v2) here: An inherent drawback is that they typically yield single solutions or very narrow solution manifolds, and that they do not combine with traditional numerical techniques well. In comparison to the Neural surrogates/operators from *Supervised Training* we've made a step backwards in some way.

E.g., the learned representation is not suitable to be refined with a classical iterative solver such as the conjugate gradient method. This means many powerful techniques that were developed in the past decades cannot be used in this context. Bringing these numerical methods back into the picture will be one of the central goals of the next sections.

✓ Pro:

- Uses physical model
- Derivatives can be conveniently computed via backpropagation

✗ Con:

- Problematic convergence
- Physical constraints are enforced only as soft constraints
- Largely incompatible with *classical* numerical methods
- Usefulness of derivatives relies on learned representation

To address these issues, we'll next look at how we can leverage existing numerical methods to improve the DL process by making use of differentiable solvers.

Part IV

Differentiable Physics

CHAPTER

THIRTEEN

INTRODUCTION TO DIFFERENTIABLE PHYSICS

As a next step towards a tighter and more generic combination of deep learning methods and physical simulations we will target incorporating *differentiable numerical simulations* into the learning process. In the following, we'll shorten these "differentiable numerical simulations of physical systems" to just "differentiable physics" (DP).

The central goal of these methods is to use existing numerical solvers to empower and improve AI systems. This requires equipping them with functionality to compute gradients with respect to their inputs. Once this is realized for all operators of a simulation, we can leverage the autodiff functionality of DL frameworks with backpropagation to let gradient information flow from a simulator into an NN and vice versa. This has numerous advantages such as improved learning feedback and generalization, as we'll outline below.

In contrast to the physics-informed loss functions of the previous chapter, it also enables handling more complex solution manifolds instead of single inverse problems. E.g., instead of using deep learning to solve single inverse problems as in the previous chapter, differentiable physics can be used to train NNs that learn to solve larger classes of inverse problems very efficiently.

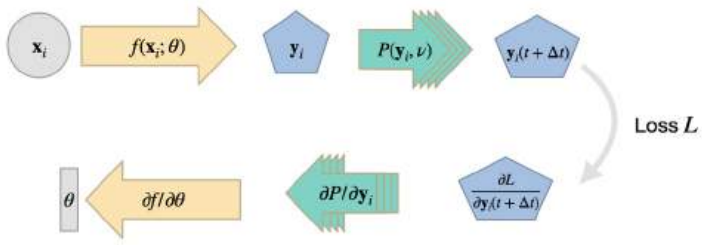


Fig. 13.1: Training with differentiable physics means that a chain of differentiable operators provide directions in the form of gradients to steer the learning process.

13.1 Differentiable operators

With DP we build on *existing* numerical solvers. I.e., the approach is strongly relying on the algorithms developed in the larger field of computational methods for a vast range of physical effects in our world. To start with, we need a continuous formulation as model for the physical effect that we'd like to simulate – if this is missing we're in trouble. But luckily, we can tap into existing collections of model equations and established methods for discretizing continuous models.

Let's assume we have a continuous formulation $\mathcal{P}^*(\mathbf{x}, \nu)$ of the physical quantity of interest $\mathbf{u}(\mathbf{x}, t) : \mathbb{R}^d \times \mathbb{R}^+ \rightarrow \mathbb{R}^d$, with model parameters ν (e.g., diffusion, viscosity, or conductivity constants). The components of $\mathbf{u}$ will be denoted by a numbered subscript, i.e., $\mathbf{u} = (u_1, u_2, \ldots, u_d)^T$.

Typically, we are interested in the temporal evolution of such a system. Discretization yields a formulation $\mathcal{P}(\mathbf{x}, \nu)$ that we re-arrange to compute a future state after a time step Δt. The state at $t + \Delta t$ is computed via sequence of operations $\mathcal{P}_1, \mathcal{P}_2 \ldots \mathcal{P}_m$ such that $\mathbf{u}(t + \Delta t) = \mathcal{P}_m \circ \ldots \mathcal{P}_2 \circ \mathcal{P}_1(\mathbf{u}(t), \nu)$, where $\circ$ denotes function decomposition, i.e. $f(g(x)) = f \circ g(x)$.

Note

In order to integrate this solver into a DL process, we need to ensure that every operator $\mathcal{P}_i$ provides a gradient w.r.t. its inputs, i.e. in the example above $\partial \mathcal{P}_i / \partial \mathbf{u}$.

Note that we typically don't need derivatives for all parameters of $\mathcal{P}(\mathbf{x}, \nu)$, e.g., we omit ν in the following, assuming that this is a given model parameter with which the NN should not interact. Naturally, it can vary within the solution manifold that we're interested in, but ν will not be the output of an NN representation. If this is the case, we can omit providing $\partial \mathcal{P}_i / \partial \nu$ in our solver. However, the following learning process naturally transfers to including ν as a degree of freedom.

13.2 Jacobians

As $\mathbf{u}$ is typically a vector-valued function, $\partial \mathcal{P}_i / \partial \mathbf{u}$ denotes a Jacobian matrix J rather than a single value:

$$\frac{\partial \mathcal{P}_i}{\partial \mathbf{u}} = \begin{bmatrix} \partial \mathcal{P}_{i,1}/\partial u_1 & \cdots & \partial \mathcal{P}_{i,1}/\partial u_d \\ \vdots & & \\ \partial \mathcal{P}_{i,d}/\partial u_1 & \cdots & \partial \mathcal{P}_{i,d}/\partial u_d \end{bmatrix}$$

where, as above, d denotes the number of components in $\mathbf{u}$. As $\mathcal{P}$ maps one value of $\mathbf{u}$ to another, the Jacobian is square here. Of course this isn't necessarily the case for general model equations, but non-square Jacobian matrices would not cause any problems for differentiable simulations.

In practice, we rely on the *reverse mode* differentiation that all modern DL frameworks provide, and focus on computing a matrix vector product of the Jacobian transpose with a vector $\mathbf{a}$, i.e. the expression: $\left(\frac{\partial \mathcal{P}_i}{\partial \mathbf{u}}\right)^T \mathbf{a}$. If we'd need to construct and store all full Jacobian matrices that we encounter during training, this would cause huge memory overheads and unnecessarily slow down training. Instead, for backpropagation, we can provide faster operations that compute products with the Jacobian transpose because we always have a scalar loss function at the end of the chain.

Given the formulation above, we need to resolve the derivatives of the chain of function compositions of the $\mathcal{P}_i$ at some current state $\mathbf{u}^n$ via the chain rule. E.g., for two of them

$$\frac{\partial (\mathcal{P}_1 \circ \mathcal{P}_2)}{\partial \mathbf{u}}\Big|_{\mathbf{u}^n} = \frac{\partial \mathcal{P}_1}{\partial \mathbf{u}}|_{\mathcal{P}_2(\mathbf{u}^n)} \; \frac{\partial \mathcal{P}_2}{\partial \mathbf{u}}|_{\mathbf{u}^n} ,$$

which is just the vector valued version of the "classic" chain rule $f(g(x))' = f'(g(x))g'(x)$, and directly extends for larger numbers of composited functions, i.e. $i > 2$.

Here, the derivatives for $\mathcal{P}_1$ and $\mathcal{P}_2$ are still Jacobian matrices, but knowing that at the "end" of the chain we have our scalar loss (cf. *Overview*), the right-most Jacobian will invariably be a matrix with 1 column, i.e. a vector. During reverse mode, we start with this vector, and compute the multiplications with the left Jacobians, $\frac{\partial \mathcal{P}_1}{\partial \mathbf{u}}$ above, one by one.

For the details of forward and reverse mode differentiation, please check out external materials such as this nice survey by Baydin et al..

13.3 Learning via DP operators

Thus, once the operators of our simulator support computations of the Jacobian-vector products, we can integrate them into DL pipelines just like you would include a regular fully-connected layer or a ReLU activation.

At this point, the following (very valid) question arises: "*Most physics solvers can be broken down into a sequence of vector and matrix operations. All state-of-the-art DL frameworks support these, so why don't we just use these operators to realize our physics solver?*"

It's true that this would theoretically be possible. The problem here is that each of the vector and matrix operations in tensorflow and pytorch is computed individually, and internally needs to store the current state of the forward evaluation for backpropagation (the "$g(x)$" above). For a typical simulation, however, we're not overly interested in every single intermediate result our solver produces. Typically, we're more concerned with significant updates such as the step from $\mathbf{u}(t)$ to $\mathbf{u}(t + \Delta t)$.

Thus, in practice it is a very good idea to break down the solving process into a sequence of meaningful but *monolithic* operators. This not only saves a lot of work by preventing the calculation of unnecessary intermediate results, it also allows us to choose the best possible numerical methods to compute the updates (and derivatives) for these operators. E.g., as this process is very similar to adjoint method optimizations, we can re-use many of the techniques that were developed in this field, or leverage established numerical methods. E.g., we could leverage the $O(n)$ runtime of multigrid solvers for matrix inversion.

The flip-side of this approach is that it requires some understanding of the problem at hand, and of the numerical methods. Also, a given solver might not provide gradient calculations out of the box. Thus, if we want to employ DL for model equations that we don't have a proper grasp of, it might not be a good idea to directly go for learning via a DP approach. However, if we don't really understand our model, we probably should go back to studying it a bit more anyway…

Also, in practice we should be *greedy* with the derivative operators, and only provide those which are relevant for the learning task. E.g., if our network never produces the parameter ν in the example above, and it doesn't appear in our loss formulation, we will never encounter a $\partial / \partial \nu$ derivative in our backpropagation step.

The following figure summarizes the DP-based learning approach, and illustrates the sequence of operations that are typically processed within a single PDE solve. As many of the operations are non-linear in practice, this often leads to a challenging learning task for the NN:

13.4 A practical example

As a simple example let's consider the advection of a passive scalar density $d(\mathbf{x}, t)$ in a velocity field $\mathbf{u}$ as physical model $\mathcal{P}^*$:

$$\frac{\partial d}{\partial t} + \mathbf{u} \cdot \nabla d = 0$$

Instead of using this formulation as a residual equation right away (as in v2 of *Physical Loss Terms*), we can discretize it with our favorite mesh and discretization scheme, to obtain a formulation that updates the state of our system over time.

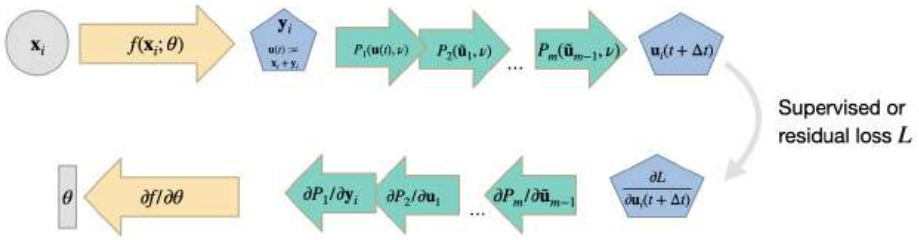


Fig. 13.2: DP learning with a PDE solver that consists of m individual operators $\mathcal{P}_i$. The gradient travels backward through all m operators before influencing the network weights θ.

This is a standard procedure for a *forward* solve. To simplify things, we assume here that $\mathbf{u}$ is only a function in space, i.e. constant over time. We'll bring back the time evolution of $\mathbf{u}$ later on.

Let's denote this re-formulation as $\mathcal{P}$. It maps a state of $d(t)$ into a new state at an evolved time, i.e.:

$$d(t+\Delta t) = \mathcal{P}(\, d(t), \mathbf{u}, t+\Delta t)$$

As a simple example of an inverse problem and learning task, let's consider the problem of finding a velocity field $\mathbf{u}$. This velocity should transform a given initial scalar density state $d^{\,0}$ at time t^0 into a state that's evolved by $\mathcal{P}$ to a later "end" time t^e with a certain shape or configuration d^{target}. Informally, we'd like to find a flow that deforms $d^{\,0}$ through the PDE model into a target state. The simplest way to express this goal is via an L^2 loss between the two states. So we want to minimize the loss function $L = |d(t^e) - d^{\text{target}}|^2$.

Note that as described here, this inverse problem is a pure optimization task: there's no NN involved, and our goal is to obtain $\mathbf{u}$. We do not want to apply this velocity to other, unseen *test data*, as would be custom in a real learning task.

The final state of our marker density $d(t^e)$ is fully determined by the evolution from $\mathcal{P}$ via $\mathbf{u}$, which gives the following minimization problem:

$$\arg\min_{\mathbf{u}} |\mathcal{P}(d^{\,0}, \mathbf{u}, t^e) - d^{\text{target}}|^2$$

We'd now like to find the minimizer for this objective by *gradient descent* (GD), where the gradient is determined by the differentiable physics approach described earlier in this chapter. Once things are working with GD, we can relatively easily switch to better optimizers or bring an NN into the picture, hence it's always a good starting point. To make things easier to read below, we'll omit the transpose of the Jacobians in the following. Unfortunately, the Jacobian is defined this way, but we actually never need the un-transposed one. Keep in mind that in practice we're dealing with transposed Jacobians $\left(\frac{\partial a}{\partial b}\right)^T$ that are "abbreviated" by $\frac{\partial a}{\partial b}$.

As the discretized velocity field $\mathbf{u}$ contains all our degrees of freedom, all we need to do is to update the velocity by an amount $\Delta\mathbf{u} = \partial L/\partial\mathbf{u}$, which is decomposed into $\Delta\mathbf{u} = \frac{\partial d}{\partial \mathbf{u}} \frac{\partial L}{\partial d}$.

The $\frac{\partial L}{\partial d}$ component is typically simple enough: we'll get

$$\frac{\partial L}{\partial d} = \frac{\partial |\mathcal{P}(d^{\,0}, \mathbf{u}, t^e) - d^{\text{target}}|^2}{\partial d} = 2(d(t^e) - d^{\text{target}}).$$

If d is represented as a vector, e.g., for one entry per cell of a mesh, $\frac{\partial L}{\partial d}$ will likewise be a column vector of equivalent size. This stems from the fact that L is always a scalar loss function, and so the Jacobian matrix will have a dimension of 1 along the L dimension. Intuitively, this vector will simply contain the differences between d at the end time in comparison to the target densities d^{target}.

The evolution of d itself is given by our discretized physical model $\mathcal{P}$, and we use $\mathcal{P}$ and d interchangeably. Hence, the more interesting component is the Jacobian $\partial d / \partial \mathbf{u} = \partial \mathcal{P} / \partial \mathbf{u}$ to compute the full $\Delta \mathbf{u} = \frac{\partial d}{\partial \mathbf{u}} \frac{\partial L}{\partial d}$. We luckily don't need $\partial d / \partial \mathbf{u}$ as a full matrix, but instead only multiplied by $\frac{\partial L}{\partial d}$.

So what is the actual Jacobian for d? To compute it we first need to finalize our PDE model $\mathcal{P}$, such that we get an expression which we can derive. In the next section we'll choose a specific advection scheme and a discretization so that we can be more specific.

13.4.1 Introducing a specific advection scheme

In the following we'll make use of a simple first order upwinding scheme on a Cartesian grid in 1D, with marker density d_i and velocity u_i for cell i. We omit the (t) for quantities at time t for brevity, i.e., $d_i(t)$ is written as d_i below. From above, we'll use our *physical model* that updates the marker density $d_i(t + \Delta t) = \mathcal{P}(d_i(t), \mathbf{u}(t), t + \Delta t)$, which gives the following:

$$
\begin{aligned}
&d_i(t + \Delta t) = d_i - \Delta t [u_i^+ (d_{i+1} - d_i) + u_i^- (d_i - d_{i-1})] \text{ with} \\
&u_i^+ = \min(u_i / \Delta x, 0) \\
&u_i^- = \max(u_i / \Delta x, 0)
\end{aligned}
$$

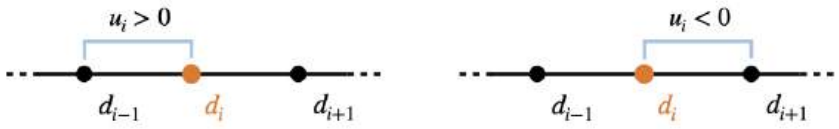


Fig. 13.3: 1st-order upwinding uses a simple one-sided finite-difference stencil that takes into account the direction of the flow

Thus, for a negative u_i, we're using u_i^+ to look in the opposite direction of the velocity, i.e., *backward* in terms of the motion. u_i^- will be zero in this case. For positive u_i it's vice versa, and we'll get a zero'ed u_i^+, and a backward difference stencil via u_i^-. To pick the former case, for a negative u_i we get

$$
\mathcal{P}(d_i(t), \mathbf{u}(t), t + \Delta t) = (1 + \frac{u_i \Delta t}{\Delta x}) d_i - \frac{u_i \Delta t}{\Delta x} d_{i+1} \tag{13.1}
$$

and hence $\partial \mathcal{P} / \partial u_i$ gives $\frac{\Delta t}{\Delta x} d_i - \frac{\Delta t}{\Delta x} d_{i+1}$. Intuitively, the change of the velocity u_i depends on the spatial derivatives of the densities. Due to the first order upwinding, we only include two neighbors (higher order methods would depend on additional entries of d)

In practice this step is equivalent to evaluating a transposed matrix multiplication. If we rewrite the calculation above as $\mathcal{P}(d_i(t), \mathbf{u}(t), t + \Delta t) = A \mathbf{u}$, then $\left(\partial \mathcal{P} / \partial \mathbf{u}\right)^T = A^T$. However, in many practical cases, a matrix free implementation of this multiplication might be preferable to actually constructing A.

Another derivative that we can consider for the advection scheme is that w.r.t. the previous density state, i.e. $d_i(t)$, which is d_i in the shortened notation. $\partial \mathcal{P} / \partial d_i$ for cell i from above gives $1 + \frac{u_i \Delta t}{\Delta x}$. However, for the full gradient we'd need to add the potential contributions from cells $i + 1$ and $i - 1$, depending on the sign of their velocities. This derivative will come into play in the next section.

13.4.2 Time evolution

So far we've only dealt with a single update step of d from time t to $t + \Delta t$, but we could of course have an arbitrary number of such steps. After all, above we stated the goal to advance the initial marker state $d(t^0)$ to the target state at time t^e, which could encompass a long interval of time.

In the expression above for $d_i(t+\Delta t)$, each of the $d_i(t)$ in turn depends on the velocity and density states at time $t-\Delta t$, i.e., $d_i(t-\Delta t)$. Thus we have to trace back the influence of our loss L all the way back to how $\mathbf{u}$ influences the initial marker state. This can involve a large number of evaluations of our advection scheme via $\mathcal{P}$.

This sounds challenging at first: e.g., one could try to insert equation (13.1) at time $t - \Delta t$ into equation (13.1) at time t and repeat this process recursively until we have a single expression relating $d^{\,0}$ to the targets. However, thanks to the linear nature of the Jacobians, we treat each advection step, i.e., each invocation of our PDE $\mathcal{P}$ as a separate, modular operation. And each of these invocations follows the procedure described in the previous section.

Given the machinery above, the backtrace is fairly simple to realize: for each of the advection steps in $\mathcal{P}$ we compute a Jacobian product with the *incoming* vector of derivatives from the loss L or a previous advection step. We repeat this until we have traced the chain from the loss with d^{target} all the way back to $d^{\,0}$. Theoretically, the velocity $\mathbf{u}$ could be a function of time like d, in which case we'd get a gradient $\Delta\mathbf{u}(t)$ for every time step t. However, to simplify things below, let's we assume we have field that is constant in time, i.e., we're reusing the same velocities $\mathbf{u}$ for every advection via $\mathcal{P}$. Now, each time step will give us a contribution to $\Delta\mathbf{u}$ which we accumulate for all steps.

$$
\begin{aligned}
\Delta\mathbf{u} = & \frac{\partial d(t^e)}{\partial \mathbf{u}} \frac{\partial L}{\partial d(t^e)} \\
& + \frac{\partial d(t^e - \Delta t)}{\partial \mathbf{u}} \frac{\partial d(t^e)}{\partial d(t^e - \Delta t)} \frac{\partial L}{\partial d(t^e)} \\
& + \cdots \\
& + \Big(\frac{\partial d(t^0)}{\partial \mathbf{u}} \cdots \frac{\partial d(t^e - \Delta t)}{\partial d(t^e - 2\Delta t)} \frac{\partial d(t^e)}{\partial d(t^e - \Delta t)} \frac{\partial L}{\partial d(t^e)} \Big)
\end{aligned}
$$

Here the last term above contains the full backtrace of the marker density to time t^0. The terms of this sum look unwieldy at first, but looking closely, each line simply adds an additional Jacobian for one time step on the left hand side. This follows from the chain rule, as shown in the two-operator case above. So the terms of the sum contain a lot of similar Jacobians, and in practice can be computed efficiently by backtracing through the sequence of computational steps that resulted from the forward evaluation of our PDE. (Note that, as mentioned above, we've omitted the transpose of the Jacobians here.)

This structure also makes clear that the process is very similar to the regular training process of an NN: the evaluations of these Jacobian vector products from nested function calls is exactly what a deep learning framework does for training an NN (we just have weights θ instead of a velocity field there). And hence all we need to do in practice is to provide a custom function the Jacobian vector product for $\mathcal{P}$.

13.5 Implicit gradient calculations

As a slightly more complex example let's consider Poisson's equation $\nabla^2 a = b$, where a is the quantity of interest, and b is given. This is a very fundamental elliptic PDE that is important for a variety of physical problems, from electrostatics to gravitational fields. It also arises in the context of fluids, where a takes the role of a scalar pressure field in the fluid, and the right hand side b is given by the divergence of the fluid velocity $\mathbf{u}$.

For fluids, we typically have $\mathbf{u}^n = \mathbf{u} - \nabla p$, with $\nabla^2 p = \nabla \cdot \mathbf{u}$. Here, $\mathbf{u}^n$ denotes the *new*, divergence-free velocity field. This step is typically crucial to enforce the hard-constraint $\nabla \cdot \mathbf{u} = 0$, and also goes under the name of *Chorin Projection*, or *Helmholtz decomposition*. It is a direct consequence of the fundamental theorem of vector calculus.

If we now introduce an NN that modifies $\mathbf{u}$ in a solver, we inevitably have to backpropagate through the Poisson solve. I.e., we need a gradient for $\mathbf{u}^n$, which in this notation takes the form $\partial \mathbf{u}^n / \partial \mathbf{u}$.

In combination, we aim for computing $\mathbf{u}^n = \mathbf{u} - \nabla \left((\nabla^2)^{-1} \nabla \cdot \mathbf{u} \right)$. The outer gradient (from ∇p) and the inner divergence ($\nabla \cdot \mathbf{u}$) are both linear operators, and their gradients are simple to compute. The main difficulty lies in obtaining the matrix inverse $(\nabla^2)^{-1}$ from Poisson's equation (we'll keep it a bit simpler here, but it's often time-dependent, and non-linear).

In practice, the matrix vector product for $(\nabla^2)^{-1} b$ with $b = \nabla \cdot \mathbf{u}$ is not explicitly computed via matrix operations, but approximated with a (potentially matrix-free) iterative solver. E.g., conjugate gradient (CG) methods are a very popular choice here. Thus, we theoretically could treat this iterative solver as a function $\mathcal{S}$, with $p = \mathcal{S}(\nabla \cdot \mathbf{u})$. It's worth noting that matrix inversion is a non-linear process, despite the matrix itself being linear. As solvers like CG are also based on matrix and vector operations, we could decompose $\mathcal{S}$ into a sequence of simpler operations over the course of all solver iterations as $\mathcal{S}(x) = \mathcal{S}_n(\mathcal{S}_{n-1}(...\mathcal{S}_1(x)))$, and backpropagate through each of them. This is certainly possible, but not a good idea: it can introduce numerical problems, and will be very slow. As mentioned above, by default DL frameworks store the internal states for every differentiable operator like the $\mathcal{S}_i()$ in this example, and hence we'd organize and keep a potentially huge number of intermediate states in memory. These states are completely uninteresting for our original PDE, though. They're just intermediate states of the CG solver.

If we take a step back and look at $p = (\nabla^2)^{-1} b$, it's gradient $\partial p / \partial b$ is just $((\nabla^2)^{-1})^T$. And in this case, (∇^2) is a symmetric matrix, and so $((\nabla^2)^{-1})^T = (\nabla^2)^{-1}$. This is the identical inverse matrix that we encountered in the original equation above, and hence we re-use our unmodified iterative solver to compute the gradient. We don't need to take it apart and slow it down by storing intermediate states. However, the iterative solver computes the matrix-vector-products for $(\nabla^2)^{-1} b$. So what is b during backpropagation? In an optimization setting we'll always have our loss function L at the end of the forward chain. The backpropagation step will then give a gradient for the output, let's assume it is $\partial L / \partial p$ here, which needs to be propagated to the earlier operations of the forward pass. Thus, we simply invoke our iterative solve during the backward pass to compute $\partial p / \partial b = \mathcal{S}(\partial L / \partial p)$. And assuming that we've chosen a good solver as $\mathcal{S}$ for the forward pass, we get exactly the same performance and accuracy in the backwards pass.

If you're interested in a code example, the differentiate-pressure example of phiflow uses exactly this process for an optimization through a pressure projection step: a flow field that is constrained on the right side, is optimized for the content on the left, such that it matches the target on the right after a pressure projection step.

The main take-away here is: it is important *not to blindly backpropagate* through the forward computation, but to think about which steps of the analytic equations for the forward pass to compute gradients for. In cases like the above, we can often find improved analytic expressions for the gradients, which we then approximate numerically.

Implicit Function Theorem & Time

IFT: The process above essentially yields an *implicit derivative*. Instead of explicitly deriving all forward steps, we've relied on the implicit function theorem to compute the derivative.

Time: we *can* actually consider the steps of an iterative solver as a virtual "time", and backpropagate through these steps. In line with other DP approaches, this enables an NN to *interact* with an iterative solver. An example is to learn initial guesses of CG solvers from [UBH+20]. Details and code can be found here.

13.6 Summary of differentiable physics so far

To summarize, using differentiable physical simulations gives us a tool to include physical equations with a chosen discretization into DL. In contrast to the residual constraints of the previous chapter, this makes it possible to let NNs seamlessly interact with physical solvers.

We'd previously fully discard our physical model and solver once the NN is trained: in the example from *Burgers Optimization with a PINN* the NN gives us the solution directly, bypassing any solver or model equation. The DP approach substantially differs from the physics-informed NNs (v2) from *Physical Loss Terms*, it has more in common with the controlled discretizations (v1). They are essentially a subset, or partial application of DP training.

However in contrast to both residual approaches, DP makes it possible to train an NN alongside a numerical solver, and thus we can make use of the physical model (as represented by the solver) later on at inference time. This allows us to move beyond solving single inverse problems, and yields NNs that quite robustly generalize to new inputs. Let's revisit the example problem from *Burgers Optimization with a PINN* in the context of DPs.

CHAPTER

FOURTEEN

BURGERS OPTIMIZATION WITH A DIFFERENTIABLE PHYSICS GRADIENT

To illustrate the process of computing gradients in a *differentiable physics* (DP) setting, we target the same inverse problem (the reconstruction task) used for the PINN example in *Burgers Optimization with a PINN*. The choice of DP as a method has some immediate implications: we start with a discretized PDE, and the evolution of the system is now fully determined by the resulting numerical solver. Hence, the only real unknown is the initial state. We will still need to re-compute all the states between the initial and target state many times, just now we won't need an NN for this step. Instead, we rely on the discretization of the model equations.

Also, as we choose an initial discretization for the DP approach, the unknown initial state consists of the sampling points of the involved physical fields, and we can simply represent these unknowns as floating point variables. Hence, even for the initial state we do not need to set up an NN. Thus, our Burgers reconstruction problem reduces to a gradient-based optimization without any NN when solving it with DP. Nonetheless, it's a very good starting point to illustrate the process.

First, we'll set up our discretized simulation. Here we employ phiflow, as shown in the overview section on *Burgers forward simulations*. [run in colab]

14.1 Initialization

phiflow directly gives us a sequence of differentiable operations, provided that we don't use the *numpy* backend. The important step here is to include `phi.tf.flow` instad of `phi.flow` (for *pytorch* you could use `phi.torch.flow`).

So, as a first step, let's set up some constants, and initialize a `velocity` field with zeros, and our constraint at $t = 0.5$ (step 16), now as a `CenteredGrid` in phiflow. Both are using periodic boundary conditions (via `extrapolation.PERIODIC`) and a spatial discretization of $\Delta x = 1/128$.

```
!pip install --upgrade --quiet phiflow==3.1
from phi.tf.flow import *

N = 128
DX = 2/N
STEPS = 32
DT = 1/STEPS
NU = 0.01/(N*np.pi)

# allocate velocity grid
velocity = CenteredGrid(0, extrapolation.PERIODIC, x=N, bounds=Box(x=(-1,1)))

# and a grid with the reference solution
REFERENCE_DATA = math.tensor([0.008612174447657694, 0.02584669669548606, 0.
 ↪043136357266407785, 0.060491074685516746, 0.07793926183951633, 0.0954779141740818,␣
 ↪0.11311894389663882, 0.1308497114054023, 0.14867023658641343, 0.1665634396808965, 0.
```

(continues on next page)

(continued from previous page)

```
↪18452263429574314, 0.20253084411376132, 0.22057828799835133, 0.23865132431365316, 0.
↪25673879161339097, 0.27483167307082423, 0.2929182325574904, 0.3109944766354339, 0.
↪3290477753208284, 0.34707880794585116, 0.36507311960102307, 0.38303584302507954, 0.
↪40094962955534186, 0.4188235294008765, 0.4366357052408043, 0.45439856841363885, 0.
↪4720845505219581, 0.4897081943759776, 0.5072391070000235, 0.5247011051514834, 0.
↪542067187709797, 0.5593576751669057, 0.5765465453632126, 0.5936507311857876, 0.
↪6106452944663003, 0.6275435911624945, 0.6443221318186165, 0.6609900633731869, 0.
↪67752574922899, 0.6939334022562877, 0.7101938106059631, 0.7263049537163667, 0.
↪7422506131457406, 0.7580207366534812, 0.7736033721649875, 0.7889776974379873, 0.
↪8041371279965555, 0.8190465276590387, 0.8337064887158392, 0.8480617965162781, 0.
↪8621229412131242, 0.8758057344502199, 0.8891341984763013, 0.9019806505391214, 0.
↪9143881632159129, 0.9261597966464793, 0.9373647624856912, 0.9476871303793314, 0.
↪9572273019669029, 0.9654367940878237, 0.9724097482283165, 0.9767381835635638, 0.
↪9669484658390122, 0.659083299684951, -0.659083180712816, -0.9669485121167052, -0.
↪9767382069792288, -0.9724097635533602, -0.9654367970450167, -0.9572273263645859, -0.
↪9476871280825523, -0.9373647681120841, -0.9261598056102645, -0.9143881718456056, -0.
↪9019807055316369, -0.8891341634240081, -0.8758057205293912, -0.8621229450911845, -0.
↪8480618138204272, -0.833706571569058, -0.8190466131476127, -0.8041372124868691, -0.
↪7889777195422356, -0.7736033858767385, -0.758020740007683, -0.7422507481169578, -0.
↪7263049162371344, -0.7101938950789042, -0.6939334061553678, -0.677525822052029, -0.
↪6609901538934517, -0.6443222327338847, -0.6275436932970322, -0.6106454472814152, -0.
↪5936507836778451, -0.5765466491708988, -0.5593578078967361, -0.5420672759411125, -0.
↪5247011730988912, -0.5072391580614087, -0.4897082914472909, -0.47208460952428394, -
↪0.4543985995006753, -0.4366355580500639, -0.41882350871539187, -0.40094955631843376,
↪ -0.38303594105786365, -0.36507302109186685, -0.3470786936847069, -0.
↪3290476440540586, -0.31099441589505206, -0.2929180880304103, -0.27483158663081614, -
↪0.2567388003912687, -0.2386513127155433, -0.22057831776499126, -0.20253089403524566,
↪ -0.18452269630486776, -0.1665634500729787, -0.14867027528284874, -0.
↪13084990929476334, -0.1131191325854089, -0.09547794429803691, -0.07793928430794522,␣
↪-0.06049114408297565, -0.0431364527809777, -0.025846763281087953, -0.
↪00861212501518312] , math.spatial('x'))
SOLUTION_T16 = CenteredGrid( REFERENCE_DATA, extrapolation.PERIODIC, x=N,␣
↪bounds=Box(x=(-1,1)))
```

Below we verify that the fields of our simulation are now backed by TensorFlow.

```
type(velocity.values.native())
```

```
tensorflow.python.framework.ops.EagerTensor
```

14.2 Gradients

The `math.gradient` operation of phiflow generates a gradient function for a scalar loss, and we use it below to compute gradients of a whole simulation with the chosen number of 32 time steps.

To use it for the Burgers case we need to compute an appropriate loss: we want the solution at $t = 0.5$ to match the reference data. Thus we simply compute an L^2 difference between step number 16 and our constraint array as `loss`. Afterwards, we evaluate the gradient function of the initial velocity state `velocity` with respect to this loss. Phiflow's `math.gradient` generates a function that returns a gradient for each parameter, and as we only have a single one in form of the velocity here, `grad[0]` represents the gradient for the initial velocity.

```
def loss_function(velocity):
    velocities = [velocity]
```

(continues on next page)

(continued from previous page)

```
    for time_step in range(STEPS):
        v1 = diffuse.explicit(1.0*velocities[-1], NU, DT, substeps=1)
        v2 = advect.semi_lagrangian(v1, v1, DT)
        velocities.append(v2)
    loss = field.l2_loss(velocities[16] - SOLUTION_T16)*2./N # MSE
    return loss, velocities

gradient_function = math.gradient(loss_function)

(loss,velocities), grad = gradient_function(velocity)

print('Loss: %f' % (loss))
```

```
Loss: 0.382915
```

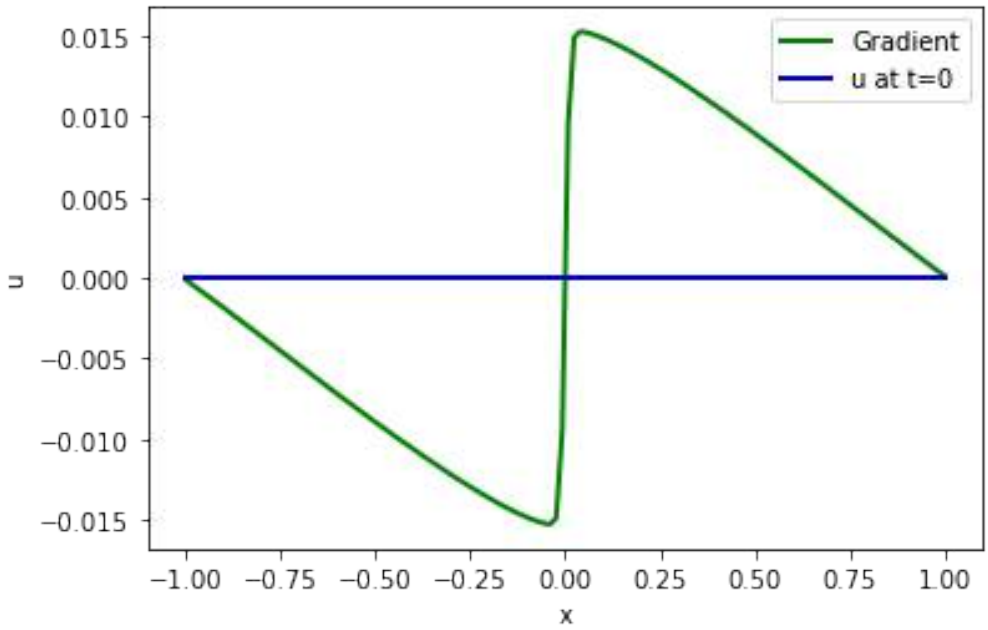
Because we're only constraining time step 16, we could actually omit steps 17 to 31 in this setup. They don't have any degrees of freedom and are not constrained in any way. However, for fairness regarding a comparison with the previous case, we include them.

Note that we've done a lot of calculations here: first the 32 steps of our simulation, and then another 16 steps backwards from the loss. They were recorded by the gradient tape, and used to backpropagate the loss to the initial state of the simulation.

Not surprisingly, because we're starting from zero, there's also a significant initial error of ca. 0.38 for the 16th simulation step.

So what do we get as a gradient here? It has the same dimensions as the velocity, and we can easily visualize it: Starting from the zero state for `velocity` (shown in blue), the first gradient is shown as a green line below. If you compare it with the solution it points in the opposite direction, as expected. The solution is much larger in magnitude, so we omit it here (see the next graph).

```
import pylab as plt

fig = plt.figure().gca()
pltx = np.linspace(-1,1,N)

# first gradient
fig.plot(pltx, grad[0].values.numpy('x')           , lw=2, color='green',      label=
 ↪"Gradient")
fig.plot(pltx, velocity.values.numpy('x'), lw=2, color='mediumblue', label="u at t=0")
plt.xlabel('x'); plt.ylabel('u'); plt.legend();

# some (optional) other fields to plot:
# fig.plot(pltx,  (velocities[16]).values.numpy('x')   , lw=2, color='cyan', label="u␣
 ↪at t=0.5")
# fig.plot(pltx,  (SOLUTION_T16).values.numpy('x')   , lw=2, color='red', label=
 ↪"solution at t=0.5")
# fig.plot(pltx,  (velocities[16] - SOLUTION_T16).values.numpy('x')   , lw=2, color=
 ↪'blue', label="difference at t=0.5")
```



This gives us a "search direction" for each velocity variable. Based on a linear approximation, the gradient tells us how to change each of them to increase the loss function (gradients *always* point "upwards"). Thus, we can use the gradient to run an optimization and find an initial state `velocity` that minimizes our loss.

14.3 Optimization

Equipped with the gradient we now run a gradient descent optimization. Below, we're using a learning rate of `LR=5`, and we're re-evaluating the loss for the updated state to track convergence.

In the following code block, we're additionally saving the gradients in a list called `grads`, such that we can visualize them later on. For a regular optimization we could of course discard the gradient after performing an update of the velocity.

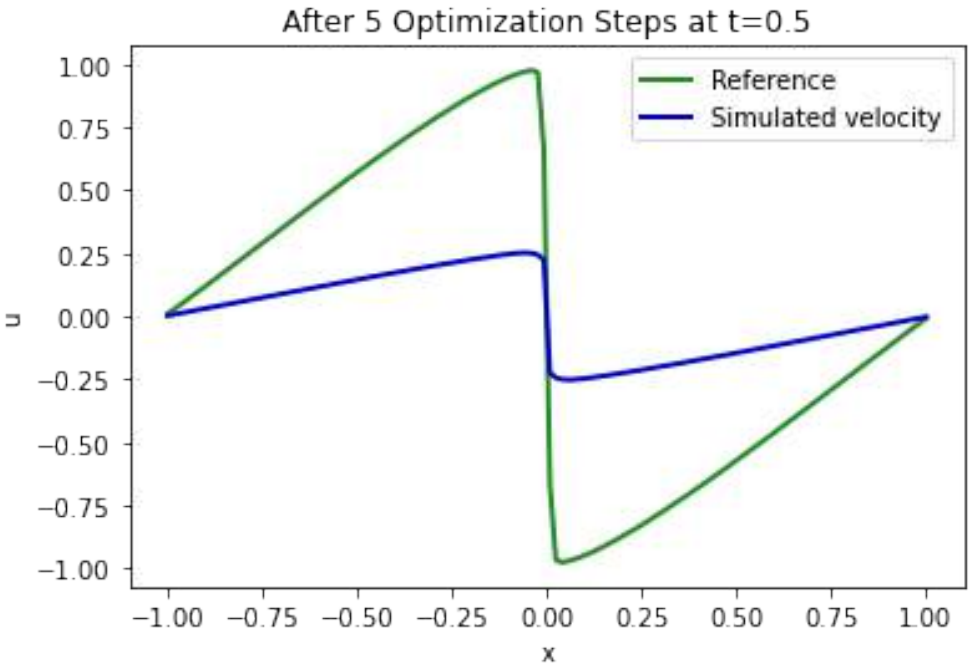
```
LR = 5.

grads=[]
for optim_step in range(5):
    (loss,velocities), grad = gradient_function(velocity)
    print('Optimization step %d, loss: %f' % (optim_step,loss))
    grads.append( grad[0] )

    velocity = velocity - LR * grads[-1]
```

```
Optimization step 0, loss: 0.382915
Optimization step 1, loss: 0.326882
Optimization step 2, loss: 0.281032
Optimization step 3, loss: 0.242804
Optimization step 4, loss: 0.210666
```

Now we'll check well the 16th state of the simulation actually matches the target after the 5 update steps. This is what the loss measures, after all. The next graph shows the constraints (i.e. the solution we'd like to obtain) in green, and the

reconstructed state after the initial state `velocity` (which we updated five times via the gradient by now) was updated 16 times by the solver.

```
fig = plt.figure().gca()

# target constraint at t=0.5
fig.plot(pltx, SOLUTION_T16.values.numpy('x'), lw=2, color='forestgreen',  label=
 ↪"Reference")

# optimized state of our simulation after 16 steps
fig.plot(pltx, velocities[16].values.numpy('x'), lw=2, color='mediumblue', label=
 ↪"Simulated velocity")

plt.xlabel('x'); plt.ylabel('u'); plt.legend(); plt.title("After 5 Optimization Steps␣
 ↪at t=0.5");
```



This seems to be going in the right direction! It's definitely not perfect, but we've only computed 5 GD update steps so far. The two peaks with a positive velocity on the left side of the shock and the negative peak on the right side are starting to show.

This is a good indicator that the backpropagation of gradients through all of our 16 simulated steps is behaving correctly, and that it's driving the solution in the right direction. The graph above only hints at how powerful the setup is: the gradient that we obtain from each of the simulation steps (and each operation within them) can easily be chained together into more complex sequences. In the example above, we're backpropagating through all 16 steps of the simulation, and we could easily enlarge this "look-ahead" of the optimization with minor changes to the code.

14.4 More optimization steps

Before moving on to more complex physics simulations, or involving NNs, let's finish the optimization task at hand, and run more steps to get a better solution.

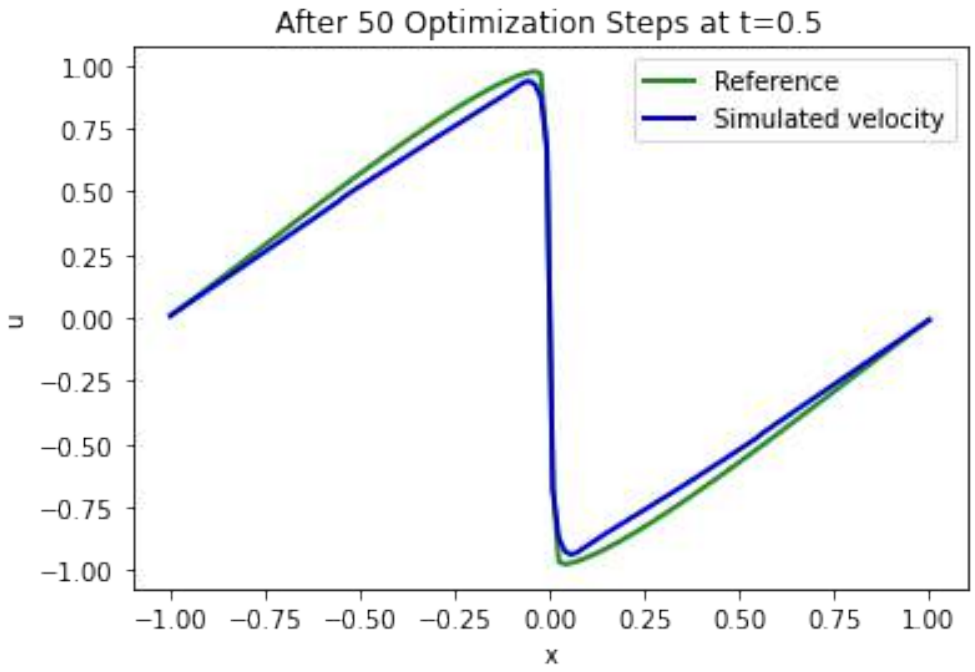
```
import time
start = time.time()

for optim_step in range(5,50):
    (loss,velocities), grad = gradient_function(velocity)
    velocity = velocity - LR * grad[0]
    if optim_step%5==0:
      print('Optimization step %d, loss: %f' % (optim_step,loss))

end = time.time()
print("Runtime {:.2f}s".format(end-start))
```

```
Optimization step 5, loss: 0.183476
Optimization step 10, loss: 0.096224
Optimization step 15, loss: 0.054792
Optimization step 20, loss: 0.032819
Optimization step 25, loss: 0.020334
Optimization step 30, loss: 0.012852
Optimization step 35, loss: 0.008185
Optimization step 40, loss: 0.005186
Optimization step 45, loss: 0.003263
Runtime 132.33s
```

Thinking back to the PINN version from *Burgers Optimization with a Differentiable Physics Gradient*, the error decreases much more strongly (by ca. two orders of magnitude) with a comparable runtime. This behavior stems from DP providing gradients for the whole solutions with all its discretization points and time steps, rather than localized updates.

Let's plot again how well our solution at $t = 0.5$ (blue) matches the constraints (green) now:

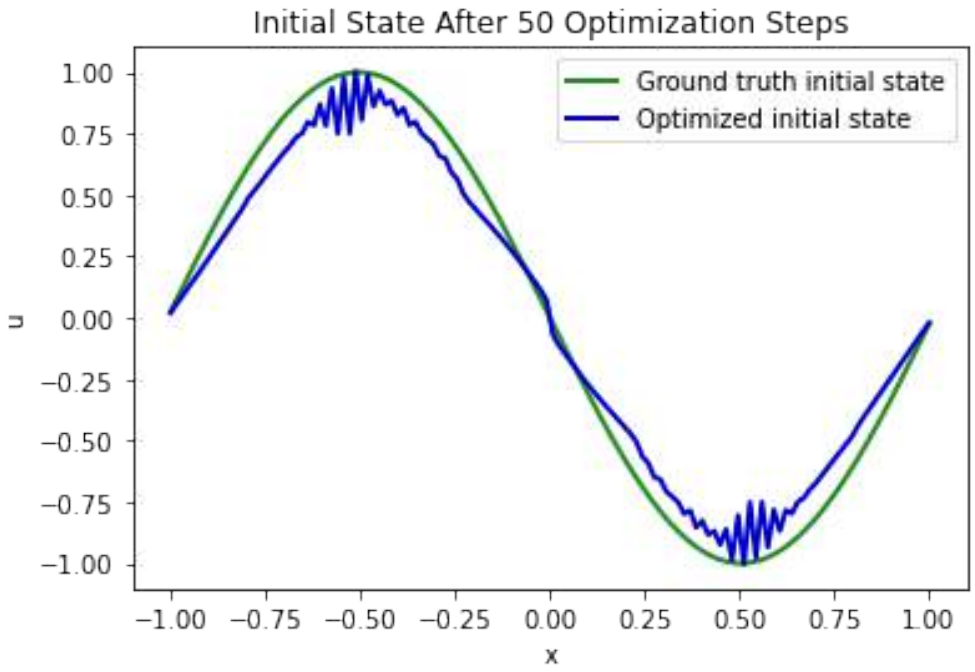
```
fig = plt.figure().gca()
fig.plot(pltx, SOLUTION_T16.values.numpy('x'), lw=2, color='forestgreen',  label=
 ↪"Reference")
fig.plot(pltx, velocities[16].values.numpy('x'), lw=2, color='mediumblue', label=
 ↪"Simulated velocity")
plt.xlabel('x'); plt.ylabel('u'); plt.legend(); plt.title("After 50 Optimization␣
 ↪Steps at t=0.5");
```



Not bad. But how well is the initial state recovered via backpropagation through the 16 simulation steps? This is what we're changing, and because it's only indirectly constrained via the observation later in time there is more room to deviate from a desired or expected solution.

This is shown in the next plot:

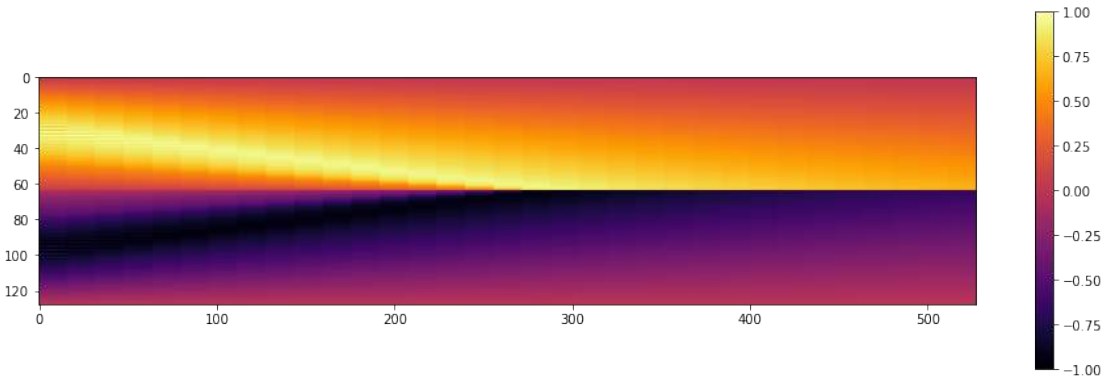
```
fig = plt.figure().gca()
pltx = np.linspace(-1,1,N)

# ground truth state at time=0 , move down
INITIAL_GT = np.asarray( [-np.sin(np.pi * x) for x in np.linspace(-1+DX/2,1-DX/2,N)]␣
↪) # 1D numpy array
fig.plot(pltx, INITIAL_GT.flatten()      , lw=2, color='forestgreen', label="Ground␣
↪truth initial state")  # ground truth initial state of sim
fig.plot(pltx, velocity.values.numpy('x'), lw=2, color='mediumblue',  label=
↪"Optimized initial state") # manual
plt.xlabel('x'); plt.ylabel('u'); plt.legend(); plt.title("Initial State After 50␣
↪Optimization Steps");
```



Naturally, this is a tougher task: the optimization receives direct feedback what the state at $t = 0.5$ should look like, but due to the non-linear model equation, we typically have a large number of solutions that exactly or numerically very closely satisfy the constraints. Hence, our minimizer does not necessarily find the exact state we started from (we can observe some numerical oscillations from the diffusion operator here with the default settings). However, the solution is still quite close in this Burgers scenario.

Before measuring the overall error of the reconstruction, let's visualize the full evolution of our system over time as this also yields the solution in the form of a numpy array that we can compare to the other versions:

```
import pylab

def show_state(a):
    a=np.expand_dims(a, axis=2)
    for i in range(4):
        a = np.concatenate( [a,a] , axis=2)
    a = np.reshape( a, [a.shape[0],a.shape[1]*a.shape[2]] )
    fig, axes = pylab.subplots(1, 1, figsize=(16, 5))
    im = axes.imshow(a, origin='upper', cmap='inferno')
    pylab.colorbar(im)

# get numpy versions of all states
vels = [ x.values.numpy('x,vector') for x in velocities]
# concatenate along vector/features dimension
vels = np.concatenate(vels, axis=-1)

# save for comparison with other methods
import os; os.makedirs("./temp",exist_ok=True)
np.savez_compressed("./temp/burgers-diffphys-solution.npz", np.reshape(vels,[N,
 ↪STEPS+1])) # remove batch & channel dimension

show_state(vels)
```



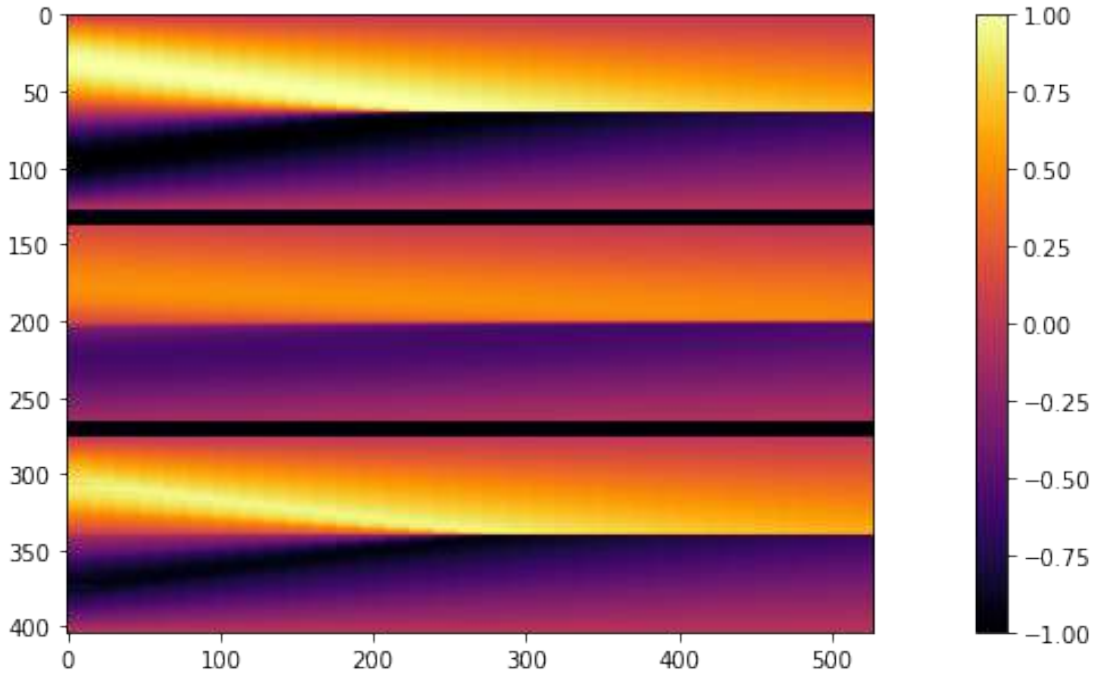
14.5 Physics-informed vs. differentiable physics reconstruction

Now we have both versions, the one with the PINN, and the DP version, so let's compare both reconstructions in more detail. (Note: The following cells expect that the Burgers-forward and PINN notebooks were executed in the same environment beforehand such that the `.npz` files in the `./temp' directory are available..)

Let's first look at the solutions side by side. The code below generates an image with 3 versions, from top to bottom: the "ground truth" (GT) solution as given by the regular forward simulation, in the middle the PINN reconstruction, and at the bottom the differentiable physics version.

```
# note, this requires previous runs of the forward-sim & PINN notebooks in the same␣
 ↪environment
sol_gt=npfile=np.load("./temp/burgers-groundtruth-solution.npz")["arr_0"]
sol_pi=npfile=np.load("./temp/burgers-pinn-solution.npz")["arr_0"]
sol_dp=npfile=np.load("./temp/burgers-diffphys-solution.npz")["arr_0"]

divider = np.ones([10,33])*-1. # we'll sneak in a block of -1s to show a black␣
 ↪divider in the image
sbs = np.concatenate( [sol_gt, divider, sol_pi, divider, sol_dp], axis=0)

print("\nSolutions Ground Truth (top), PINN (middle) , DiffPhys (bottom):")
show_state(np.reshape(sbs,[N*3+20,33,1]))
```

```
Solutions Ground Truth (top), PINN (middle) , DiffPhys (bottom):
```



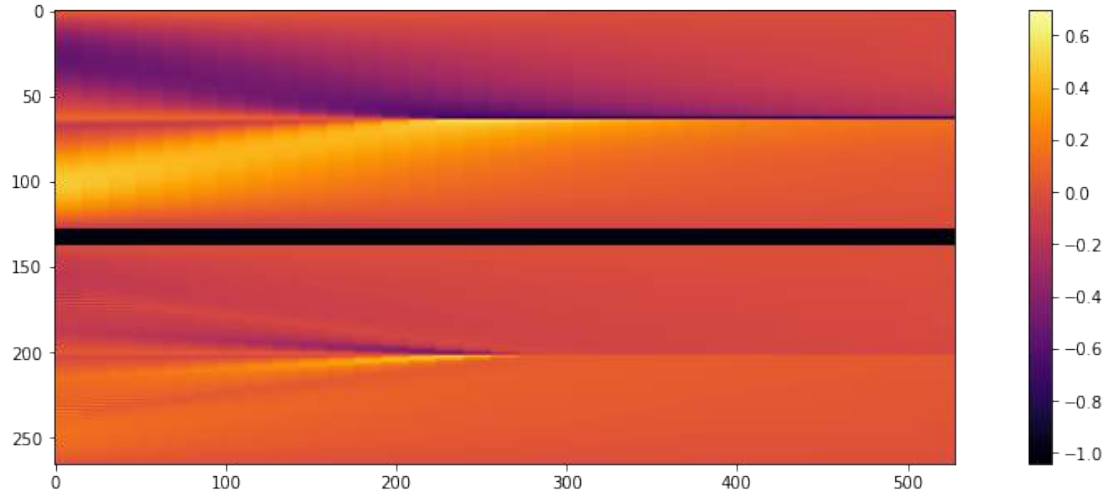
It's quite clearly visible here that the PINN solution (in the middle) recovers the overall shape of the solution, hence the temporal constraints are at least partially fulfilled. However, it doesn't manage to capture the amplitudes of the GT solution very well.

The reconstruction from the optimization with a differentiable solver (at the bottom) is much closer to the ground truth thanks to an improved flow of gradients over the whole course of the sequence. In addition, it can leverage the grid-based discretization for both forward as well as backward passes, and in this way provide a more accurate signal to the unknown initial state. It is nonetheless visible that the reconstruction lacks certain "sharper" features of the GT version, e.g., visible in the bottom left corner of the solution image.

Let's quantify these errors over the whole sequence:

```
err_pi = np.sum( np.abs(sol_pi-sol_gt)) / (STEPS*N)
err_dp = np.sum( np.abs(sol_dp-sol_gt)) / (STEPS*N)
print("MAE PINN: {:7.5f} \nMAE DP:   {:7.5f}".format(err_pi,err_dp))

print("\nError GT to PINN (top) , GT to DiffPhys (bottom):")
show_state(np.reshape( np.concatenate([sol_pi-sol_gt, divider, sol_dp-sol_gt],axis=0)␣
↪,[N*2+10,33,1]))
```

```
MAE PINN: 0.19298
MAE DP:   0.06382

Error GT to PINN (top) , GT to DiffPhys (bottom):
```



That's a pretty clear result: the PINN error is more than 3 times larger than the one from the Differentiable Physics (DP) reconstruction.

This difference also shows clearly in the jointly visualized image at the bottom: the magnitudes of the errors of the DP reconstruction are much closer to zero, as indicated by the purple color above.

A simple direct reconstruction problem like this one is always a good initial test for a DP solver. It can be tested independently before moving on to more complex setups, e.g., coupling it with an NN. If the direct optimization does not converge, there's probably still something fundamentally wrong, and there's no point involving an NN.

Now we have a first example to show similarities and differences of the two approaches. In the next section, we'll present a discussion of the findings so far, before moving to more complex cases in the following chapter.

14.6 Next steps

As before, there's variety of things that can be improved and experimented with using the code above:

- You can try to adjust the training parameters to further improve the reconstruction.
- Activate a different optimizer, and observe the changing (not necessarily improved) convergence behavior.
- Vary the number of steps, or the resolution of the simulation and reconstruction.
- Try adding `@jit_compile` in a line before `loss_function`. This will incude a one-time compilation cost, but greatly speed up the optimization.

CHAPTER

FIFTEEN

SO FAR SO GOOD - A FIRST DISCUSSION

In the previous section we've seen an example reconstructions that used physical residuals as soft constraints, in the form of the variant 2 (PINNs), and reconstructions that used a differentiable physics (DP) solver. While both methods can find minimizers for similar inverse problems, the obtained solutions differ substantially, as does the behavior of the non-linear optimization problem that we get from each formulation. In the following we discuss these differences in more detail, and we will combine conclusions drawn from the behavior of the Burgers case of *Burgers Optimization with a PINN* and *Burgers Optimization with a Differentiable Physics Gradient* with observations from external research papers [HKT19].

15.1 Compatibility with existing numerical methods

It is very obvious that the PINN implementation is quite simple, which is a positive aspect, but at the same time it differs strongly from "typical" discretizations and solution approaches that are usually employed to solve PDEs like Burgers equation. The derivatives are computed via the neural network, and hence rely on a fairly accurate representation of the solution to provide a good direction for optimization problems.

The DP version on the other hand inherently relies on a numerical solver that is tied into the learning process. As such it requires a discretization of the problem at hand, and via this discretization can employ existing, and potentially powerful numerical techniques. This means solutions and derivatives can be evaluated with known and controllable accuracy, and can be evaluated efficiently.

15.2 Discretization

The reliance on a suitable discretization requires some understanding and knowledge of the problem under consideration. A sub-optimal discretization can impede the learning process or, worst case, lead to diverging training runs. However, given the large body of theory and practical realizations of stable solvers for a wide variety of physical problems, this is typically not an unsurmountable obstacle.

The PINN approaches on the other hand do not require an a-priori choice of a discretization, and as such seems to be "discretization-less". This, however, is only an advantage on first sight. By now, researchers are trying to "re-integrate" discretizations into PINN training. Generally, PINNs inevitably yield solutions in a computer and thus *have* to discretize the problem. They construct this discretization over the course of the training process, in a way that lies at the mercy of the underlying nonlinear optimization, and is not easily controllable from the outside. Thus, the resulting accuracy is determined by how well the training manages to estimate the complexity of the problem for realistic use cases, and how well the training data approximates the unknown regions of the solution.

E.g., as demonstrated with the Burgers example, the PINN solutions typically have significant difficulties propagating information *backward* in time. This is closely coupled to the efficiency of the method.

15.3 Efficiency

The PINN approach also results in fundamentally more difficult training tasks that causes convergence problems. PINNs typically perform a localized sampling and correction of the solutions, which means the corrections in the form of weight updates are likewise typically local. The fulfillment of boundary conditions in space and time can be correspondingly slow, leading to long training runs in practice.

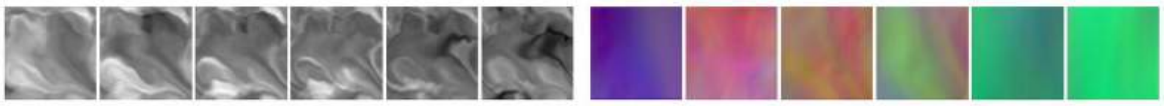
A well-chosen discretization of a DP approach can remedy this behavior, and provide an improved flow of gradient information. At the same time, the reliance on a computational grid means that solutions can be obtained very quickly. Given an interpolation scheme or a set of basis functions, the solution can be sampled at any point in space or time given a very local neighborhood of the computational grid. Worst case, this can lead to slight memory overheads, e.g., by repeatedly storing mostly constant values of a solution.

For the PINN representation with fully-connected networks on the other hand, we need to make a full pass over the potentially large number of values in the whole network to obtain a sample of the solution at a single point. The network effectively needs to encode the full high-dimensional solution, and its size likewise determines the efficiency of derivative calculations.

15.4 Efficiency continued

That being said, because the DP approaches can cover much larger solution manifolds, the structure of these manifolds is typically also difficult to learn. E.g., when training a network with a larger number of iterations (i.e. a long look-ahead into the future), this typically represents a signal that is more difficult to learn than a short look ahead.

As a consequence, these training runs not only take more computational resources per NN iteration, they also often need longer to converge. Regarding resources, each computation of the look-ahead potentially requires a large number of simulation steps, and typically a similar amount of resources for the backpropagation step. Thus, while they may seem costly and slow to converge at times, this is usually caused by the the more complex signal that needs to be learned.



15.5 Summary

The following table summarizes these pros and cons of physics-informed (PI) and differentiable physics (DP) approaches:

Method	? Pro	? Con
PI	- Analytic derivatives via backpropagation.	- Expensive evaluation of NN, and very costly derivative calculations.
	- Easy to implement.	- Incompatible with existing numerical methods.
		- No control of discretization.
DP	- Leverage existing numerical methods.	- More complicated implementation.
	- Efficient evaluation of simulation and derivatives.	- Require understanding of problem to choose suitable discretization.

As a summary, both methods are definitely interesting, and have a lot of potential. There are numerous more complicated extensions and algorithmic modifications that change and improve on the various negative aspects we have discussed for both sides.

However, as of this writing, the PINN approach has clear limitations when it comes to performance and compatibility with existing numerical methods. Thus, when knowledge of the problem at hand is available, which typically is the case when we choose a suitable PDE model to constrain the learning process, employing a differentiable physics solver to train Neural operators can significantly improve the training process as well as the quality of the obtained solution. So, in the following we'll focus on DP variants, and illustrate their capabilities with more complex scenarios in the next chapters. First, we'll consider a case that very efficiently computes space-time gradients for a transient fluid simulations.

CHAPTER

SIXTEEN

DIFFERENTIABLE FLUID SIMULATIONS

We now target a more complex example with the Navier-Stokes equations as physical model. In line with *Navier-Stokes Forward Simulation*, we'll target a 2D case.

As optimization objective we'll consider a more difficult variant of the previous Burgers example: the state of the observed density s should match a given target after $n = 20$ steps of simulation. In contrast to before, the observed quantity in the form of the marker field s cannot be changed in any way. Only the initial state of the velocity $\mathbf{u}_0$ at $t = 0$ can be modified. This gives us a split between observable quantities for the loss formulation and quantities that we can interact with during the optimization (or later on via NNs). [run in colab]

16.1 Physical Model

We'll use an inviscid Navier-Stokes model with velocity $\mathbf{u}$, no explicit viscosity term, and a smoke marker density s that drives a simple Boussinesq buoyancy term ηd adding a force along the y dimension. Due to a lack of an explicit viscosity, the equations are equivalent to the Euler equations. This gives:

$$\begin{aligned}
\frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x &= -\frac{1}{\rho} \nabla p \\
\frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y &= -\frac{1}{\rho} \nabla p + \eta d \\
\text{s.t.} \quad \nabla \cdot \mathbf{u} &= 0,
\end{aligned}$$

With an additional transport equation for the passively advected marker density s:

$$\frac{\partial s}{\partial t} + \mathbf{u} \cdot \nabla s = 0$$

16.2 Formulation

With the notation from *Models and Equations* the inverse problem outlined above can be formulated as a minimization problem

$$\arg\min_{\mathbf{u}_0} \sum_i \left(f(x_{t_e,i}; \mathbf{u}_0) - y^*_{t_e,i} \right)^2,$$

where $y^*_{t_e,i}$ are samples of the reference solution at a targeted time t_e, and $x_{t_e,i}$ denotes the estimate of our simulator at the same sampling locations and time. The index i here runs over all discretized, spatial degrees of freedom in our fluid solver (we'll have 32×40 below).

In contrast to before, we are not dealing with pre-computed quantities anymore, but now $x_{t_e,i}$ is a complex, non-linear function itself. More specifically, the simulator starts with the initial velocity $\mathbf{u}_0$ and density s_0 to compute the $x_{t_e,i}$, by n

evaluations of the discretized PDE $\mathcal{P}$. This gives as simulated final state $y_{t_e,i} = s_{t_e} = \mathcal{P}^n(\mathbf{u}_0, s_0)$, where we will leave s_0 fixed in the following, and focus on $\mathbf{u}_0$ as our degrees of freedom. Hence, the optimization can only change $\mathbf{u}_0$ to align $y_{t_e,i}$ with the references $y^*_{t_e,i}$ as closely as possible.

16.3 Starting the Implementation

First, let's get the loading of python modules out of the way. By importing `phi.torch.flow`, we get fluid simulation functions that work within pytorch graphs and can provide gradients (`phi.tf.flow` would be the alternative for tensorflow).

```
!pip install --upgrade --quiet phiflow==3.1
from phi.torch.flow import *
import pylab # for visualizations later on
```

16.4 Batched simulations

Now we can set up the simulation, which will work in line with the previous "regular" simulation example from the *Navier-Stokes Forward Simulation*. However, now we'll directly include an additional dimension, similar to a mini-batch used for NN training. For this, we'll introduce a named dimension called `inflow_loc`. This dimension will exist "above" the previous spatial dimensions `y`, `x` which are declared as dimensions for the `vector` channel. As indicated by the name `inflow_loc`, the main differences for this dimension will lie in different locations of the inflow, in order to obtain different flow simulations. The named dimensions in phiflow make it very convenient to broadcast information across matching dimensions in different tensors.

```
# closed domain
INFLOW_LOCATION = tensor([(12, 4), (13, 6), (14, 5), (16, 5)], batch('inflow_loc'),␣
 ↪channel(vector="x,y"))
INFLOW = (1./3.) * CenteredGrid(Sphere(center=INFLOW_LOCATION, radius=3),␣
 ↪extrapolation.BOUNDARY, x=32, y=40, bounds=Box(x=(0,32), y=(0,40)))
BND = extrapolation.ZERO # closed, boundary conditions for velocity grid below

# uncomment this for a slightly different open domain case
#INFLOW_LOCATION = tensor([(11, 6), (12, 4), (14, 5), (16, 5)], batch('inflow_loc'),␣
 ↪channel(vector="x,y"))
#INFLOW = (1./4.) * CenteredGrid(Sphere(center=INFLOW_LOCATION, radius=3),␣
 ↪extrapolation.BOUNDARY, x=32, y=40, bounds=Box(x=(0,32), y=(0,40)))
#BND = extrapolation.BOUNDARY # open boundaries

INFLOW.shape
```

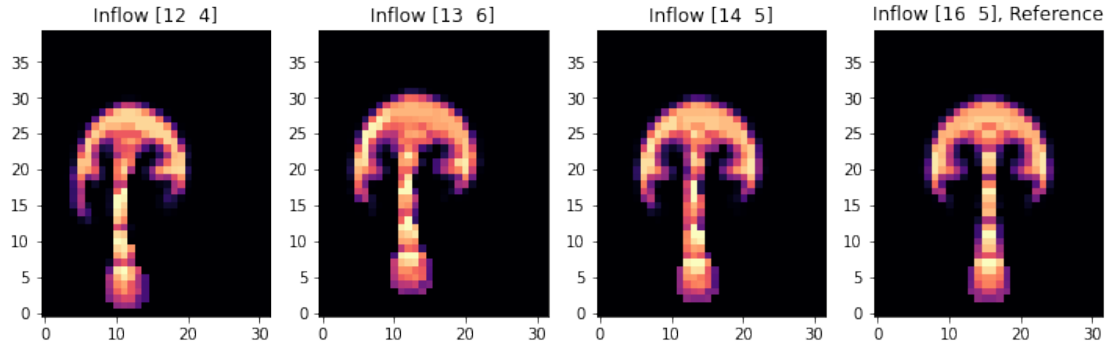
```
(inflow_loc=4, x=32, y=40)
```

The last statement verifies that our `INFLOW` grid likewise has an `inflow_loc` dimension in addition to the spatial dimensions `x` and `y`. You can test for the existence of a tensor dimension in phiflow with the `.exists` boolean, which can be evaluated for any dimension name. E.g., above `INFLOW.inflow_loc.exists` will give `True`, while `INFLOW.some_unknown_dim.exists` will give `False`. The ^{b} superscript indicates that `inflow_loc` is a batch dimension.

Phiflow tensors are automatically broadcast to new dimensions via their names, and hence typically no-reshaping operations are required. E.g., you can easily add or multiply tensors with differing dimensions. Below we'll multiply a staggered grid with a tensor of ones along the `inflow_loc` dimension to get a staggered velocity that has `x,y,inflow_loc` as dimensions via `StaggeredGrid(...) * math.ones(batch(inflow_loc=4))`.

We can easily simulate a few steps now starting with these different initial conditions. Thanks to the broadcasting, the exact same code we used for the single forward simulation in the overview chapter will produce four simulations with different smoke inflow positions.

```
smoke = CenteredGrid(0, extrapolation.BOUNDARY, x=32, y=40, bounds=Box(x=(0,32), y=(0,
 ↪40)))  # sampled at cell centers
velocity = StaggeredGrid(0, BND, x=32, y=40, bounds=Box(x=(0,32), y=(0,40)))  #␣
 ↪sampled in staggered form at face centers

def step(smoke, velocity):
  smoke = advect.mac_cormack(smoke, velocity, dt=1) + INFLOW
  buoyancy_force = (smoke * (0, 1)).at(velocity)
  velocity = advect.semi_lagrangian(velocity, velocity, dt=1) + buoyancy_force
  velocity, _ = fluid.make_incompressible(velocity)
  return smoke, velocity

for _ in range(20):
    smoke,velocity = step(smoke,velocity)

# store and show final states (before optimization)
smoke_final = smoke
fig, axes = pylab.subplots(1, 4, figsize=(10, 6))
for i in range(INFLOW.shape.get_size('inflow_loc')):
  axes[i].imshow(smoke_final.values.numpy('inflow_loc,y,x')[i,...], origin='lower',␣
 ↪cmap='magma')
  axes[i].set_title(f"Inflow {INFLOW_LOCATION.numpy('inflow_loc,vector')[i]}" + (",␣
 ↪Reference" if i==3 else ""))
pylab.tight_layout()
```



The last image shows the state of the advected smoke fields after 20 simulation steps. The final smoke shape of simulation `[3]` with an inflow at `(16,5)`, with the straight plume on the far right, will be our **reference state** below. The initial velocity of the other three will be modified in the optimization procedure below to match this reference.

(As a small side note: phiflow tensors will keep track of their chain of operations using the backend they were created for. E.g. a tensor created with NumPy will keep using NumPy/SciPy operations unless a PyTorch or TensorFlow tensor is also passed to the same operation. Thus, it is a good idea to verify that tensors are using the right backend once in a while, e.g., via `GRID.values.default_backend`.)

16.5 Gradients

Let's look at how to get gradients from our simulation. The first trivial step taken care of above was to include `phi.torch.flow` to import differentiable operators from which to build our simulator.

Now we want to optimize the initial velocities so that all simulations arrive at a final state that is similar to the simulation on the right, where the inflow is located at `(16, 5)`, i.e. centered along `x`. To achieve this, we record the gradients during the simulation and define a simple L^2 based loss function. The loss function we'll use is given by $L = |s_{t_e} - s^*_{t_e}|^2$, where s_{t_e} denotes the smoke density, and $s^*_{t_e}$ denotes the reference state from the fourth simulation in our batch (both evaluated at the last time step t_e). When evaluating the loss function we treat the reference state as an external constant via `field.stop_gradient()`. As outlined at the top, s is a function of $\mathbf{u}$ (via the advection equation), which in turn is given by the Navier-Stokes equations. Thus, via a chain of multiple time steps s depends in the initial velocity state $\mathbf{u}_0$.

It is important that our initial velocity has the `inflow_loc` dimension before we record the gradients, such that we have the full "mini-batch" of four versions of our velocity (three of which will be updated via gradients in our optimization later on). To get the appropriate velocity tensor, we initialize a `StaggeredGrid` with a tensor of zeros along the `inflow_loc` batch dimension. As the staggered grid already has `y,x` and `vector` dimensions, this gives the desired four dimensions, as verified by the print statement below.

Phiflow provides a unified API for gradients across different platforms by using functions that need to return a loss values, in addition to optional state values. It uses a loss function based interface, for which we define the `simulate` function below. `simulate` computes the L^2 error outlined above and returns the evolved `smoke` and `velocity` states after 20 simulation steps.

```
initial_smoke = CenteredGrid(0, extrapolation.BOUNDARY, x=32, y=40, bounds=Box(x=(0,
 ↪32), y=(0,40)))
initial_velocity = StaggeredGrid(math.zeros(batch(inflow_loc=4)), BND, x=32, y=40,␣
 ↪bounds=Box(x=(0,32), y=(0,40)))
print("Velocity dimensions: "+format(initial_velocity.shape))

def simulate(smoke: CenteredGrid, velocity: StaggeredGrid):
    for _ in range(20):
        smoke,velocity = step(smoke,velocity)

    loss = field.l2_loss(smoke - field.stop_gradient(smoke.inflow_loc[-1]) )
    # optionally, use smoother loss with diffusion steps - no difference here, but␣
 ↪can be useful for more complex cases
    #loss = field.l2_loss(diffuse.explicit(smoke - field.stop_gradient(smoke.inflow_
 ↪loc[-1]), 1, 1, 10))

    return loss, smoke, velocity
```

```
Velocity dimensions: (inflow_loc⍰=4, xˢ=32, yˢ=40, vector⍰=x,y)
```

Phiflow's `field.gradient()` function is the central function to compute gradients. Next, we'll use it to obtain the gradient with respect to the initial velocity. Since the velocity is the second argument of the `simulate()` function, we pass `wrt=[1]`. (Phiflow also has a `field.spatial_gradient` function which instead computes derivatives of tensors along spatial dimensions, like `x,y`.)

`gradient` generates a gradient function. As a demonstration, the next cell evaluates the gradient once with the initial states for smoke and velocity. The last statement prints a summary of a part of the resulting gradient tensor.

```
sim_grad = field.gradient(simulate, wrt=[1], get_output=False)
(velocity_grad,) = sim_grad(initial_smoke, initial_velocity)
```

(continues on next page)

(continued from previous page)

```
print("Some gradient info: " + format(velocity_grad))
print(format(velocity_grad.values.inflow_loc[0].vector[0])) # one example, location 0,
↪ x component, automatically prints size & content range
```

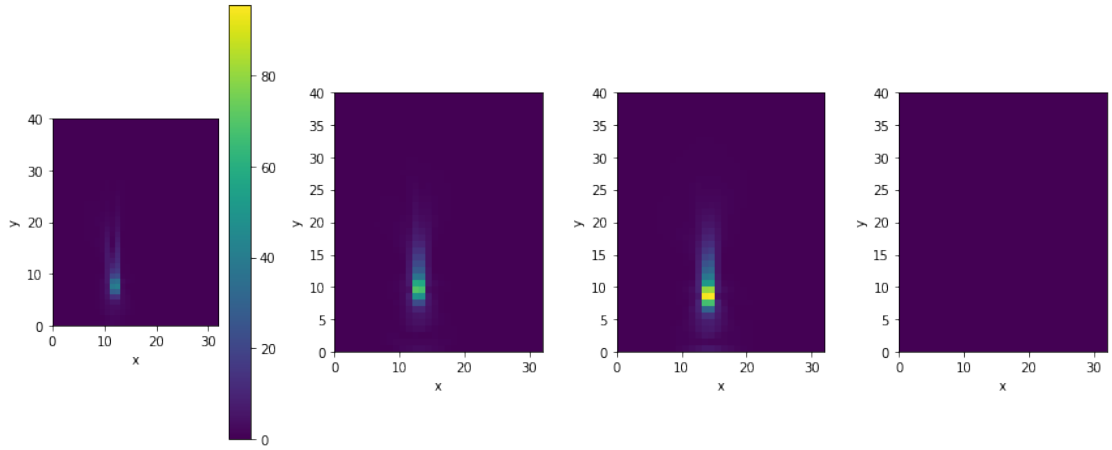
```
Some gradient info: StaggeredGrid[(inflow_loc⍰=4, xˢ=32, yˢ=40, vector⍰=x,y),␣
↪size=(x=32, y=40) int64, extrapolation=0]
(xˢ=31, yˢ=40) 2.61e-08 ± 8.5e-01 (-2e+01...1e+01)
```

The last two lines just print some information about the resulting gradient field. Naturally, it has the same shape as the velocity itself: it's a staggered grid with four inflow locations. The last line shows how to access the x-components of one of the gradients.

We could use this to take a look at the content of the computed gradient with regular plotting functions, e.g., by converting the x component of one of the simulations to a numpy array via `velocity_grad.values.inflow_loc[0].vector[0].numpy('y,x')`. An interactive alternative would be phiflow's `view()` function, which automatically analyzes the grid content and provides UI buttons to choose different viewing modes. You can use them to show arrows, single components of the 2-dimensional velocity vectors, or their magnitudes. Because of its interactive nature, the corresponding image won't show up outside of Jupyter, though, and hence we're showing the vector length below via `plot()` instead.

```
# neat phiflow helper function:
v = vis.plot(field.vec_length(velocity_grad)) # show magnitude
```

```
<Figure size 864x360 with 5 Axes>
```



Not surprisingly, the fourth gradient on the left is zero (it's already matching the reference). The other three gradients have detected variations for the initial round inflow positions shown as positive and negative regions around the circular shape of the inflow. The ones for the larger distances on the left are also noticeably larger.

16.6 Optimization

The gradient visualized above is just the linearized change that points in the direction of an increasing loss. Now we can proceed by updating the initial velocities in the opposite direction to minimize the loss, and iterate to find a minimizer.

This is a difficult task: the simulation is producing different dynamics due to the differing initial spatial density configuration. Our optimization should now find a single initial velocity state that gives the same state as the reference simulation at $t = 20$. Thus, after 20 non-linear update steps the simulation should reproduce the desired marker density state. It would be much easier to simply change the position of the marker inflow to arrive at this goal, but – to make things more difficult and interesting here – the inflow is *not* a degree of freedom. The optimizer can only change the initial velocity $\mathbf{u}_0$.

The following cell implements a simple steepest gradient descent optimization: it re-evaluates the gradient function, and iterates several times to optimize $\mathbf{u}_0$ with a learning rate (step size) `LR`.

`field.gradient` has a parameter `get_output` that determines whether the original results of the function (`simulate()` in our case) are returned, or only the gradient. As it's interesting to track how the loss evolves over the course of the iterations, let's redefine the gradient function with `get_output=True`.

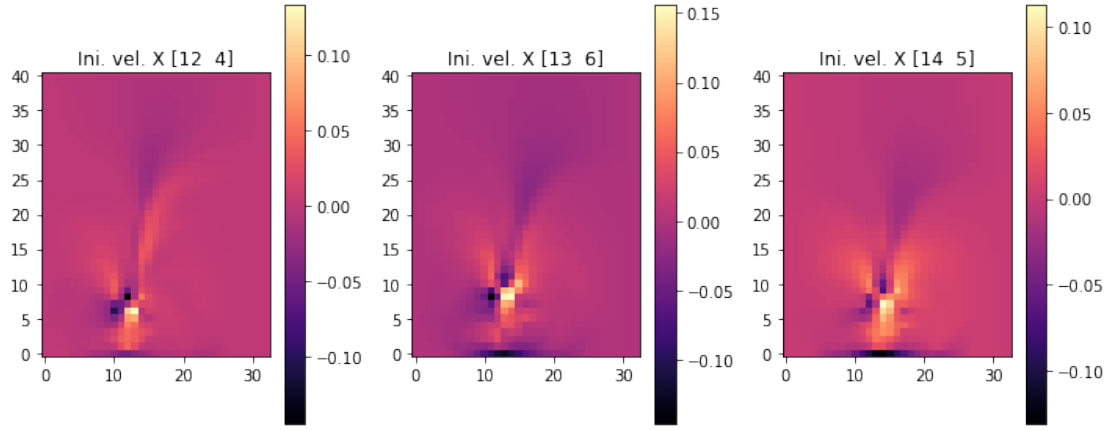
```
sim_grad_wloss = field.gradient(simulate, wrt=[1], get_output=True) # if we need␣
↪outputs...

LR = 1e-03
for optim_step in range(80):
    (loss, _smoke, _velocity), (velocity_grad,) = sim_grad_wloss(initial_smoke,␣
↪initial_velocity)
    initial_velocity = initial_velocity - LR * velocity_grad
    if optim_step<3 or optim_step%10==9: print('Optimization step %d, loss: %f' %␣
↪(optim_step, np.sum(loss.numpy()) ))
```

```
Optimization step 0, loss: 298.286163
Optimization step 1, loss: 291.454376
Optimization step 2, loss: 276.057831
Optimization step 9, loss: 233.706482
Optimization step 19, loss: 232.652145
Optimization step 29, loss: 178.186951
Optimization step 39, loss: 176.523254
Optimization step 49, loss: 169.360931
Optimization step 59, loss: 167.578674
Optimization step 69, loss: 175.005310
Optimization step 79, loss: 169.943680
```

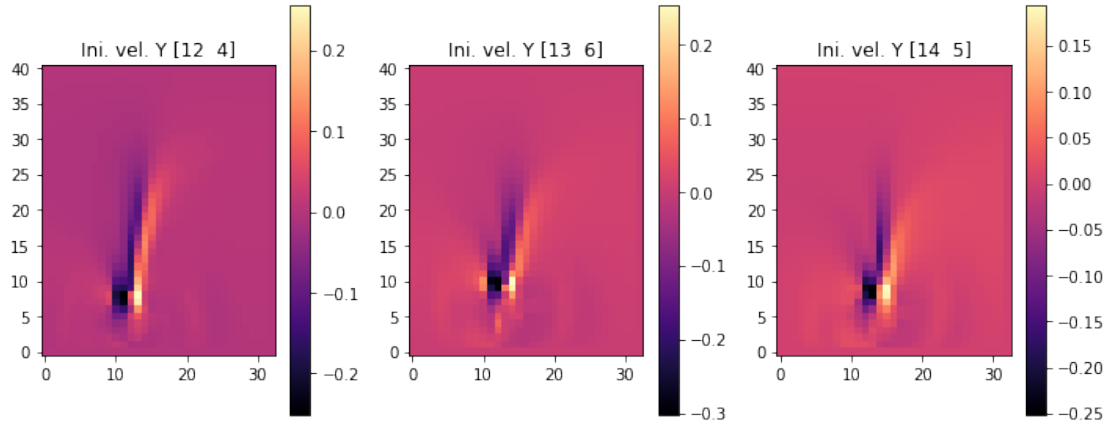
The loss should have gone down significantly, from almost 300 to below 170, and now we can also visualize the initial velocities that were obtained in the optimization.

The following images show the resulting three initial velocities in terms of their x (first set), and y components (second set of images). We're skipping the fourth set with `inflow_loc[0]` because it only contains zeros.

```
fig, axes = pylab.subplots(1, 3, figsize=(10, 4))
for i in range(INFLOW.shape.get_size('inflow_loc')-1):
  im = axes[i].imshow(initial_velocity.staggered_tensor().numpy('inflow_loc,y,x,vector
↪')[i,...,0], origin='lower', cmap='magma')
  axes[i].set_title(f"Ini. vel. X {INFLOW_LOCATION.numpy('inflow_loc,vector')[i]}")
  pylab.colorbar(im,ax=axes[i])
pylab.tight_layout()
```




```
fig, axes = pylab.subplots(1, 3, figsize=(10, 4))
for i in range(INFLOW.shape.get_size('inflow_loc')-1):
  im = axes[i].imshow(initial_velocity.staggered_tensor().numpy('inflow_loc,y,x,vector
↪')[i,...,1], origin='lower', cmap='magma')
  axes[i].set_title(f"Ini. vel. Y {INFLOW_LOCATION.numpy('inflow_loc,vector')[i]}")
  pylab.colorbar(im,ax=axes[i])
pylab.tight_layout()
```



16.7 Re-simulation

We can also visualize how the full simulation over the course of 20 steps turns out, given the new initial velocity conditions for each of the inflow locations. This is what happened internally at optimization time for every gradient calculation, and what was measured by our loss function. Hence, it's good to get an intuition for which solutions the optimization has found.

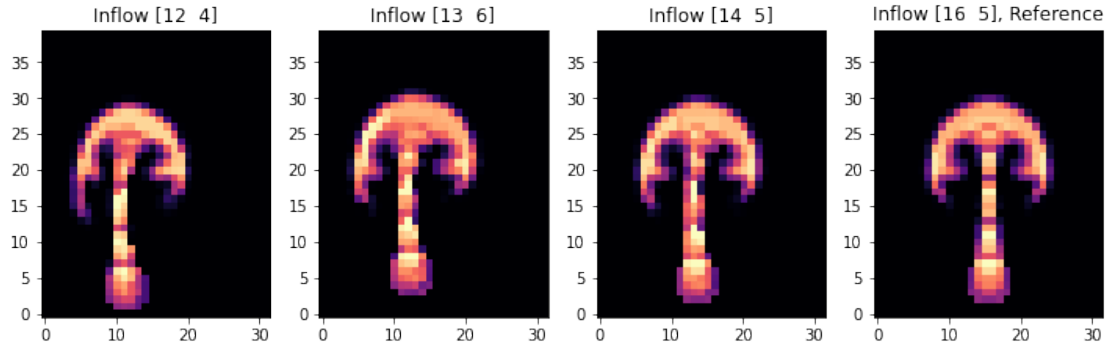
Below, we re-run the forward simulation with the new initial conditions from `initial_velocity`:

```
smoke = initial_smoke
velocity = initial_velocity

for _ in range(20):
    smoke,velocity = step(smoke,velocity)
```

(continues on next page)

(continued from previous page)

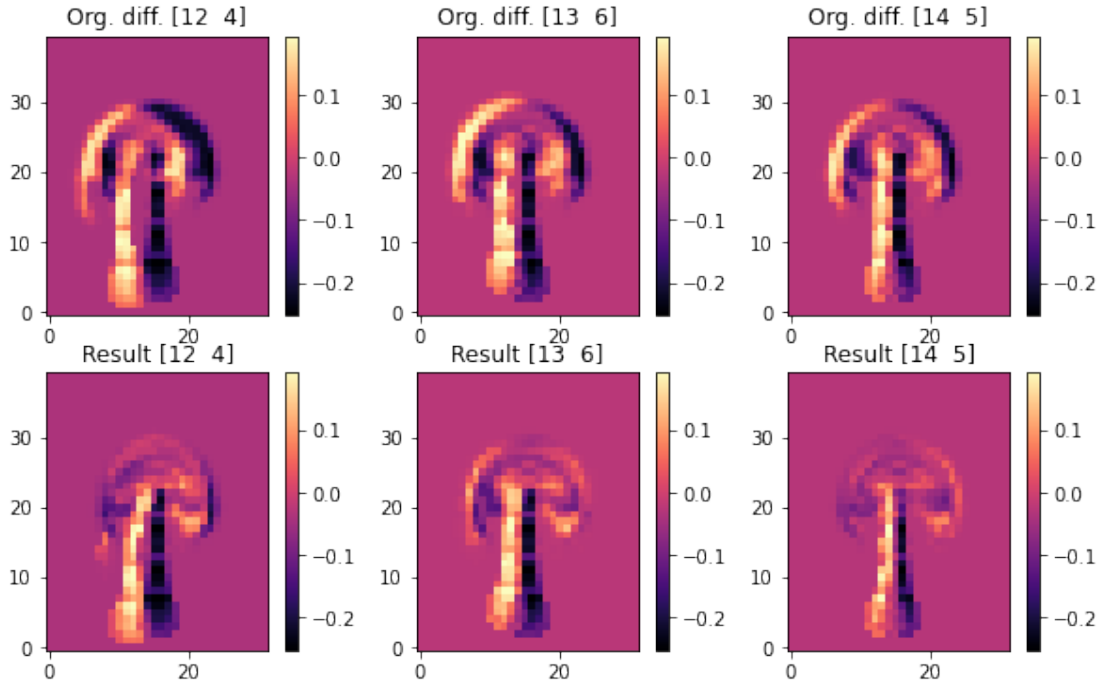
```
fig, axes = pylab.subplots(1, 4, figsize=(10, 6))
for i in range(INFLOW.shape.get_size('inflow_loc')):
  axes[i].imshow(smoke_final.values.numpy('inflow_loc,y,x')[i,...], origin='lower',␣
 ↪cmap='magma')
  axes[i].set_title(f"Inflow {INFLOW_LOCATION.numpy('inflow_loc,vector')[i]}" + (",␣
 ↪Reference" if i==3 else ""))
pylab.tight_layout()
```



Naturally, the image on the right is the same (this is the reference), and the other three simulations now exhibit a shift towards the right. As the differences are a bit subtle, let's visualize the difference between the target configuration and the different final states.

The following images contain the difference between the evolved simulated and target density. Hence, dark regions indicate where the target should be, but isn't. The top row shows the original states with the initial velocity being zero, while the bottom row shows the versions after the optimization has tuned the initial velocities. Hence, in each column you can compare before (top) and after (bottom):

```
fig, axes = pylab.subplots(2, 3, figsize=(10, 6))
for i in range(INFLOW.shape.get_size('inflow_loc')-1):
  axes[0,i].imshow(smoke_final.values.numpy('inflow_loc,y,x')[i,...] - smoke_final.
 ↪values.numpy('inflow_loc,y,x')[3,...], origin='lower', cmap='magma')
  axes[0,i].set_title(f"Org. diff. {INFLOW_LOCATION.numpy('inflow_loc,vector')[i]}")
  pylab.colorbar(im,ax=axes[0,i])
for i in range(INFLOW.shape.get_size('inflow_loc')-1):
  axes[1,i].imshow(smoke.values.numpy('inflow_loc,y,x')[i,...] - smoke_final.values.
 ↪numpy('inflow_loc,y,x')[3,...], origin='lower', cmap='magma')
  axes[1,i].set_title(f"Result {INFLOW_LOCATION.numpy('inflow_loc,vector')[i]}")
  pylab.colorbar(im,ax=axes[1,i])
```



These difference images clearly show that the optimization managed to align the upper region of the plumes very well. Each original image (at the top) shows a clear misalignment in terms of a black halo, while the states after optimization largely overlap the target smoke configuration of the reference, and exhibit differences closer to zero for the front of each smoke cloud.

Note that all three simulations need to "work" with a fixed inflow, hence they cannot simply "produce" marker density out of the blue to match the target. Also each simulation needs to take into account how the non-linear model equations change the state of the system over the course of 20 time steps. So the optimization goal is quite difficult, and it is not possible to exactly satisfy the constraints to match the reference simulation in this scenario. E.g., this is noticeable at the stems of the smoke plumes, which still show a black halo after the optimization. The optimization was not able to shift the inflow position, and hence needs to focus on aligning the upper regions of the plumes.

16.8 Conclusions

This example illustrated how the differentiable physics approach can easily be extended towards significantly more complex PDEs. Above, we've optimized for a mini-batch of 20 steps of a full Navier-Stokes solver. ⍰

This is a powerful basis to bring NNs into the picture. Above, the degrees of freedom were still a regular grid, and we've jointly solved a single inverse problem. There were three cases to solve as a mini-batch, of course, but nonetheless the setup still represents a direct *optimization*. Thus, in line with the PINN example of *Burgers Optimization with a PINN* we've not really dealt with a *machine learning* task here. However, DP training allows for a range of flexible combinations with NNs that will be the topic of the next chapters.

16.9 Next steps

Based on the code example above, we can recommend experimenting with the following:

- Modify the setup of the simulation to differ more strongly across the four instances, run longer, or use a finer spatial discretization (i.e. larger grid size). Note that this will make the optimization problem tougher, and hence might not converge directly with this simple setup.
- As a larger change, add a multi-resolution optimization to handle cases with larger differences. I.e., first solve with a coarse discretization, and then uses this solution as an initial guess for a finer discretization.

CHAPTER

SEVENTEEN

INTEGRATING DP INTO NN TRAINING

We'll now target integrations of differentiable physics (DP) setups into NNs. When using DP approaches for learning applications, there is a lot of flexibility w.r.t. the combination of DP and NN building blocks. As some of the differences are subtle, the following section will go into more detail. We'll especially focus on solvers that repeat the PDE and NN evaluations multiple times, e.g., to compute multiple states of the physical system over time. In classical numerics, this would be called an iterative time stepping method, while in the context of AI, it's an *autoregressive* method.

Hint: Correction vs Prediction

The problems that are best tackled with DP approaches are very fundamental. The combination of a imperfect physical model and an *improvement term* classically goes under many different names: *closure problems* in fluid dynamics and turbulence, *homogenization* or *coarse-graining* in material science, while it's called *parametrization* in climate and weather.

In the following, we'll generically denote all these tasks containing NN+solver as **correction** task, in contrast to pure **prediction** tasks for cases where no solver is involved at inference time.

To re-cap, here's the previous figure about combining NNs and DP operators. In the figure these operators look like a loss term: they typically don't have weights, and only provide a gradient that influences the optimization of the NN weights:

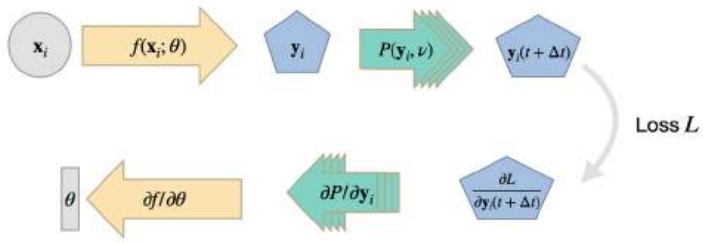


Fig. 17.1: The DP approach as described in the previous chapters. A network produces an input to a PDE solver $\mathcal{P}$, which provides a gradient for training during the backpropagation step.

This setup can be seen as the network receiving information about how it's output influences the outcome of the PDE solver. I.e., the gradient will provide information how to produce an NN output that minimizes the loss. Similar to the previously described *Physical Loss Terms*, this can, e.g., mean upholding a conservation law or generally a PDE-based constraint over time.

17.1 Switching the order

However, with DP, there's no real reason to be limited to this setup. E.g., we could imagine a swap of the NN and DP components, giving the following structure:

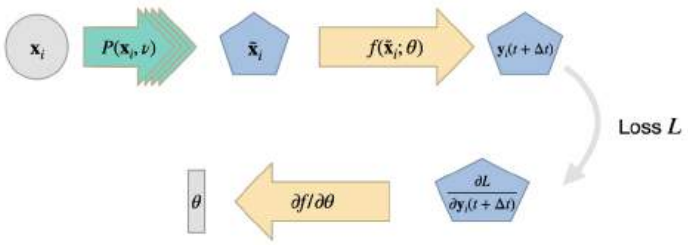


Fig. 17.2: A PDE solver produces an output which is processed by an NN.

In this case the PDE solver essentially represents an *on-the-fly* data generator. That's not necessarily always useful: this setup could be replaced by a pre-computation of the same inputs, as the PDE solver is not influenced by the NN. Hence, there's no backpropagation through $\mathcal{P}$, and it could be replaced by a simple "loading" function. On the other hand, evaluating the PDE solver at training time with a randomized sampling of input parameters can lead to an excellent sampling of the data distribution of the input. If we have realistic ranges for how the inputs vary, this can improve the NN training. If implemented correctly, the solver can also alleviate the need to store and load large amounts of data, and instead produce them more quickly at training time, e.g., directly on a GPU. Recent methods explore this direction in the context of *Active Learning*.

However, this version does not leverage the gradient information from a differentiable solver, which is why the following variant is more interesting.

17.2 Recurrent evaluation

A combination that makes particular sense is to **unroll** the iterations of a time stepping process of a simulator, and let the state of a system be influenced by an NN. (In general, there's no combination of NN layers and DP operators that is *forbidden* (as long as their dimensions are compatible).)

In the case of unrolling, we compute a (potentially very long) sequence of PDE solver steps in the forward pass. In-between these solver steps, an NN modifies the state of our system, which is then used to compute the next PDE solver step. During the backpropagation pass, we move backwards through all of these steps to evaluate contributions to the loss function (it can be evaluated in one or more places anywhere in the execution chain), and to backprop the gradient information through the DP and NN operators. This unrollment of solver iterations essentially gives feedback to the NN about how it's "actions" influence the state of the physical system and resulting loss. Here's a visual overview of this form of combination:

Due to the iterative nature of this process, errors will start out very small, and then (for modes with eigenvalues larger than one in the Jacobian) slowly increase exponentially over the course of iterations. Hence they are extremely difficult to detect in a single evaluation, e.g., with a simpler supervised training setup. Rather, it is crucial to provide feedback to the NN at training time how the errors evolve over course of the iterations. Additionally, a pre-computation of the states is not possible for such iterative cases, as the iterations depend on the state of the NN. Naturally, the NN state is unknown before training time and changes while being trained. This is the classic ML problem of **data shift**. Hence, a DP-based

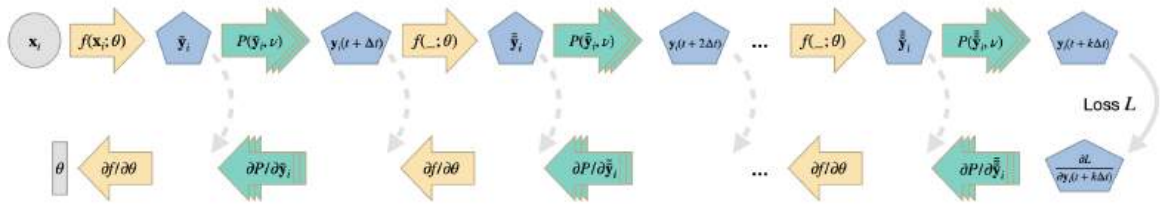
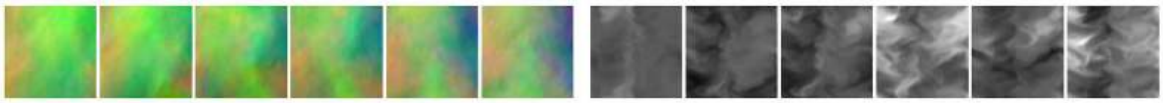


Fig. 17.3: Time stepping with interleaved DP and NN operations for k solver iterations. The dashed gray arrows indicate optional intermediate evaluations of loss terms (similar to the solid gray arrow for the last step k), and intermediate outputs of the NN are indicated with a tilde.

training is crucial in these recurrent settings to provide the NN with gradients about how its current state influences the solver iterations, and correspondingly, how the weights should be changed to better achieve the learning objectives.

DP setups with many time steps can be difficult to train: the gradients need to backpropagate through the full chain of PDE solver evaluations and NN evaluations. Typically, each of them represents a non-linear and complex function. Hence for larger numbers of steps, the vanishing and exploding gradient problem can make training difficult. Some practical considerations for alleviating this will follow int *Reducing Numerical Errors with Neural Operators*.



17.3 Composition of NN and solver

One question that we have ignored so far is how the merge the output of the NN into the iterative solving process. In the images above, it looks like the NN f produces a full state of the physical system, that is used as input to $\mathcal{P}$. That means for a state $x(t + j\Delta t)$ at step j, the NN yields an intermediate state $\tilde{x}(t + j\Delta t) = f(x(t + j\Delta t); \theta)$, with which the solver produces the new state for the following step: $x(t + (j + 1)\Delta t) = \mathcal{P}(\tilde{x}(t + j\Delta t))$.

While this approach is possible, it is not necessarily the best in all cases. Especially if the NN should produce only a correction of the current state, we can reuse parts of the current state. This avoids allocating resources of the NN in the form of parts of θ to infer the parts that are already correct. Along the lines of skip connections in a U-Net and the residuals of a ResNet, in these cases it's better to use an operator $\circ$ that merges x and $\tilde{x}$, i.e. $x(t + (j + 1)\Delta t) = \mathcal{P}(x(t+j\Delta t) \circ \tilde{x}(t+j\Delta t))$. In the simplest case, we can define $\circ$ to be an addition, in which case $\tilde{x}$ represents an additive correction of x. In short, we evaluate $\mathcal{P}(x + \tilde{x})$ to compute the next state. Here the network only needs to update the parts of x that don't yet satisfy the learning objective.

In general, we can use any differentiable operator for $\circ$, it could be a multiplication or an integration scheme. Similar to the loss function, this choice is problem dependent, but an addition is usually a good starting point.

17.4 In equation form

Next, we'll formalize the descriptions of the previous paragraphs. Specifically, we'll answer the question: what does the resulting update step for θ look like in terms of Jacobians? Given mini batches with an index i, a loss function L, we'll use k to denote the total number of steps that are unrolled for an iteration. To shorten the notation, $x_{i,j} = x_i(t + j\Delta t)$ denotes a state x of batch i at time step j. With this notation we can write the gradient of the network weights as:

$$\frac{\partial L}{\partial \theta} = \sum_i \sum_{m=1}^{k} \Big[\frac{\partial L}{\partial x_{i,k}} \Big(\prod_{n=k}^{m+1} \frac{\partial x_{i,n}}{\partial x_{i,n-1}} \Big) \frac{\partial x_{i,m}}{\partial \tilde{x}_{i,m-1}} \frac{\partial \tilde{x}_{i,m-1}}{\partial \theta} \Big] \tag{17.1}$$

This doesn't look too intuitive on first sight, but this expression has a fairly simple structure: the first sum for i simply accumulates all entries of a mini batch. Then we have an outer summation over m (the brackets) that accounts for all time steps from 1 to k. For each m, we'll trace the chain from the final state k back to each m by multiplying up all Jacobians along the way (with index n, in parentheses). Each step along the way is made up of a Jacobian w.r.t. x for each time step, which in turn depends on the correction from the NN $\tilde{x}$ (not written out).

At each last step m for the neural network we "branch-off" and determine the change in terms of the network output $\tilde{x}$ and its weights θ at the m'th time step. All these contributions for different m are added up to give a final update $\Delta\theta$ that is used in the optimizer of our training process.

It's important to keep in mind that for large m, the recurrently applied Jacobians of $\mathcal{P}$ and f strongly influence the contributions of later time steps, and hence it is critical to stabilize the training to prevent exploding gradients, in particular. This is a topic we will re-visit several time later on.

In terms of implementation, all deep learning frameworks will re-use the *overlapping* parts that repeat for different m. This is automatically handled in the backprop evaluation, and in practice, the sum will be evaluated from large to small m, such that we can "forget" the later steps when moving towards small m. So the backprop step definitely increases the computational cost, but it's usually on a similar order as the forward pass, provided that we have suitable operators to compute the derivatives of $\mathcal{P}$.

17.5 Backpropagation through solver steps

Now that we have all this machinery set up, a good question to ask is: *"How much does training with a differentiable physics simulator really improve things? Couldn't we simply unroll a supervised setup, along the lines of standard recurrent training, without using a differentiable solver?"* Or to pose it differently, how much do we really gain by backpropagating through multiple steps of the solver?

In short, quite a lot! The next paragraphs show an evaluation for a turbulent mixing layer from List et al. [LCT22], case to illustrate this difference. Before going into details, it worth noting that this comparison uses a differentiable second-order semi-implicit flow solver with a set of custom turbulence loss terms. So it's not a toy problem, but shows the influence of differentiability for a complex, real-world case.

The nice thing about this case is that we can evaluate it in terms of established statistic measurements for turbulence cases, and quantify the differences in this way. The energy spectrum of the flow is typically a starting point here, but we'll skip it and refer to the original paper [LCT22], and rather focus on two metrics that are more informative. The graphs below show the Reynolds stresses and the turbulence kinetic energy (TKE), both in terms of resolved quantities for a cross section in the flow. The reference solution is shown with orange dots.

Especially in the regions indicated by the colored arrows, the red curve of the "unrolled supervised" training deviates more strongly from the reference solution. Both measurements are taken after 1024 time steps of simulation using the fluid solver in conjunction with a trained NN. Hence, both solutions are quite stable, and fare significantly better than the unmodified output of the solver, which is shown in blue in the graphs.

The differences are also very obvious visually, when qualitatively comparing visualizations of the vorticity fields:

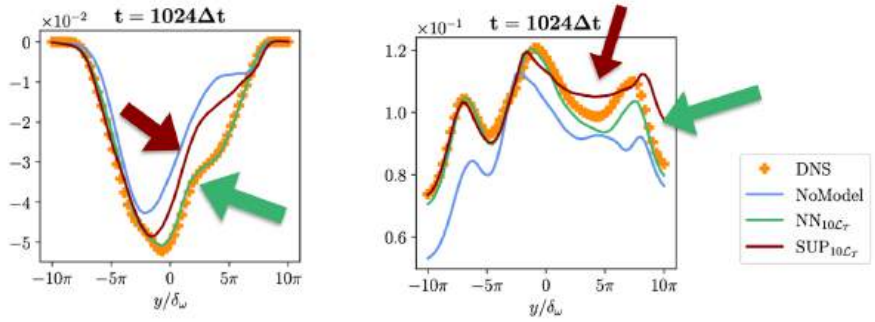


Fig. 17.4: Quantified evaluation with turbulence metrics: Reynolds stresses (L) and TKE (R). The red curve of the training without a differentiable solver deviates more strongly from the ground truth (orange dots) than the training with DP (green).

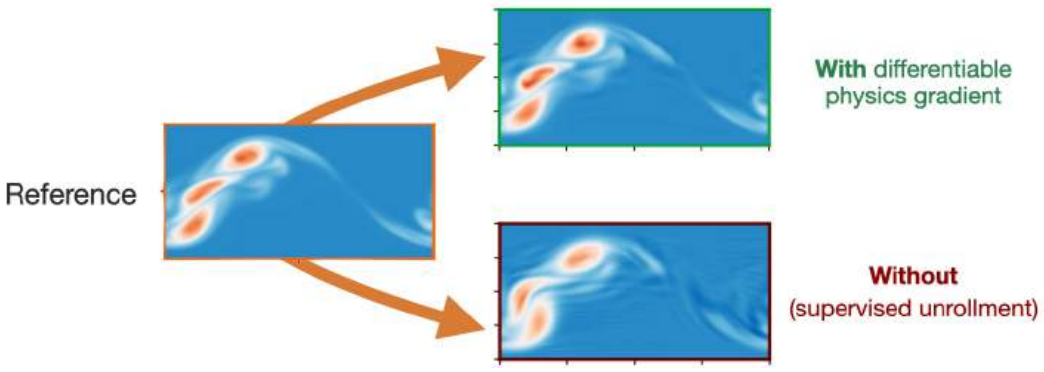


Fig. 17.5: Qualitative , visual comparison in terms of vorticity. The training with a differentiable physics solver (top) results in structures that better preserve those of the reference solution obtained via a direct numerical simulation.

Both versions, with and without solver gradients strongly benefit from unrollment, for 10 steps in this comparison. However, the supervised variant without DP cannot use longer-term information about the effects of the NN at training time, and hence its capabilities are limited. The version trained with the differentiable solver receives feedback for the whole course of the 10 unrolled steps, and in this way can infer corrections the give an improved accuracy for the resulting, NN-powered solver.

As an outlook, this case also highlights the practical advantages of incorporating NNs into solvers: we can measure how long a regular simulation would take to achieve a certain accuracy in terms of turbulence statistics. For this case it would require more than 14x longer than the solver with the NN [LCT22]. While this is just a first data point, it's good to see that, once a network is trained, real-world improvements in terms of performance can be achieved more or less out-of-the-box.

17.6 Alternatives: noise

Other works have proposed perturbing the inputs and the iterations at training time with noise [SGGP+20], somewhat similar to regularizers like dropout. This can help to prevent overfitting to the training states, and in this way can help to stabilize training iterative solvers.

However, the noise is very different in nature. It is typically undirected, and hence not as accurate as training with the actual evolutions of simulations. So noise can be a good starting point for training setups that tend to overfit. However, if possible, it is preferable to incorporate the actual solver in the training loop via a DP approach to give the network feedback about the time evolution of the system.

With the current state of affairs, generative modeling approaches (denoising diffusion or flow matching) or provide a better founded approach for incorporating noise. We'll look into this topic in more detail in *Unconditional Stability*.

17.7 Complex examples

The following sections will give code examples of more complex cases to show what can be achieved via differentiable physics training.

First, we'll show a scenario that employs deep learning to represent the errors of numerical simulations, following Um et al. [UBH+20]. This is a very fundamental task, and requires the learned model to closely interact with a numerical solver. Hence, it's a prime example of situations where it's crucial to bring the numerical solver into the deep learning loop.

Next, we'll show how to let NNs solve tough inverse problems, namely the long-term control of a Navier-Stokes simulation, following Holl et al. [HKT19]. This task requires long term planning, and hence needs two networks, one to *predict* the evolution, and another one to *act* to reach the desired goal. (Later on, in *Controlling Burgers' Equation with Reinforcement Learning* we will compare this approach to another DL variant using reinforcement learning.)

Both cases require quite a bit more resources than the previous examples, so you can expect these notebooks to run longer (and it's a good idea to use check-pointing when working with these examples).

CHAPTER

EIGHTEEN

REDUCING NUMERICAL ERRORS WITH NEURAL OPERATORS

In this example we will target numerical errors that arise in the discretization of a continuous PDE $\mathcal{P}^*$, i.e. when we formulate $\mathcal{P}$. This approach will demonstrate that, despite the lack of closed-form descriptions, discretization errors often are functions with regular and repeating structures and, thus, can be learned by a discretized neural operator. Once trained, the neural network (NN) can be evaluated locally to improve the solution of a PDE-solver, i.e., to reduce its numerical error. The resulting method is a hybrid one: it will always perform (a coarse) PDE solve, and then improve it at runtime with corrections inferred by an NN.

Pretty much all numerical methods contain some form of iterative process: repeated updates over time for explicit solvers, or within a single update step for implicit or steady-state solvers. An example for the second case could be found here, but below we'll target the first case, i.e. iterations over time. [run in colab]

18.1 Problem formulation

In the context of reducing errors, it's crucial to have a *differentiable physics solver*, so that the learning process can take the updates of the solver into account. This interaction is not possible with supervised- or PINN-based training. Even small inference errors of a supervised NN accumulate over time, and lead to a data distribution that differs from the distribution of the pre-computed data. This distribution shift leads to sub-optimal results, or even cause blow-ups of the solver.

In order to learn the error function, we'll consider two different discretizations of the same PDE $\mathcal{P}^*$: a *reference* version, which we assume to be accurate and high fidelity, with a discretized version $\mathcal{P}_r$, and solutions $\mathbf{r} \in \mathcal{R}$, where $\mathcal{R}$ denotes the manifold of solutions of $\mathcal{P}_r$. In parallel to this, we have a lower fidelty solver for the same PDE, which we'll refer to as the *source* version, as this will be the solver that our NN should later on interact with. Analogously, we have $\mathcal{P}_s$ with solutions $\mathbf{s} \in \mathcal{S}$. After training, we'll obtain a *hybrid* solver that the source solver $\mathcal{P}_s$ in conjunction with a trained operator to obtain improved solutions, i.e., solutions that are closer to the ones produced by $\mathcal{P}_r$.

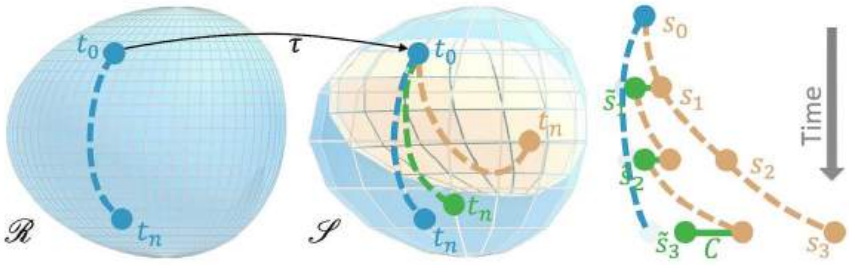


Fig. 18.1: Visual overview of coarse and reference manifolds

Let's assume $\mathcal{P}$ advances a solution by a time step Δt, and let's denote n consecutive steps by a superscript: $\mathcal{P}_s^n(\mathcal{T}\mathbf{r}_t) = \mathcal{P}_s(\mathcal{P}_s(\cdots \mathcal{P}_s(\mathcal{T}\mathbf{r}_t)\cdots))$. The corresponding state of the simulation is $\mathbf{s}_{t+n} = \mathcal{P}^n(\mathcal{T}\mathbf{r}_t)$. Here we assume a mapping operator $\mathcal{T}$ exists that transfers a reference solution to the source manifold. This could, e.g., be a simple downsampling

operation. Especially for longer sequences, i.e. larger n, the source state $\mathbf{s}_{t+n}$ will deviate from a corresponding reference state $\mathbf{r}_{t+n}$. This is what we will address with an NN-based operator in the following.

As before, we'll use an L^2-norm to quantify the deviations, i.e., an error function $e(\mathbf{s}_t, \mathcal{T}\mathbf{r}_t) = \|\mathbf{s}_t - \mathcal{T}\mathbf{r}_t\|_2$. Our learning goal is to train at a correction operator $\mathcal{C}(\mathbf{s})$ such that a solution to which the correction is applied has a lower error than the original unmodified (source) solution: $e(\mathcal{P}_s(\mathcal{C}(\mathcal{T}\mathbf{r}_t)), \mathcal{T}\mathbf{r}_{t+1}) < e(\mathcal{P}_s(\mathcal{T}\mathbf{r}_t), \mathcal{T}\mathbf{r}_{t+1})$.

The correction operator $\mathcal{C}(\mathbf{s}|\theta)$ is represented as a deep neural network with weights θ and receives the state $\mathbf{s}$ to infer an additive correction field with the same dimension. To distinguish the original states $\mathbf{s}$ from the corrected ones, we'll denote the latter with an added tilde $\tilde{\mathbf{s}}$. The overall learning goal now becomes

$$\arg\min_\theta \left((\mathcal{P}_s \mathcal{C})^n (\mathcal{T}\mathbf{r}_t) - \mathcal{T}\mathbf{r}_{t+n} \right)^2$$

To simplify the notation, we've dropped the sum over different samples here (the i from previous versions). A crucial bit that's easy to overlook in the equation above, is that the correction depends on the modified states, i.e. it is a function of $\tilde{\mathbf{s}}$, so we have $\mathcal{C}(\tilde{\mathbf{s}}|\theta)$. These states actually evolve over time when training. They don't exist beforehand.

TL;DR: We'll train a neural operator $\mathcal{C}$ to reduce the numerical errors of a simulator with respect to a more accurate reference. It's crucial to have the *source* solver realized as a differential physics operator, such that it provides gradients for an improved training of $\mathcal{C}$.

18.2 Getting started with the implementation

The following replicates an experiment from Solver-in-the-loop: learning from differentiable physics to interact with iterative pde-solvers [UBH+20], further details can be found in section B.1 of the appendix of the paper.

First, let's import the necessary libraries, most importantly phiflow and PyTorch, and let's get the device handling out of the way, so that we can focus on the *interesting* parts…

```
try:
    import google.colab  # to ensure that we are inside colab
    !pip install --upgrade --quiet phiflow==3.3
    #!pip install --upgrade --quiet git+https://github.com/tum-pbs/PhiFlow@develop

    # for pbdl-dataset:
    !pip install --upgrade --quiet git+https://github.com/tum-pbs/pbdl-dataset

except ImportError:
    print("This notebook is running locally, please make sure the necessary pip␣
 ↪packages are installed.")
    pass
```

```
This notebook is running locally, please make sure the necessary pip packages are␣
 ↪isntalled.
```

```
import os, sys, logging, argparse, pickle, glob, random, pylab, time
from tqdm import tqdm
from phi.torch.flow import *

random.seed(42)
np.random.seed(42)

math.seed(42) # phiflow seed
```

(continues on next page)

(continued from previous page)

```
math.set_global_precision(32) # single precision

USE_CPU = 0
TORCH.set_default_device("GPU")
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
if USE_CPU > 0:
    device = 'cpu'
device = torch.device(device)
print("Using device: "+str(device))
```

```
Using device: cuda:0
```

And while we're at it, we also set the random seed - obviously, 42 is the ultimate choice here 🙂

18.3 Simulation setup

Now we set up the *source* simulation $\mathcal{P}_s$. Note that we won't deal with $\mathcal{P}_r$ in this notebook: the downsampled reference data is from the high-fidelty solve is already contained in the training data set. It was generated with a four times finer spatial and temporal discretization. Below we're focusing on the interaction of the source solver and the NN.

The `KarmanFlow` solver below simulates a relatively standard Navier-Stokes wake flow case with a spherical obstacle in a rectangular domain, and an explicit viscosity solve to obtain different Reynolds numbers. This is the geometry of the setup:

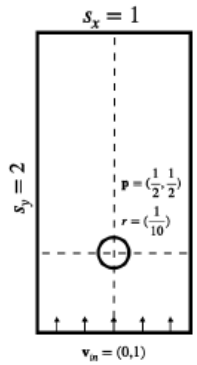


Fig. 18.2: Domain setup for the wake flow case (sizes in the imlpementation are using an additional factor of 100).

The solver applies inflow boundary conditions for the y-velocity with a pre-multiplied mask (`vel_BcMask`), to set the y components at the bottom of the domain during the simulation step. This mask is created with the `HardGeometryMask` from phiflow, which initializes the spatially shifted entries for the components of a staggered grid correctly. The simulation step is quite straight forward: it computes contributions for viscosity, inflow, advection and finally makes the resulting motion divergence free via an implicit pressure solve:

```
RE_FAC_SOL = 10/(128*128) # factor to compensate for the original scaling from the␣
 ↪original solver-in-the-loop paper

class KarmanFlow():
    def __init__(self, domain):
        self.domain = domain
```

(continues on next page)

(continued from previous page)

```
        self.vel_BcMask = self.domain.staggered_grid( HardGeometryMask( Box(y=(None,␣
↪5), x=None) ) )

        self.inflow = self.domain.scalar_grid(Box(y=(5,10), x=(25,75)) ) # scale with␣
↪domain if necessary!
        self.obstacles = [Obstacle(Sphere(center=tensor([50, 50], channel(vector="y,x
↪")), radius=10))]

    def step(self, marker_in, velocity_in, Re, res, buoyancy_factor=0, dt=1.0):
        velocity = velocity_in
        marker   = marker_in
        Re_phiflow = Re / RE_FAC_SOL # rescale for phiflow

        # viscosity
        velocity = phi.flow.diffuse.explicit(u=velocity, diffusivity=1.0/Re_
↪phiflow*dt*res*res, dt=dt)

        # inflow boundary conditions
        velocity = velocity*(1.0 - self.vel_BcMask) + self.vel_BcMask * (1,0)

        # advection
        marker = advect.semi_lagrangian(marker+ 1. * self.inflow, velocity, dt=dt)
        velocity = advected_velocity = advect.semi_lagrangian(velocity, velocity,␣
↪dt=dt)

        # mass conservation (pressure solve)
        pressure = None
        velocity, pressure = fluid.make_incompressible(velocity, self.obstacles)
        self.solve_info = { 'pressure': pressure, 'advected_velocity': advected_
↪velocity }

        return [marker, velocity]
```

Note that the `marker` density here denotes a passively advected marker field, and not the density of the fluid. Below we'll only be focusing on the velocity for the correction task. The marker density is tracked purely for visualization purposes.

18.4 Network and transfer functions

We'll also define a simple neural networks to represent the operator $\mathcal{C}$. We'll use fully convolutional networks, i.e. networks without any fully-connected (MLP) layers. We'll use phiflow's network tools to set up a `conv_net` with a given number of layers as specified in the `layers` list. The inputs to the network are 3 fields:

- 2 fields with x,y velocity
- the Reynolds number as constant channel.

The output is:

- a 2 component field containing the x,y velocity.

In the conv-net, the input dimensions are determined from input tensor (it has three channels: u,v, and Re). Then we process the data via the sequence of conv layers and activations with 32 (and 48) features each, before reducing to 2 channels in the output. The code below also re-initializes the convolutions with a uniform Xavier initializer that is downscaled with a gain of `0.1`. The simplifies training by avoiding overly large values at the beginning. With it, we can

direclty activate unrolling multiple steps. Without it, we'd need to add a curriculum and make sure the network is trained for a bit to find a suitable range for the corrections, before being applied to longer sequences via unrolling.

While we're at it, we (of course) also checking the number of paramters in the network. This is a crucial metric for the approximate computational cost of the NN.

```
layers = [32,32,32] # small
#layers = [32,48,48,48,32] # uncomment for a somewhat larger and deeper network
#network = conv_net(in_channels=3,out_channels=2,layers=layers) # a simpler variant
network = res_net(in_channels=3,out_channels=2,layers=layers)
print(network)

# reinit
import torch.nn as nn
for m in network.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.xavier_uniform_(m.weight, gain=0.1)

print("Total number of trainable parameters: "+ str( sum(p.numel() for p in network.
↪parameters()) ))
```

```
ResNet(
  (Res_in): ResNetBlock(
    (sample_input): Conv2d(3, 32, kernel_size=(1, 1), stride=(1, 1))
    (bn_sample): Identity()
    (bn1): Identity()
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): Identity()
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (Res1): ResNetBlock(
    (sample_input): Identity()
    (bn_sample): Identity()
    (bn1): Identity()
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): Identity()
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (Res2): ResNetBlock(
    (sample_input): Identity()
    (bn_sample): Identity()
    (bn1): Identity()
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): Identity()
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (Res_out): Conv2d(32, 2, kernel_size=(1, 1), stride=(1, 1))
)
Total number of trainable parameters: 47330
```

Next, we're defining transfer functions which are pretty important: they don't modify any content, but transform the simulation state into suitable data structures for the different software packages that are used: staggered grids for the solver, pytorch tensors for the NN, and another helper to turn the numpy output of the dataloader into phiflow grids and values that can easily be used in the NS solver.

The `to_phiflow` function takes a pytorch tensor `batch` and transforms the two relevant channels (1 and 2 in the second dimension for x and y) into a staggered grid for phiflow. The only caveat here is the size of the arrays involved: due to the closed and open boundaries and the staggered grid, phiflow expects arrays of size `[64, 31]` and `[65,`

32] for x and y, respectively. In phiflow's math.tensor call we then have to tell phiflow about the batch and spatial dimensions.

The to_pytorch function receives a small list with [marker,velocity], and uses the two vector components of the staggered grid velocity via vector['x'] and vector['y'] to discard the outermost layer of the velocity field grids. This gives two tensors of equal size that are stacked along the channels dimension. It also adds a constant channel via math.ones that is multiplied by the desired Reynolds number in ext_const_channel onto the stack. The resulting stack of grids is turned into a single pytorch tensor via native(order='batch,channels,y,x') and represents the input to the neural network.

The last function is a helper to transform the output of the dataloader into data structurs for phiflow. The data loader produces numpy arrays, and they're transformed into a velocity grid with the to_phiflow function from above. At the same time, we're normalizing the Reynolds number for the NN operator, and we keep the original one as a scalar value for the phiflow solver, as the latter uses "physical units", in contrast to the network. Each batch for training will contain multiple samples with the same Reynolds number. This is taken care of by the dataloader.

```
def to_phiflow(batch):
    vx = batch[:,1,:-1,:-1]
    vy = batch[:,2,:,:] # fine

    #print("v_dims "+str([vx.shape,vy.shape])) # example for debugging
    # v_dims should be vx [torch.Size([B, 64, 31]),  vy torch.Size([B, 65, 32])]

    vel = domain.staggered_grid( math.stack( [
                            math.tensor(vy, math.batch('batch'), math.spatial('y, x
↪')),
                            math.tensor(vx, math.batch('batch'), math.spatial('y, x
↪')),
                        ], math.dual(vector="y,x")
                    ) )
    return vel

def to_pytorch(marker_vel, Re):
    # align the sides the staggered velocity grid making its size the same as the␣
↪centered grid
    grid = math.stack(
        [
            math.pad( marker_vel[1].vector['x'].values, {'x':(0,1)} , math.
↪extrapolation.ZERO), # x component
            marker_vel[1].vector['y'].y[:-1].values,                                ␣
↪           # y component
            math.ones(marker_vel[0].shape)*Re                                       ␣
↪           # constant Re
        ],
        math.channel('channels')
    ).native(order='batch,channels,y,x')
    return grid

def to_solver(inputs):
    marker_in = inputs[:,0,:-1,:]
    marker_in = domain.scalar_grid( math.tensor(marker_in, math.batch('batch'), math.
↪spatial('y, x')) )
    v_in = to_phiflow(inputs)
    Re = math.tensor(inputs[0,3, 0,0].detach()) # Scalar , get first index 0,0

    Re_norm = (Re - math.tensor(DATA_RE_MEAN)) / math.tensor(DATA_RE_STD)
    Re_norm = float(Re_norm.native().detach()) # we just need a single number
```

(continues on next page)

(continued from previous page)

```
    return marker_in, v_in, Re, Re_norm
```

18.5 Training setup

Now we also need to take care to set up the training, i.e. intialize data sets and optimizers. The latter is relatively easy, we'll use an Adam optimizer with a given learning rate, and this is also a good time to define the batch size (3 is a good default here).

```
LEARNING_RATE = 1e-3
optimizer = adam(network, LEARNING_RATE)

# one of the most crucial parameters: how many simulation steps to look into the␣
↪future in each training step
MSTEPS = 4

BATCH_SIZE = 3
```

The most important and interesting parameter is `MSTEPS`. It defines the number of simulation steps that are unrolled at each training iteration. This directly influences the runtime of each training step, as we first have to simulate all steps forward, and then backpropagate the gradient through all `MSTEPS` simulation steps interleaved with the NN evaluations. However, this is where we'll receive important feedback in terms of gradients how the inferred corrections actually influence a running simulation. Hence, larger `msteps` are typically better. Ideally, the training also increases the number of `MSTEPS` over time in the form of a curriculum, but we'll omit that to keep the code simpler.

For the training data itself, we can use PBDL's data loader class, which automatically downloads the data from HuggingFace (if necessary), and returns a class that we can easily iterate over to get a new batch for training. Here we need to specify the `BATCH_SIZE`, and we select simulations 0 to 5 from the dataset, in order to keep cases 6 to 9 for testing later on. The first 6 contain an intermediate range of Reynolds numbers, so that we can keep some new ones outside of this range for testing generalization later on. The dataloader also needs to provide enough steps of the ground truth reference to compute the loss for the full unrolled trajectory. This is ensured via `time_steps=MSTEPS+1, intermediate_time_steps=True`.

```
import pbdl
import pbdl.torch.loader

dataloader = pbdl.torch.loader.Dataloader("solver-in-the-loop-wake-flow", MSTEPS,␣
↪shuffle=True, sel_sims=[0,1,2,3,4,5],
                                          batch_size=BATCH_SIZE, normalize_const="std
↪", normalize_data="std", intermediate_time_steps=True)

# workaround for using un-normalized and normalized values in one script:
#    save the normalization constants of the Reynolds number conditioning , then turn␣
↪off (norm=None);
#    the Re values will be normalized manually later on
DATA_RE_MEAN = dataloader.dataset.norm_strat_const.const_mean[0][0]
DATA_RE_STD  = dataloader.dataset.norm_strat_const.const_std[0][0]
print([DATA_RE_MEAN,DATA_RE_STD])
dataloader.dataset.norm_strat_const = None
dataloader.dataset.norm_strat_data = None
```

```
download completed          ———————————————————————————————————————— 100
 ↪%6m
Success: Loaded solver-in-the-loop-wake-flow with 10 simulations (6 selected) and␣
 ↪496 samples each.
Info: No precomputed normalization data found (or not complete). Calculating data..
 ↪.
[np.float64(1237.79296875), np.float64(1453.7359614526729)]
```

Additionally, we defined several global variables to control the training and the simulation in the next code cell.

The fluid solver object is called `simulator` below. In order to easily create grids in phiflow it uses a phiflow `Domain` object, which mostly exists for convenience purposes: it stores resolution, physical size, and boundary conditions of the domain. This information needs to be passed to every grid, and hence it's convenient to have it in one place in the form of the `Domain`. For the setup described above, we need different boundary conditions along x and y: closed walls, and free flow in and out of the domain, respecitvely.

```
# this is the actual resolution in terms of cells for phiflow (not too crucial)
SOURCE_RES = [64,32]

# this is the physical size in terms of abstract units for the bounding box of the␣
 ↪domain (it's important for conversions between computational and physical units)
LENGTH = 100.

# for readability
from phi.physics._boundaries import Domain, OPEN, STICKY as CLOSED

BNDS = {
    'y':(phi.physics._boundaries.OPEN,  phi.physics._boundaries.OPEN) ,
    'x':(phi.physics._boundaries.STICKY,phi.physics._boundaries.STICKY)
}

domain = phi.physics._boundaries.Domain(y=SOURCE_RES[0], x=SOURCE_RES[1],␣
 ↪boundaries=BNDS, bounds=Box(y=2*LENGTH, x=LENGTH))
simulator = KarmanFlow(domain=domain)
```

```
/tmp/ipykernel_774262/1641432920.py:8: FutureWarning: Domain (phi.physics._
 ↪boundaries) is deprecated and will be removed in a future release.
Please create grids directly, replacing the domain with a dict, e.g.
    domain = dict(x=64, y=128, bounds=Box(x=1, y=1))
    grid = CenteredGrid(0, **domain)
  from phi.physics._boundaries import Domain, OPEN, STICKY as CLOSED
/tmp/ipykernel_774262/1641432920.py:15: DeprecationWarning: Domain is deprecated␣
 ↪and will be removed in a future release. Use a dict instead, e.g.␣
 ↪CenteredGrid(values, extrapolation, **domain_dict)
  domain = phi.physics._boundaries.Domain(y=SOURCE_RES[0], x=SOURCE_RES[1],␣
 ↪boundaries=BNDS, bounds=Box(y=2*LENGTH, x=LENGTH))
/tmp/ipykernel_774262/1641432920.py:15: FutureWarning: Domain is deprecated and␣
 ↪will be removed in a future release. Use a dict instead, e.g.␣
 ↪CenteredGrid(values, extrapolation, **domain_dict)
  domain = phi.physics._boundaries.Domain(y=SOURCE_RES[0], x=SOURCE_RES[1],␣
 ↪boundaries=BNDS, bounds=Box(y=2*LENGTH, x=LENGTH))
/tmp/ipykernel_774262/1529251970.py:7: DeprecationWarning: HardGeometryMask and␣
 ↪SoftGeometryMask are deprecated. Use field.mask or field.resample instead.
  self.vel_BcMask = self.domain.staggered_grid( HardGeometryMask( Box(y=(None, 5),␣
 ↪x=None) ) )
```

18.6 Interleaving simulation and NN

In order to efficiently run training with non-trivial simulations, it's a good idea to keep the runtime in mind. For efficient runs it's especially important to involve the GPUs used for training, and keep the data on the GPU as much as possible. For phiflow, this can largely be achieved by jit-compiling the central steps, and in the next code cell we'll do this for the Navier-Stokes simulation step. It involves an implicit pressure solve, among others, and is potentially called multiple times for each forward and backwards pass. Hence this is a good candidate for jit-compilation. (Try removing the `@jit_compile` statement to experience the slow-down yourself.)

```
@jit_compile
def simulation_step(marker,velocity,Re,resolution):
    m,v = simulator.step(
        marker_in=marker,
        velocity_in=velocity,
        Re=Re, res=resolution
    )
    return m,v
```

Next comes the **most crucial** step in the whole setup: we define a function that encapsulates the chain of simulation steps and network evaluations in each training step. With the helper functions we've set up so far, it's actually pretty simple: we loop `MSTEPS` times, calling the simulator via `simulator_step` for an input state, and afterwards evaluate the correction operator via `network(to_pytorch(...))`. The NN correction is then added to the last simulation state in the `prediction` list of states. This list keeps around the marker density and velocity for each time step.

Note that apart from the Reynolds number, we're not normalizing the states themselves as they're already in the -1 to 1 range. For other simulations it would be a good idea to normalize before invoking the network, and de-normalizing afterwards for the subsequent physics solving step.

```
def training_step(inputs_targets):
    [inputs, targets] = inputs_targets
    marker_in, v_in, Re, Re_norm = to_solver(inputs)
    prediction = [ [marker_in,v_in] ]
    loss = 0

    for i in range(MSTEPS):
        m2,v2 = simulation_step(
            marker=prediction[-1][0],
            velocity=prediction[-1][1],
            Re=Re, resolution=SOURCE_RES[1]
        )

        net_in = to_pytorch([m2,v2],Re_norm)
        net_out = network(net_in)

        cy = net_out[:,1,:,:] # pad y
        cy = torch.nn.functional.pad(input=cy, pad=(0,0, 0,1), mode='constant',␣
↪value=0)
        cx = net_out[:,0,:,:-1]

        v_corr = domain.staggered_grid( math.stack( [
                            math.tensor(cy, math.batch('batch'), math.spatial('y, x
↪')),
                            math.tensor(cx, math.batch('batch'), math.spatial('y, x
↪')),
                        ], math.dual(vector="y,x")
                    ) )
```

(continues on next page)

(continued from previous page)

```
        prediction += [ [domain.scalar_grid(m2) , v2 + v_corr] ]
        vdiff = prediction[-1][1] - to_phiflow(targets[:,i,...])
        loss += field.l2_loss(vdiff)

    return loss, prediction
```

The `training_step` function above also directly evaluates and returns the loss. Here, we simply use an L^2 loss over the grids (in phiflow `fields`) for the whole sequence, i.e. over the unrolled `msteps`. In `vdiff` we're simply computing the difference between the targets and the current prediction, and then compute its `l2_loss`.

With the training step, the training is quite simple: all that's left to do is to let the optimizer compute the gradients to minimize the loss. Phiflow provides a helper function for this: `update_weights`. We provide the neural network, the optimizer, and the function to compute the loss (the first return value of `training_step`). We simply loop `EPOCHS` times over enumerating the full dataset from the `dataloader`. The progress bar `pbar` below is simply eyecandy to track the progress of the training. Because the jit compilation of the simulator is triggered in the very first step takes a bit longer, but the subsequent ones should be substantially faster. The code below also saves the network state every epoch N in a file `net-N.pickle`.

```
EPOCHS = 5

pbar = tqdm(initial=0, total=EPOCHS*len(dataloader), ncols=96)

for epoch in range(EPOCHS):
    for b, (input_cpu, targets_cpu) in enumerate(dataloader):
        input = torch.tensor(input_cpu, dtype=torch.float32).to(device);
        targets = torch.tensor(targets_cpu, dtype=torch.float32).to(device)

        loss, prediction = update_weights(network, optimizer, training_step, [input,␣
↪targets])

        pbar.set_description("loss "+str(np.sum(loss.numpy("batch"))), refresh=False);
↪ pbar.update(1)

    torch.save(network.state_dict(), "net-"+str(epoch)+".pickle")

pbar.close()
```

```
  0%|                                                                          | 0/4960␣
↪[00:00<?, ?it/s]/tmp/ipykernel_774262/4154370188.py:7: UserWarning: To copy␣
↪construct from a tensor, it is recommended to use sourceTensor.clone().detach()␣
↪or sourceTensor.clone().detach().requires_grad_(True), rather than torch.
↪tensor(sourceTensor).
  input = torch.tensor(input_cpu, dtype=torch.float32).to(device);
/tmp/ipykernel_774262/4154370188.py:8: UserWarning: To copy construct from a␣
↪tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.
↪clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  targets = torch.tensor(targets_cpu, dtype=torch.float32).to(device)
anaconda3/envs/torch24/lib/python3.12/site-packages/phiml/math/_optimize.py:631:␣
↪UserWarning: Possible rank deficiency detected. Matrix might be singular which␣
↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
↪).
  warnings.warn("Possible rank deficiency detected. Matrix might be singular which␣
↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
↪).")
anaconda3/envs/torch24/lib/python3.12/site-packages/phiml/backend/torch/_torch_
```

(continues on next page)

(continued from previous page)

```
 ↪backend.py:803: UserWarning: Sparse CSR tensor support is in beta state. If you␣
 ↪miss a functionality in the sparse tensor support, please submit a feature␣
 ↪request to https://github.com/pytorch/pytorch/issues. (Triggered internally at ..
 ↪/aten/src/ATen/SparseCsrTensorImpl.cpp:53.)
  return torch.sparse_csr_tensor(row_pointers, column_indices, values, shape,␣
 ↪device=values.device)
anaconda3/envs/torch24/lib/python3.12/site-packages/phiml/math/_optimize.py:631:␣
 ↪UserWarning: Possible rank deficiency detected. Matrix might be singular which␣
 ↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
 ↪).
  warnings.warn("Possible rank deficiency detected. Matrix might be singular which␣
 ↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
 ↪).")
loss 15.62617:   0%|                                                  | 1/4960 [00:12
 ↪<16:36:58, 12.06s/it]/tmp/ipykernel_774262/4154370188.py:7: UserWarning: To copy␣
 ↪construct from a tensor, it is recommended to use sourceTensor.clone().detach()␣
 ↪or sourceTensor.clone().detach().requires_grad_(True), rather than torch.
 ↪tensor(sourceTensor).
  input = torch.tensor(input_cpu, dtype=torch.float32).to(device);
/tmp/ipykernel_774262/4154370188.py:8: UserWarning: To copy construct from a␣
 ↪tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.
 ↪clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  targets = torch.tensor(targets_cpu, dtype=torch.float32).to(device)
loss 0.23079132: 100%|████████████████████████████████████| 4960/4960 [3:10:57
 ↪<00:00,  2.31s/it]
```

Note that each call to `update_weights` internally performs all 4 interleaved simulation steps and neural network calls, and back-propagates gradients for the whole chain. Due to the order, solver first, then network, the first step does not go back through the solver. However, for the next 3 steps, the first network call will receive feedback whether it's inferred correction influenced the 3 subsequent states towards the right direction or not.

With the default parameters, the loss should go down from above 10 initially to around 1.0 (more epochs and larger/deeper networks will bring it down further). The decreasing loss is a good sign, but of course it's even more important to see how the resulting NN-solver combination fares on new inputs. With this training approach we've realized a hybrid solver, consisting of a regular *source* simulator, and a network operator that was trained to specifically interact with this simulator for a chosen domain of simulation cases.

18.7 Test evaluation

In order to evaluate the performance of our "AI-powered solver", we essentially only need to repeat the inner loop of each training iteration for more steps. While we were limited to `msteps` evaluations at training time, we can now run our solver for arbitrary lengths. This is a good test for how well our solver has learned to keep the data within the desired distribution, and represents a generalization test for longer rollouts.

We also need a set of new data. Below, we'll use the previously unused last four simulations of the dataset that was downloaded above. We simply instantiate a new dataloader with `sel_sims=[6,7,8,9]`.

```
del dataloader # close hdf5 file handle

# get new & unseen test data
dataloader_test = pbdl.torch.loader.Dataloader("solver-in-the-loop-wake-flow", 200,␣
 ↪batch_size=1, shuffle=True, sel_sims=[6,7,8,9],
                                        normalize=False, intermediate_time_
 ↪steps=True)
```

```
download completed          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100
↪%6m
Success: Loaded solver-in-the-loop-wake-flow with 10 simulations (4 selected) and␣
↪300 samples each.
```

In case you have a pre-trained network, this is a good point to load a model. By default, we assume the training above was completed, so it's not necessary to load anything.

```
# optionally load
if False:
    fn = "net-"+str(EPOCHS-1)+".pickle" # load last
    network.load_state_dict(torch.load(fn, map_location=device, weights_only=True))
    print("Loaded "+fn)
```

We can reuse a lot of the solver code from above, but in the following, we will consider two simulated versions: for comparison, we'll run one reference simulation in the *source* manifold (i.e. based on $\mathcal{P}_s$, without any corrections applied). The second version is the actual result, we'll repeatedly compute the source solver plus the learned correction.

The `run_sim` function below switches between these two variants depending on whether a neural operator is provided in `network`. Without it, it simply runs the source solver and appends the states. With a `network` it runs the full hybrid solver. Both cases compute error w.r.t. reference along the way. For analysis and visualization later on, the function returns the correction and references in addition to the relative errors and the actual states computed by the solver.

```
def run_sim(inputs, targets, steps, network=None):
    marker_in, v_in, Re, Re_norm = to_solver(inputs)

    simtype = "With corr."
    if (network==None): simtype = "Sim. only"
    print("Running test with Re="+str(Re)+", "+simtype)

    prediction = [ [marker_in,v_in] ]
    correction = [ [marker_in,v_in] ]
    refs = [ v_in ]
    errors = []

    for i in tqdm(range(steps), desc=simtype, ncols = 64):
        marker_sim,v_sim = simulation_step(
            marker=prediction[-1][0],
            velocity=prediction[-1][1],
            Re=Re, resolution=SOURCE_RES[1]  # take Re from constant field
        )

        if network: # run hybrid solver with trained Neural op
            net_in = to_pytorch([marker_sim,v_sim],Re_norm)
            net_out = network(net_in)

            cy = net_out[:,1,:,:] # pad y
            cy = torch.nn.functional.pad(input=cy, pad=(0,0, 0,1), mode='constant',␣
↪value=0)
            cx = net_out[:,0,:,:-1]

            v_corr = domain.staggered_grid( math.stack( [
                                math.tensor(cy, math.batch('batch'), math.spatial('y,␣
↪x')),
                                math.tensor(cx, math.batch('batch'), math.spatial('y,␣
↪x')),
                            ], math.dual(vector="y,x")
```

(continues on next page)

(continued from previous page)

```
                        ) )

            prediction += [ [domain.scalar_grid(marker_sim) , v_sim + v_corr] ]
            correction += [ [domain.scalar_grid(marker_sim) , v_corr] ]

        else: # only low-fidelity solver
            prediction += [ [domain.scalar_grid(marker_sim) , v_sim ] ]

        refs += [to_phiflow(targets[:,i,...])]
        vdiff = prediction[i][1] - refs[-1]

        error_phi = field.l1_loss(vdiff)
        errors += [float( error_phi.native("batch")[0] / field.l1_loss(refs[-1]).
↪native("batch")[0] )]

    return errors, prediction, refs, correction
```

With `next(iter(dataloader_test))` we'll get a new state from the dataloader with a previously unseen Reynolds number. Then we'll run source and hybrid solver for `ROLLOUT_STEPS` iterations starting from the same initial state. Similar to training, we'll first get an initial state and reference states from the dataloader in `input` and `targets`. (Due to the random sampling, you might need to run this multiple times to e.g. get one of the higher Re cases.)

```
# get a sample
(input_cpu, targets_cpu) = next(iter(dataloader_test))
input = torch.tensor(input_cpu, dtype=torch.float32).to(device)
targets = torch.tensor(targets_cpu, dtype=torch.float32).to(device)
print("Re ",math.tensor(input[0,3, 0,0].detach()))
```

```
Re  73.24219
```

```
/tmp/ipykernel_774262/1514360431.py:3: UserWarning: To copy construct from a␣
↪tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.
↪clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  input = torch.tensor(input_cpu, dtype=torch.float32).to(device)
/tmp/ipykernel_774262/1514360431.py:4: UserWarning: To copy construct from a␣
↪tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.
↪clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  targets = torch.tensor(targets_cpu, dtype=torch.float32).to(device)
```

The interesting question of course is: how much does the NN operator actually improve the accuracy of the low-fidelity *source* solver? This is captured by the relative L_2 errors computed each time by the `run_sim` function. It measures the squared distance in comparison to the squared magnitudes of the reference velocities. The cell above directly plots the aggregated errors.

Below, we run the hybrid solver with its trained Neural correction operator and the low-fidelity solver alone for comparison. `ROLLOUT_STEPS` below determines the number of time steps to compute with both variants. The cell also directly outputs the mean relative error. Due to the stochastic training and numerical round-off errors, the results can vary slightly over different runs, but in general the hybrid solver with its Neural correction operator should show a significantly reduced numerical error: typically 5-6x lower than the low-fidelity solver.

This shows the central objective of this training setup has been achieved: the hybrid solver yields substantially reduced numerical errors and generalizes to new Reynolds numbers ❓

```
ROLLOUT_STEPS = 100
```

(continues on next page)

(continued from previous page)

```
err_lowfid_only, prediction_lowfid_only, refs, _   = run_sim(input, targets, ROLLOUT_
↪STEPS)
err_corrected,   prediction_corrected  , _ , corrs = run_sim(input, targets, ROLLOUT_
↪STEPS, network)
print("\n Rel. L2 errors: low-fidelity:", float(np.mean(err_lowfid_only)),"  corrected:
↪", float(np.mean(err_corrected)) )
```

```
Running test with Re=73.24219, Sim. only
```

```
Sim. only:   0%|                          | 0/100 [00:00<?, ?it/s]anaconda3/envs/
↪torch24/lib/python3.12/site-packages/phiml/math/_optimize.py:631: UserWarning:␣
↪Possible rank deficiency detected. Matrix might be singular which can lead to␣
↪convergence problems. Please specify using Solve(rank_deficiency=...).
  warnings.warn("Possible rank deficiency detected. Matrix might be singular which␣
↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
↪).")
anaconda3/envs/torch24/lib/python3.12/site-packages/phiml/math/_optimize.py:631:␣
↪UserWarning: Possible rank deficiency detected. Matrix might be singular which␣
↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
↪).
  warnings.warn("Possible rank deficiency detected. Matrix might be singular which␣
↪can lead to convergence problems. Please specify using Solve(rank_deficiency=...
↪).")
Sim. only: 100%|████████████████| 100/100 [00:31<00:00,  3.14it/s]
```

```
Running test with Re=73.24219, With corr.
```

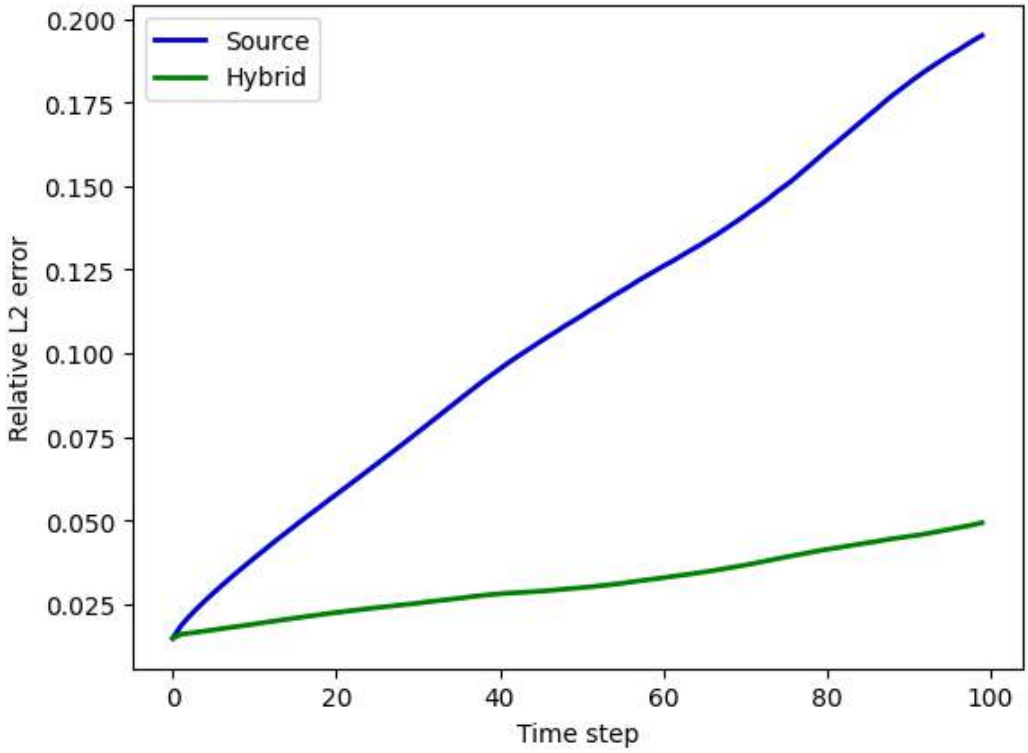
```
With corr.: 100%|████████████████| 100/100 [00:52<00:00,  1.89it/s]
```

```
 Rel. L2 errors: low-fidelity: 0.10863638424314559  corrected: 0.031211425131186844
```

Now we'll plot and compare the two versions in more detail. The next graph shows how the errors evolve over time. The relative errors of the low-fidelity solver rise linearly over time, while the hybrid solver yields a much slower increase. For larger networks and successful training runs, it can pretty much suppress any increase, and keep the errors at a very low level. This indicates, that it can successfully nudge the low-fidelity solver to preserve the accuracy of the targets. Note that this job would be much more difficult without the base solver: then the NN would need to do *all* the work. Here, it can rely on the prediction of the coupled solver, and a small correction typically suffices.

```
fig = pylab.figure().gca()
pltx = np.linspace(0,ROLLOUT_STEPS-1,ROLLOUT_STEPS)
fig.plot(pltx, err_lowfid_only, lw=2, color='mediumblue', label='Source')
fig.plot(pltx, err_corrected,   lw=2, color='green', label='Hybrid')
pylab.xlabel('Time step'); pylab.ylabel('Relative L2 error'); fig.legend()
```

```
<matplotlib.legend.Legend at 0x7de77dc7b770>
```



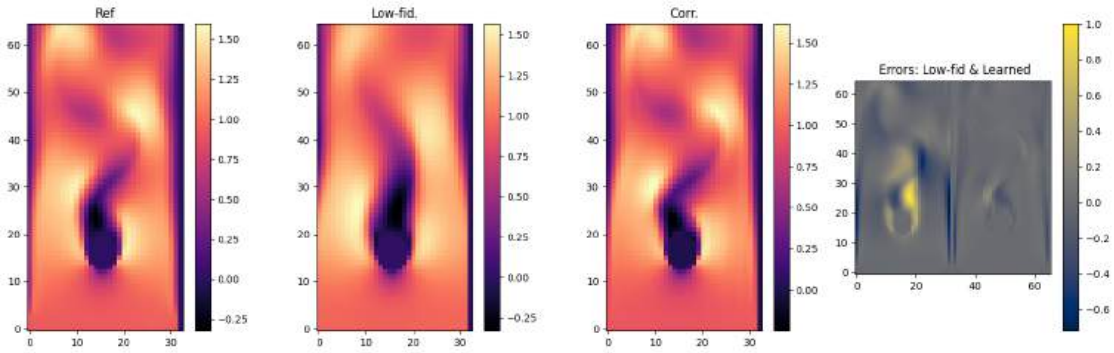
While the quantified results give an important summary of the performance of our Neural operator, it's important to sanity check these results to make sure the NN works as intended. In the next cell, we'll plot the states of the reference, the low-fidelity solver and the hybrid solver side-by-side. Additionally, we'll plot the errors made by both solvers on the right side.

```
# which step from which batch to show , by default shows last step from first case
STEP = ROLLOUT_STEPS
BATCH = 0
NUM_SHOW = 4
PRINT_STATS = False # optional, print statistics

fig, axes = pylab.subplots(1, 4, figsize=(16, 5))
i = 0

v = refs[STEP].staggered_tensor().numpy('batch,y,x,vector')[BATCH,:,:,0]
if PRINT_STATS: print(["reference ", BATCH, i, np.mean(v),  np.min(v) ,  np.max(v)])
axes[i].set_title(f"Ref")
im=axes[i].imshow( v , origin='lower', cmap='magma') ;
pylab.colorbar(im) ; i=i+1; vy_ref=v

v = prediction_lowfid_only[STEP][1].staggered_tensor().numpy('batch,y,x,vector
↪')[BATCH,:,:,0]
if PRINT_STATS: print(["low-fid. ", BATCH, i, np.mean(v),  np.min(v) ,  np.max(v)])
axes[i].set_title(f"Low-fid.")
im=axes[i].imshow( v , origin='lower', cmap='magma') ;
pylab.colorbar(im) ; i=i+1; vy_lowfid=v

v = prediction_corrected[STEP][1].staggered_tensor().numpy('batch,y,x,vector')[BATCH,
↪:,:,0]
```

(continues on next page)

(continued from previous page)

```
if PRINT_STATS: print(["corrected", BATCH, i, np.mean(v),  np.min(v) ,  np.max(v)])
axes[i].set_title(f"Corr.")
im=axes[i].imshow( v , origin='lower', cmap='magma') ;
pylab.colorbar(im) ; i=i+1; vy_corr=v

# show error side by side
err_lf   = vy_ref - vy_lowfid
err_corr = vy_ref - vy_corr
v = np.concatenate([err_lf,err_corr], axis=1)
axes[i].set_title(f" Errors: Low-fid & Learned")
im=axes[i].imshow( v , origin='lower', cmap='cividis') ;
pylab.colorbar(im) ; i=i+1

pylab.tight_layout()
```



This shows very clearly how the pure source simulation in the middle deviates from the reference on the left. The learned version stays much closer to the reference solution.

The two per-cell error images on the right also illustrate this: the source version has much larger errors (i.e. brighter colors) that show how it systematically underestimates the vortices that should form. The error for the learned version is much more evenly distributed and significantly smaller in magnitude.

This concludes our evaluation. Note that the improved behavior of the AI-powered hybrid solver can be difficult to reliably measure with simple vector norms such as an MAE or L^2 norm. To improve this, we'd need to employ other, domain-specific metrics. In this case, metrics for fluids based on vorticity and turbulence properties of the flow would be applicable. However, in this text, we instead want to focus on DL-related topics and target another inverse problem with differentiable physics solvers in the next chapter.

18.8 Next steps

- Turn off the differentiable physics training (by setting `msteps=1`), and compare it with the unrolled version. This yields a *supervised* training, as no gradients need to flow through the solver anymore. The relative errors will be substantially larger.
- Likewise, try training a network with a larger `msteps` settings, e.g., 8 or 16. Note that due to the recurrent nature of the training, you'll probably have to load a pre-trained state to stabilize the first iterations (this effectively adds "curriculum learning").
- Use the external github code to generate tougher test data, and run your trained NN on these cases. You'll see that a reduced training error not always directly correlates with an improved test performance.

CHAPTER

NINETEEN

SOLVING INVERSE PROBLEMS WITH NNS

Inverse problems encompass a large class of practical applications in science. In general, the goal here is not to directly compute a physical field like the velocity at a future time (this is the typical scenario for a *forward* solve), but instead more generically compute one or more parameters in the model equations such that certain constraints are fulfilled. A very common objective is to find the optimal setting for a single parameter given some constraints. E.g., this could be the global diffusion constant for an advection-diffusion model such that it fits measured data as accurately as possible. Inverse problems are encountered for any model parameter adjusted via observations, or the reconstruction of initial conditions, e.g., for particle imaging velocimetry (PIV). More complex cases aim for computing boundary geometries w.r.t. optimal conditions, e.g. to obtain a shape with minimal drag in a fluid flow.

A key aspect below will be that we're not aiming for solving only a *single instance* of an inverse problem, but we'd like to use deep learning to solve a *larger collection* of inverse problems. Thus, unlike the physics-informed example of *Burgers Optimization with a PINN* or the differentiable physics (DP) optimization of *Differentiable Fluid Simulations*, where we've solved an optimization problem for specific instances of inverse problems, we now aim for training an NN that learns to solve a larger class of inverse problems, i.e., a whole solution manifold. Nonetheless, we of course need to rely on a certain degree of similarity for these problems, otherwise there's nothing to learn (and the implied assumption of continuity in the solution manifold breaks down).

Below we will run a very challenging test case as a representative of these inverse problems: we will aim for computing a high dimensional control function that exerts forces over the full course of an incompressible fluid simulation in order to reach a desired goal state for a passively advected marker in the fluid. This means we only have very indirect constraints to be fulfilled (a single state at the end of a sequence), and a large number of degrees of freedom (the control force function is a space-time function with the same degrees of freedom as the flow field itself).

The *long-term* nature of the control is one of the aspects which makes this a tough inverse problem: any changes to the state of the physical system can lead to large change later on in time, and hence a controller needs to anticipate how the system will behave when it is influenced. This means an NN also needs to learn how the underlying physics evolve and change, and this is exactly where the gradients from the DP training come in to guide the learning task towards solution that can reach the goal.

[*Warning*: This code is a very "classic" one by now, and requires quite a few legacy APIs that are not supported on colab anymore (most importantly Python3.6 with TF1.14 and phiflow 1.4.1). Hence it's recommended to run this example in a local conda environment rather than colab.]

19.1 Formulation

With the notation from *Models and Equations* this gives the minimization problem

$$\arg\min_\theta \sum_m \sum_i (f(x_{m,i};\theta) - y^*_{m,i})^2,$$

where $y^*_{m,i}$ denotes the samples of the target state of the marker field, and $x_{m,i}$ denotes the simulated state of the marker density. As before, the index i samples our solution at different spatial locations (typically all grid cells), while the index m here indicates a large collection of different target states.

Our goal is to train two networks OP and CFE with weights θ_{OP} and θ_{CFE} such that a sequence

$$\mathbf{u}_n, d_n = \mathcal{P}(\text{CFE}(\ \mathcal{P}(\text{CFE}(\cdots \mathcal{P}(\text{CFE}(\mathbf{u}_0, d_0, d_{OP}))\cdots)))) = (\mathcal{P}\ \text{CFE})^n(\mathbf{u}_0, d_0, d_{OP}).$$

minimizes the loss above. The OP network is a predictor that determines the state d_{OP} that the action of the CFE should aim for, i.e., it does the longer term planning from which to determine the action. Given the target d^*, it computes $d_{OP} = \text{OP}(d, d^*) = f_{\text{OP}}(d, d^*; \theta_{\text{OP}})$. The CFE acts additively on the velocity field by computing $\mathbf{u} + f_{\text{CFE}}(\mathbf{u}, d, f_{\text{OP}}(d, d^*; \theta_{\text{OP}}); \theta_{\text{CFE}})$, where we've used f_{OP} and f_{CFE} to denote the NN representations of OP and CFE, respectively, and d^* to denote the target density state. θ_{OP} and θ_{CFE} denote the corresponding network weights.

For this problem, the model PDE $\mathcal{P}$ contains a discretized version of the incompressible Navier-Stokes equations in two dimensions for a velocity $\mathbf{u}$:

$$\begin{aligned}
\frac{\partial u_x}{\partial t} + \mathbf{u}\cdot\nabla u_x &= -\frac{1}{\rho}\nabla p \\
\frac{\partial u_y}{\partial t} + \mathbf{u}\cdot\nabla u_y &= -\frac{1}{\rho}\nabla p \\
\text{s.t.} \quad \nabla\cdot\mathbf{u} &= 0,
\end{aligned}$$

without explicit viscosity, and with an additional transport equation for the marker density d given by $\frac{\partial d}{\partial t} + \mathbf{u}\cdot\nabla d = 0$.

To summarize, we have a predictor OP that gives us a direction, an actor CFE that exerts a force on a physical model $\mathcal{P}$. They all need to play hand in hand to reach a given target after n iterations of the simulation. As apparent from this formulation, it's not a simple inverse problem, especially due to the fact that all three functions are non-linear. This is exactly why the gradients from the DP approach are so important. (The viewpoint above also indicates that *reinforcement learning* is a potential option. In *Controlling Burgers' Equation with Reinforcement Learning* we'll compare DP with these alternatives.)

19.2 Control of incompressible fluids

The next sections will walk you through all the necessary steps from data generation to network training using ΦFlow. Due to the complexity of the control problem, we'll start with a supervised initialization of the networks, before switching to a more accurate end-to-end training with DP. (*Note: this example uses an older version 1.4.1 of ΦFlow.*)

The code below replicates an inverse problem example (the shape transitions experiment) from Learning to Control PDEs with Differentiable Physics [HKT19], further details can be found in section D.2 of the paper's appendix.

First we need to load phiflow and check out the *PDE-Control* git repository, which also contains some numpy arrays with initial shapes.

```
!pip install --quiet phiflow==1.4.1

import matplotlib.pyplot as plt
from phi.flow import *

if not os.path.isdir('PDE-Control'):
  print("Cloning, PDE-Control repo, this can take a moment")
  os.system("git clone --recursive https://github.com/holl-/PDE-Control.git")

# now we can load the necessary phiflow libraries and helper functions
import sys; sys.path.append('PDE-Control/src')
from shape_utils import load_shapes, distribute_random_shape
from control.pde.incompressible_flow import IncompressibleFluidPDE
from control.control_training import ControlTraining
from control.sequences import StaggeredSequence, RefinedSequence
```

19.3 Data generation

Before starting the training, we have to generate a data set to train with, i.e., a set of ground truth time sequences u^*. Due to the complexity of the training below, we'll use a staged approach that pre-trains a supervised network as a rough initialization, and then refines it to learn control looking further and further ahead into the future. (This will be realized by training specialized NNs that deal with longer and longer sequences.)

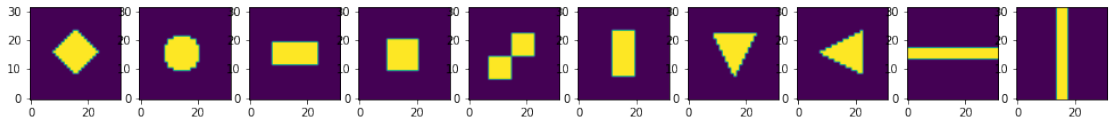
First, let's set up a domain and basic parameters of the data generation step.

```
domain = Domain([64, 64])  # 1D Grid resolution and physical size
step_count = 16  # how many solver steps to perform
dt = 1.0  # Time increment per solver step
example_count = 1000
batch_size = 100
data_path = 'shape-transitions'
pretrain_data_path = 'moving-squares'
shape_library = load_shapes('PDE-Control/notebooks/shapes')
```

The `shape_library` in the last line contains ten different shapes that we'll use to intialize a marker density with at random positions.

This is what the shapes look like:

```
import pylab
pylab.subplots(1, len(shape_library), figsize=(17, 5))
for t in range(len(shape_library)):
    pylab.subplot(1, len(shape_library), t+1)
    pylab.imshow(shape_library[t], origin='lower')
```



The following cell uses these shapes to create the dataset we want to train our network with. Each example consists of a start and target (end) frame which are generated by placing a random shape from the `shape_library` somewhere within the domain.

```
for scene in Scene.list(data_path): scene.remove()

for _ in range(example_count // batch_size):
    scene = Scene.create(data_path, count=batch_size, copy_calling_script=False)
    print(scene)
    start = distribute_random_shape(domain.resolution, batch_size, shape_library)
    end__ = distribute_random_shape(domain.resolution, batch_size, shape_library)
    [scene.write_sim_frame([start], ['density'], frame=f) for f in range(step_count)]
    scene.write_sim_frame([end__], ['density'], frame=step_count)
```

```
shape-transitions/sim_000000
shape-transitions/sim_000100
shape-transitions/sim_000200
shape-transitions/sim_000300
shape-transitions/sim_000400
shape-transitions/sim_000500
shape-transitions/sim_000600
shape-transitions/sim_000700
shape-transitions/sim_000800
shape-transitions/sim_000900
```

Since this dataset does not contain any intermediate frames, it does not allow for supervised pretraining. This is because to pre-train a CFE network, two consecutive frames are required while to pretrain an OP_n network, three frames with a distance of $n/2$ are needed.

Instead, we create a second dataset which contains these intermediate frames. This does not need to be very close to the actual dataset since it's only used for network initialization via pretraining. Here, we linearly move a rectangle around the domain.

```
for scene in Scene.list(pretrain_data_path): scene.remove()

for scene_index in range(example_count // batch_size):
    scene = Scene.create(pretrain_data_path, count=batch_size, copy_calling_
↪script=False)
    print(scene)
    pos0 = np.random.randint(10, 56, (batch_size, 2))  # start position
    pose = np.random.randint(10, 56, (batch_size, 2))  # end position
    size = np.random.randint(6,  10,  (batch_size, 2))
    for frame in range(step_count+1):
        time = frame / float(step_count + 1)
        pos = np.round(pos0 * (1 - time) + pose * time).astype(np.int)
        density = AABox(lower=pos-size//2, upper=pos-size//2+size).value_at(domain.
↪center_points())
        scene.write_sim_frame([density], ['density'], frame=frame)
```

```
moving-squares/sim_000000
moving-squares/sim_000100
moving-squares/sim_000200
moving-squares/sim_000300
moving-squares/sim_000400
moving-squares/sim_000500
moving-squares/sim_000600
moving-squares/sim_000700
moving-squares/sim_000800
moving-squares/sim_000900
```

19.4 Supervised initialization

First we define a split of the 1000 data samples into 100 test, 100 validation, and 800 training samples.

```
test_range = range(100)
val_range = range(100, 200)
train_range = range(200, 1000)
```

The following cell trains all OP_n $\forall n \in \{2, 4, 8, 16\}$. Here the n indicates the number of time steps for which the network predicts the target. In order to cover longer time horizons, we're using factors of two here to hierarchically divide the time intervals during which the physical system should be controlled.

The `ControlTraining` class is used to set up the corresponding optimization problem. The loss for the supervised initialization is defined as the observation loss in terms of velocity at the center frame:

$$L_o^{\text{sup}} = \left| \text{OP}\left(d_{t_i}, d_{t_j}\right) - d^*_{(t_i+t_j)/2} \right|^2 .$$

Consequently, no sequence needs to be simulated (`sequence_class=None`) and an observation loss is required at frame $\frac{n}{2}$ (`obs_loss_frames=[n // 2]`). The pretrained network checkpoints are stored in `supervised_checkpoints`.

Note: The next cell will run for some time. The PDE-Control git repo comes with a set of pre-trained networks. So if you want to focus on the evaluation, you can skip the training and load the pretrained networks instead by commenting out the training cells, and uncommenting the cells for loading below.

```
supervised_checkpoints = {}

for n in [2, 4, 8, 16]:
    app = ControlTraining(n, IncompressibleFluidPDE(domain, dt),
                          datapath=pretrain_data_path, val_range=val_range, train_
 ↪range=train_range, trace_to_channel=lambda _: 'density',
                          obs_loss_frames=[n//2], trainable_networks=['OP%d' % n],
                          sequence_class=None).prepare()
    for i in range(1000):
        app.progress()  # Run Optimization for one batch
    supervised_checkpoints['OP%d' % n] = app.save_model()
```

```
supervised_checkpoints # this is where the checkpoints end up when re-training:
```

```
{'OP16': '/root/phi/model/control-training/sim_000003/checkpoint_00001000',
 'OP2': '/root/phi/model/control-training/sim_000000/checkpoint_00001000',
 'OP4': '/root/phi/model/control-training/sim_000001/checkpoint_00001000',
 'OP8': '/root/phi/model/control-training/sim_000002/checkpoint_00001000'}
```

```
# supervised_checkpoints = {'OP%d' % n: 'PDE-Control/networks/shapes/supervised/OP%d_
 ↪1000' % n for n in [2, 4, 8, 16]}
```

This concludes the pretraining of the OP networks. These networks make it possible to at least perform a rough planning of the motions, which will be refined via end-to-end training below. However, beforehand we'll initialize the CFE networks such that we can perform *actions*, i.e., apply forces to the simulation. This is completely decoupled from the OP networks.

19.5 CFE pretraining with differentiable physics

To pretrain the CFE networks, we set up a simulation with a single step of the differentiable solver.

The following cell trains the CFE network from scratch. If you have a pretrained network at hand, you can skip the training and load the checkpoint by running the cell after.

```
app = ControlTraining(1, IncompressibleFluidPDE(domain, dt),
                      datapath=pretrain_data_path, val_range=val_range, train_
↪range=train_range, trace_to_channel=lambda _: 'density',
                      obs_loss_frames=[1], trainable_networks=['CFE']).prepare()
for i in range(1000):
    app.progress()  # Run Optimization for one batch
supervised_checkpoints['CFE'] = app.save_model()
```

```
# supervised_checkpoints['CFE'] = 'PDE-Control/networks/shapes/CFE/CFE_2000'
```

Note that we have not actually set up a simulation for the training, as the CFE network only infers forces between pairs of states.

19.6 End-to-end training with differentiable physics

Now that first versions of both network types exist, we can initiate the most important step of the setup at hand: the coupled end-to-end training of both networks via the differentiable fluid solver. While the pretraining stages relied on supervised training, the next step will yield a significantly improved quality for the control.

To initiate an end-to-end training of the CFE and all OP_n networks with the differentiable physics loss in phiflow, we create a new `ControlTraining` instance with the staggered execution scheme.

The following cell builds the computational graph with `step_count` solver steps without initializing the network weights.

```
staggered_app = ControlTraining(step_count, IncompressibleFluidPDE(domain, dt),
                                datapath=data_path, val_range=val_range, train_
↪range=train_range, trace_to_channel=lambda _: 'density',
                                obs_loss_frames=[step_count], trainable_networks=['CFE
↪', 'OP2', 'OP4', 'OP8', 'OP16'],
                                sequence_class=StaggeredSequence, learning_rate=5e-4).
↪prepare()
```

```
App created. Scene directory is /root/phi/model/control-training/sim_000005 (INFO),
 ↪ 2021-04-09 00:41:17,299n

Sequence class: <class 'control.sequences.StaggeredSequence'> (INFO), 2021-04-09␣
 ↪00:41:17,305n

Partition length 16 sequence (from 0 to 16) at frame 8
Partition length 8 sequence (from 0 to 8) at frame 4
Partition length 4 sequence (from 0 to 4) at frame 2
Partition length 2 sequence (from 0 to 2) at frame 1
Execute -> 1
Execute -> 2
Partition length 2 sequence (from 2 to 4) at frame 3
Execute -> 3
```

(continues on next page)

(continued from previous page)

```
Execute -> 4
Partition length 4 sequence (from 4 to 8) at frame 6
Partition length 2 sequence (from 4 to 6) at frame 5
Execute -> 5
Execute -> 6
Partition length 2 sequence (from 6 to 8) at frame 7
Execute -> 7
Execute -> 8
Partition length 8 sequence (from 8 to 16) at frame 12
Partition length 4 sequence (from 8 to 12) at frame 10
Partition length 2 sequence (from 8 to 10) at frame 9
Execute -> 9
Execute -> 10
Partition length 2 sequence (from 10 to 12) at frame 11
Execute -> 11
Execute -> 12
Partition length 4 sequence (from 12 to 16) at frame 14
Partition length 2 sequence (from 12 to 14) at frame 13
Execute -> 13
Execute -> 14
Partition length 2 sequence (from 14 to 16) at frame 15
Execute -> 15
Execute -> 16
Target loss: Tensor("truediv_16:0", shape=(), dtype=float32) (INFO), 2021-04-09␣
 ↪00:41:44,654n

Force loss: Tensor("truediv_107:0", shape=(), dtype=float32) (INFO), 2021-04-09␣
 ↪00:41:51,312n

Supervised loss at frame 16: Tensor("truediv_108:0", shape=(), dtype=float32)␣
 ↪(INFO), 2021-04-09 00:41:51,332n

Setting up loss (INFO), 2021-04-09 00:41:51,338n

Preparing data (INFO), 2021-04-09 00:42:32,417n

Initializing variables (INFO), 2021-04-09 00:42:32,443n

Model variables contain 0 total parameters. (INFO), 2021-04-09 00:42:36,418n

Validation (000000): Learning_Rate: 0.0005, GT_obs_16: 399498.75, Loss_reg_
 ↪unscaled: 1.2424506, Loss_reg_scale: 1.0, Loss: 798997.5 (INFO), 2021-04-09␣
 ↪00:42:59,618n
```

The next cell initializes the networks using the supervised checkpoints and then trains all networks jointly. You can increase the number of optimization steps or execute the next cell multiple times to further increase performance.

Note: The next cell will run for some time. Optionally, you can skip this cell and load the pretrained networks instead with code in the cell below.

```
staggered_app.load_checkpoints(supervised_checkpoints)
for i in range(1000):
    staggered_app.progress()  # run staggered Optimization for one batch
staggered_checkpoint = staggered_app.save_model()
```

```
# staggered_checkpoint = {net: 'PDE-Control/networks/shapes/staggered/all_53750' for␣
```

(continues on next page)

(continued from previous page)

```
↪net in ['CFE', 'OP2', 'OP4', 'OP8', 'OP16']}
# staggered_app.load_checkpoints(staggered_checkpoint)
```

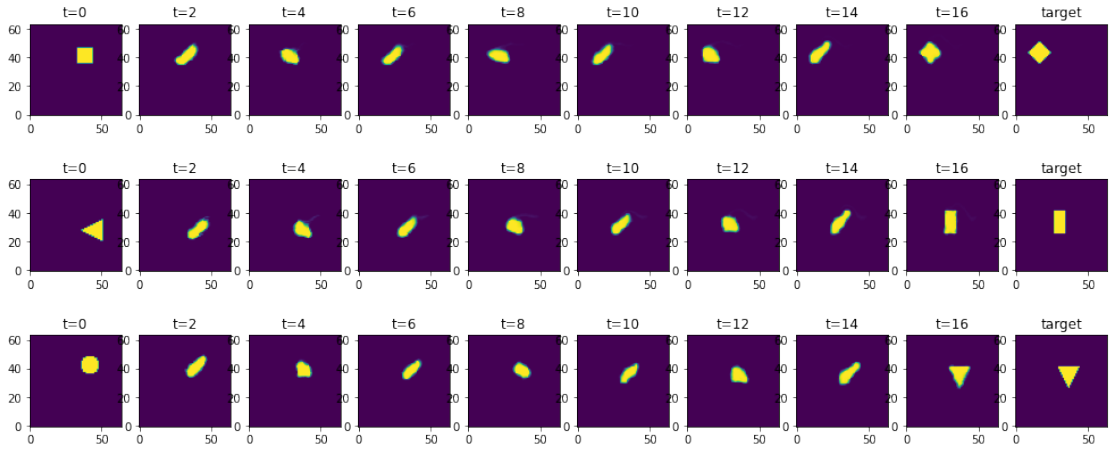
Now that the network is trained, we can infer some trajectories from the test set. (This corresponds to Fig 5b and 18b of the original paper.)

The following cell takes the first one hundred configurations, i.e. our test set as defined by `test_range`, and let's the network infer solutions for the corresponding inverse problems.

```
states = staggered_app.infer_all_frames(test_range)
```

Via the index list `batches` below, you can choose to display some of the solutions. Each row shows a temporal sequence starting with the initial condition, and evolving the simulation with the NN control forces for 16 time steps. The last step, at $t = 16$ should match the target shown in the image on the far right.

```
batches = [0,1,2]

pylab.subplots(len(batches), 10, sharey='row', sharex='col', figsize=(14, 6))
pylab.tight_layout(w_pad=0)

# solutions
for i, batch in enumerate(batches):
    for t in range(9):
        pylab.subplot(len(batches), 10, t + 1 + i * 10)
        pylab.title('t=%d' % (t * 2))
        pylab.imshow(states[t * 2].density.data[batch, ..., 0], origin='lower')

# add targets
testset = BatchReader(Dataset.load(staggered_app.data_path,test_range), staggered_app.
↪_channel_struct)[test_range]
for i, batch in enumerate(batches):
        pylab.subplot(len(batches), 10, i * 10 + 10)
        pylab.title('target')
        pylab.imshow(testset[1][i,...,0], origin='lower')
```



As you can see in the two right-most columns, the network does a very good job at solving these inverse problems: the fluid marker is pushed to the right spot and deformed in the right way to match the target.

What looks fairly simple here is actually a tricky task for a neural network: it needs to guide a full 2D Navier-Stokes simulation over the course of 16 time integration steps. Hence, if the applied forces are slightly off or incoherent, the

fluid can start swirling and moving chaotically. However, the network has learned to keep the motion together, and guide the marker density to the target location.

Next, we quantify the achieved error rate by comparing the mean absolute error in terms of the final density configuration relative to the initial density. With the standard training setup above, the next cell should give a relative residual error of 5-6%. Vice versa, this means that more than ca. 94% of the marker density ends up in the right spot!

```
errors = []
for batch in enumerate(test_range):
  initial = np.mean( np.abs( states[0].density.data[batch, ..., 0] - testset[1][batch,
↪...,0] ))
  solution = np.mean( np.abs( states[16].density.data[batch, ..., 0] -␣
↪testset[1][batch,...,0] ))
  errors.append( solution/initial )
print("Relative MAE: "+format(np.mean(errors)))
```

```
Relative MAE: 0.05450168251991272
```

19.7 Next steps

For further experiments with this source code, you can, e.g.:

- Change the `test_range` indices to look at different examples, or test the generalization of the trained controller networks by using new shapes as targets.
- Try using a `RefinedSequence` (instead of a `StaggeredSequence`) to train with the prediction refinement scheme. This will yield a further improved control and reduced density error.

CHAPTER

TWENTY

DISCUSSION OF DIFFERENTIABLE PHYSICS

The previous sections have explained the *differentiable physics* approach for deep learning, and have given a range of examples: from a very basic gradient calculation, all the way to complex learning setups powered by advanced simulations. This is a good time to take a step back and evaluate: in the end, the differentiable physics components of these approaches are not too complicated. They are largely based on existing numerical methods, with a focus on efficiently using those methods not only to do a forward simulation, but also to compute gradient information. What is primarily exciting in this context are the implications that arise from the combination of these numerical methods with deep learning.

20.1 Integration

Most importantly, training via differentiable physics allows us to seamlessly bring the two fields together: we can obtain *hybrid* methods, that use the best numerical methods that we have at our disposal for the simulation itself, as well as for the training process. We can then use the trained model to improve forward or backward solves. Thus, in the end, we have a solver that combines a *traditional* solver and a *learned* component that in combination can improve the capabilities of numerical methods.

20.2 Reducing data shift via interaction

One key aspect that is important for these hybrids to work well is to let the NN *interact* with the PDE solver at training time. Differentiable simulations allow a trained model to "explore and experience" the physical environment, and receive directed feedback regarding its interactions throughout the solver iterations.

This addresses the classic **data shift** problem of machine learning: rather than relying on a *a-priori* specified distribution for training the network, the training process generates new trajectories via unrolling on the fly, and computes training signals from them. This can be seen as an *a-posteriori* approach, and makes the trained NN significantly more resilient to unseen inputs. As we'll evaluate in more detail in *Unconditional Stability*, it's actually hard to beat a good unrolling setup with other approaches.

Note that the topic of *differentiable physics* nicely fits into the broader context of machine learning as *differentiable programming*.

20.3 Generalization

The hybrid approach also bears particular promise for simulators: it improves generalizing capabilities of the trained models by letting the PDE-solver handle large-scale *changes to the data distribution*. This allows the learned model to focus on localized structures not captured by the discretization. While physical models generalize very well, learned models often specialize in data distributions seen at training time. Hence, this aspect benefits from the previous reduction of data shift, and effectively allows for even larger differences in terms of input distribution. If the NN is set up correctly, these can be handled by the classical solver in a hybrid approach.

These benefits were, e.g., shown for the models reducing numerical errors of *Reducing Numerical Errors with Neural Operators*: the trained models can deal with solution manifolds with significant amounts of varying physical behavior, while simpler training variants would deteriorate over the course of recurrent time steps.

To summarize, the pros and cons of training NNs via DP:

✓ Pro:

- Uses physical model and numerical methods for discretization.
- Efficiency and accuracy of selected methods carries over to training.
- Very tight coupling of physical models and NNs possible.
- Improved resilience and generalization.

✗ Con:

- Not compatible with all simulators (need to provide gradients).
- Require more heavy machinery (in terms of framework support) than previously discussed methods.

Outlook: the last negative point (regarding heavy machinery) is strongly improving at the moment. Many existing simulators, e.g. the popular open source framework *OpenFoma*, as well as many commercial simulators are working on tight integrations with NNs. However, there's still plenty room for improvement, and in this book we're focusing on examples using phiflow, which was designed for interfacing with deep learning frameworks from ground up.

The training via differentiable physics (DP) allows us to integrate full numerical simulations into the training of deep neural networks. This effectively provides **hard constraints**, as the coupled solver can project and enforce constraints just like classical solvers would. It is a very generic approach that is applicable to a wide range of combinations of PDE-based models and deep learning.

In the next chapters, we will first expand the scope of the learning tasks to incorporate uncertainties, i.e. to work with full distributions rather than single deterministic states and trajectories. Afterwards, we'll also compare DP training to reinforcement learning, and target the underlying learning process to obtain even better NN states.

Part V

Probabilistic Learning

CHAPTER

TWENTYONE

INTRODUCTION TO PROBABILISTIC LEARNING

So far we've treated the target function $f(x) = y$ as being deterministic, with a unique solution y for every input. That's certainly a massive simplification: in practice, solutions can be ambiguous, our learned model might mix things up, and both effects could show up in combination. This all calls for moving towards a probabilistic setting, which we'll address here. The machinery from previous sections will come in handy, as the probabilistic viewpoint essentially introduces another dimension for the problem. Instead of a single y, we now have a multitude of solutions drawn from a distribution Y, each with a probability $p_Y(y)$, often shortened to $p(y)$. Samples $y \sim p(y)$ drawn from the distribution should follow this probability, so that we can distinguish rare and and frequent cases.

To summarize, instead of individual solutions y we're facing a large number of samples $y \sim p(y)$.

21.1 Uncertainty

All measurements, models, and discretizations that we are working with exhibit uncertainties. For measurements and observations, they typically appear in the form of measurement errors. Model equations, on the other hand, usually encompass only parts of a system we're interested in (leaving the remainder as an uncertainty), while for numerical simulations we inevitably introduce discretization errors. In the context of machine learning, we additionally have errors introduced by the trained model. All these errors and unclear aspects make up the uncertainties of the predicted outcomes, the *predictive uncertainty*. For practical applications it's crucial to have means for quantifying this uncertainty. This is a central motivation for working with probabilistic models, and for adjacent fields such as in "uncertainty quantification" (UQ).

Note

Aleatoric vs. Epistemic Uncertainty. The predictive uncertainty in many cases can be distinguished in terms of two types of uncertainty:

- *Aleatoric* uncertainty denotes uncertainty within the data, e.g., noise in measurements.
- *Epistemic* uncertainty, on the other hand, describes uncertainties within a model such as a trained neural network.

A word of caution is important here: while this distinction seems clear cut, both effects overlay and can be difficult to tell apart. E.g., when facing discretization errors, uncertain outcomes could be caused by unknown ambiguities in the data, or by a suboptimal discrete representation. These aspects can be very difficult to disentangle in practice.

Closely aligned, albeit taking a slightly different perspective, are so-called *simulation-based inference* (SBI) methods. Here the main motivation is to estimate likelihoods in computer-based simulations, so that reliable probability distributions for the solutions can be obtained. The SBI viewpoint provides a methodological approach for working with computer simulations and uncertainties, and will provide a red thread for the following sections.

21.2 Forward or Backward?

At this point it's important to revisit the central distinction between forward and inverse ("backward") problems: most classic numerical methods target ➡ **forward** ➡ problems to compute solutions for steady-state or future states of a system.

Forward problems arise in many settings, but across the board, at least as many problems are ⬅ **inverse** ⬅ problems, where a forward simulation plays a central role, but the main question is not a state that it generates, but rather the value of parameter of simulator to explain a certain measurement or observation. To formalize this, our simulator f is parametrized by a set of inputs ν, e.g., a viscosity, and takes states x to produce a modified state y. We have an observation $\tilde{y}$ and are interested in the value of ν to produce the observation. In the easiest case this inverse problem can tackled as a minimization problem $\arg\min_\nu |f(x;\nu) - \tilde{y}|_2^2$. Solving it would tell us the viscosity of an observed material, and similar problems arise in pretty much all fields, from material science to cosmology. To simplify the notation, we'll merge ν into x, and minimize for x correspondingly, but it's important to keep in mind that x can encompass any set of parameters or state samples that we'd like to solve for with our inverse problem.

In the following, we will focus on inverse problems, as these best illustrate the capabilities of the probabilistic modeling, but the algorithms discussed are not exclusively applicable to inverse problems (an example will follow).

21.3 Simulation-based Inference

For inverse problems, it is in practice not sufficient to match a single observation $\tilde{y}$. Rather, we'd like to ensure that the parameter we obtain explains a wide range of observations, and we might be interested in the possibility of multiple values explaining our observations. Similarly, quantifying the uncertainty of the estimate is important in real world settings: is the observation explained by only a very narrow range of parameters, or could the parameter vary by orders of magnitude without really influencing the observation? These questions require a statistical analysis, typically called *inference*, to draw conclusions about the results obtained from the inverse problem solve. To connect this viewpoint with the distinction regarding epistemic and aleatoric uncertainties above, we're primarily addressing the latter here: which uncertainties lie in our observations, given a scientific hypothesis in the form of a simulator.

To formalize these inverse problems let's consider a vector-valued input x that can contain states and / or the aforementioned parameters (like ν). We also have a distribution of latent variables $z \sim p(z|x)$ that describes the unknown part of our system. Examples for z are unobservable and stochastic variables , intermediate simulation steps, or the control flow of simulator.

Note

Bayes theorem is fundamental for all of the following. For completeness, here it is: $p(x|y)\ p(y) = p(y|x)\ p(x)$. And it's worth keeping in mind that both sides are equivalent to the joint probabilities, i.e. $\ldots = p(x,y) = p(y,x)$.

For x there is a prior distribution X with a probability density $p(x)$ for the inputs, and the simulator produces an observation or output $y \sim p(y|x,z)$. Thus, x can take different values, maybe it contains some noise, and the z is out of our control, and can likewise influence the y that are produced.

The function for the conditional probability $p(y|x)$ is called the **likelihood** function, and is a crucial value in the following. Note that it does not depend on z, as these latent states are out of our control. So we actually need to compute the marginal likelihood $p(y|x) = \int p(y,z|x)dz$ by integrating over all possible z. This is necessary because the likelihood function shouldn't depend on z, otherwise we'd need to know the exact values of z before being able to calculate the likelihood. Unfortunately, this is often intractable, as z can be difficult to sample, and in some case we can't even control it in a reasonable way. Some algorithms have been proposed to compute likelihoods, one popular one is Approximate Bayesian Computation (ABC), but all approaches are highly expensive and require a lot of expert knowledge to set up. They suffer from the *curse of dimensionality*, i.e. become very expensive when facing larger numbers of degrees of freedom. Thus, obtaining good approximations of the likelihood will be a topic that we'll revisit below.

With a function for the likelihood we can compute the **distribution of the posterior**, the main quantity we're after, in the following way: $p(x|y) = \frac{p(y|x)p(x)}{\int p(y|x')p(x')dx'}$, where the denominator $\int p(y|x')p(x')dx'$ is called the *evidence*. The evidence is just $p(y)$, which shows that the equation for the posterior follows directly from Bayes' theorem $p(x|y) = p(y|x)p(x)/p(y)$.

The evidence can be computed with stochastic methods such as Markov Chain Monte Carlo (MCMC). It primarily "normalizes" our posterior distribution and is typically easier to obtain than the likelihood, but nonetheless still a challenging term.

Leveraging Deep Learning

This is were deep learning turns out to be extremely useful: we can use it to train a conditional density estimator $q_\theta(x|y)$ for the posterior $p(x|y)$ that allows sampling, and can be trained from simulations $y \sim p(y|x)$ alone.

Deep learning has been instrumental to provide new ways of addressing the classic challenges of obtaining accurate estimates of posterior distributions, and this is what we'll focus on in this chapter. Previously, we called our neural networks f_θ, but in the following we'll use $q_\theta = f_\theta$ to make clear we're dealing with a learned probability. Specifically, we'll target neural networks that learn a probability density, i.e. $\int q_\theta(x)dx = 1$. We'll often first target unconditional densities, and then show how they can be modified to learn conditional versions $q_\theta(x|y)$.

Looking ahead, the learned SBI methods, i.e. approaches for computing posterior distributions, have the following properties:

✓ Pro:

- Fast inference (once trained)
- Less affected by curse of dimensionality
- Can represent arbitrary priors

✗ Con:

- Require costly upfront training
- Lacks rigorous theoretical guarantees

In the following we'll explain how to obtain and derive a very popular and powerful family of methods that can be summarized as **diffusion models**. We could simply provide the final algorithm (which will turn out to be surprisingly

simple), but it's actually very interesting to see where it all comes from. We'll focus on the basics, and leave the *physics-based extensions* (i.e. including differentiable simulators) for a later section. The path towards diffusion models also introduces a few highly interesting concepts from machine learning along the way, and provides a nice "red thread" for discussing seminal papers from the past few years. Here we go…

Note

Historic Alternative: Bayesian Neural Networks

A classic variant that should be mentioned here are "Bayesian Neural Networks". They follow Bayes more closely, and pre-scribe a prior distribution on the neural network parameters to learn the posterior distribution. Every weight and bias in the NN are assumed to be Gaussian with an own mean and variance, which are adjusted at training time. For inference, we can then "sample" a network, and use it like any regular NN. Despite being a very good idea on paper, this method turned out to have problems with learning complex distributions, and requires careful tuning of the hyperparameters involved. Hence, these days, it's strongly recommended to use flow matching (or at least a diffusion model) instead. If you're interested in details, BNNs with a code example can be found, e.g., in v0.3 of PBDL: https://arxiv.org/abs/2109.05237v3 .

CHAPTER

TWENTYTWO

LEARNING A PROBABILITY DISTRIBUTION

First, let's target a general, and very basic question: how can we learn a *probability distribution*? If we have some knowledge about the problem at hand, we can choose a distribution, e.g. a Gaussian in the simplest case, and we have samples drawn from the target distribution, we can train by computing a difference between the target and the parametrized distribution to be learned.

22.1 Fundamentals: A Training Objective

A particularly simple and useful metric here is the *Kullback-Leibler (KL) divergence* between two probability distributions P and Q with densities p and q. You've surely comes across it already as it's a widely used loss term. The KL divergence is defined as $\mathrm{KL}(p||q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$.

It is strictly larger than zero, $\mathrm{KL}(p||q) \geq 0$, and has the nice property that $\mathrm{KL}(p||q) = 0$ if and only if P and Q are identical.

Now let's say there is a family of densities $\{q_\theta\}_\theta$ which is parameterized by θ. It is important to stress here that all q_θ need to be a valid density. That means that q_θ is non-negative and if we integrate q_θ over the entire probability space, we obtain 1, i.e. $\int q_\theta(x)dx = 1$. Now among all these densities in our family $\{q_\theta\}_\theta$, we want to find the density parameterized by θ that is as close to p as possible. We can find such a density parameterized by θ by minimizing$\mathrm{KL}(p||q_\theta)$.

The training objective from the KL divergence can be rewritten as

$$\begin{aligned}\mathrm{KL}(p||q_\theta) &= \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \\ &= \mathbb{E}_{x\sim p(x)}\left[\log\left(\frac{p(x)}{q_\theta(x)}\right)\right] \\ &= \mathbb{E}_{x\sim p(x)}[\log p(x)] - \mathbb{E}_{x\sim p(x)}[\log q_\theta(x)].\end{aligned}$$

Looking at these two terms, the first one doesn't even depend on θ. The training objective for θ can be simplified to minimizing the second term only $\mathbb{E}_{x\sim p(x)}[-\log q_\theta(x)]$. This means we can train $q_\theta(x)$ simply by sampling from p, and minimizing the negative log-likelihood for $q_\theta(x)$.

22.2 From Unconditional to Conditional

This very simply setup is an unconditional one. That means there is no additional information or inputs available and the distribution P that we want to learn is fixed. However, in many cases we want to condition the distribution P on an observation or additional input y. This is the case when we want to learn the posterior distribution $P|Y$ with density $p(x|y)$ depending on y. Instead of working with the unconditional densities $p(x)$, we consider the conditional densities $p(x|y)$. Note that we also need to include the information y in our family of proposal densities $\{p(x|y)_\theta\}_\theta$. In the updated objective, we include an additional expectation for sampling y.

$$\mathbb{E}_{y\sim p(y)}\left[\mathbb{E}_{x\sim p(x|y)}[-\log q_\theta(x|y)]\right] \ .$$

With Bayes' theorem, we can directly rewrite this as

$$\begin{aligned}
\mathbb{E}_{y\sim p(y)}\left[\mathbb{E}_{x\sim p(x|y)}[-\log q_\theta(x|y)]\right] &= -\int\int p(y)p(x|y)\ \log p(x|y)dxdy \\
&= -\int\int p(y,x)\ \log p(x|y)dxdy \\
&= -\int\int p(x)p(y|x)\ \log p(x|y)dydx \\
&= \mathbb{E}_{x\sim p(x),y\sim p(y|x)}[-\log q_\theta(x|y)] \ .
\end{aligned}$$

In simulation-based inference and scientific machine learning, $p(x)$ represents our prior and y an observation. Given a realization from the prior $x \sim p(x)$, we can sample an observation $y \sim p(y|x)$ using for example numerical simulations. The most interesting but also difficult quantity to find for the scientist is the posterior $p(x|y)$.

The above equations tell us that we can learn the posterior $p(x|y)$ by finding an optimal θ in our family of proposal densities $\{p(x|y)_\theta\}_\theta$. For that, we need to minimize $\$\mathbb{E}_{x\sim p(x),y\sim p(y|x)}[-\log q_\theta(x|y)].\$$

This only requires sampling from the prior x and drawing observations $y \sim p(y|x)$. Those are two things that we know how to do them!

Hence, even for conditional probabilities like the posterior of our inverse problems, we can use an extremely simple training objective … *if and only if* we can make sure that q_θ is a probability density. How to enforce this is the topic of the next section. It's worth pointing out that *negative log-likelihood* training for Gaussian densities is actually equivalent to minimizing an L^2 error. This is a nice conceptual connection towards earlier topics like *Supervised Training*, but below we'd like to get away from being restricted to simple Guassian distributions.

22.3 Learning Distributions with Normalizing Flows

Various ways to learn distributions and probabilities have been proposed in the deep learning area, and *Normalizing Flows* are a particularly powerful one [KPB20]. These flows generally target transformations of a simple base distribution $p_Z{}^{\circ_{\mathrm{B}_j}}$ into a potentially complicated target distribution ${}^{\circ_{\mathrm{B}_j}}p_Y$. The key idea is to use a sequence of invertible and differentiable mappings as layers for the neural network. Hence the *"normalizing"* in the name. Here, p_Z resembles the latent states from the SBI section above, but a restriction is that it needs to be a (simple) distribution we can easily sample from.

For a single, invertible mapping $g : \mathbb{R}^D \to \mathbb{R}^D{}^{\circ_{\mathrm{B}_j}}$ and inverse function $f = g^{-1}{}^{\circ_{\mathrm{B}_j}}$, we have $y = g(y)^{\circ_{\mathrm{B}_j}}$ and, vice versa, ${}^{\circ_{\mathrm{B}_j}}\, z = f(y)$. The probability density $p_Y(y)$ can be computed as $p_Y(y) = p_Z(f(y))\left|\det\frac{\partial f}{\partial y}\right|$. Here $\frac{\partial f}{\partial y}$ is the Jacobian of f, and the magnitude of its determinant provides the scaling of y due to f. $P_Z{}^{\circ_{\mathrm{B}_j}}$ is usually a normal Gaussian, so luckily evaluating $p_Z\big(f(\,y)\big)^{\circ_{\mathrm{B}_j}}$ is easy.

If p_Y is a function with internal parameters, this similarly holds for the log likelihoods w.r.t. its parameters: $\log p_Y(y) = \log p_Z(f(y)) + \log\left|\det\frac{\partial f}{\partial y}\right|$. Hence a convenient way to perform maximum likelihood training later is via a KL divergence

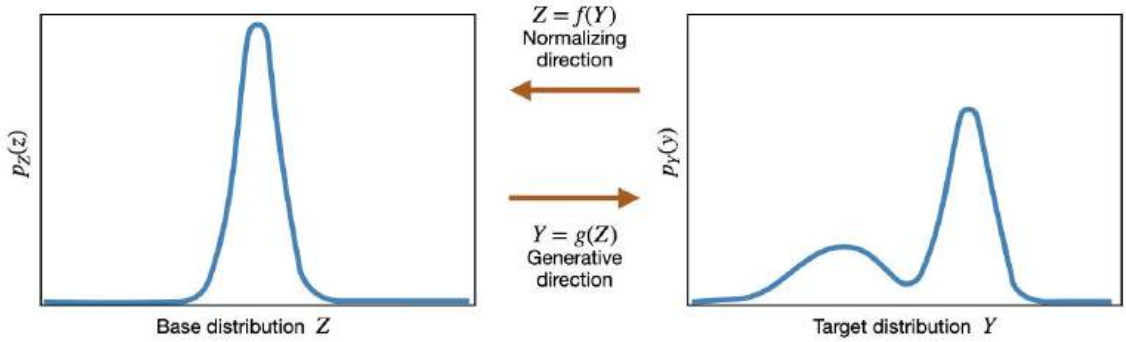


Fig. 22.1: A visual example: a simple Gaussian (left) is transformed into a non-trivial target distribution (right).

term (unconditional or conditional) as described just above. Note that maximizing probability densities is equivalent to maximizing likelihoods (this follows from the fundamental theorem of calculus), and hence many works (and the following explanations) switch back and forth between them.

Of course, we're not restricted to single functions g and f. The same holds for a sequence of mappings $g = g_1 \circ g_2 \circ ... \circ g_n$ and $f = f_n \circ f_{n-1} \circ ... \circ f_1$, where each g_i is the inverse of f_i. The probability density $p_Y(y)$ can still be written as $p_Y(y) = p_Z(f(Z)) \prod_{i=1}^{n} \left| \det \frac{\partial f_i}{\partial y_i} \right|$.

Sampling is also very convenient in this settings: draw a random vector from $p_Z(z)$, which is usually a normal Gaussian. We obtain a sample from the target distribution via $y = g(z)$, and it's probability is computed by transforming $p_Z(z)$ into p_Y, as outlined above. For y we use the "forward" sequence with all the g_i, while the "backward" sequence f_i, together with its Jacobians, provides the right probability density. This is great, and works for *unconditional* sampling as well as *conditional* sampling with the simple negative log likelihood objective.

22.4 Practical Example: Learning Gaussians

Let's use this theoretical knowledge to learn a probability distribution. First, we define a probability distribution P that consists of multiple Gaussians. We want to be able to sample from this distribution and evaluate the likelihoods.

```
import numpy as np

class GaussianMixture:
    def __init__(self, parameters):

        self.parameters = parameters
        self.distributions = [
            {
                'mean': np.array(dist['mean']),
                'std': np.array(dist['std']),
                'cov': np.diag(np.array(dist['std']) ** 2)
            }
            for dist in parameters
        ]

    def sample(self, num_samples):
        samples = []
        num_distributions = len(self.distributions)
```

(continues on next page)

(continued from previous page)

```
        for _ in range(num_samples):
            idx = np.random.randint(num_distributions)  # Choose a random Gaussian
            dist = self.distributions[idx]
            sample = np.random.multivariate_normal(mean=dist['mean'], cov=dist['cov'])
            samples.append(sample)
        return np.array(samples)

    def likelihood(self, points):
        likelihoods = np.zeros(points.shape[0])
        for dist in self.distributions:
            mean = dist['mean']
            cov = dist['cov']
            inv_cov = np.linalg.inv(cov)
            det_cov = np.linalg.det(cov)

            # Multivariate Gaussian PDF
            factor = 1 / (2 * np.pi * np.sqrt(det_cov))
            diff = points - mean
            exponents = -0.5 * np.sum(diff @ inv_cov * diff, axis=1)
            likelihoods += factor * np.exp(exponents)

        return likelihoods
```

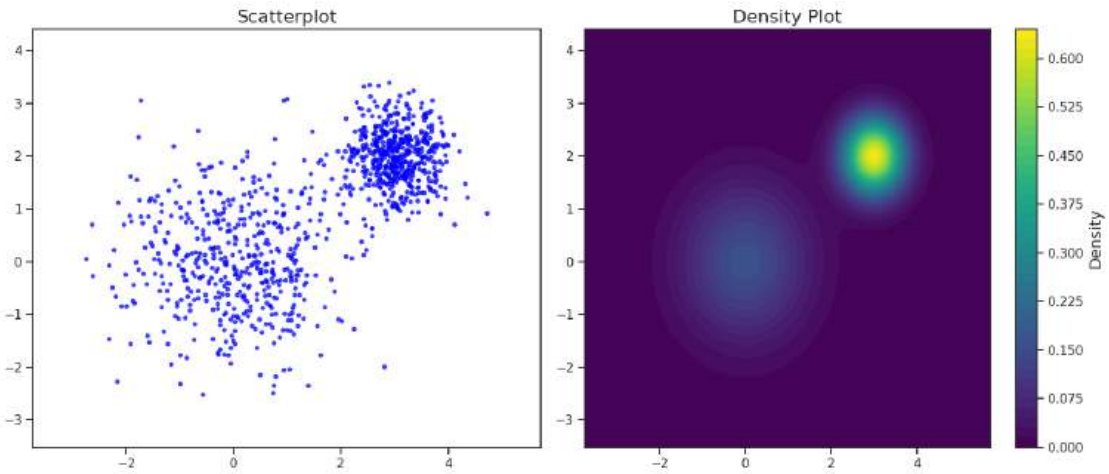
The above class `GaussianMixture` (GM) can be used to sample from multiple 2D Gaussian distributions and we can use it to evaluate the likelihood at different positions. Now, let's visualize how the samples are distributed and what the likelihood looks like. For this, we define a function for plotting first and then use it to plot a model consisting of two Gaussians with different variances and means.

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

sns.set_theme(style="ticks")

def plot_gaussian_mixture(gm, samples, grid_size=100):

    x_min, x_max = np.min(samples[:, 0]) - 1, np.max(samples[:, 0]) + 1
    y_min, y_max = np.min(samples[:, 1]) - 1, np.max(samples[:, 1]) + 1
    x = np.linspace(x_min, x_max, grid_size)
    y = np.linspace(y_min, y_max, grid_size)
    X, Y = np.meshgrid(x, y)
    points = np.column_stack([X.ravel(), Y.ravel()])
    densities = gm.likelihood(points).reshape(grid_size, grid_size)

    fig, axes = plt.subplots(1, 2, figsize=(14, 6))

    ax1 = axes[0]
    ax1.scatter(samples[:, 0], samples[:, 1], s=10, alpha=0.7, color="blue")
    ax1.set_title("Scatterplot", fontsize=16)
    ax1.set_xlim(x_min, x_max)
    ax1.set_ylim(y_min, y_max)

    ax2 = axes[1]
    contour = ax2.contourf(X, Y, densities, cmap="viridis", levels=50)
    cbar = fig.colorbar(contour, ax=ax2)
    cbar.set_label("Density", fontsize=14)
```

(continues on next page)

(continued from previous page)

```
    ax2.set_title("Density Plot", fontsize=16)
    ax2.set_xlim(x_min, x_max)
    ax2.set_ylim(y_min, y_max)

    plt.tight_layout()
    plt.show()

parameters = [
    {"mean": [0, 0], "std": [1, 1]},
    {"mean": [3, 2], "std": [0.5, 0.5]}
]
gm = GaussianMixture(parameters)

samples = gm.sample(1000)
plot_gaussian_mixture(gm, samples)
```



We'll use this distribution as a starting point for the following code examples.

22.5 A Simple Normalizing Flow based on Affine Couplings

Let's build a simple network that puts these ideas to use. We'll use a fully connected NN (`FCNN` below) with three layers and ReLU activations as a building block to turn it into an invertible layer as described above. The cell below then provides a base class `RealNVP2D` that concatenates multiple of these building blocks (6 in our example b elow) to form an NN that we can train to learn our toy GM distribution shown just above.

```
import torch.nn as nn

class FCNN(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):

        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
```

(continues on next page)

(continued from previous page)

```
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
        return self.net(x)

class NVPBlock2D(nn.Module):
    def __init__(self, dim_flow, hidden_dim=256, flip=False):
        super().__init__()
        self.dim_flow = dim_flow
        self.hidden_dim = hidden_dim
        self.flip = flip

        self.f = FCNN((dim_flow // 2), dim_flow, hidden_dim)

    def shift_and_log_scale_fn(self, x1):
        s = self.f(x1)
        shift, log_scale = torch.chunk(s, 2, dim=1)
        return shift, log_scale

    def forward(self, x, ldj=None):
        d = self.dim_flow // 2
        x1, x2 = x[:, :d], x[:, d:]
        if self.flip:
            x1, x2 = x2, x1

        fcnn_input = x1

        shift, log_scale = self.shift_and_log_scale_fn(fcnn_input)
        y2 = x2 * torch.exp(log_scale) + shift

        if self.flip:
            x1, y2 = y2, x1
        z = torch.cat([x1, y2], dim=-1)

        if ldj is not None:
            ldj = ldj + log_scale.sum(dim=-1)

        return z, ldj

    def inverse(self, z, ldj=None):

        d = self.dim_flow // 2
        y1, y2 = z[:, :d], z[:, d:]
        if self.flip:
            y1, y2 = y2, y1

        fcnn_input = y1

        shift, log_scale = self.shift_and_log_scale_fn(fcnn_input)
        x2 = (y2 - shift) * torch.exp(-log_scale)  # Apply inverse affine␣
↪transformation

        if self.flip:
            y1, x2 = x2, y1
        x = torch.cat([y1, x2], dim=-1)
```

(continues on next page)

(continued from previous page)

```
        if ldj is not None:
            ldj = ldj - log_scale.sum(dim=-1)

        return x, ldj

class RealNVP2D(nn.Module):
    def __init__(self, dim_flow, steps=6, hidden_dim=256):
        super().__init__()
        self.flows = nn.ModuleList()
        flip = False

        for _ in range(steps):
            self.flows.append(NVPBlock2D(dim_flow, hidden_dim, flip=flip))
            flip = not flip

    def forward(self, x, num_layers=None):

        if num_layers is None:
            num_layers = len(self.flows)

        ldj = torch.zeros(x.shape[0], device=x.device)
        for flow in self.flows[:num_layers]:
            x, ldj = flow(x, ldj)
        return x, ldj

    def inverse(self, z, num_layers=None):

        if num_layers is None:
            num_layers = len(self.flows)

        ldj = torch.zeros(z.shape[0], device=z.device)
        for flow in list(reversed(self.flows[:num_layers])):
            z, ldj = flow.inverse(z, ldj)
        return z, ldj
```

22.5.1 Setup Dataset and Train the Normalizing Flow

As dataset we'll simply sample from the GM, allocate a `RealNVP` model, and train it for the chosen number of epochs (50 below).

```
import torch
from torch.utils.data import DataLoader, TensorDataset

def generate_2d_gaussian_mixture(num_samples, gm):
    samples = gm.sample(num_samples)
    return torch.tensor(samples, dtype=torch.float32)

def train_model(model, dataloader, optimizer, num_epochs=50, device="cuda"):
    model.train()
    model.to(device)
    losses = []
    for epoch in range(num_epochs):
        epoch_loss = 0.0
        for x in dataloader:
            x = x[0].to(device)
```

(continues on next page)

(continued from previous page)

```
            optimizer.zero_grad()

            z, ldj = model(x)
            prior = (-0.5 * z ** 2).sum(-1) - 0.5 * torch.log(torch.tensor(2.0 *␣
↪torch.pi))
            loss = (-prior - ldj).mean()

            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(dataloader)
        losses.append(avg_loss)
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.4f}")
    return losses

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

from sklearn.utils import shuffle

samples = generate_2d_gaussian_mixture(50000, gm)
samples = shuffle(samples.numpy())
dataset = TensorDataset(torch.tensor(samples, dtype=torch.float32))
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

dim_flow = 2
steps = 6
hidden_dim = 256

realnvp_model = RealNVP2D(dim_flow, steps, hidden_dim).to(device)
optimizer = torch.optim.Adam(realnvp_model.parameters(), lr=2e-4)

# Step 3: Train the model
num_epochs = 50
losses = train_model(realnvp_model, dataloader, optimizer, num_epochs=num_epochs,␣
↪device=device)
```

```
Epoch 1/50, Loss: 2.6399
Epoch 2/50, Loss: 1.9512
Epoch 3/50, Loss: 1.9392
Epoch 4/50, Loss: 1.9355
Epoch 5/50, Loss: 1.9310
Epoch 6/50, Loss: 1.9323
Epoch 7/50, Loss: 1.9339
Epoch 8/50, Loss: 1.9270
Epoch 9/50, Loss: 1.9281
Epoch 10/50, Loss: 1.9275
...
Epoch 40/50, Loss: 1.9234
Epoch 41/50, Loss: 1.9278
Epoch 42/50, Loss: 1.9236
Epoch 43/50, Loss: 1.9236
Epoch 44/50, Loss: 1.9216
Epoch 45/50, Loss: 1.9222
Epoch 46/50, Loss: 1.9236
Epoch 47/50, Loss: 1.9215
```

(continues on next page)

(continued from previous page)

```
Epoch 48/50, Loss: 1.9226
Epoch 49/50, Loss: 1.9202
Epoch 50/50, Loss: 1.9213
```

22.5.2 Visualizing the Likelihood of the Trained Normalizing Flow

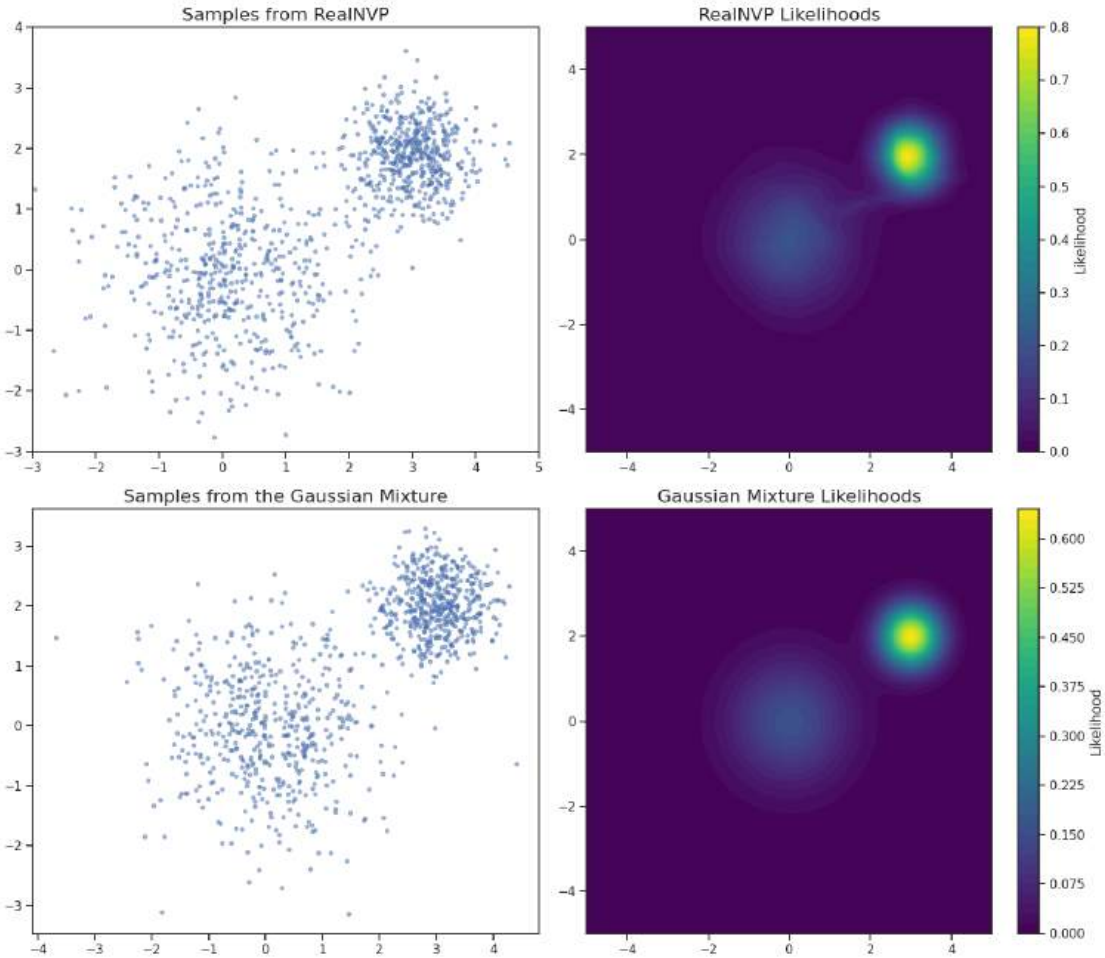
A main motivation for the simple Gaussian mixture distribution as learning target is that we can easily verify learning success with visualizations. Hence, the cell below plots samples from the original and the learned distribution to qualitatively verify that the normalizing flow model has learned to approximate the target distribution. Also, we can now visualize the likelihoods by sampling the distributions in a dense grid. The corresponding images are shown on the right.

```
def visualize_training_results(model, gm, grid_size=100, dim=2, model_desc='Model'):

    model.eval()
    with torch.no_grad():
        z = torch.randn(1000, dim).to(device)
        samples, _ = model.inverse(z)

    samples = samples.cpu().numpy()
    gm_samples = gm.sample(1000)

    x = np.linspace(-5, 5, grid_size)
    y = np.linspace(-5, 5, grid_size)
    X, Y = np.meshgrid(x, y)
    points = np.column_stack([X.ravel(), Y.ravel()])

    with torch.no_grad():

        points_tensor = torch.tensor(points, device=device, dtype=torch.float32)
        z, ldj = model(points_tensor)
        prior = (-0.5 * z ** 2).sum(-1) - 0.5 * torch.log(torch.tensor(2.0 * torch.
↪pi))
        model_likelihoods = torch.exp(prior + ldj).cpu().numpy().reshape(grid_size,␣
↪grid_size)

        gm_likelihoods = gm.likelihood(points).reshape(grid_size, grid_size)

    fig, axes = plt.subplots(2, 2, figsize=(14, 12))

    axes[0, 0].scatter(samples[:, 0], samples[:, 1], s=10, alpha=0.5, label="")
    axes[0, 0].set_title(f"Samples from {model_desc}", fontsize=16)
    axes[0, 0].set_xlabel('')
    axes[0, 0].set_ylabel('')

    contour = axes[0, 1].contourf(X, Y, model_likelihoods, levels=50, cmap="viridis")
    fig.colorbar(contour, ax=axes[0, 1], label="Likelihood")
    axes[0, 1].set_title(f"{model_desc} Likelihoods", fontsize=16)
    axes[0, 1].set_xlabel('')
    axes[0, 1].set_ylabel('')

    axes[1, 0].scatter(gm_samples[:, 0], gm_samples[:, 1], s=10, alpha=0.5, label="")
    axes[1, 0].set_title("Samples from the Gaussian Mixture", fontsize=16)
    axes[1, 0].set_xlabel('')
    axes[1, 0].set_ylabel('')
```

(continues on next page)

(continued from previous page)

```
    contour = axes[1, 1].contourf(X, Y, gm_likelihoods, levels=50, cmap="viridis")
    fig.colorbar(contour, ax=axes[1, 1], label="Likelihood")
    axes[1, 1].set_title("Gaussian Mixture Likelihoods", fontsize=16)
    axes[1, 1].set_xlabel('')
    axes[1, 1].set_ylabel('')

    for i in range(1):
        for j in range(1):
            axes[i, j].set_xlim([-3,5])
            axes[i, j].set_ylim([-3,4])

    plt.tight_layout()
    plt.show()

visualize_training_results(realnvp_model, gm, model_desc='RealNVP')
```



As expected, our network has learned likelihoods that nicely approximate the reference likelihoods.

22.5.3 Visualizing Different Layers

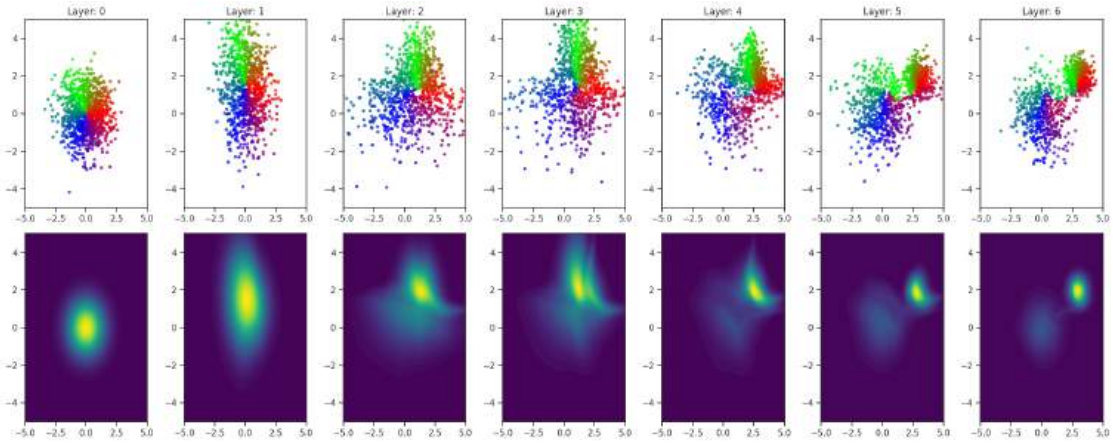
The invertible RealNVP network consisted of six layers, that step by step transform the prior distribution into the posterior. As the mapping of each layer is density-mass preserving, we can inspect what happens step by step. This is shown via the cell below:

```
import matplotlib.pyplot as plt

def get_angle_colors(positions):
    angles = np.arctan2(positions[:, 1], positions[:, 0])
    angles_deg = (np.degrees(angles) + 360) % 360
    colors = np.zeros((len(positions), 3))
    for i, angle in enumerate(angles_deg):
        segment = int(angle / 120)
        local_angle = angle - segment * 120  # angle within segment [0, 120]
        if segment == 0:
            colors[i] = [1 - local_angle/120, local_angle/120, 0]
        elif segment == 1:
            colors[i] = [0, 1 - local_angle/120, local_angle/120]
        else:
            colors[i] = [local_angle/120, 0, 1 - local_angle/120]

    return colors

def visualize_progression_with_layers_and_likelihoods(model, grid_size=100, num_
↪layers_max=6, num_samples=1000):

    model.eval()
    fig, axes = plt.subplots(2, num_layers_max + 1, figsize=(20, 8))

    x = np.linspace(-5, 5, grid_size)
    y = np.linspace(-5, 5, grid_size)
    X, Y = np.meshgrid(x, y)
    points = np.column_stack([X.ravel(), Y.ravel()])
    points_tensor = torch.tensor(points, dtype=torch.float32).to(device)

    for num_layers in range(num_layers_max + 1):

        z = torch.randn(num_samples, model.flows[0].dim_flow).to(device)

        c = get_angle_colors(z.detach().cpu().numpy())

        with torch.no_grad():
            samples, _ = model.inverse(z, num_layers=num_layers)

        samples = samples.cpu().numpy()

        scatter_ax = axes[0, num_layers]
        scatter_ax.scatter(samples[:, 0], samples[:, 1], s=10, alpha=0.7, c=c)
        scatter_ax.set_title(f"Layer: {num_layers}")
        scatter_ax.set_xlim(-5, 5)
        scatter_ax.set_ylim(-5, 5)
        scatter_ax.set_xlabel("")
        scatter_ax.set_ylabel("")

        with torch.no_grad():
            z, ldj = model(points_tensor, num_layers=num_layers)
            prior = (-0.5 * z ** 2).sum(-1) - 0.5 * torch.log(torch.tensor(2.0 *␣
```

(continues on next page)

(continued from previous page)

```
↪torch.pi))
            likelihoods = torch.exp(prior + ldj).cpu().numpy().reshape(grid_size,␣
↪grid_size)

        likelihood_ax = axes[1, num_layers]
        contour = likelihood_ax.contourf(X, Y, likelihoods, levels=50, cmap="viridis")

        likelihood_ax.set_xlim(-5, 5)
        likelihood_ax.set_ylim(-5, 5)
        likelihood_ax.set_xlabel("")
        likelihood_ax.set_ylabel("")

    plt.tight_layout()
    plt.show()

visualize_progression_with_layers_and_likelihoods(realnvp_model, grid_size=100, num_
↪layers_max=6, num_samples=1000)
```



Interestingly, the network does not take a very intuitive "route" to reach the target: the prior distribution is warped in somewhat arbitrary ways, which however, step by step move closer to the target distribution. As there are no constraints on the intermediate distributions, these strongly depend on the random initialization, and the network only receives no gradients about the final distribution. Hence, the intermediates can retain their arbitrary shapes as long as the final state matches.

22.6 Neural ODEs: Making Normalizing Flows Continuous

In the Normalizing Flow based on affine coupling layers that we had considered in the previous section, there are a total of 6 layers and we could observe how the initial Gaussian distribution is transformed step-by-step to the target distribution.

A big limitation of this type of normalizing flow is that the architecture needs to be invertible. Most efficient architectures in deep learning are not invertible.

Our next goal is to get rid of this limitation so that we can use an arbitrary architecture. At the same time, this will address another shortcoming. Right now, the number of layers is fixed. Is it possible that instead of using a fixed number of layers, we can make the number of layers variable? If we imagine the transformation of the sampling distribution to the target distribution to be a continuous process instead of a sequence of invertible mappings, we can introduce an artificial "time" t. Our normalizing flow should transform the sampling distribution smoothly from time $t = 0$ to the target distribution at time $t = 1$.

The way in which we can achieve this are *Neural ODEs*[CRBD19]. They're especially interesting in the context of physics simulations. Mathematically speaking, they replace the mapping g_i with a learned velocity predictor g_θ. This means

$$\frac{\partial z(t)}{\partial t} = g_\theta(z(t), t).$$

Then the sequence of g_i steps that made up g can be replaced by integrating the velocity, which gives a simple ODE integration. The continuous time axis is introduced, with $t = 0$ starting at a normal Gaussian distribution, to $t = 1$ for the target distribution. The ODE is solved along this timeline, e.g. to transform the base distribution p_Z into the target p_Y by querying g for a velocity at each time point along the way. Even better, for an ODE solve there's an analytic formulation for the gradient of the backpropagation path. This is a neat example of a differentiable physics solver (the ODE solve), providing an efficient way to compute a gradient, and aligns with the topics discussed in *Scale-Invariance and Inversion*.

The change in probabilities over time can also be computed conveniently via the trace of the learned function:

$$\frac{\partial \log p(z(t))}{\partial t} = -\mathrm{Tr}\left(\frac{\partial g_\theta}{\partial z(t)}\right).$$

Compared to the Normalizing Flows above, an important difference in the NeuralODE picture is that now we have a single function $g_\theta(\cdot, t)$ that is repeatedly evaluated at different points in the time interval $t \in [0, 1]$. This might seem like a trivial change at first, but it's a crucial step towards more powerful probabilistic models such as diffusion models. It turns out to be important that we can re-use a learned function, instead of having to manually construct many layers with large numbers of trainable parameters.

22.6.1 Building a Continuous Normalizing Flow

In the next cell's we'll use the *Free-form Jacobian of Reversible Dynamics* (FFJORD) architecture (from here) to implement a continuous normalizing flow.

```
import torch.nn as nn

def kernel_init_fn():
    return nn.init.xavier_uniform_

def bias_init_fn():
    return nn.init.zeros_

class ConcatSquash(nn.Module):
    def __init__(self, in_size, out_size):
        super().__init__()
        self.out_size = out_size

        self.lin1 = nn.Linear(in_size, out_size)
        self.lin2 = nn.Linear(1, out_size)
        self.lin3 = nn.Linear(1, out_size, bias=False)

        kernel_init = kernel_init_fn()
        kernel_init(self.lin1.weight)
        kernel_init(self.lin2.weight)
        kernel_init(self.lin3.weight)

        bias_init = bias_init_fn()
        bias_init(self.lin1.bias)
        bias_init(self.lin2.bias)

    def forward(self, t, y):
        if t.dim() == 0:
```

(continues on next page)

(continued from previous page)

```
            t = t.view(1, 1)
        elif t.dim() == 1:
            t = t.view(-1, 1)

        return self.lin1(y) * torch.sigmoid(self.lin2(t)) + self.lin3(t)

class FFJORD(nn.Module):
    def __init__(self, data_size, width_size, depth):
        super().__init__()
        self.data_size = data_size
        self.width_size = width_size
        self.depth = depth

        layers = []

        if self.depth == 0:
            layers.append(ConcatSquash(in_size=data_size, out_size=self.data_size))
        else:
            layers.append(ConcatSquash(in_size=data_size, out_size=self.width_size))
            for _ in range(self.depth - 1):
                layers.append(ConcatSquash(in_size=width_size, out_size=self.width_
↪size))
            layers.append(ConcatSquash(in_size=width_size, out_size=self.data_size))

        self.layers = nn.ModuleList(layers)

    def forward(self, t, y):
        for layer in self.layers[:-1]:
            y = layer(t, y)
            y = torch.tanh(y)
        y = self.layers[-1](t, y)
        return y
```

22.6.2 The Continuous Normalizing Flow Network

For ODE integration, we'll make use of the differentiable ODE solvers from the `torchdiffeq` package.

```
try:
    import google.colab  # only to ensure that we are inside colab
    %pip install --quiet torchdiffeq
except ImportError:
    print("This notebook is running locally, please install torchdiffeq manually.")
```

The `ContinuousNormalizingFlow` class implements the basic functionality to integrate the neural velocity estimator via `odeint` in a differentiable manner. The latter is important to allow for backpropagating the gradients from the loss (and the output of the integration step) back to the weights of the FFJORD network.

```
import torch
import torch.nn as nn
from torchdiffeq import odeint

class CNFVelocityFn(nn.Module):

    def __init__(self, input_dim, hidden_dim, num_layers=8):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.net = FFJORD(data_size=input_dim, width_size=hidden_dim, depth=num_
↪layers)

    def forward(self, t, combined_state):
        y, ldj = combined_state

        with torch.set_grad_enabled(True):
            y.requires_grad_(True)
            t.requires_grad_(True)

            t = torch.unsqueeze(t.repeat(y.shape[0]), 1)

            velocity = self.net(t, y)

            divergence = 0.0
            for i in range(y.shape[1]):
                divergence += torch.autograd.grad(velocity[:, i].sum(), y, create_
↪graph=True)[0][:, i]

        return velocity, divergence.view(velocity.shape[0], 1)

class ContinuousNormalizingFlow(nn.Module):

    def __init__(self, input_dim, hidden_dim,
                 time_0=0.0, time_T=1.0):

        super().__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.time_0 = time_0
        self.time_T = time_T

        self.velocity_fn = CNFVelocityFn(input_dim=input_dim, hidden_dim=hidden_dim)

    def solveODE(self, x, t):

        batch_size, dim = x.shape
        assert dim == self.input_dim, "Input dimension mismatch!"

        y0 = x
        ldj0 = torch.zeros(batch_size, device=x.device)

        combined_state = (y0, ldj0)

        result = odeint(self.velocity_fn, combined_state, t,
                        method='dopri5',
                                    atol=[1e-5, 1e-5],
                                    rtol=[1e-5, 1e-5],
                        )

        final_y, final_ldj = result

        return final_y[-1], final_ldj[-1]

    def forward(self, x):
```

(continues on next page)

(continued from previous page)

```
        t = torch.tensor([self.time_0, self.time_T], device=x.device)

        return self.solveODE(x, t)

    def inverse(self, x):

        t = torch.tensor([self.time_T, self.time_0], device=x.device)

        return self.solveODE(x, t)
```

22.6.3 Training

The training step can re-use the `train_model` function from above, as all basic modalities (data format, loss, etc.) stay the same. We're only replacing the "discrete" step-by-step transformation with the continuously integrated version.

```
samples = generate_2d_gaussian_mixture(5000, gm) # use fewer samples because training␣
↪takes longer
samples = shuffle(samples.numpy())
dataset = TensorDataset(torch.tensor(samples, dtype=torch.float32))
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

input_dim = 2
hidden_dim = 128
time_0 = 0.0
time_T = 1.0

cnf_model = ContinuousNormalizingFlow(input_dim=input_dim, hidden_dim=hidden_dim,␣
↪time_0=time_0, time_T=time_T)

learning_rate = 2e-4
optimizer = torch.optim.Adam(cnf_model.parameters(), lr=learning_rate)

num_epochs = 50
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
losses = train_model(cnf_model, dataloader, optimizer, num_epochs=num_epochs,␣
↪device=device)
```

```
Epoch 1/50, Loss: 4.8563
Epoch 2/50, Loss: 2.7663
Epoch 3/50, Loss: 2.4124
Epoch 4/50, Loss: 2.3910
Epoch 5/50, Loss: 2.3755
Epoch 6/50, Loss: 2.3799
Epoch 7/50, Loss: 2.3673
Epoch 8/50, Loss: 2.3654
Epoch 9/50, Loss: 2.3439
Epoch 10/50, Loss: 2.3458
...
Epoch 40/50, Loss: 1.9560
Epoch 41/50, Loss: 1.9537
Epoch 42/50, Loss: 1.9505
Epoch 43/50, Loss: 1.9367
Epoch 44/50, Loss: 1.9524
Epoch 45/50, Loss: 1.9343
```

(continues on next page)

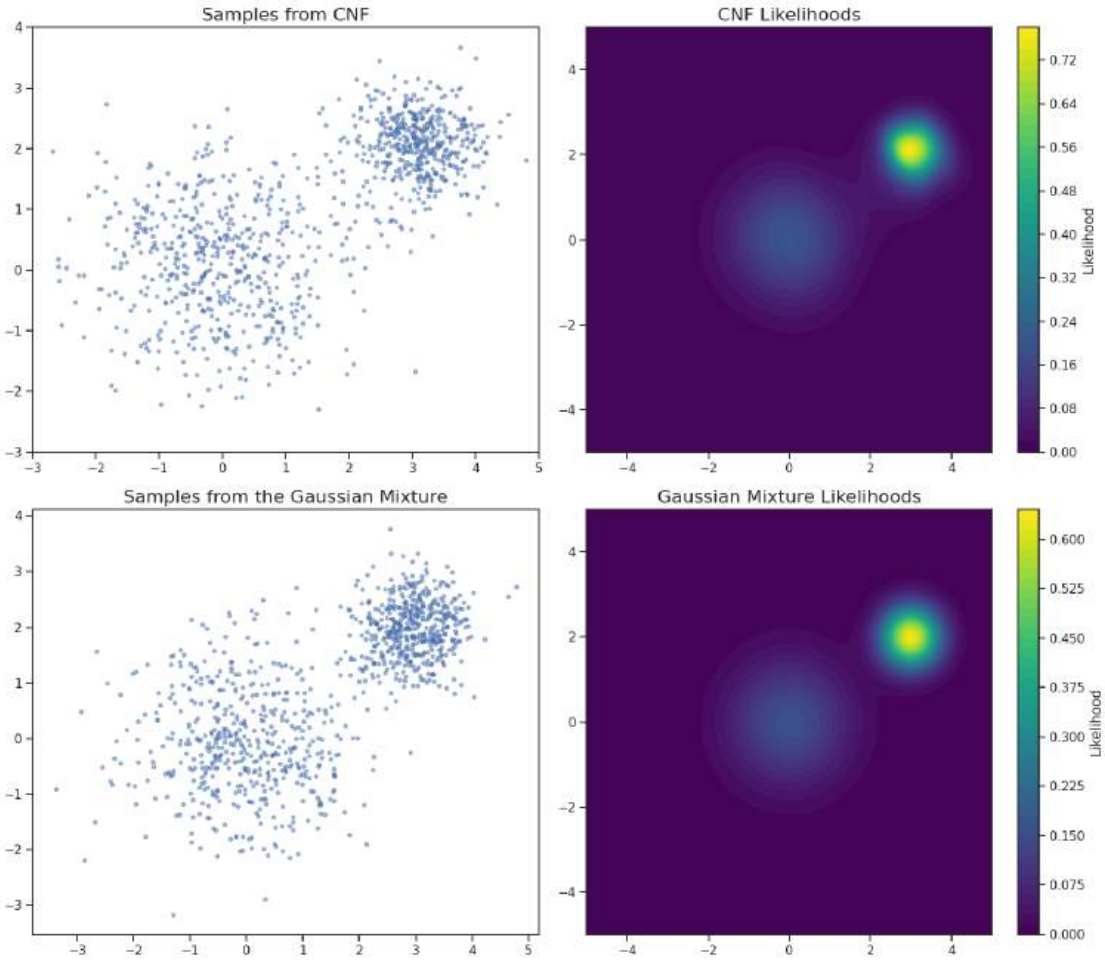
(continued from previous page)

```
Epoch 46/50, Loss: 1.9524
Epoch 47/50, Loss: 1.9574
Epoch 48/50, Loss: 1.9440
Epoch 49/50, Loss: 1.9543
Epoch 50/50, Loss: 1.9453
```

22.6.4 Visualization of the Continuous Normalizing Flow

Now we can repeat the same steps as before to visualize what this second network has learned.

```
visualize_training_results(cnf_model.to(device), gm, 100, model_desc='CNF')
```



This looks good, how does the continuous version treat the intermediate distributions? This will be visualized below.

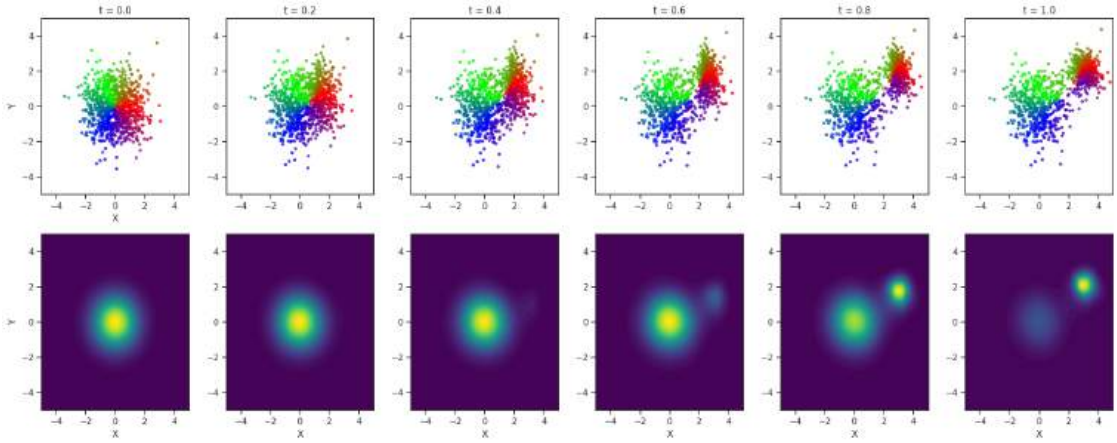
```
def visualize_progression_with_time_and_likelihoods(model, grid_size=100, num_
↪timepoints=6, num_samples=1000):
    model.eval()

    timepoints = torch.linspace(0.0, 1.0, num_timepoints)
```

(continues on next page)

(continued from previous page)

```
    fig, axes = plt.subplots(2, num_timepoints, figsize=(20, 8))

    x = np.linspace(-5, 5, grid_size)
    y = np.linspace(-5, 5, grid_size)
    X, Y = np.meshgrid(x, y)
    points = np.column_stack([X.ravel(), Y.ravel()])
    points_tensor = torch.tensor(points, dtype=torch.float32).to(device)

    z = torch.randn(num_samples, 2).to(device)

    c = get_angle_colors(z.cpu().numpy())

    eps = 1e-5

    for i, t in enumerate(timepoints):
        t_tensor = torch.tensor([0.0, t+eps]).to(device)
        t_tensor_inv = torch.tensor([1.0, 1.0-t-eps]).to(device)

        with torch.no_grad():

            samples, _ = model.solveODE(z, t_tensor_inv)
            samples = samples.cpu().numpy()

            z_t, ldj = model.solveODE(points_tensor, t_tensor)

            prior = (-0.5 * z_t ** 2).sum(-1) - 0.5 * torch.log(torch.tensor(2.0 *␣
↪torch.pi))

            likelihoods = torch.exp(prior + ldj).cpu().numpy().reshape(grid_size,␣
↪grid_size)

        scatter_ax = axes[0, i]
        scatter_ax.scatter(samples[:, 0], samples[:, 1], s=10, alpha=0.7, c=c)
        scatter_ax.set_title(f"t = {t:.1f}")
        scatter_ax.set_xlim(-5, 5)
        scatter_ax.set_ylim(-5, 5)
        scatter_ax.set_xlabel("X" if i == 0 else "")
        scatter_ax.set_ylabel("Y" if i == 0 else "")

        likelihood_ax = axes[1, i]
        contour = likelihood_ax.contourf(X, Y, likelihoods, levels=50, cmap="viridis")
        likelihood_ax.set_xlim(-5, 5)
        likelihood_ax.set_ylim(-5, 5)
        likelihood_ax.set_xlabel("X")
        likelihood_ax.set_ylabel("Y" if i == 0 else "")

    plt.tight_layout()
    plt.show()
```

```
visualize_progression_with_time_and_likelihoods(
    cnf_model,
    grid_size=100,
    num_timepoints=6,
    num_samples=1000
)
```



The figure shows how the Gaussian sampling distribution is transformed to the target distribution much more smoothly using the continuous-time normalizing flow than the implementation with discrete layers.

22.7 Summary of Normalizing Flows

This is a great result. Using our knowledge about ODE solving, we can choose basically any neural network architecture for the velocity. We can trade off speed against accuracy when solving the ODE by choosing different solvers and step sizes depending on our current computational budget.

However, there are also disadvantages with this approach. In order to train our continuous normalizing flow, we need to solve the entire ODE transporting the samples from our target distribution from $t = 1$ until $t = 0$ to the Gaussian distribution to evaluate their likelihoods with high accuracy. This requires a large number of network evaluations and is computationally expensive. As such, it is difficult to scale neural ODEs to high dimensional data and large neural networks.

The focus of the next section will be on more scalable methods that can be combined with very large networks and high-dimensional data.

CHAPTER

TWENTYTHREE

SCORE MATCHING

A first important step is to realize that there's a convenient alternative to probability densities that let's us work with unnormalized functions: the so-called *score*. This is the name that was established for the gradient of the log likelihood function: $\nabla_x \log p(x)$. When we learn this function with a neural network, which we'll call $s_\theta(x)$ to indicate this is a learned *score*, the great thing is that we don't need to worry about normalization anymore. When integrating we'd need to have the right constant offset, but for the "local" gradients at x, all that matters is the derivative at this point.

23.1 Gaussian Toy Dataset with Analytic Scores

We'll re-use the same Gaussian mixture case from the previous sections, but for experiments with the score function we'll expand the base class with a `score()` function so that we can check how well learned approximations fare. Luckily, we can compute the reference score directly from the Gaussian mixture model.

```
import numpy as np

class GaussianMixture:
    def __init__(self, parameters):

        self.parameters = parameters
        self.distributions = [
            {
                'mean': np.array(dist['mean']),
                'std': np.array(dist['std']),
                'cov': np.diag(np.array(dist['std']) ** 2)
            }
            for dist in parameters
        ]

    def sample(self, num_samples):
        samples = []
        num_distributions = len(self.distributions)
        for _ in range(num_samples):
            idx = np.random.randint(num_distributions)
            dist = self.distributions[idx]
            sample = np.random.multivariate_normal(mean=dist['mean'], cov=dist['cov'])
            samples.append(sample)
        return np.array(samples)

    def likelihood(self, points):
        likelihoods = np.zeros(points.shape[0])
        for dist in self.distributions:
            mean = dist['mean']
```

(continues on next page)

(continued from previous page)

```
            cov = dist['cov']
            inv_cov = np.linalg.inv(cov)
            det_cov = np.linalg.det(cov)

            factor = 1 / (2 * np.pi * np.sqrt(det_cov))
            diff = points - mean
            exponents = -0.5 * np.sum(diff @ inv_cov * diff, axis=1)
            likelihoods += factor * np.exp(exponents)

        return likelihoods

    def score(self, points):

        scores = np.zeros_like(points)
        likelihoods = np.zeros(points.shape[0])

        component_likelihoods = []
        for dist in self.distributions:
            mean = dist['mean']
            cov = dist['cov']
            inv_cov = np.linalg.inv(cov)
            det_cov = np.linalg.det(cov)

            factor = 1 / (2 * np.pi * np.sqrt(det_cov))
            diff = points - mean
            exponents = -0.5 * np.sum(diff @ inv_cov * diff, axis=1)
            comp_likelihood = factor * np.exp(exponents)

            component_likelihoods.append(comp_likelihood)
            likelihoods += comp_likelihood

        for dist, comp_likelihood in zip(self.distributions, component_likelihoods):
            mean = dist['mean']
            inv_cov = np.linalg.inv(dist['cov'])
            weights = comp_likelihood / (likelihoods + 1e-8)
            diff = points - mean
            component_score = -(diff @ inv_cov)
            scores += np.multiply(weights[:, np.newaxis], component_score)

        return scores
```

23.1.1 Visualizing Samples, Likelihoods and Scores

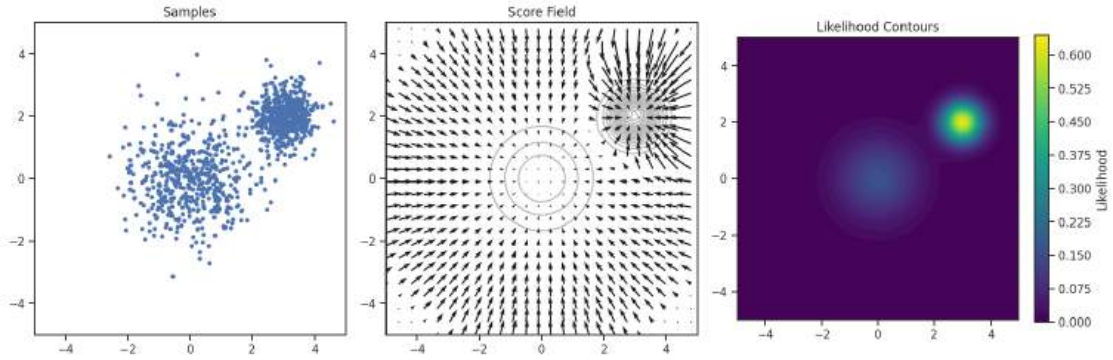
We'll also directly define a helper function that visualizes samples, the score field and the likelihood in a single graph:

```
from matplotlib import pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks")

def visualize_gaussian_mixture_with_score(mixture, num_samples=1000, grid_size=50):
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 6))

    # Create grid for visualization
    x = np.linspace(-5, 5, grid_size)
    y = np.linspace(-5, 5, grid_size)
```

(continues on next page)

(continued from previous page)

```
    X, Y = np.meshgrid(x, y)
    points = np.column_stack([X.ravel(), Y.ravel()])

    # Compute likelihoods and scores
    likelihoods = mixture.likelihood(points)
    scores = mixture.score(points)
    samples = mixture.sample(num_samples)

    # Reshape for plotting
    likelihoods = likelihoods.reshape(grid_size, grid_size)
    scores_x = scores[:, 0].reshape(grid_size, grid_size)
    scores_y = scores[:, 1].reshape(grid_size, grid_size)

    ax1.scatter(samples[:, 0], samples[:, 1], s=10, alpha=1)
    ax1.set_title("Samples")

    skip = 2
    ax2.quiver(X[::skip, ::skip], Y[::skip, ::skip],
               scores_x[::skip, ::skip], scores_y[::skip, ::skip],
               scale=100, width=0.005)

    # Add likelihood contours to score plot for reference
    ax2.contour(X, Y, likelihoods, levels=20, colors='k', alpha=0.3)
    ax2.set_title("Score Field")

    # Plot likelihood contours
    contour = ax3.contourf(X, Y, likelihoods, levels=50, cmap='viridis')
    ax3.set_title("Likelihood Contours")
    plt.colorbar(contour, ax=ax3, label='Likelihood', fraction=0.046)

    # Plot score field
    # Subsample the grid for clearer arrows


    # Set consistent limits and labels
    for ax in [ax1, ax2, ax3]:
        ax.set_xlim(-5, 5)
        ax.set_ylim(-5, 5)
        ax.set_aspect('equal')
        ax.set_xlabel("")
        ax.set_ylabel("")

    plt.tight_layout()
    plt.show()

parameters = [
    {"mean": [0, 0], "std": [1, 1]},
    {"mean": [3, 2], "std": [0.5, 0.5]}
]
mixture = GaussianMixture(parameters)
visualize_gaussian_mixture_with_score(mixture)
```



It is important to keep in mind that we only know samples from the target distribution and we typically don't know the likelihood of the samples. It turns out that in this case it is much easier to obtain the score than the likelihood. Why is this the case?

To estimate the likelihood of a single individual sample, we need a large number of independent samples from the target distribution for comparison. The main difficulty here is that we have a proposal density q_θ for the target density p, then we need to make sure that if we integrate over the entire probability space, the result equals 1, i.e. one of the standard requirements of densities $\int q_\theta(x)dx = 1$. To get the density of a single sample x right, it needs to be normalized in the sense that the density still integrates to 1.

With normalizing flows, this is always satisfied by the way the model and networks are constructed. However, as we have seen in the previous section, normalizing flows do not scale well to high dimensional data and are computationally expensive.

For general and flexible methods, making sure that the model q_θ integrates to 1 becomes extremely difficult. What makes the score so interesting is that it depends on local information only. That means, because it considers the *gradient* of the log likelihood, its value only depends on the local values of the likelihood. This makes it much easier for a neural network to learn the score.

We will first explore how to learn the score from samples of the target distribution using a neural network. Then we will introduce a first method how to use the score to sample from the target distribution.

23.2 Learning the Score

We could directly learn $\nabla_x \log p(x)$, the gradient of the log likelihood function, with an L^2 term. In this context, L^2 is known as the "Fisher divergence": $\mathbb{E}_{x\sim p(x)}[||\nabla_x \log p(x) - s_\theta(x)||^2]$. However, this would require having access to the ground truth gradients $\nabla_x \log p(x)$?, which unfortunately is not the case for relevant settings. The trick established in ML to compute targets for learning is to slightly perturb the dataset with Gaussian noise [Vin11]. This turns it from a collection of point-wise samples into a continuous function of which we can compute the gradients.

So, for the dataset $\{x_1, ..., x_n\}$ we consider the perturbed dataset $\{\tilde{x}_1, ..., \tilde{x}_n\}$ by adding Gaussian noise $\tilde{x} = x+\sigma z$ with $z \sim \mathcal{N}(0, I)$. For now we keep the noise level $\sigma > 0$ fixed at a certain value, but this is a crucial parameter that we'll revisit soon.

We can write the perturbed distribution p_σ that we get from the perturbed samples as a conditional distribution marginalized over the condition: $p_\sigma(\tilde{x}) = \int p_\sigma(\tilde{x}|x)p(x)dx = \mathbb{E}_{x\sim p(x)}[p_\sigma(\tilde{x}|x)]$. The smaller the noise level σ is, the closer the densities of the perturbed p_σ and the original ones p will be, i.e. $\lim_{\sigma\to 0} \mathrm{KL}(p_\sigma||p) = 0$. Intuitively, we need the noise to compute a gradient, but it shouldn't be too large to distort the target distribution.

Now we can leverage the construction of the perturbed density with a Gaussian to compute the gradient for the score: the conditional density $p_\sigma(\tilde{x}|x)$ has the analytic form

$$p_\sigma(\tilde{x}|x) = \frac{1}{\sqrt{(2\pi)^D \sigma^D}} \exp\left(-\frac{1}{2\sigma^2}(\tilde{x}-x)^T(\tilde{x}-x)\right), \tag{23.1}$$

and its score is surprisingly simple:

$$\nabla_{\tilde{x}} \log p_\sigma(\tilde{x}|x) = -\frac{\tilde{x}-x}{\sigma^2}. \tag{23.2}$$

We can train s_θ to approximate the score of the perturbed dataset with the conditional density using the identity

$$\begin{aligned} &\arg\min_\theta \mathbb{E}_{\tilde{x}\sim p_\sigma(\tilde{x})}\left[||s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log p_\sigma(\tilde{x})||^2\right] = \\ &\arg\min_\theta \mathbb{E}_{x\sim p(x), \tilde{x}\sim p_\sigma(\tilde{x}|x)}\left[||s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log p_\sigma(\tilde{x}|x)||^2\right] \end{aligned}$$

23.2.1 Define Score Network and Dataset

To implement these ideas for our Gaussian mixture problem, we first define a simple score network with four hidden layers, and a dataset class that perturbs our samples as described above. The `DenoisingScoreMatchingDataset` class has a parameter `sigma` to control the amount of perturbation.

```
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

class ScoreNetwork(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, input_dim)
        )

    def forward(self, x):
        return self.net(x)

class DenoisingScoreMatchingDataset(Dataset):
    def __init__(self, gaussian_mixture, num_samples=10000, sigma=0.1):
        self.samples = torch.FloatTensor(gaussian_mixture.sample(num_samples))
        self.sigma = sigma

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        x = self.samples[idx]

        noise = torch.randn_like(x) * self.sigma
        x_noisy = x + noise

        return x_noisy, noise, x
```

23.2.2 Training

Now we can train the score network to estimate the gradients of the perturbed dataset, which are given by the perturbation in the dataset (`noise`) below, divided by σ^2.

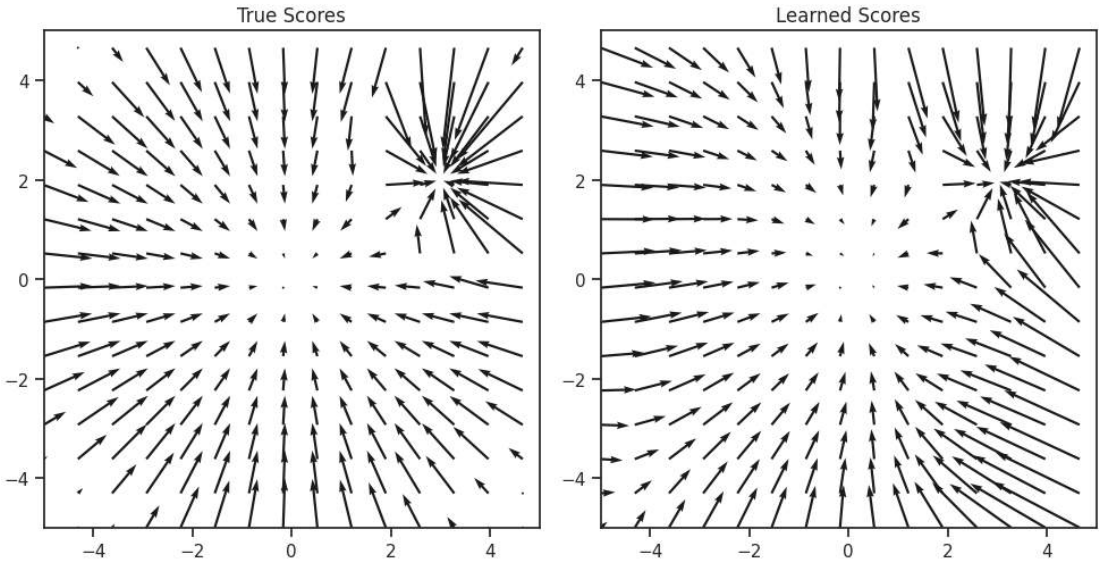
```
import torch
from torch import optim

def train_score_network(model, train_loader, num_epochs=100, sigma=0.1, device='cpu'):
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    model.train()

    losses = []

    for epoch in range(num_epochs):
        epoch_loss = 0
        for batch_idx, (x_noisy, noise, x_clean) in enumerate(train_loader):
            x_noisy, noise = x_noisy.to(device), noise.to(device)

            optimizer.zero_grad()

            pred_score = model(x_noisy)
            target_score = -noise / (sigma**2)
            loss = torch.mean(torch.sum((pred_score - target_score)**2, dim=1))

            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(train_loader)
        losses.append(avg_loss)

        if (epoch + 1) % 10 == 0:
            print(f'Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}')

    return losses

# Create dataset and dataloader
dataset = DenoisingScoreMatchingDataset(mixture, num_samples=10000, sigma=0.1)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

# Initialize and train the model
device = 'cuda' if torch.cuda.is_available() else 'cpu'
score_net = ScoreNetwork().to(device)

losses = train_score_network(score_net, dataloader, num_epochs=100, device=device)
```

```
Epoch 10/100, Loss: 192.8589
Epoch 20/100, Loss: 197.8407
Epoch 30/100, Loss: 196.8402
Epoch 40/100, Loss: 198.7428
Epoch 50/100, Loss: 194.3992
Epoch 60/100, Loss: 198.6634
Epoch 70/100, Loss: 194.6108
Epoch 80/100, Loss: 197.0947
Epoch 90/100, Loss: 191.6466
Epoch 100/100, Loss: 197.7943
```

23.2.3 Compare Learned and Reference Scores

Now that we have a trained network, let's qualitatively evaluate whether the predicted score is accurate or not. As shown above, we have a ground truth vector field for the score. The next cell plots it next to the one predicted by our trained network.

```
# Visualization function for learned scores
def visualize_learned_scores(mixture, score_net, grid_size=30, device='cpu'):
    """
    Visualize true and learned scores side by side
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

    # Create grid for visualization
    x = np.linspace(-5, 5, grid_size)
    y = np.linspace(-5, 5, grid_size)
    X, Y = np.meshgrid(x, y)
    points = np.column_stack([X.ravel(), Y.ravel()])

    # Compute true scores
    true_scores = mixture.score(points)
    true_scores_x = true_scores[:, 0].reshape(grid_size, grid_size)
    true_scores_y = true_scores[:, 1].reshape(grid_size, grid_size)

    # Compute learned scores
    score_net.eval()
    with torch.no_grad():
        learned_scores = score_net(torch.FloatTensor(points).to(device)).cpu().numpy()
    learned_scores_x = learned_scores[:, 0].reshape(grid_size, grid_size)
    learned_scores_y = learned_scores[:, 1].reshape(grid_size, grid_size)

    # Plot true scores
    skip = 2
    ax1.quiver(X[::skip, ::skip], Y[::skip, ::skip],
               true_scores_x[::skip, ::skip], true_scores_y[::skip, ::skip],
               scale=50, width=0.005)
    ax1.set_title("True Scores")

    # Plot learned scores
    ax2.quiver(X[::skip, ::skip], Y[::skip, ::skip],
               learned_scores_x[::skip, ::skip], learned_scores_y[::skip, ::skip],
               scale=50, width=0.005)
    ax2.set_title("Learned Scores")

    # Set consistent limits and labels
    for ax in [ax1, ax2]:
        ax.set_xlim(-5, 5)
        ax.set_ylim(-5, 5)
        ax.set_aspect('equal')
        ax.set_xlabel("")
        ax.set_ylabel("")

    plt.tight_layout()
    plt.show()

# Visualize the results
visualize_learned_scores(mixture, score_net, device=device)
```



This motivates the non-trivial construction via the perturbed, conditional density: all steps required in the second equation can be computed efficiently. Hence, this gives a practical method for learning score functions, provided that we know a suitable value for σ.

This leads us to the next step, let's assume we have successfully trained our network, how can we use ${}^{\circ}\nabla_{\tilde{x}} \log p_\sigma(\tilde{x})$ to obtain a generative model for $p(x)^{\circ}$? That is, how can we produce actual samples x with the score?

23.3 Langevin Dynamics

To arrive at a practical method, we turn to *Langevin Dynamics*. They're traditionally used for molecular systems with deterministic and stochastic forces. It was shown there that iteration steps along the score, with just the right amount of additional perturbation at each step actually converge to samples from $p(x)$.

Specifically, for the update

$$x_{i+1} \leftarrow x_i + \epsilon \nabla_x \log p(x) + \sqrt{2\epsilon} z_i \tag{23.3}$$

with $i = 0, 1, ..., K$ and $z_i \sim \mathcal{N}(0, I)$, the iterate x_K converges to a sample from $p(x)^{\circ}$ as $K \to \infty$ and $\epsilon \to 0$ under a set of regularity conditions. For the theory, we should also consider how x_0 was sampled from a distribution for initialization, typically called ${}^{\circ}\pi(x)$. For updating x_i, we can of course make use of the trained network s_θ to obtain the score $\nabla_x \log p(x)$ in each step of the iteration.

23.3.1 Langevin Dynamics Algorithm

Let's give this a try with our score function! The `langevin_dynamics` function below implements the integration steps for a varying number of steps (0 to 500) to show how it converges. We're starting with points on a dense regular grid, to show whether (and how) points from all over domain move towards the densities of the underlying distribution via the score gradients.

```
def langevin_dynamics(score_net, n_steps=1000, n_samples=35, step_size=0.01):

    device = next(score_net.parameters()).device
```

(continues on next page)

(continued from previous page)

```
    x = torch.linspace(-5, 5, n_samples)
    y = torch.linspace(-5, 5, n_samples)
    X, Y = torch.meshgrid(x, y)
    x = torch.stack([X.flatten(), Y.flatten()], dim=1)

    x = x.to(device)

    trajectory = [x.cpu().detach().numpy()]

    score_net.eval()
    with torch.no_grad():
        for step in range(n_steps):

            score = score_net(x)
            noise = torch.randn_like(x)
            x = x + step_size * score + np.sqrt(2 * step_size) * noise

            if step in [0, 100, 200, 300, 400, 500]:
                trajectory.append(x.cpu().detach().numpy())

    return trajectory


trajectory = langevin_dynamics(
    score_net,
    n_steps=501,    # run for 501 steps to include step 500
    n_samples=35,
    step_size=0.01,
)

samples =  trajectory[-1]
```

23.3.2 Visualize the Trajectories

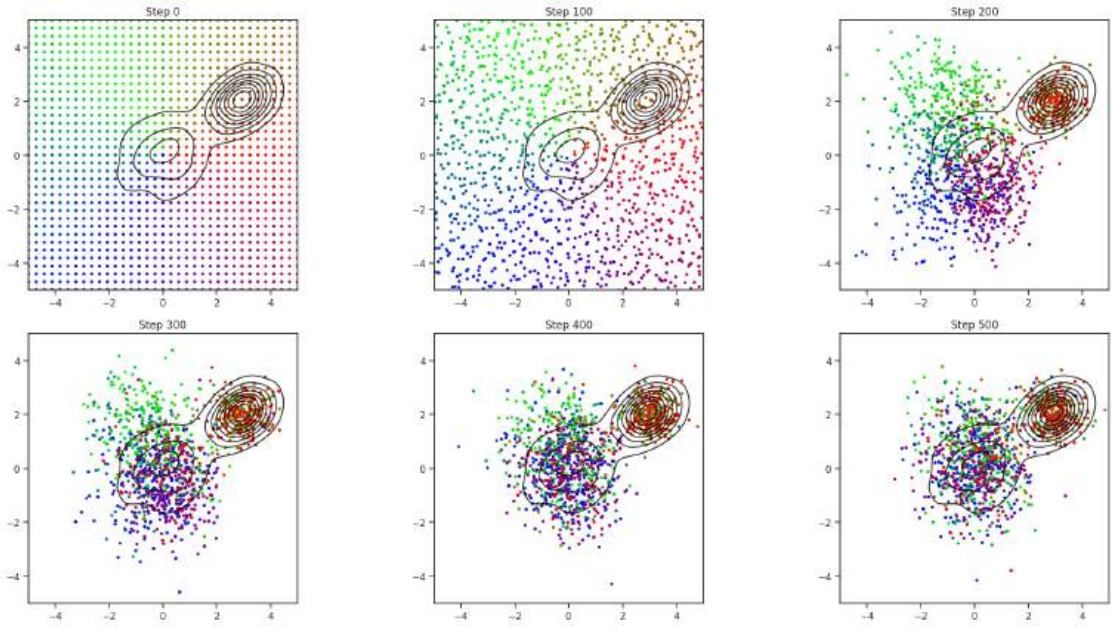
Having the sample points ready in the `samples` array, we can visualize them as colored dots over the score contours. We'll color the initial grid setup with a smooth RGB gradient, so that the colors of dots later on show where initial positions have ended up across the target density landscape.

```
def get_angle_colors(positions):
    angles = np.arctan2(positions[:, 1], positions[:, 0])
    angles_deg = (np.degrees(angles) + 360) % 360
    colors = np.zeros((len(positions), 3))
    for i, angle in enumerate(angles_deg):
        segment = int(angle / 120)
        local_angle = angle - segment * 120
        if segment == 0:    # 0 degrees to 120 degrees (R->G)
            colors[i] = [1 - local_angle/120, local_angle/120, 0]
        elif segment == 1:  # 120 degrees to 240 degrees (G->B)
            colors[i] = [0, 1 - local_angle/120, local_angle/120]
        else:               # 240 degrees to 360° (B->R)
            colors[i] = [local_angle/120, 0, 1 - local_angle/120]
```

(continues on next page)

(continued from previous page)

```
    return colors

def visualize_langevin_trajectory(trajectory, mixture, figsize=(20, 10)):

    n_plots = len(trajectory)
    fig, axes = plt.subplots(2, 3, figsize=figsize)
    axes = axes.ravel()
    real_samples = mixture.sample(1000)
    from scipy.stats import gaussian_kde
    xx, yy = np.mgrid[-5:5:100j, -5:5:100j]
    positions = np.vstack([xx.ravel(), yy.ravel()])
    kernel = gaussian_kde(real_samples.T)
    density = np.reshape(kernel(positions).T, xx.shape)

    steps = [0, 100, 200, 300, 400, 500]

    for idx, (samples, step) in enumerate(zip(trajectory, steps)):
        ax = axes[idx]

        if idx == 0:
            c = get_angle_colors(samples)

        ax.contour(xx, yy, density, levels=10, alpha=0.8, colors='black')
        ax.scatter(samples[:, 0], samples[:, 1], alpha=1, s=7, color=c)

        ax.set_xlim(-5, 5)
        ax.set_ylim(-5, 5)
        ax.set_aspect('equal')
        ax.set_title(f'Step {step}')

    plt.tight_layout()
    plt.show()

visualize_langevin_trajectory(trajectory, mixture)
```



This seems to work well! The regularly spaced points correctly move towards the higher density regions, and the reddish points on the right primarily end up in the high-density cluster on the right side, while other points (blue, green and red all mixed) end up in the lower density peak in the center.

23.4 Annealed Langevin Dynamics

While this provably converges in the limit, it causes some difficulties in practice: the approach requires a fairly large amount of noise σ to ensure the gradients actually "lead" the samples towards the correct targets. If σ is too small, we won't have a score (no direction from the gradient), and if it's too large, the overlapping Gaussians will reduce the quality of our data samples (drowning important details in noise).

Considering the space of all x we get from $\pi(x)$, increasing noise levels σ of the perturbed dataset will cover larger regions of the perturbed data space. This will give gradients wherever we start out, and is important in practice: we shouldn't have to make a lot of assumptions to find good initial points for our iterations. This would only shift the problem from generating the outputs to generating suitable inputs for the algorithm. This is way many classic works need to carefully specify prior distributions to make sure the algorithms converge. It's a delicate balance: working with large σ is important for practical applications, but if the perturbation is too large we have $p(x) \approx p_\theta(x)$ and the samples we produce will be useless in the worst case.

This poses the question: why should we be restricted to a single σ? It turns out that we can resolve the problem of *no score VS bad quality* by going from large to small amounts of noise. Let's consider multiple noise scales $0 < \sigma_1 < \sigma_2 < \ldots < \sigma_L$, giving what's known as *Annealed Langevin Dynamics*. (Looking ahead, this also bring us closer to a central topic in the area of diffusion models: the different and changing levels of noise.)

We repeatedly apply Langevin Dynamics for each noise scale, starting from the largest noise σ_L until the smallest noise σ_1, and train a network $s_\theta(x, \sigma_j)$ with the noise scale σ_j as an additional input. Now we can iterate over x_i, and while updating it we also use smaller and smaller σ_j to make sure we converge to an accurate sample $x \sim p_\theta(\tilde{x})$.

23.4.1 Network and Dataset Definition with Noise Level

Let's implement this idea. First, we need to modify our score network to be aware of the noise level. For this, it will receive an additional parameter σ as input. Likewise, our dataset is extended to cover different amounts of noise.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

class ScoreNetworkWithSigma(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim + 1, hidden_dim),  # +1 for sigma
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, input_dim)
        )

    def forward(self, x, sigma):
        sigma = sigma.view(-1, 1)
        x_sigma = torch.cat([x, sigma], dim=1)
        return self.net(x_sigma)

class AnnealedDenoisingScoreMatchingDataset(Dataset):
    def __init__(self, gaussian_mixture, num_samples=10000, sigmas=[4.0, 2.0, 1.0, 0.
↪5, 0.2, 0.01]):
        self.samples = torch.FloatTensor(gaussian_mixture.sample(num_samples))
        self.sigmas = sigmas

    def __len__(self):
        return len(self.samples) * len(self.sigmas)

    def __getitem__(self, idx):
        sample_idx = idx // len(self.sigmas)
        sigma_idx = idx % len(self.sigmas)

        x = self.samples[sample_idx]
        sigma = self.sigmas[sigma_idx]

        # Add noise to the sample
        noise = torch.randn_like(x) * sigma
        x_noisy = x + noise

        return x_noisy, noise, x, torch.tensor(sigma).float()
```

23.4.2 Training

With these two ingredients, training proceeds just like before…

```
def train_annealed_score_network(model, train_loader, num_epochs=100, device='cpu'):
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    model.train()

    losses = []

    for epoch in range(num_epochs):
        epoch_loss = 0
        for batch_idx, (x_noisy, noise, x_clean, sigma) in enumerate(train_loader):
            x_noisy, noise = x_noisy.to(device), noise.to(device)
            sigma = sigma.to(device)

            optimizer.zero_grad()

            pred_score = model(x_noisy, sigma)

            target_score = -noise / (sigma.view(-1, 1) ** 2)
            loss = torch.mean(torch.sum((pred_score - target_score) ** 2, dim=1))

            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(train_loader)
        losses.append(avg_loss)

        if (epoch + 1) % 10 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.4f}')

    return losses

sigmas = [4.0, 2.0, 1.0, 0.5, 0.2, 0.01]

dataset = AnnealedDenoisingScoreMatchingDataset(mixture, num_samples=10000,␣
↪sigmas=sigmas)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

device = 'cuda' if torch.cuda.is_available() else 'cpu'
score_net = ScoreNetworkWithSigma().to(device)

losses = train_annealed_score_network(score_net, dataloader, num_epochs=100,␣
↪device=device)
```

```
Epoch 10/100, Loss: 3306.6889
Epoch 20/100, Loss: 3374.6170
Epoch 30/100, Loss: 3296.4916
Epoch 40/100, Loss: 3328.9254
Epoch 50/100, Loss: 3302.5448
Epoch 60/100, Loss: 3362.3191
Epoch 70/100, Loss: 3349.0398
Epoch 80/100, Loss: 3337.2041
Epoch 90/100, Loss: 3354.9199
Epoch 100/100, Loss: 3328.9864
```

For sampling with Langevin Dynamics and varying noise levels, we modify the sampling process to reduce the σ values step by step, and then leverage the noise-aware score network to provide the right gradients.

```
def annealed_langevin_dynamics(score_net, sigmas,
                               n_steps_each=100, n_samples=1000,
                               step_size=0.001):
    device = next(score_net.parameters()).device

    x = torch.linspace(-10, 10, n_samples)
    x = torch.stack(torch.meshgrid(x, x), dim=-1).reshape(-1, 2)

    x = x.to(device)

    trajectory = [x.cpu().detach().numpy()]

    score_net.eval()
    with torch.no_grad():
        for sigma in sigmas:
            alpha = step_size * (sigma / sigmas[-1]) ** 2

            for step in range(n_steps_each):
                sigma_tensor = torch.full((n_samples*n_samples,), sigma,␣
↪device=device)
                score = score_net(x, sigma_tensor)

                noise = torch.randn_like(x)
                x = x + alpha * score + np.sqrt(2 * alpha) * noise

            trajectory.append(x.cpu().detach().numpy())

    return trajectory

trajectory = annealed_langevin_dynamics(
        score_net,
        sigmas=sigmas,
        n_steps_each=150,
        n_samples=35,
        step_size=0.00001
        )
```

23.4.3 Visualization of Trajectories

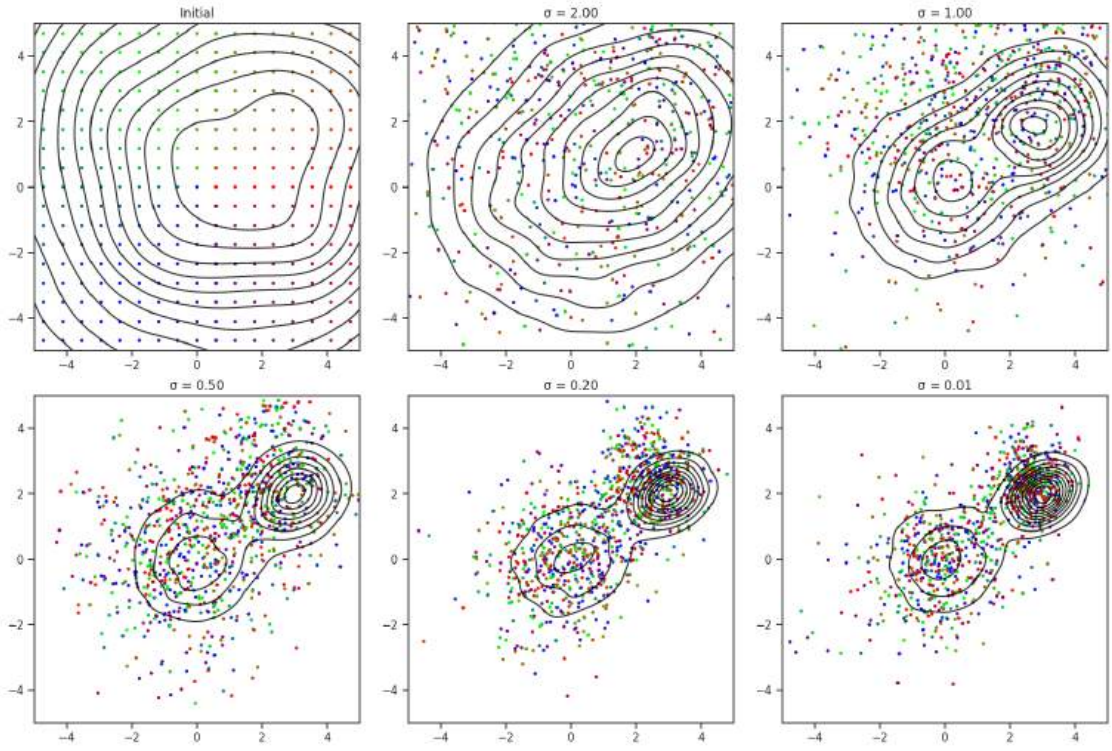
Now we can finally visualize the trajectories. In addition to the moving sample locations, the contours of the underlying score function will now change correspondingly with the varying σ perturbations.

```
def visualize_annealed_langevin_trajectory(trajectory, mixture, sigmas, figsize=(15,␣
↪10)):

    fig, axes = plt.subplots(2, 3, figsize=figsize)
    axes = axes.ravel()

    from scipy.stats import gaussian_kde
    xx, yy = np.mgrid[-5:5:100j, -5:5:100j]
    positions = np.vstack([xx.ravel(), yy.ravel()])

    for idx, (samples, sigma) in enumerate(zip(trajectory, sigmas)):
```

(continues on next page)

(continued from previous page)

```
        ax = axes[idx]

        if idx == 0:
            c = get_angle_colors(samples)

        real_samples = torch.tensor(mixture.sample(5000))
        real_samples += torch.randn_like(real_samples) * sigma

        kernel = gaussian_kde(real_samples.T)
        density = np.reshape(kernel(positions).T, xx.shape)

        ax.contour(xx, yy, density, levels=10, alpha=0.8, colors='black')

        ax.scatter(samples[:, 0], samples[:, 1], alpha=1.0, s=5, color=c)

        ax.set_xlim(-5, 5)
        ax.set_ylim(-5, 5)
        ax.set_aspect('equal')
        ax.set_title(f'σ = {sigma:.2f}' if idx > 0 else 'Initial')

    plt.tight_layout()
    plt.show()

visualize_annealed_langevin_trajectory(trajectory, mixture, sigmas)
```



These plots nicely show how we start with a very smoothly varying score function, as shown by the wide contour lines, towards the sharper peaks of the target distribution. The annealing successfully removes the need for a manually prescribed perturbation.

23.5 Score Summary

Now we're almost at the goal, but before making the last step to derive the famous "denoising" task, let's summarize the ground we've covered so far:

- Denoising score matching works well even for high-dimensional data such as images.
- There's no need to backpropagate gradients through many steps, i.e. the method is much more scalable than CNFs or NeuralODEs.
- We have a way to sample from $p(x)$ to produce samples without the need for assumptions in the form of non-trivial prior distributions.

Despite these important steps forward, we're left with a few challenging aspects:

- Specifying a good sequence of noise scales is critical, and unclear to obtain so far.
- We can sample from $p(x)$, but not directly compute likelihoods.
- The "convenient" maximum likelihood training is not applicable anymore.

There's also the practical aspect of performance: inference actually requires many evaluations of $s_\theta(x, \sigma_j)$, and this can make producing samples expensive. As a first step, we'll make sure we can get *accurate* samples, but we'll also revisit the question of computational efficiency later on.

CHAPTER

TWENTYFOUR

DENOISING

From our perspective of annealed Langevin dynamics, let's consider the extreme case: let's aim for starting with *pure* noise, and treat the annealing steps as a continuous time dimension. We define our pure noise starting time as $t = T$, and then reduce the noise to zero with a continuous *noising schedule* at $t = 0$. When we focus on Gaussian noise $\mathcal{N}$, combining multiple instances of Gaussian noise is still Gaussian. Hence, this actually gives us a Markov chain with a **forward** chain that adds noise in a very controllable way. We'll call the forward process q and denote the function for the standard deviation with β, giving

$$q(x_{1:T}|x_0) := \prod_{t=1}^{T} q(x_t|x_{t-1}) \text{ with } q(x_t|x_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}x_{t-1}, \beta_t I) \ .$$

The process that we've been considering in the previous sections above (the "Langevin dynamics") then represents the **reverse** process. We'll directly denote it with p_θ as this is what we want to learn later on. The reverse process over the diffusion time t is given by:

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \text{ with } p_\theta(x_{t-1}|x_t) = \mathcal{N}(\mu_\theta(x_t, t), \beta_t I)$$

which looks quite simple at first, but we've simply hidden all the difficulties in μ_θ, the learned mean of our distributions, giving the key building block of p_θ.

Thanks to our construction with the Gaussians having standard deviation β_t, we have an analytic form for every step along the way: Given a data point $x_0 \sim q(x_0)$, we can sample the noisy latent state $x_t \sim q(x_t)$ from the forward Markov chain via

$$q(x_t|x_0) = \mathcal{N}(x_t, \sqrt{\overline{\alpha}_t}x_0, (1-\overline{\alpha}_t)I)),$$

with the inverted weights $\alpha_t = 1 - \beta_t$ and alphas accumulated for time t denoted by $\overline{\alpha}_t = \prod_{s=1}^{t} \alpha_s$.

24.1 Latent Variable Models

Conceptually, this formulation gives us what's called a *latent variable model* in the ML community. Instead of the somewhat arbitrary in between states of the Annealed Langevin Dynamics above, we now have explicitly modeled *latent* states along the diffusion time t. Our distribution for targets $x_0 \sim q(x_0)$ is of the form $p_\theta(x_0) = \int p_\theta(x_{0:T})dx_{1:T}$, where $x_1, ..., x_T \sim q(x_t)$ are latents with the same dimensionality as x_0.

Above we've directly trained models with the gradient of the perturbed data to get the score. How does this work in the context of these continually changing latents? The exact marginal likelihood of the diffusion process involves an

intractable integral, but the step-wise accumulation of Gaussian noise represents a variational lower bound that we can use instead. Via *Jensen's inequality* for the log marginal likelihood, we obtain an expression for the *evidence lower bound* (ELBO)

$$\log p_\theta(x_0) \geq \mathbb{E}_{q(x_1,\dots,x_T|x_0)} \left[\log \frac{p_\theta(x_0, x_1, \dots, x_T)}{q(x_1, \dots, x_T|x_0)} \right].$$

Expanding and simplifying the terms we get a lower bound for the maximum likelihood objective: $\mathbb{E}[-\log p_\theta(x_0)] \leq \mathbb{E}\left[-\log p(x_T) - \sum_{t\geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right]$

Note that we've also made use of the Bayes conditioning trick from the top here. This training objective can be reformulated in terms of a KL divergence, to give three terms:

$$\begin{aligned}
&\mathbb{E}\left[L_T + L_{t-1} + L_1\right] = \\
&\mathbb{E}\left[\mathrm{KL}(q(x_T|x_0)||p(x_T)) + \sum_{t>1} \mathrm{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1)\right]
\end{aligned}$$

This looks complicated, but the three terms have distinct meanings:

- the first one, L_T does not depend on θ;
- the last one, L_1, is easy to train with;
- and the middle L_{t-1} term is just the KL divergence between two known Gaussian distributions.

Hence, the middle term L_{t-1} is key, and we actually have the Gaussians under control with the α and β terms above. For the forward process the corresponding expressions are a bit lengthy, but we can write them out. This gives the conditional Gaussian $q(x_{t-1}|x_t, x_0) = \mathcal{N}(\tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I)$, with mean and standard deviation being

$$\tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\overline{\alpha}_{t-1}}\beta_t}{1-\overline{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\overline{\alpha}_{t-1})}{1-\overline{\alpha}_t} x_t \text{ and } \tilde{\beta}_t = \frac{1-\overline{\alpha}_{t-1}}{1-\overline{\alpha}_t}\beta_t$$

This actually gives us a feasible training objective that has been used in a number of papers, but we don't need to stop there. Rather we can further simplify this construction by realizing that we're dealing with noise ϵ being added to our means ($x + \epsilon$). Instead of predicting the means (the signal x itself), predicting the noise ϵ on top of it is easier, and by subtracting it we likewise get x. In the terminology above, we get

$$x_t(x_0, \epsilon) = \sqrt{\overline{a}_t} x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon \text{ for } \epsilon \sim \mathcal{N}(0, I), \tag{24.1}$$

and instead of predicting the mean $\mu_\theta(x_t, t)$ we predict the noise $\epsilon_\theta(x_t, t)$ from which we can compute the mean via

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}} \epsilon_\theta(x_t, t) \right).$$

This simplifies the loss term L_{t-1} from above to

$$\mathbb{E}_{x_0,\epsilon}\left[\frac{\beta_t^2}{2\beta_t\alpha_t(1-\overline{\alpha}_t)} ||\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t} x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon, t)||^2 \right] + C,$$

and even better, it turns out we can remove the first β^2 factor. People have noticed that it actually provides a sub-optimal scaling, and removing it increases the weighting for samples with small β values at the end of the reverse chain. This gives a simplified denosing loss of

$$L_{\text{DM}}(\theta) = \mathbb{E}_{t,x_0,\epsilon}\left[||\epsilon - \epsilon_\theta(x_t(x_0, \epsilon), t)||^2 \right],$$

where can easily compute correct values for x_t by mixing $\sqrt{\overline{a}_t} x_0$ with the right amount of noise $\sqrt{1-\overline{\alpha}_t}\epsilon$, as outlined above in equation (24.1).

24.2 Full Denoising Algorithm

This extremely simple L^2 loss is really the reason for the break-through success of this approach, as it's very simple to compute, and very stable. This training variant often goes under the name *denoising diffusion probablistic models* (DDPM). Unlike, e.g. GANs, for which one needs to train with a fragile balance of two networks, we only need to consider a fully supervised loss for training DDPMs. This comes at the cost of increased computational resources, though. Before addressing the computational challenges (these have also been mostly resolved by now), let's re-cap that algorithms for training and inference that we get from this denoising objective.

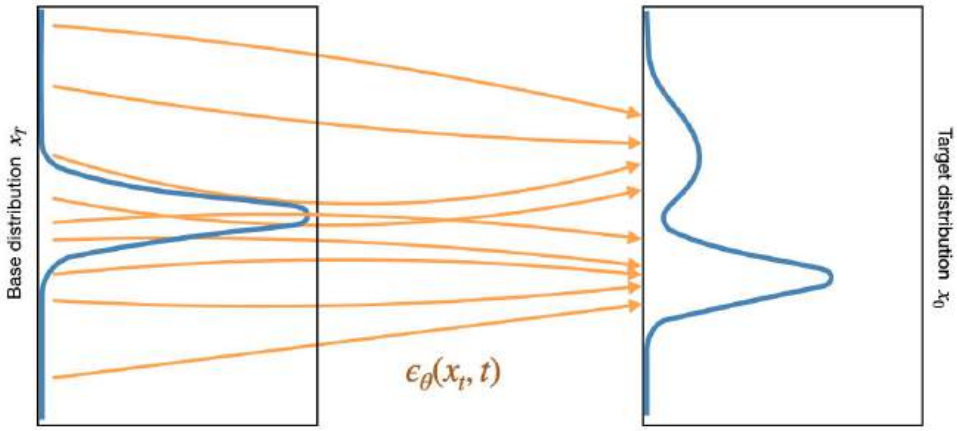


Fig. 24.1: DDPM process (in contrast to CNFs): denoising the simple base distribution (left) over a large number of denoising steps to arrive at a sample from the target distribution (right).

The training procedure can be summarized in pseudo-code with:

Algorithm 1 (DDPM Training)

Inputs Target distribution q, noise schedule $\overline{\alpha}$

Output Trained noise estimator NN ϵ_θ

1. repeat until converged:
    2. $x_0 \sim q(x_0)$
    3. $t \sim \text{Uniform}(1, \dots, T)$
    4. $\epsilon \sim \mathcal{N}(0, I)$
    5. $x_t = \sqrt{\overline{\alpha}_t} x_0 + \sqrt{1 - \overline{\alpha}_t}\epsilon$
    6. optimizer_step($\nabla_\theta ||\epsilon - \epsilon_\theta(x_t, t)||^2$)
2. return ϵ_θ

We uniformly sample an integer time t, and compute the noise sample x_t. This is quite simple, just blend a sample from step 0 of our forward process, x_0 with uniform noise according to the weight $\overline{\alpha}_t$. Let the network try to estimate the noise given x_t and the time t, and then do a step with your favorite optimizer (usually Adam) along the gradient of the L^2 loss. Note that the noise schedule $\overline{\alpha}_t$ and the corresponding coefficients can be precomputed for a chosen number of steps T. This reduces the calculation to a simple lookup at training time. The loss is particularly simple as we can omit the weighting of the noise schedule as explained above.

For inference, we need to properly evaluate the weighting terms, which makes the process slightly more complicated. The weighting can be precomputed just like for training, but the expensive bit is that we now need to go through all T steps to fully denoise the sample:

Algorithm 2 (DDPM Inference)

Inputs Noise estimator ϵ_θ, noise schedule $\alpha, \overline{\alpha}, \sigma$

Output Sample from distribution p

1. $x \sim \mathcal{N}(0, I)$
2. for $t = T, \dots, 1$:
    3. if $t > 1$: $z \sim \mathcal{N}(0, I)$
    4. else: $z = 0$
    5. $x = \frac{1}{\sqrt{\alpha_t}} \left(x - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}} \epsilon_\theta(x, t) \right) + \sigma_t z$
3. return x

This means we have to sequentially evaluate the trained neural network ϵ_θ for T times, and the original DDPM papers actually used $T = 1000$ to obtain very high quality samples. In practice, T can be reduced significantly, by ca. one order of magnitude, without while still giving excellent results, but this nonetheless means we have to evaluate the network around 100 times.

While this is clearly more expensive than the deterministic networks we have used in previous chapters, which produced the mean output in a single pass, the powerful, fundamental change is that the DDPM network can learn and reliably reproduce distributions. I.e., if your data for a single flow input contains 20% solutions that spin left, and 80% that turn right, the trained DDPM network will reproduce these solutions with the right probability. When you repeatedly run inference with different initial noise values for x, you will see left spins with a probability of 0.2, and right spinning solutions with 0.8. The network itself is of course still deterministic, for a constant initial x in line 1 of the inference algorithm, it will always produce the same output.

Another powerful aspect that we'll get back to is also that these networks can be reliable conditioned to steer the generated samples. This is highly important for our initial goals from simulation-based inference: we like to learn the posterior distribution $p(x|y)$ that produces the correct outputs for a given observation y. Looking ahead, three important extensions of the algorithm so far are to be addressed:

- Faster inference speed, primarily getting away with fewer network calls.
- Conditioning on external parameters and observations.
- And in the context of physics simulations, we'd of course like to bring back our prior knowledge in the form of PDE-base constraints.

Nonetheless, we've made the most important first step with DDPMs now: we have turned the deterministic networks into tools for probabilistic inference that robustly learn and reproduce complex distributions. Before starting with source code examples, let's address the computational overhead: DDPMs require a *large* number of denoising steps. The resulting models can be orders of magnitude more expensive to run than their deterministic counterparts. Hence the central question is, how can we reduce the number of required steps without loosing accuracy for the posterior?

24.3 Training with DDPM

Let's see how well these ideas work for our Gaussian mixture (GM) case. To make this a standalone notebook, we first initialize the same two-peak GM case that was also used for the normalizing flows and score tests.

```
import numpy as np

class GaussianMixture:
    def __init__(self, parameters):
        self.parameters = parameters
        self.distributions = [
            {
                'mean': np.array(dist['mean']),
                'std': np.array(dist['std']),
                'cov': np.diag(np.array(dist['std']) ** 2)
            }
            for dist in parameters
        ]

    def sample(self, num_samples):
        samples = []
        num_distributions = len(self.distributions)
        for _ in range(num_samples):
            idx = np.random.randint(num_distributions)  # Choose a random Gaussian
            dist = self.distributions[idx]
            sample = np.random.multivariate_normal(mean=dist['mean'], cov=dist['cov'])
            samples.append(sample)
        return np.array(samples)

parameters = [
    {"mean": [0, 0], "std": [1, 1]},
    {"mean": [3, 2], "std": [0.5, 0.5]}
]
mixture = GaussianMixture(parameters)
```

24.3.1 Define the Forward Process

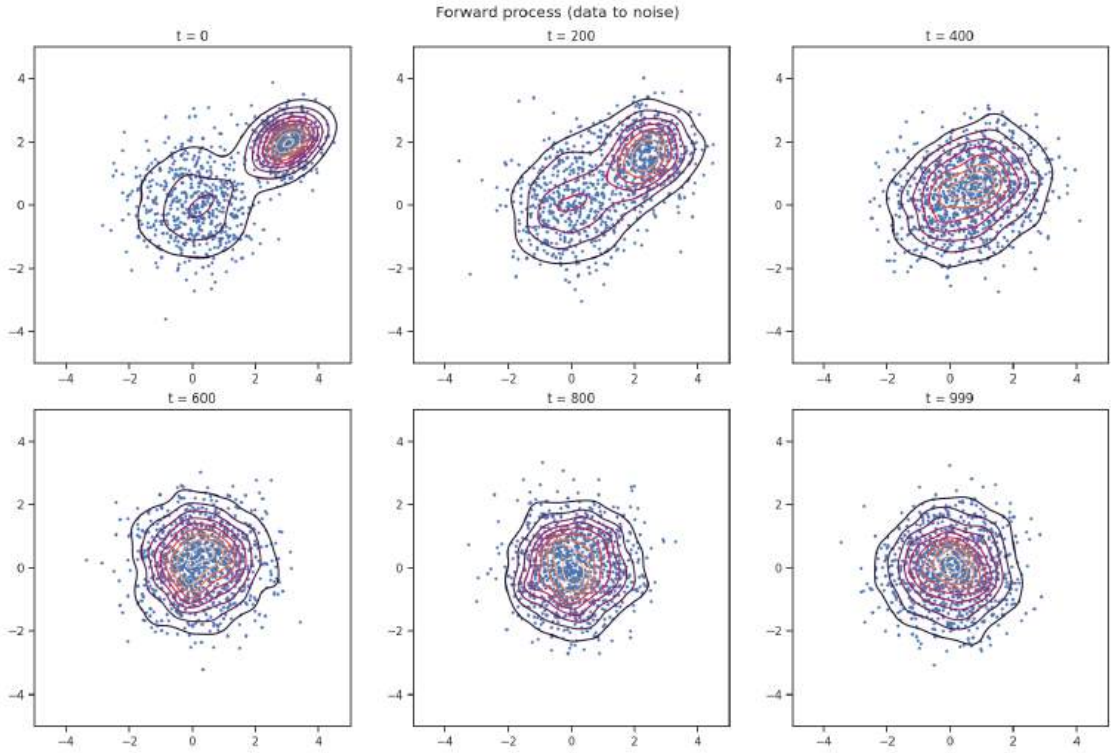
Next, we implement the forward process for DDPM, which takes our GM distribution, and adds increasing amounts of noise until all traces of information left in the samples are removed. The visualizations below show this destructive process in action…

```
import torch
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks")

class DDPMForwardProcess:
    def __init__(self, num_timesteps=1000):
        self.num_timesteps = num_timesteps

        # Standard beta schedule from DDPM paper
        scale = 1000 / num_timesteps
        beta_start = scale * 0.0001
        beta_end = scale * 0.02
        self.betas = torch.linspace(beta_start, beta_end, num_timesteps)
```

(continues on next page)

(continued from previous page)

```
        self.alphas = 1 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
        self.sqrt_betas = torch.sqrt(self.betas)
        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1 - self.alphas_cumprod)

    def forward_process(self, x_0, t):
        epsilon = torch.randn_like(x_0)
        mean = self.sqrt_alphas_cumprod[t] * x_0
        std = self.sqrt_one_minus_alphas_cumprod[t]
        return mean + std * epsilon, epsilon

    def visualize_forward_process(self, mixture, num_samples=1000):

        timesteps_to_show=[0, 200, 400, 600, 800, 999]
        x_0 = torch.FloatTensor(mixture.sample(num_samples))

        fig, axes = plt.subplots(2, 3, figsize=(15, 10))
        axes = axes.ravel()

        xx, yy = np.mgrid[-5:5:100j, -5:5:100j]
        positions = np.vstack([xx.ravel(), yy.ravel()])

        for idx, t in enumerate(timesteps_to_show):
            x_t, _ = self.forward_process(x_0, t)
            samples = x_t.numpy()

            from scipy.stats import gaussian_kde
            kernel = gaussian_kde(samples.T)
            density = np.reshape(kernel(positions).T, xx.shape)

            ax = axes[idx]

            ax.contour(xx, yy, density, levels=10, alpha=1.0)

            ax.scatter(samples[:, 0], samples[:, 1], alpha=1.0, s=4)

            ax.set_xlim(-5, 5)
            ax.set_ylim(-5, 5)
            ax.set_aspect('equal')
            ax.set_title(f't = {t}')

        plt.suptitle("Forward process (data to noise)")

        plt.tight_layout()
        plt.show()

ddpm = DDPMForwardProcess(num_timesteps=1000)
ddpm.visualize_forward_process(mixture)
```



It's visible here that the two peaks from $t = 0$ start to disappear after ca. 200 steps of adding noise, and, visually, form a single Guassian peak centered around zero.

24.3.2 Dataset and Dataloader

The next code cell defines a small dataset class that we will use to train the DDPM. It precomputes the correct alpha and beta values for each t so that we can quickly pull up samples with varying amounts of noise for training.

```
from torch import nn
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
import numpy as np

class ForwardDiffusionDataset(Dataset):
    def __init__(self, num_samples, input_dim,
                 T, mixture, beta_value=0.02):
        super().__init__()
        self.num_samples = num_samples
        self.input_dim = input_dim
        self.T = T
        self.beta_value = beta_value
        self.gm = mixture

        self.betas = beta_value * torch.ones(T)
        self.alphas = 1 - self.betas
        self.alpha_bar = torch.cumprod(self.alphas, dim=0)

        self.x0 = torch.FloatTensor(self.gm.sample(num_samples))
```

(continues on next page)

(continued from previous page)

```
    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        x0_sample = self.x0[idx]
        t = torch.randint(low=1, high=self.T + 1, size=(1,)).item()
        eps = torch.randn(self.input_dim)
        sqrt_alpha_bar = torch.sqrt(self.alpha_bar[t - 1])
        sqrt_one_minus_alpha_bar = torch.sqrt(1 - self.alpha_bar[t - 1])

        x_t = sqrt_alpha_bar * x0_sample + sqrt_one_minus_alpha_bar * eps

        return x_t, torch.tensor(t, dtype=torch.float32), eps
```

Next, we'll define a simple network, much in line with the previous score networks. In this case, three hidden layers suffice to learn the denoising task. The main difference now is that the network receives the denoising time `t` instead of the perturbation amount from before.

```
class DdpmNet(nn.Module):
    def __init__(self, in_dim=2, time_dim=1, hidden_dim=128, out_dim=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim + time_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, out_dim)
        )

    def forward(self, x, t):
        t = t.unsqueeze(1)
        xt = torch.cat([x, t], dim=1)
        return self.net(xt)
```

24.3.3 Initialization and Training

```
num_samples = 10000
input_dim = 2
T = 1000
dataset = ForwardDiffusionDataset(num_samples=num_samples, input_dim=input_dim, T=T,␣
↪mixture=mixture, beta_value=0.02)
batch_size = 64
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# %% Training the DDPM model
time_dim = 1
hidden_dim = 128
model = DdpmNet(in_dim=input_dim, time_dim=time_dim, hidden_dim=hidden_dim, out_
↪dim=input_dim)
optimizer = Adam(model.parameters(), lr=1e-3)
loss_fn = nn.MSELoss()

epochs = 100
```

(continues on next page)

(continued from previous page)

```
model.train()
for epoch in range(1, epochs + 1):
    epoch_loss = 0
    for x_t_batch, t_batch, eps_batch in data_loader:
        optimizer.zero_grad()
        # Normalize time t to range [0,1]
        t_norm = t_batch / T
        eps_pred = model(x_t_batch, t_norm)
        loss = loss_fn(eps_pred, eps_batch)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    avg_loss = epoch_loss / len(data_loader)
    print(f"Epoch {epoch}/{epochs}, Loss: {avg_loss:.4f}")
```

```
Epoch 1/100, Loss: 0.1529
Epoch 2/100, Loss: 0.0871
Epoch 3/100, Loss: 0.0863
Epoch 4/100, Loss: 0.0747
Epoch 5/100, Loss: 0.0759
Epoch 6/100, Loss: 0.0737
Epoch 7/100, Loss: 0.0715
Epoch 8/100, Loss: 0.0742
Epoch 9/100, Loss: 0.0706
Epoch 10/100, Loss: 0.0704
...
Epoch 90/100, Loss: 0.0631
Epoch 91/100, Loss: 0.0592
Epoch 92/100, Loss: 0.0597
Epoch 93/100, Loss: 0.0595
Epoch 94/100, Loss: 0.0570
Epoch 95/100, Loss: 0.0632
Epoch 96/100, Loss: 0.0609
Epoch 97/100, Loss: 0.0570
Epoch 98/100, Loss: 0.0580
Epoch 99/100, Loss: 0.0606
Epoch 100/100, Loss: 0.0624
```

24.3.4 Inference with the Reverse Process

Above, it was enough to have single steps of the denoising process for training the network. At inference time, we now need to perform the whole denoising process: starting from an initial pure noise sample, updating it over the course of 1000 iterations for denoising, to finally obtain a sample from the target distribution at the end.

The code below implements this process, and records different snapshots of the denoising process for visualization.

```
n_infer = 1000

betas = 0.02 * torch.ones(T)
alphas = 1 - betas
alpha_bar = torch.cumprod(alphas, dim=0)

with torch.no_grad():
    x = torch.randn(n_infer, input_dim)
```

(continues on next page)

(continued from previous page)

```
    record_steps = {999, 500, 200, 100, 50, 1}
    snapshots = {}

    for t in range(T, 0, -1):

        t_tensor = (torch.ones(n_infer) * t / T).float()
        eps_theta = model(x, t_tensor)

        beta_t = betas[t - 1]
        alpha_t = alphas[t - 1]
        alpha_bar_t = alpha_bar[t - 1]

        coef = beta_t / torch.sqrt(1 - alpha_bar_t)
        x = (1 / torch.sqrt(alpha_t)) * (x - coef * eps_theta)

        if t > 1:
            sigma_t = torch.sqrt(beta_t)
            x = x + sigma_t * torch.randn_like(x)

        if t in record_steps:
            snapshots[t] = x.clone().cpu().numpy()
```

24.3.5 Visualize the Sampled Trajectories

Next, we visualize how the initial distribution slowly turns into the target distribution from our GM case.

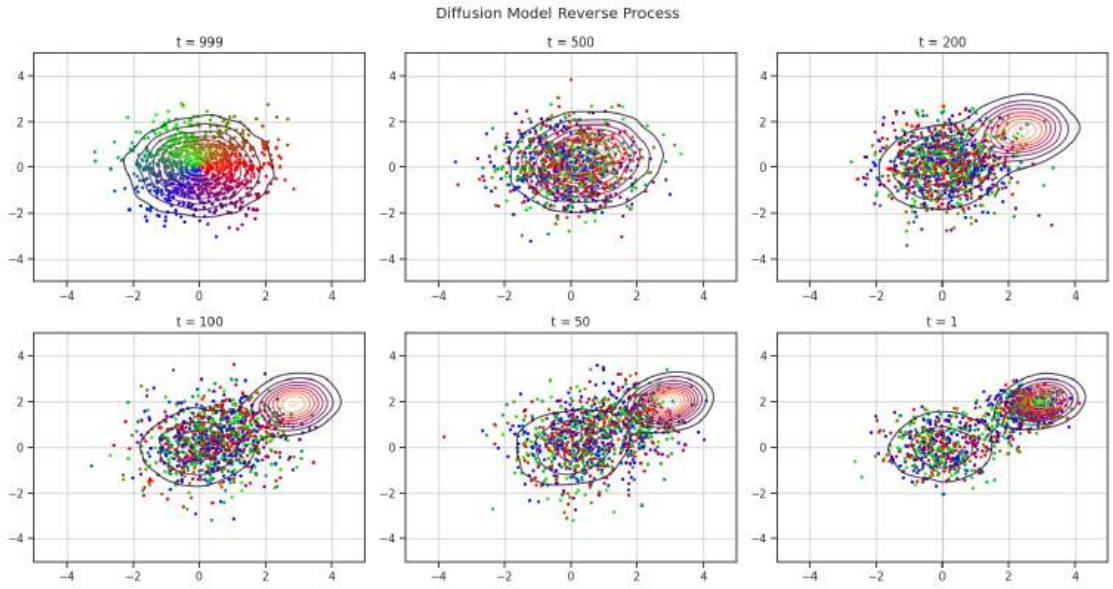
```
import matplotlib.pyplot as plt

def get_angle_colors(positions):
    angles = np.arctan2(positions[:, 1], positions[:, 0])
    angles_deg = (np.degrees(angles) + 360) % 360
    colors = np.zeros((len(positions), 3))
    for i, angle in enumerate(angles_deg):
        segment = int(angle / 120)
        local_angle = angle - segment * 120
        if segment == 0:    # 0 degrees to 120 degrees (R->G)
            colors[i] = [1 - local_angle/120, local_angle/120, 0]
        elif segment == 1:  # 120 degrees to 240 degrees (G->B)
            colors[i] = [0, 1 - local_angle/120, local_angle/120]
        else:               # 240 degrees to 360° (B->R)
            colors[i] = [local_angle/120, 0, 1 - local_angle/120]

    return colors

def visualize_snapshots(snapshots):

    sorted_steps = sorted(snapshots.keys(), reverse=True)
    n_plots = len(sorted_steps)

    fig, axes = plt.subplots(2, 3, figsize=(5 * 3, 8))
    axes = axes.ravel()

    if n_plots == 1:
        axes = [axes]
```

(continues on next page)

(continued from previous page)

```
    # Create grid for density visualization
    xx, yy = np.mgrid[-5:5:100j, -5:5:100j]
    positions = np.vstack([xx.ravel(), yy.ravel()])

    idx = 0
    for ax, t in zip(axes, sorted_steps):

        samples = snapshots[t]

        if idx == 0:
            c = get_angle_colors(samples)
            idx += 1

        x_0 = torch.FloatTensor(mixture.sample(num_samples))
        x_t, _ = ddpm.forward_process(x_0, t)
        samples_ = x_t.numpy()

        from scipy.stats import gaussian_kde
        kernel = gaussian_kde(samples_.T)
        density = np.reshape(kernel(positions).T, xx.shape)

        ax.contour(xx, yy, density, levels=10, alpha=0.8)

        ax.scatter(samples[:, 0], samples[:, 1], alpha=1.0, s=5, color=c)
        ax.set_title(f"t = {t}")
        ax.set_xlabel("")
        ax.set_ylabel("")
        ax.grid(True)

    plt.suptitle("Diffusion Model Reverse Process")
    plt.tight_layout()
    plt.show()
```

```
visualize_snapshots(snapshots)
```



Qualitatively, the DDPM network manages to accurate capture the two peaks in our distribution, and - best of all - it does so with a very simple and stable training process. All we need are the initial samples, and controlled perturbations with the noising schedule added on top!

Looking ahead, three important extensions of the DDPM algorithm still need to be addressed:

- Faster inference speed, primarily getting away with fewer network calls.
- Conditioning on external parameters and observations.
- And in the context of physics simulations, we'd of course like to bring back our prior knowledge in the form of PDE-base constraints.

Nonetheless, we've made the most important first step with DDPMs now: we have turned the deterministic networks into tools for probabilistic inference that robustly learn and reproduce complex distributions. So we've arrived at a very capable and stable method to learn distributions from data. The only caveat left is the computational cost. The probabilistic DDPM network require many sequential evaluations to produce a sample. These *neural function evaluations* (NFEs) are of course costly, and while many practical cases can do with fewer than the 1000 we've used above, even 10 mean that the network is 10x slower than the "regular" deterministic version. This brings us to the last remaining step in this *generative AI journey*: how can we make the inference process faster, without negatively affecting the accuracy of our target, the inferred posterior distribution?

CHAPTER

TWENTYFIVE

FLOW MATCHING

To reduce the many function evaluations of DDPM, we'll turn to *flow matching* [LCBH+22]. To motivate the transition from denoising to flow matching, the score formulation from (23.2) comes in handy: as a reminder, there we were integrating the gradient ∇_x of the log likelihoods to transform noise into a target distribution over the course of a continuous virtual time axis. To briefly re-cap, we integrated $\nabla_{\tilde{x}} \log p_\sigma(\tilde{x})$ over time with Langevin dynamics $dx/dt = \epsilon \nabla_x \log p(x) + \sqrt{2\epsilon} z_i$, and used a training objective of $\arg\min_\theta \mathbb{E}\left[||s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log p_\sigma(\tilde{x})||^2\right]$ to learn the score $s_\theta(\tilde{x})$. Score modeling leveraged the mathematical equivalence between the probability flows of diffusion processes and the trajectories of probability densities in the form of ordinary differential equations (ODEs).

25.1 Learning Flows with Velocities

For *flow matching*, our goal is to construct a *flow* in a velocity field, and the gradients in score matching can actually readily be interpreted as a *velocity*: neglecting the stochastic terms of the Langevin time evolution above, we're left with the velocity $dx/dt = \nabla_x \log p(x)$ of x being the score. The main challenge we were fighting with in the context of score matching was how to obtain a good reference velocity, so that we could train a neural network.

Now, with the powerful machinery of *denoising* at hand, we can revisit the concept of *flows and velocities*: we aim for learning a continuous-time flow, for which we prescribe as-simple-as-possible reference velocities. As before, the velocity viewpoint frees us from any constraints on the network architecture, and the simple velocities will make training and inference substantially faster than before.

Let's formalize this: methods such as *flow matching* are typically categorized as "continuous-time flow models", and transform samples x from a sampling distribution p_0 to samples of a target or posterior distribution p_1. This mapping is expressed via the ODE

$$dx/dt = v_\theta(x, t),$$

where $v_\theta(x, t)$ represents a neural network with parameters θ. In flow matching, the network v_θ is trained by regressing a vector field that generates probability paths that map from p_0 to p_1.

We say that a smooth vector field $u : [0, 1] \times \mathbb{R}^d \to \mathbb{R}^d$, called *velocity*, generates the probability paths p_t, if it satisfies the continuity equation $\partial p/\partial t = -\nabla \cdot (p_t u_t)$. Informally, this means that we can sample from the distribution p_t by sampling $x_0 \sim p_0$ and then solving the ODE $dx = u(x, t)dt$ with initial condition x_0. In the following, we will denote $u(x, t)$ by $u_t(x)$. To regress the velocity field, we define the *flow matching* objective

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{t\sim\mathcal{U}(0,1), x\sim p_t(x)} \,||\, v_\theta(x, t) - u_t(x) \,||^2. \tag{25.1}$$

In order to evaluate this loss, we need to sample from the probability distribution $p_t(x)$ and we need to know the velocity $u_t(x)$. In contrast to the perturbed data that we used for score matching, we'll solve this problem differently for flow matching: we apply a trick by introducing a latent variable z distributed according to $q(z)$ and define the conditional likelihoods $p_t(x|z)$ that depend on the latent variable so that $p_t(x) = \int p_t(x|z)q(z)dz$. Interestingly, if the conditional likelihoods are generated by the velocities $u_t(x|z)$, then the velocity $u_t(x)$ can be expressed in terms of $u_t(x|z)$ and

$p_t(x|z)$ as the following expectation: $u_t(x) = \mathbb{E}_{q(z)}[u_t(x|z)p_t(x|z)/p_t(x)]$. Here we have the freedom to choose paths $p_t(x|z)$ that are easy to sample from and for which we know the generating velocities $u_t(x|z)$. So, as long as we can generate enough velocities conditioned on z, we can obtain the expectation over the course of producing all these samples, and obtain the right generating velocity for training our network.

Flow networks can then be trained with the *conditional flow matching* loss

$$\mathcal{L}_{\text{CFM}}(\phi) = \mathbb{E}_{q(z,t),p_t(x|z)} \mid\mid v_\phi(x,t) - u_t(x|z) \mid\mid^2. \tag{25.2}$$

This version is tractable and can be used for actual training runs, in contrast to the un-conditional objective from equation (25.1). This means that we can train $v_\theta(x,t)$ to regress $u_t(x)$ generating the mapping from p_0 to the target distribution p_1.

25.2 Mappings and Conditioning

Especially important: we have a lot of freedom when specifying the mapping from p_0 to p_1 via the conditioning variable z and the conditional likelihoods p_t in this formulation. So how do we best make use of this freedom? It turns out, the "simplest possibility" of aiming for straight, non-crossing paths is a great choice. The goal is a velocity from the current state x directly towards the target x_1 at time t with $(x_1 - x)/(1-t)$. The only slight complication is that we need to consider a minimal amount of noise $\sigma_{\text{min}} > 0$ at time $t = 1$ to ensure that despite having discrete samples, we keep a continuous distribution. This aspect is in line with the perturbed samples from score matching above. Hence σ_{min} influences the end points of the prescribed paths and its probability distribution.

Plugging it into the equations, we consider the coupling $q(z) = p_1(x)$ together with conditional probability and generating velocity

$$\begin{aligned} p_t(x|x_1) &= \mathcal{N}(tx_1, (1-(1-\sigma_{\text{min}})t)I) \\ u_t(x|x_1) &= \frac{x_1 - (1-\sigma_{\text{min}})x}{1-(1-\sigma_{\text{min}})t} \,. \end{aligned} \tag{25.3}$$

Conditioned on x_1, this coupling transports a point $x_0 \sim \mathcal{N}(0, I)$ from the sampling distribution to the posterior distribution on the linear trajectory tx_1 ending in x_1. At the same time it decreases the standard deviation from 1 to a smoothing constant σ_{min}. In this case, the transport path coincides with the optimal transport between two Gaussian distributions.

So it might seem - on first sight - that we've made a huge step back: we're back to transforming distributions, a concept for which we argued above in the section on *normalizing flows* that it was a bad idea. However, with the methodology from score matching and denoising we've arrived at a fundamentally different (and more powerful) way to transform distributions. E.g., we're freed from constraints to preserve densities, and have a very tractable and convenient learning objective. In contrast to *Neural-ODEs* we can train with single steps, instead of having to backpropagate through the full chain from end to start.

In addition, a fundamental advantage of flow matching over denoising diffusion models comes from a smart choice of the generating velocity paths: aiming for a linear velocity towards the target means we would obtain a straight path from an original sample towards our goal point. **Linear** here means that we could take a **single** Euler step to compute it from any starting point! As illustrated in the picture right above, this stands in huge contrast to the hundreds of steps we might have to make to follow one of the curved (unconstrained) denoising paths. It turns out we don't always obtain perfectly straight paths, so a single step can be sub-optmial (or might require further fine tuning / distillation of the flows). Nonetheless, flow matching typically works with a substantially reduced number of iterations compared to denoising. As inference time is directly proportional to the number of steps, this will result in a corresponding speed up when computing a sample from our posterior distribution. And, most importantly, we've not sacrificied any of the original goals: a convenient sampling procedure and an accurate coverage of the posterior just from data.

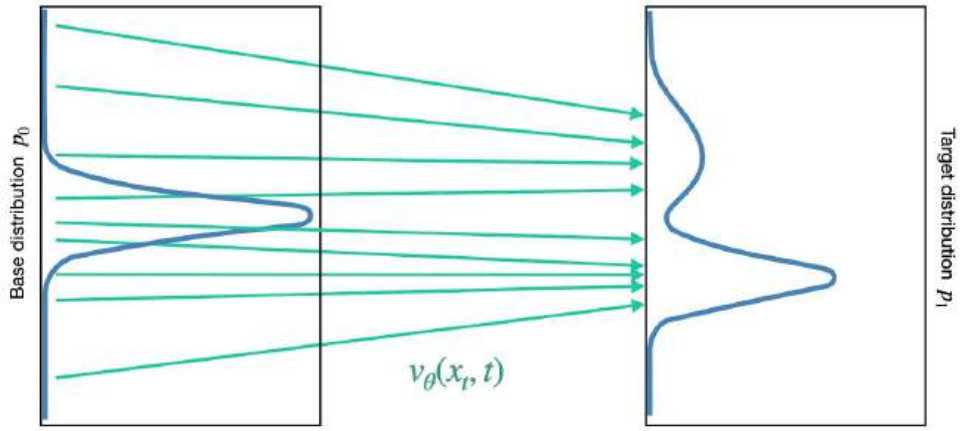


Fig. 25.1: Flow matching (in contrast to CNFs and DDPMs): the simple base distribution (left) is transfored by ODE integration of the inferred flow velocity to obtain a sample from the target distribution (right). Due to the linear paths, only very few integration steps are necessary.

25.3 Implementing Flow Matching

Just like for DDPM, we first need to initialize our standard Gaussian Mixture (GM) case:

```
import numpy as np

class GaussianMixture:
    def __init__(self, parameters):
        self.parameters = parameters
        self.distributions = [
            {
                'mean': np.array(dist['mean']),
                'std': np.array(dist['std']),
                'cov': np.diag(np.array(dist['std']) ** 2)
            }
            for dist in parameters
        ]

    def sample(self, num_samples):
        samples = []
        num_distributions = len(self.distributions)
        for _ in range(num_samples):
            idx = np.random.randint(num_distributions)  # Choose a random Gaussian
            dist = self.distributions[idx]
            sample = np.random.multivariate_normal(mean=dist['mean'], cov=dist['cov'])
            samples.append(sample)
        return np.array(samples)

parameters = [
    {"mean": [0, 0], "std": [1, 1]},
    {"mean": [3, 2], "std": [0.5, 0.5]}
]
mixture = GaussianMixture(parameters)
```

Setting up the training dataset is even simpler than for the denoising task. We don't need to include a noising schedule,

but rather, we simply compute a straight velocity `u_t` for each requested sample after selecting a random time `t`.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class FlowMatchingDataset(Dataset):
    def __init__(self, mixture, n_samples=1000, sigma_min=1e-4):
        super().__init__()
        self.n_samples = n_samples
        self.mixture = mixture
        self.sigma_min = sigma_min
        self.x1 = mixture.sample(n_samples)

    def __len__(self):
        return self.n_samples

    def __getitem__(self, idx):

        x0 = np.random.multivariate_normal([0.0, 0.0], np.eye(2), 1)[0]
        t = np.random.rand()  # scalar in [0,1]
        x1 = self.x1[idx]

        x_t = (1 - ( 1 - self.sigma_min) * t) * x0 + t * x1
        u_t = (x1 - x0)
        x_t = torch.tensor(x_t, dtype=torch.float32)
        t = torch.tensor([t], dtype=torch.float32)
        u_t = torch.tensor(u_t, dtype=torch.float32)

        return x_t, t, u_t
```

Similar to before, we'll use a simple neural network with three layers as "backbone" for the flow matching task:

```
class VelocityNet(nn.Module):
    def __init__(self, in_dim=2, time_dim=1, hidden_dim=128, out_dim=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim + time_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, out_dim)
        )

    def forward(self, x, t):
        xt = torch.cat([x, t], dim=1)
        return self.net(xt)
```

25.3.1 Training a Velocity Model

Now we can start training the velocity model. This means with simply let the network predict the "supervised" ground truth velocity we receive from the dataset class. Just like for denoising, we have a very well-defined and stable learning task. The code cell below runs this training for 50 epochs:

```
n_samples = 10000
batch_size = 128
n_epochs = 50
learning_rate = 1e-3

dataset = FlowMatchingDataset(mixture, n_samples=n_samples)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

model = VelocityNet().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()

for epoch in range(n_epochs):
    running_loss = 0.0
    for x_t, t, u_t in dataloader:
        x_t = x_t.to(device)
        t = t.to(device)
        u_t = u_t.to(device)
        optimizer.zero_grad()
        pred_v = model(x_t, t)
        loss = criterion(pred_v, u_t)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * x_t.size(0)
    running_loss /= len(dataset)
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {running_loss:.4f}")
```

```
Epoch 1/50, Loss: 2.8302
Epoch 2/50, Loss: 2.2797
Epoch 3/50, Loss: 2.2231
Epoch 4/50, Loss: 2.1215
Epoch 5/50, Loss: 2.1258
Epoch 6/50, Loss: 2.0737
Epoch 7/50, Loss: 2.1223
Epoch 8/50, Loss: 2.0625
Epoch 9/50, Loss: 2.0656
Epoch 10/50, Loss: 2.0527
...
Epoch 40/50, Loss: 2.0522
Epoch 41/50, Loss: 2.0709
Epoch 42/50, Loss: 2.0460
Epoch 43/50, Loss: 2.0688
Epoch 44/50, Loss: 2.0273
Epoch 45/50, Loss: 2.0711
Epoch 46/50, Loss: 2.0120
Epoch 47/50, Loss: 2.0344
Epoch 48/50, Loss: 2.0347
Epoch 49/50, Loss: 2.0537
Epoch 50/50, Loss: 2.0440
```

25.3.2 Inference via Solving the ODE

At inference time, we actually only have to integrate the velocities produced by the network starting from a random sample. While this could be done with simple Euler steps, we'll again use the ODE solvers from the `torchdiffeq` package for additional flexibility. (Feel free to experiment with other integration schemes below.)

```
try:
    import google.colab  # only to ensure that we are inside colab
    %pip install --quiet torchdiffeq
except ImportError:
    print("This notebook is running locally, please install torchdiffeq manually.")
```

The integration itself is done in `odeint` using the trained network in the `ode_func` wrapper. By default, we're integrating the trajectories over 100 steps, but in practice, much fewer should still give good results. Nonetheless, this is also an order of magnitude less that what we used for the DDPM version.

```
import torch
from torchdiffeq import odeint

def integrate_flow(model, x0, t_span=(0.0, 1.0), n_steps=100, method='dopri5'):

    t = torch.linspace(t_span[0], t_span[1], n_steps).to(x0.device)

    def ode_func(t, x):
        t_tensor = t.expand(x.shape[0], 1)
        return model(x, t_tensor)

    trajectory = odeint(ode_func, x0, t, method=method,
                        atol=1e-5, rtol=1e-5)

    return trajectory, t

n_gen = 500

x0_gen = torch.randn(n_gen, 2).to(device)
trajectory, time_points = integrate_flow(model, x0_gen)
```

25.3.3 Visualize the Sampling Trajectories

We start with points from a standard gaussian, and color them based on their initial position. The plots below again show states at six different times of the trajectories.

```
import numpy as np
import matplotlib.pyplot as plt

import seaborn as sns
sns.set_theme(style="ticks", palette="pastel")

def get_angle_colors(positions):
    angles = np.arctan2(positions[:, 1], positions[:, 0])
    angles_deg = (np.degrees(angles) + 360) % 360
    colors = np.zeros((len(positions), 3))
    for i, angle in enumerate(angles_deg):
        segment = int(angle / 120)
        local_angle = angle - segment * 120
        if segment == 0:    # 0 degrees to 120 degrees (R->G)
```

(continues on next page)

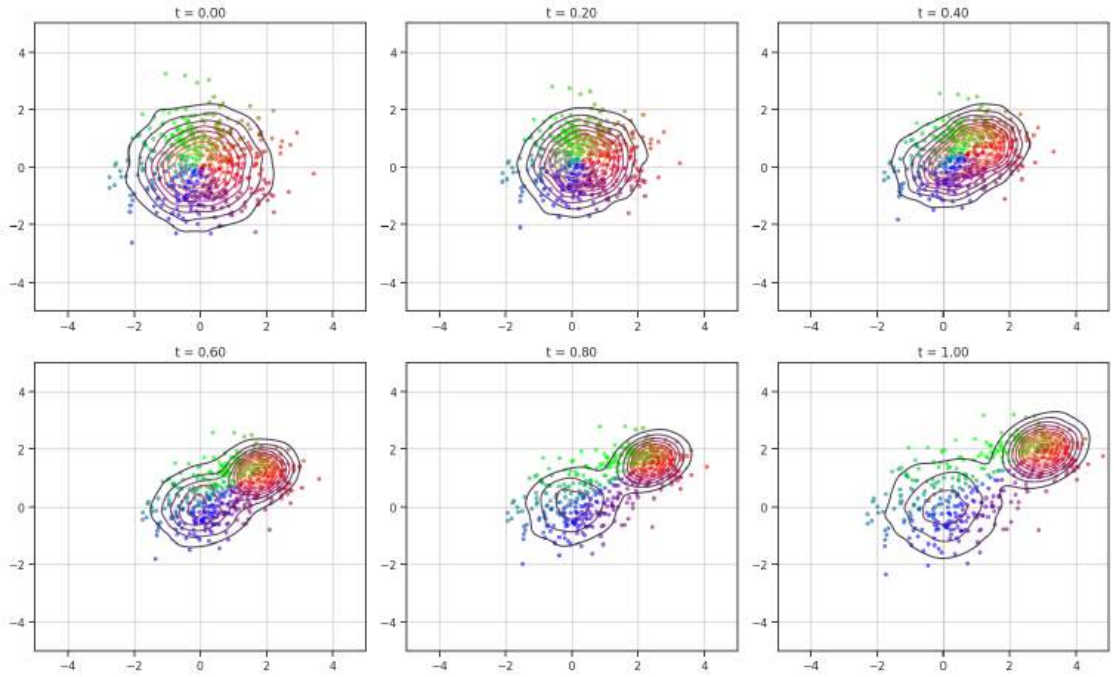
(continued from previous page)

```
            colors[i] = [1 - local_angle/120, local_angle/120, 0]
        elif segment == 1:  # 120 degrees to 240 degrees (G->B)
            colors[i] = [0, 1 - local_angle/120, local_angle/120]
        else:               # 240 degrees to 360° (B->R)
            colors[i] = [local_angle/120, 0, 1 - local_angle/120]

    return colors

# Desired time points to visualize
desired_times = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

# Determine indices in the trajectory corresponding to these times.
# We assume time_points is a 1D tensor of size n_steps.
time_np = time_points.detach().cpu().numpy()
n_steps = len(time_np)

# Get the index for each desired time: here we choose the index with minimum absolute␣
↪difference.
indices = [np.argmin(np.abs(time_np - t_val)) for t_val in desired_times]

# Create subplots: We'll use 2 rows and 3 columns, one subplot per time.
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))
axes = axes.ravel()  # flatten the 2D array for easier indexing

# Create grid for density visualization
xx, yy = np.mgrid[-5:5:100j, -5:5:100j]
positions = np.vstack([xx.ravel(), yy.ravel()])

for i, idx in enumerate(indices):
    ax = axes[i]
    # Get the samples at this time point: shape (batch, 2)
    x_t = trajectory[idx].detach().cpu().numpy()

    if i == 0:
        c = get_angle_colors(x_t)

    x_0 = mixture.sample(5000)
    t = time_np[idx]
    eps = np.random.randn(5000, 2)
    x_t_forward = t * x_0 + (1-t) * eps
    samples_ = x_t_forward

    # Compute KDE for visualization
    from scipy.stats import gaussian_kde
    kernel = gaussian_kde(samples_.T)
    density = np.reshape(kernel(positions).T, xx.shape)

    # Plot density contours
    ax.contour(xx, yy, density, levels=10, alpha=0.8)

    # Scatter plot: plot each sample as a point
    ax.scatter(x_t[:, 0], x_t[:, 1], alpha=0.5, s=10, color=c)
    ax.set_title(f"t = {time_np[idx]:.2f}")
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.grid(True)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



This looks just like it should - both density peaks are well represented, and with *flow matching*, we now have a method at hand that yields accurate samples from the posterior with a (relatively) mild computational overhead in comparison to a deterministic model.

25.4 Summary

The **flow matching** algorithm is an important milestone in the field of diffusion models, and it concludes our trip through the history of generative modeling approaches in deep learning. Interestingly, after they were proposed, denoising, flow matching, and other variants were shown to be more similar than one would think based on the derivation. Nonetheless, they're easier to understand via their respective views on the problem, rather than from a more generic mathematical framework. Nonetheless, this field is a very exciting. We can recommend browsing recent developments online!

Nonetheless, it's time to show what these methods can do with specific and more complex examples than our GM toy problem. The following notebook will do just that. We'll start by comparing denoising diffusion models with flow matching models for a RANS airfoil task with uncertainty.

CHAPTER

TWENTYSIX

DENOISING AND FLOW MATCHING SIDE-BY-SIDE

To show the capabilities of **denoising diffusion** and **flow matching**, we'll be use a learning task where we can reliably generate arbitrary amounts of ground truth data. This ensures we can quantify how well the target distribution was learned. Specifically, we'll focus on Reynolds-averaged Navier-Stokes simulations around airfoils, which have the interesting characteristic that typical solvers (such as OpenFoam) transition from steady solutions to oscillating ones for larger Reynolds numbers. This transition is exactly what we'll give as a task to diffusion models below. (Details can be found in our diffusion-based flow prediction repository.) Also, to make the notebook self-contained, we'll revisit the most important concepts from the previous section. [run in colab]

Note

If you're directly continuing reading from the previous chapter, note that there's an important difference: we'll deviate from the _simulation-based inference viewpoint, and for simplicity we'll apply denoising and flow-matching to a **forward** problem. We won't be aiming to recover x for an observation y, but rather assume we have initial conditions x from which we want to compute a solution y. So don't be surprised by the switched x and y below.

26.1 Intro

Diffusion models have been rising stars ⋆ in the deep learning field in the past years, and have made it possible to train powerful generative models with surprisingly simple and robust training setups. Within this sub-field of deep learning, a very promising new development are flow-based approaches, typically going under names such as *flow matching* [LCBH+22] and *rectified flows* [LGL22] . We'll stick to the former here for simplicity, and denote this class of models with *FM*.

For the original diffusion models, especially the *denoising* tasks were extremely successful: a neural network learns to restore a signal from pure noise. Score functions provided an alternate viewpoint, but ultimately also resulted in denoising tasks. Instead, flow-based approaches aim for transforming distributions. The goal is to transform a known one, such as gaussian noise, into one that represents the distribution of the signal or target function we're interested in. Despite these seemingly different viewpoints, all viewpoints above effectively do the same: starting with noise, they step by step turn it into samples for our target signal. Interestingly, the FM-perspective is not only more stable at training time, it also speeds up inference by orders of magnitude thanks to yielding straighter paths. And even better: if you have a working DM setup, it's surprisingly simple to turn it into an FM one.

Below, we'll highlight the similarities and differences, and evaluate both methods with the RANS-based flow setup outlined above.

26.2 Problem statement

Instead of the previous supervised learning tasks, we'll need to consider distributions. For "classic" supervised tasks, we have unique input-output pairs (x, y) and train a model to provide y given x based on internal parameters θ, i.e. $y = f(x; \theta)$.

In contrast, let's assume there is *some* hidden state ψ, that varies for a single x. This could e.g. be measurement noise, the starting point of an optimization for inverse problems, or the non-converging solution of a RANS solver (our scenario here). Now we can phrase our problem in terms of random variable Y, and our solution is drawn from the distribution $y \sim P_Y(Y)$ that we typically specify as a marginalized distribution, in terms of samples with varying ψ for any given x. From a probabilistic perspective, it is important to capture the conditional probabilty of our solutions, i.e. $p(y|x)$, where we marginalize over ψ. (We don't need to know about the specifics of ψ in practice.)

This conditional distribution $p(y|x)$, the *posterior*, is exactly what our generative model should learn: when we repeatedly evaluate our model, it should give us samples from the posterior with the right probabilities. And it should do so efficiently, without wasting computations…

26.3 Implementation and Setup

First, we need to install the required packages and clone the repository:

```
try:
    import google.colab  # to ensure that we are inside colab
    !pip install --upgrade --quiet einops bayesian_torch
except ImportError:
    print('This notebook is running locally, please make sure the necessary pip␣
 ↪packages are installed.')
    pass
!git clone https://github.com/tum-pbs/Diffusion-based-Flow-Prediction.git
%cd Diffusion-based-Flow-Prediction/
```

```
site-packages/IPython/core/magics/osm.py:417: UserWarning: This is now an optional␣
 ↪IPython functionality, setting dhist requires you to install the `pickleshare`␣
 ↪library.
  self.shell.db['dhist'] = compress_dhist(dhist)[-100:]
```

We also need to prepare the training dataset. The one below can be generated with OpenFoam, but is downloaded below for convenience. The data structure and the `DataFiles` class that are used to organize the dataset come from the diffusion-based flow prediction repository, details can be found here if you're interested.

```
import zipfile
from airfoil_diffusion.airfoil_datasets import *
from airfoil_diffusion.networks import *
from airfoil_diffusion.trainer import *
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Using device: "+str(device))

if not os.path.exists("./datasets/1_parameter/data/"):
    files=[file for file in os.listdir("./datasets/1_parameter/") if file.endswith(".
 ↪zip")]
    for file in tqdm(files):
        f=zipfile.ZipFile("./datasets/1_parameter/"+file,'r')
```

(continues on next page)

(continued from previous page)

```
        for file in f.namelist():
            f.extract(file,"./datasets/1_parameter/data/")
        f.close()

df_train=FileDataFiles("./datasets/1_parameter/train_cases.txt",base_path="./datasets/
↪1_parameter/data/")
train_dataset = AirfoilDataset(df_train,data_size=32)
```

```
Using device: cuda:0
```

```
Loading data: 100%|██████████| 125/125 [00:00<00:00, 341.19it/s]
```

Next, we'll implement the denoising diffusion model, so that we can compare with flow matching afterwards.

26.4 Denoising with Diffusion Models

At it's core, the denoising task is as simple as the name implies: we learn to estimate the noise ϵ by minimizing $\| \epsilon - f_\theta(x,t) \|^2$, the trick is primarily to carefully control the noise so that it can be learned easily.

Note that for all equations here, we'll omit the i subscript that previously denoted the different samples of a batch or dataset in *Models and Equations*. Thus, we'll e.g. shorten y_i to y, and leave out summations over i.

To get started with denoising, we'll define a forward process that adds noise to give a perfect Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$ at $t = 1$. It starts with a data sample y at $t = 0$, i.e. $\mathcal{N}(y, 0)$ and then turns it into noise as t increases. Note that the noising / denoising time t is a *virtual* one, it has no physical meaning. The change of mean and standard deviation over time t is controlled by a *noise schedule* $\beta^t \in (0,1)$, that gradually increases from 0 to $\beta^t = 1$ at the end of the chain, where $t = 1$.

By choosing a Gaussian distribution, we can decouple the steps to give a Markov chain for the distribution q of the form

$$q\left(y^{0:T}\right) = q(y^0) \prod_{t=1}^{T} q\left(y^t \mid y^{t-1}\right),$$

where

$$q\left(y^t \mid y^{t-1}\right) = \mathcal{N}\left(\sqrt{1-\beta^t} y^{t-1}, \beta^t \mathbf{I}\right).$$

It's fairly obvious that we can destroy any input signal y by accumulating more and more noise. What's more interesting is the reverse process that removes the noise, i.e. the denoising. We can likewise formulate a reverse Markov chain for the distribution p_θ. The subscript already indicates that we'll learn the transition and parameterize it by a set or parameters θ:

$$p_\theta\left(y^{0:T}\right) = p(y^T) \prod_{t=1}^{T} p_\theta\left(y^{t-1} \mid y^t\right)$$

with

$$y^t = \sqrt{\bar{\alpha}^t} y^0 + \sqrt{(1-\bar{\alpha}^t)} \epsilon. \tag{26.1}$$

We can calculate the correct coefficients α from the noise schedule of the forward chain via $\alpha^t = 1 - \beta^t$ and $\bar{\alpha}^t = \prod_{i=1}^{t} \alpha^i$.

Each step $p_\theta\left(y^{t-1} \mid y^t\right)$ along the way now has the specific form

$$p_\theta\left(y^{t-1} \mid y^t\right) = \mathcal{N}\left(\mu(f_\theta), \sigma_\theta\right) \tag{26.2}$$

where we're employing a neural network f_θ to predict the noise. We could also call the network ϵ_θ here, but for consistency we'll stick to f_θ. The noise inferred by our network parametrizes the mean

$$\mu(\epsilon) = \frac{1}{\sqrt{\alpha^t}}\left(y^t - \frac{\beta^t}{\sqrt{1-\bar{\alpha}^t}}\epsilon\right).$$

The standard deviation interestingly does not depend on the noise (and our network), but evolves over time with

$$\sigma = \frac{1-\bar{\alpha}^{t-1}}{1-\bar{\alpha}^t}\beta^t \mathbf{I}.$$

Thus, given a pair x, y, we can directly compute the right amount of noise for time $t-1$ and t, and generate y^t and y^{t-1}. In practice, we're not only interested in retrieving an arbitrary y, but the one that corresponds to some global paramters like a chosen Reynolds number. These conditions, together with e.g. the shape of an airfoil are actually our x from $f(x) = y$ at the top. So, we'll also condition f on x to have the form $f_\theta(y^t, x, t)$.

In practice, we simply choose a time t, compute the right amount of noise ϵ, and let our network predict the noise given y^t computed by linear interpolation with $\bar{\alpha}^t$, as given above in equation `ref`(eq-yt). This gives the loss function: $\mathcal{L}_{\text{DM}}(\theta) = \mathbb{E}_{x,\epsilon\sim\mathcal{N}(\mathbf{0},\mathbf{I}),t}\left[\| \epsilon - f_\theta(y^t, x, t) \|^2\right]$

26.5 Implementing DDPM

We'll split the implementation into a helper class that computes the coefficients for the denoising schedule, and a trainer class that takes care of the training loop.

The helper class is callsed `MyDiffuser` below, and starts by computing the beta coefficients for a so called *cosine-beta* schedule. It has the form $\beta = cos\left((t/T + s) \;/\; (1+s) * \pi/2\right)^2$. The offset s below is chosen such that the standard deviation of the $\sqrt{\beta}$ is smaller than the color step of $1/256$ for a typical RGB image. The `generate_parameters_from_beta()` function takes the precomputed list of `beta` coefficients, and turns them into PyTorch tensors, computing the correct `alpha` and `alpha_bar` values along the way.

The class also implements a function `forward_diffusion()` that calculates forward diffusion step using the precomputed `alpha_bar` values, given an input y and t. Hence, this function computes y^t from above.

```
class MyDiffuser():
    def __init__(self, steps, device):
        self.device = device
        self.steps = steps
        self.name = "Cos2ParamsDiffuser"

        s = 0.008
        tlist = torch.arange(1, self.steps+1, 1)
        temp1 = torch.cos((tlist/self.steps+s)/(1+s)*np.pi/2)
        temp1 = temp1*temp1
        temp2 = np.cos(((tlist-1)/self.steps+s)/(1+s)*np.pi/2)
        temp2 = temp2*temp2
        self.beta_source = 1-(temp1/temp2)
        self.beta_source[self.beta_source > 0.999] = 0.999
        self.generate_parameters_from_beta()


    def generate_parameters_from_beta(self):
        self.betas = torch.cat( (torch.tensor([0]), self.beta_source), dim=0)
        self.betas = self.betas.view(self.steps+1, 1, 1, 1)
        self.betas = self.betas.to(self.device)
```

(continues on next page)

(continued from previous page)

```
        self.alphas = 1-self.betas
        self.alphas_bar = torch.cumprod(self.alphas, 0)
        self.one_minus_alphas_bar = 1 - self.alphas_bar
        self.sqrt_alphas = torch.sqrt(self.alphas)
        self.sqrt_alphas_bar = torch.sqrt(self.alphas_bar)
        self.sqrt_one_minus_alphas_bar = torch.sqrt(self.one_minus_alphas_bar)


    def forward_diffusion(self, y0, t, noise):
        yt = self.sqrt_alphas_bar[t]*y0+self.sqrt_one_minus_alphas_bar[t]*noise
        return yt
```

Now we're ready to start training. The `MyDiffusionTrainer` class derives from a `Trainer` base class from the airfoil diffusion repository. This base class primarily handles parameters, book-keeping and a few other mundane tasks. Effectively, it makes sure we have a batch to train with, and then calls `train_step()`, which is the most interesting function below.

It implements exactly the procedure outlines above: given a y, we compute noise, and y^t with a forward step for a random t. Then we let our network predict the noise ϵ from y^t, the condition x, and t. All that's left afterwards is to compute an MSE loss on the true noise versus the precicted one, and let PyTorch backpropagate the gradient to update the weights of our neural network.

```
class MyDiffusionTrainer(TrainerStepLr):

    def __init__(self) -> None:
        super().__init__()

    def set_configs_type(self):
        super().set_configs_type()
        self.configs_handler.add_config_item("diffusion_step",value_type=int,default_
↪value=200,description="The number of diffusion steps.")

    def event_before_training(self,network):
        self.diffuser = MyDiffuser(steps=self.configs.diffusion_step,device=self.
↪configs.device)

    def train_step(self, network: torch.nn.Module, batched_data, idx_batch: int, num_
↪batches: int, idx_epoch: int, num_epoch: int):
        condition = batched_data[0].to(device=self.configs.device)
        targets = batched_data[1].to(device=self.configs.device)
        t = torch.randint(1, self.diffuser.steps+1,
                          size=(targets.shape[0],), device=self.configs.device)
        noise = torch.randn_like(targets, device=self.configs.device)
        yt = self.diffuser.forward_diffusion(targets, t, noise)
        predicted_noise = network(yt, t, condition)
        loss=torch.nn.functional.mse_loss(predicted_noise, noise)
        return loss

diffusion_trainer = MyDiffusionTrainer()

dif_network = AifNet("./pre_trained/single_parameter/32/diffusion/network_configs.yaml
↪")
```

At the end of the cell above we directly instantiate a trainer object, and initialiaze a neural network. The `AifNet` class implements a *state-of-the-art* U-net with all the latest tricks for diffusion modeling, but in the end it's "just a U-net", and we'll skip the details here.

More importantly, we can finally starting training our DDPM model. For that we can use the `train_from_scratch()` function of the trainer class, which we'll call in the next cell. The training with a default of 10000 steps can take a while, but shouldn't take much longer than half an hour on current GPUS. If you're not patient enough, feel free to skip this step and load one of the pre-trained models from our DBFP repository with the commented-out code at the bottom.

```
diffusion_trainer.train_from_scratch(name="diffusion", #device="cpu",
                                     network=dif_network,
                                     train_dataset=train_dataset,
                                     path_config_file="./pre_trained/train_configs.
 ↪yaml",
                                     save_path="./training/single_parameter/32/",␣
 ↪epochs=10000)

# alternative load
#dif_network.load_state_dict(torch.load("./pre_trained/single_parameter/32/diffusion/
 ↪weights_0.pt"))
```

```
Trainer created at 2024-11-05-03_25_26
Working path:./training/single_parameter/32/diffusion/2024-11-05-03_25_26/
Random seed: 1730773526
Training configurations saved to ./training/single_parameter/32/diffusion/2024-11-
 ↪05-03_25_26/configs.yaml
Network has 1185218 trainable parameters
There are 5 training batches in each epoch
Batch size for training:25
Training epochs:10000
Total training iterations:50000
learning rate:0.0001
Optimizer:AdamW
Learning rate scheduler:step
Training start!
  0%|          | 0/10000 [00:00<?, ?it/s]/tmp/ipykernel_65435/3287631965.py:11:␣
 ↪DeprecationWarning: __array_wrap__ must accept context and return_scalar␣
 ↪arguments (positionally) in the future. (Deprecated NumPy 2.0)
  temp2 = np.cos(((tlist-1)/self.steps+s)/(1+s)*np.pi/2)
lr:1.000e-05 train loss:0.00100: 100%|██████████| 10000/10000 [30:40<00:00,  5.
 ↪43it/s]
Training finished!
```

Before we investigate the capabilities of this model, let's directly train a flow matching version, so that we can compare.

26.6 Flow Matching

Instead of adding and removing noise, flow matching transforms probability distributions. Let's consider a time-dependent differentiable mapping $\phi : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$ that transforms samples $x_0 \in \mathbb{R}^d$ from an initial distribution $p(x_0)$ to samples x_t from a distribution $p(x_t)$. In short: $x_t = \phi(x_0)$.

Later on, x_0 will represent samples from a simple distribution, such as a Gaussian distribution, similar to what we used for the diffusion models previously. x_1, on the other hand, corresponds to samples from the target distribution, i.e., samples from our training dataset (y in the notation above). As we're going from Gaussian noise towards a target, the progression is similar to what we saw for denoising: from very noisy to no noise, despite the original flow matching formulation not necessarily aiming for this behavior. For the transformation of distributions, it's convenient to consider continuously changing distributions $p(x_t)$ for varying t. Just keep in mind that for $t = 1$, we're at x_1 which is identical to y. I.e. $p(x_t)|_{t=1} = p(x_1) = p(y)$.

Flow matching learns the time derivative of this transformation, the *flow*, as a time-dependent vector field $u : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d$, where $u_t(x_t) = \frac{d}{dt} x_t$. For a neural network $f_\theta(x,t)$ the loss function is simply an L^2 between predicted and target velocities:

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{t\sim[0,1],\ x_t\sim p(x_t)} \| f_\theta(x_t,t) - u_t(x_t) \|^2,$$

where p_t denotes the intermediate distributions at time t with $t \sim [0,1]$.

Looks surprisingly simple so far, and a lot like the loss for our noise estimation problem above. However, tithout additional tricks this loss function is intractable since we don't know the distributions $p(x_t)$ and the correct velocities u_t.

Luckily, it was shown in previous work that we can make use of the known samples x_1 to drive the procedure without distorting the distributions. We can construct a conditional vector field $u_t(x_t|x_1)$ based on the x_1 samples. Then the intermediate probability density and vector fields can be marginalized by integrating over x_1 as follows: $p(x_t) = \int p(x_t|x_1) p(x_1) dx_1$, and $u_t(x_t) = \int u_t(x_t|x_1) \frac{p(x_t|x_1)p(x_1)}{p(x_t)} dx_1$. With this marginalization, it was demonstrated that learning this conditional flow is mathematically equivalent to learning the original flow.

We now have some freedom to prescribe flows, and it turns out that straight, rectified motions are particularly interesting. They can be derived from optimal transport, which to define a **linear** mapping between samples from $p(x_0)$ and $p(x_1)$. Starting with a normalized Gaussian distribution at $t = 0$, we then want the standard deviations σ_t to linearly decrease with $1 - t$, so that we're left with no randomness at $t = 1$. At the same time, the mean μ_t should change from zero to x_1, i.e. $\mu_t(x_1) = t\, x_1$.

This gives the mapping:

$$\phi_t(x_0) = \sigma_t(x_t) x_0 + \mu_t(x_t),$$

with it's time derivative being the velocity:

$$u_t(x_t|x_1) = \frac{d}{dt}\phi_t(x_0) = \sigma_t'(x_1) x_0 + \mu_t'(x_1) \ .$$

In practice, we also introduce a threshold σ_{min}, to ensure that the standard deviation stays above zero. In practice, σ_{min} is chosen sufficiently small so that $p(x_1|x_1)$ is representing a very concentrated Gaussian distribution centered at x_1. The time evolution of the standard deviation is then computed with $\sigma_t(x_1) = 1 - (1 - \sigma_{\text{min}})t$.

The great advantage of this setup is that it actually provides a "straight" motions with a constant vector field. The vector field is independent of time t! If things work out as planned, that means that we can compute the result of the transformation in a single Euler step, directly from x_0 to x_1. This is in stark contrast to the denoising above, where the network can learn arbitrary paths, and correspondingly requires a larger number of steps to arrive at the target. In practice, the *single-step* inference requires a few more tricks, but we'll see below that flow matching works with much fewer steps than denoising, even with this simple, basic formulation.

Now we have all necessary ingredients to compute the target velocities u_t as

$$u_t(x_t|x_1) = x_1 - (1 - \sigma_{\text{min}}) x_0 \ ,$$

and we can formulate the conditional version of the loss function above:

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t\sim[0,1],\ x_1\sim p(x_1),\ x_t\sim p(x_t|x_1)} \left\| f_\theta(x_t,t,x_1) - u_t(x_t|x_1) \right\|^2$$

And once trained, we can query our network for the vector field to generate x_1 samples from x_0 via integration in time: $x_1 = x_0 + \int_0^1 f_\theta(x_t,t)dt$. To integrate this ODE, any ODE solver can be used. For simplicity, we'll use Euler steps below, but you can try a variety of higher-order methods in the DBFP code.

26.7 Implementing Flow Matching

For the implementation, we'll again split the core functionality and the training code. The former is handled by the helper class `MyFlowMatcher`. It's even simpler than the previous one for denoising: `phi_t()` computes the linear forward step by interpolating two samples `x_0` and `x_1`. `u_t()` instead computes the time derivative, as explained above. Because we're aiming for a straight motion, it's constant in time, and `u_t` does not depend on `t` anymore.

The last function `cfm_loss()` computes the target velocity and evaluates the conditional flow matching loss for a training step.

```
class MyFlowMatcher():

    def __init__(self):
        self._unifrom_sampler = torch.distributions.uniform.Uniform(0.,1.)
        self.sig_min = 0.001

    def phi_t(self, x_0, x_1, t):
        return (1-(1-self.sig_min)*t)*x_0 + t*x_1

    def u_t(self, x_0, x_1): # note - linear flow, does not depend on t anymore!
        return x_1-(1-self.sig_min)*x_0

    def cfm_loss(self, network, x_1, x_0=None, *args,**kwargs):
        x_0 = torch.randn_like(x_1) if x_0 is None else x_0
        t = self._unifrom_sampler.sample([x_1.shape[0]]+[1]*(x_1.dim()-1)).to(x_1.
↪device)
        x_t = self.phi_t(x_0,x_1,t)
        v_t = self.u_t(x_0,x_1)
        return torch.mean( (network(x_t,t.view(t.shape[0]),*args,**kwargs) - v_t) **2)
```

The flow matching trainer relies on the `trainer` base class, and primarily has the job transfer targets and conditioning data to the PyTorch device, and call the `cfm_loss()` function. Not much left to do here…

```
class MyFMTrainer(TrainerStepLr):

    def __init__(self) -> None:
        super().__init__()

    def event_before_training(self,network):
        self.flow_matcher = MyFlowMatcher()

    def train_step(self, network: torch.nn.Module, batched_data, idx_batch: int, num_
↪batches: int, idx_epoch: int, num_epoch: int):
        condition = batched_data[0].to(device=self.configs.device)
        targets = batched_data[1].to(device=self.configs.device)
        loss=self.flow_matcher.cfm_loss(network=network,x_1=targets,
↪condition=condition)
        return loss
```

Next we can instantiate a trainer object, and allocate a network. We're using a U-net that's identical to the one previously used for denoising, so that we can make a fair comparison between the two training methodologies.

```
fmatching_trainer=MyFMTrainer()

network = AifNet("./pre_trained/single_parameter/32/diffusion/network_configs.yaml")
```

Now we can start training. Similar to before, this should take around half an hour for 10000 epochs, but if you want to skip this step, you can find code for loading one of the models from the github repo below.

```
fmatching_trainer.train_from_scratch(name="flowmatching",
                                     network=network,
                                     train_dataset=train_dataset,
                                     path_config_file="./pre_trained/train_configs.
 ↪yaml",
                                     save_path="./training/single_parameter/32/",␣
 ↪epochs=10000)

# uncomment to load the checked in model
#network.load_state_dict(torch.load("./pre_trained/single_parameter/32/flow_matching/
 ↪weight.pt"))
```

```
Trainer created at 2024-11-05-03_56_06
Working path:./training/single_parameter/32/flowmatching/2024-11-05-03_56_06/
Random seed: 1730775366
Training configurations saved to ./training/single_parameter/32/flowmatching/2024-
 ↪11-05-03_56_06/configs.yaml
Network has 1185218 trainable parameters
There are 5 training batches in each epoch
Batch size for training:25
Training epochs:10000
Total training iterations:50000
learning rate:0.0001
Optimizer:AdamW
Learning rate scheduler:step
Training start!
lr:1.000e-05 train loss:0.00430: 100%|██████████| 10000/10000 [30:44<00:00,  5.
 ↪42it/s]
Training finished!
```

We finally have to trained models that we can evaluate side by side.

26.8 Test Evaluation

To evaluate the trained models on inputs that weren't used for training we first need to download some more data. This is what happens in the next cell. The `scale_factor=0.25` parameters of the `read_single_file()` function below make sure that we get fields of size 32×32 like the ones in the training data set. However, the test set has previously unseen Reynolds numbers, so that we can check how well the model generalizes. While loading the data, the code also computes statistics for the ground truth mean and standard deviations (`mean_field_test_gd` and `std_field_test_gd`). This data will be used to quantify differences between the trained models later on.

```
df_test=FileDataFiles("./datasets/1_parameter/test_cases.txt",base_path="./datasets/1_
 ↪parameter/data/")
df_test.sort()
std_field_test_gd=[]
```

(continues on next page)

(continued from previous page)

```
mean_field_test_gd=[]
inputs_test=[]
samples_gd=[]
for case in df_test.get_simulation_cases():
    datas=[]
    selected_cases=df_test.select_simulation_cases([case])
    for case in selected_cases:
        raw_data=read_single_file(case['path']+case['file_name'],model="dimless",
↪scale_factor=0.25)
        datas.append(
            raw_data[3:]
            )
    # scale factor is 0.25 to get 32x32 data
    inputs_test.append(read_single_file(case['path']+case['file_name'],model=
↪"normalized",scale_factor=0.25)[0:3])
    samples_gd.append(np.stack(datas,axis=0))
    std_field_test_gd.append(samples_gd[-1].std(axis=0))
    mean_field_test_gd.append(samples_gd[-1].mean(axis=0))
std_field_test_gd=np.stack(std_field_test_gd,axis=0)
mean_field_test_gd=np.stack(mean_field_test_gd,axis=0)

df_all=DataFiles(df_train.case_list+df_test.case_list)
df_all.sort()
std_value_gd=[]
for case in df_all.get_simulation_cases():
    datas=[]
    selected_cases=df_all.select_simulation_cases([case])
    for case in selected_cases:
        datas.append(
            read_single_file(case['path']+case['file_name'],model="dimless",scale_
↪factor=0.25)[3:]
            )
    std_value_gd.append(np.stack(datas,axis=0).std(axis=0).mean())
```

The next cell defines two helper functions to compute the Euler integration for a chosen number of steps of a trained FM model. It simply evalutes the target samples via ODE integration steps to compute $x_1 = x_0 + \int_0^1 v_\theta(x,t)dt$.

It does so using Euler steps (for simplicity) for a whole batch of 25 samples, while the main `sample_flowmatching()` function breaks down larger inputs into chunks of 25. More advanced ODE integration methods are interesting to try here, of course, and a variety of integrators can be found in the accompanying github repository.

It's worth explicitly pointing out a key difference between denoising and flow matching here that is not so obvious from the equations above: for denoising, we repeatedly add noise again over the course of the denoising steps. Flow matching, in contrast, only works with the initial noise, and follows the trajectory prescribed by the learned vector field. Hence, *no noise is added* over the course of the flow matching steps at inference time.

```
def integrate_euler( f, x_0, t_0, t_1, dt):
    t_0 = torch.as_tensor(t_0, dtype=x_0.dtype, device=x_0.device)
    t_1 = torch.as_tensor(t_1, dtype=x_0.dtype, device=x_0.device)
    dt  = torch.as_tensor(dt,  dtype=x_0.dtype, device=x_0.device)
    with torch.no_grad():
        t=t_0
        x=x_0
        while (t_1 - t) > 0:
            dt = torch.min(abs(dt), abs(t_1 - t))
            x, t = x + dt * f(t,x), t + dt
```

(continues on next page)

(continued from previous page)

```
    return x

def fm_sample( network, x_0, dt, condition):
    with torch.no_grad():

        def wrapper(t,x):
            return network(x,
                        t*torch.ones((x.shape[0],)).to(x_0.device),
                        condition=condition)

        return integrate_euler( f=wrapper, x_0=x_0,  t_0=0.,  t_1=1., dt=dt)

def sample_flowmatching(network, input_field, dt, num_sample=100):
    network.eval();network.to(device);predictions=[]
    batch_size=25;N_all=num_sample

    while N_all>0:
        batch_size_now=min(batch_size,N_all)
        N_all-=batch_size
        condition=input_field.to(device).repeat(batch_size_now,1,1,1)
        noise=torch.randn_like(condition)
        prediction_batch=normalized2dimless(
            fm_sample(x_0=noise,
                      network=network, dt=dt,
                      condition=condition)
            )
        predictions.append(prediction_batch.detach().cpu().numpy())
    predictions=np.concatenate(predictions,axis=0)
    return np.mean(predictions,axis=0), np.std(predictions,axis=0), predictions
```

We'll directly test our FM model with a varying number of integration steps. As promised above, FM can produce results in very few steps, so this is an interesting hyperparameter to vary. The next cell collects results via `sample_flowmatching()` with `step` varying from `1` to `100`. For qualitative comparison, we'll only do this for a single test sample, so that we can visualize the results next to the ground truth.

For each integration step count, we'll collect 100 samples produces with different noise as starting point, in order to gather the mean and standard deviation statistics.

```
index=3
input_field=inputs_test[index].unsqueeze(0)

titles=[]
result_fms=[]
for step in tqdm([1,5,20,100]):
    mean_fm,std_fm,samples_fm=sample_flowmatching(network,input_field,dt=1/step)
    titles.append("Flow Matching {}".format(step))
    result_fms.append(np.concatenate([mean_fm,std_fm],axis=0))
```

```
100%|██████████| 4/4 [00:05<00:00,  1.36s/it]
```

Next, we repeat this process and define helper functions to produce samples with the diffusion model.

As mentioned just above, DDPM adds the correct amount of noise for the current noise schedule in the `DDPM_sample_step()` calls.

```
def diffusion_sample_from_noise(diffuser, model, condition):
    with torch.no_grad():
```

(continues on next page)

(continued from previous page)

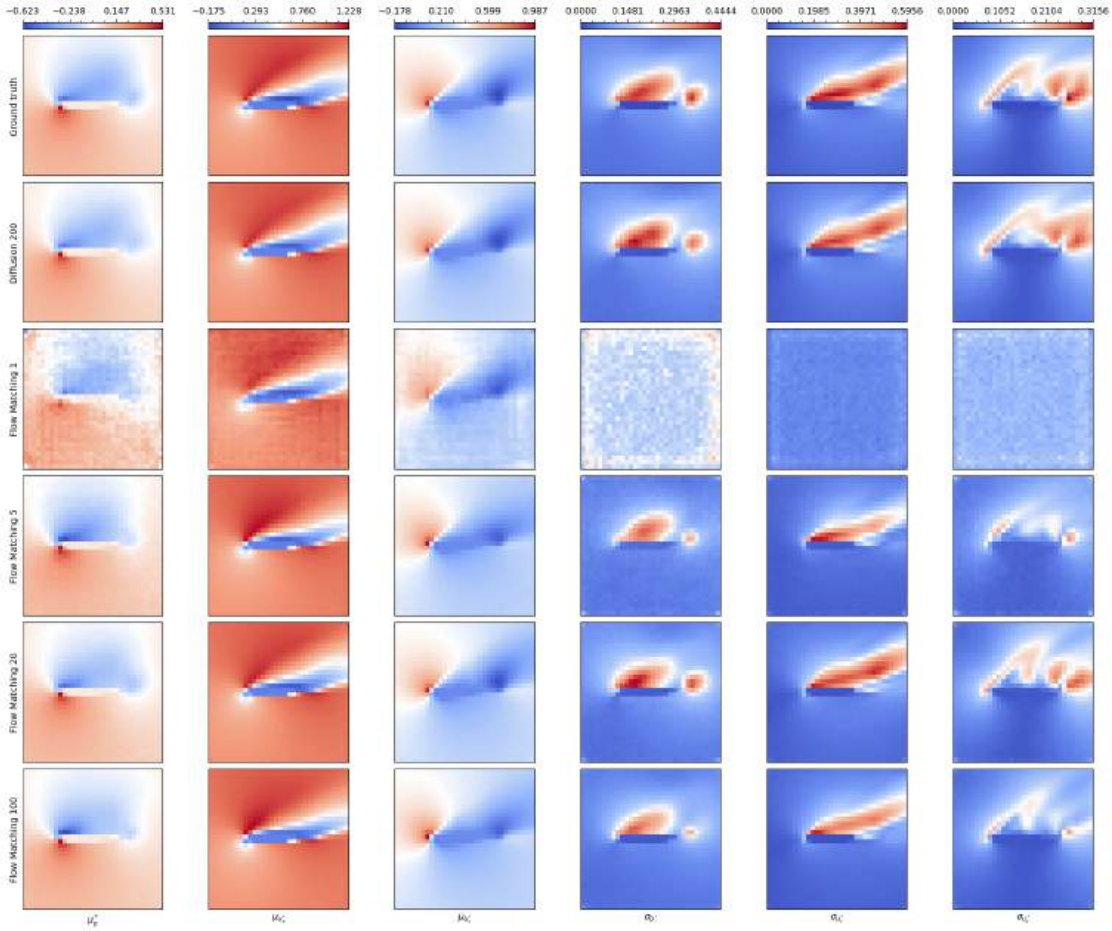
```
        x_t=torch.randn_like(condition)
        t_now = torch.tensor([diffuser.steps], device=diffuser.device).repeat(x_t.
↪shape[0])
        t_pre = t_now-1
        for t in range(diffuser.steps):
            predicted_noise=model(x_t,t_now,condition)
            x_t=DDPM_sample_step(diffuser, x_t,t_now,t_pre,predicted_noise)
            t_now=t_pre
            t_pre=t_pre-1
        return x_t

def DDPM_sample_step(d, x_t, t, t_pre, noise):
    coef1 = 1/d.sqrt_alphas[t]
    coef2 = d.betas[t]/d.sqrt_one_minus_alphas_bar[t]
    sig = torch.sqrt(d.betas[t]) *d.sqrt_one_minus_alphas_bar[t_pre] /d.sqrt_one_
↪minus_alphas_bar[t]
    return coef1*(x_t-coef2*noise) + sig*torch.randn_like(x_t)
```

Note that these snippest closely follow the sampling of the original airfoil paper, e.g. sample.ipynb. Below we compute a single batch of outputs for the diffusion model in `result_diffusion`.

```
def sample_diffusion(diffuser, network,input_field,num_diffusion_sample=100):
    network.eval();network.to(device);predictions=[]
    batch_size=25;N_all=num_diffusion_sample
    while N_all>0:
        batch_size_now=min(batch_size,N_all)
        N_all-=batch_size
        prediction_batch=normalized2dimless(
            diffusion_sample_from_noise(diffuser, network,
                                        input_field.to(device).repeat(batch_size_now,1,
↪1,1) ))
        predictions.append(prediction_batch.detach().cpu().numpy())
    predictions=np.concatenate(predictions,axis=0)
    return np.mean(predictions,axis=0),np.std(predictions,axis=0),predictions

mean,std,samples_diffusion=sample_diffusion(diffusion_trainer.diffuser,dif_network,
↪input_field)
result_diffusion=np.concatenate([mean,std],axis=0)
```

Let's first do some qualitative comparisons first by plotting the mean and standard deviation fields of a single test case.

```
result_ground_truth=np.concatenate([mean_field_test_gd[index],std_field_test_
↪gd[index]],axis=0)

CHANNEL_NAME_MEAN=[r"$\mu_p^*$",r"$\mu_{u_x^*}$",r"$\mu_{u_y^*}$"]
CHANNEL_NAME_STD=[r"$\sigma_{p^*}$",r"$\sigma_{u_x^*}$",r"$\sigma_{u_y^*}$"]

from airfoil_diffusion.plotter import *
show_each_channel([result_ground_truth,result_diffusion]+result_fms,
                  channel_names=CHANNEL_NAME_MEAN+CHANNEL_NAME_STD,
                  case_names=["Ground truth","Diffusion 200"]+titles,transpose=True,
↪inverse_y=True)
```



The code above produced 100 samples with each method, and the image of the previous cell shows the mean and standard deviation for each spatial point in the regular grid. This illustrates that the mean is relatively easy to get right for all methods. The corresponding fields don't vary too much, and even the 1-step FM variant gets this mostly right (with a bit of noise).

The standard deviation across the samples is more difficult: 1-step FM completely fails here, but, e.g., the 20-step FM version already does very well. This version only uses one tenth of steps compared to the DDPM version. The latter uses 200 steps, so the FM version is effectively 10x faster, while yielding a comparable accuracy here.

26.9 Quantified Results

So far, we've focused on a single test case, and this could have been a "lucky" one for FM. Hence, below we'll repeat the evaluation for different cases across different Reynolds numbers to obtain quantified results. In total, the test set has six different Reynolds numbers, the middle four being interpolations of the training parameters, the first and last being extrapolations.

The `do_test()` helper function defined in the next cell directly computes the statistics for a given network over the whole test set.

```
def do_test(sample_func):
    mean_predictions=[]
```

(continues on next page)

(continued from previous page)

```
    std_predictions=[]
    std_a_predictions=[]
    for input_field in tqdm(inputs_test):
        mean_fields,std_fields,_=sample_func(input_field.unsqueeze(0))
        mean_predictions.append(mean_fields)
        std_predictions.append(std_fields)
        std_a_predictions.append(np.mean(std_fields))
    return mean_predictions,std_predictions,std_a_predictions
```

Because of the larger number of test cases, the following cells can take a bit longer, especially for the diffusion model with its 200 steps.

```
std_a_fms=[]
labels=[]
for step in [1,5,20,100]:
    _,_,std_a_fm_i = do_test(lambda x:sample_flowmatching( network, x, dt=1/step, num_
 ↪sample=500))
    std_a_fms.append(std_a_fm_i)
    labels.append("Flow Matching {}".format(step))
```

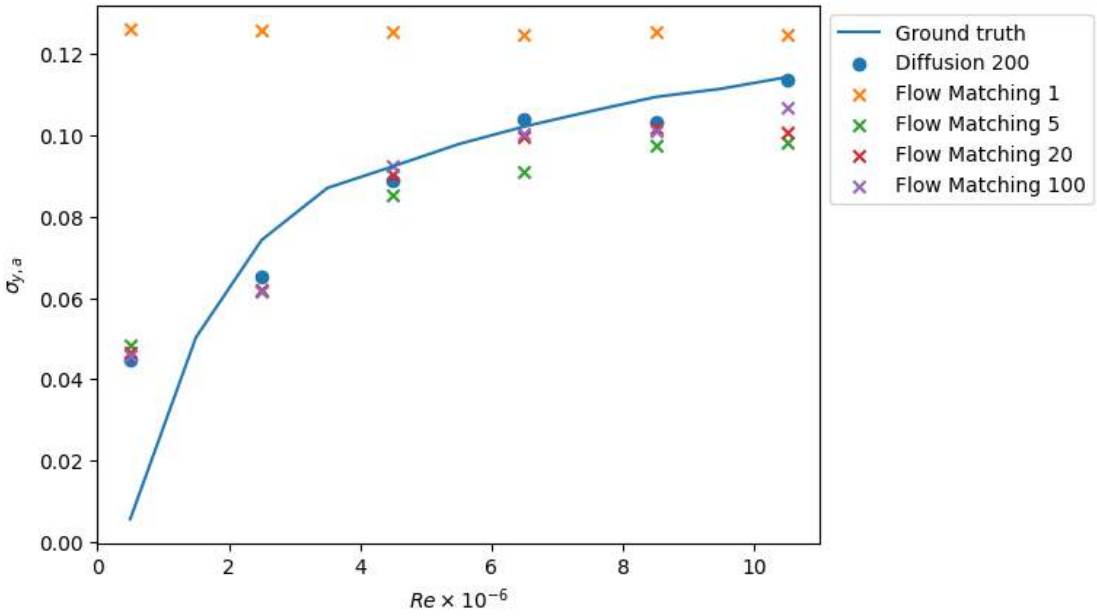
```
100%|██████████| 6/6 [00:01<00:00,  4.35it/s]
100%|██████████| 6/6 [00:06<00:00,  1.08s/it]
100%|██████████| 6/6 [00:25<00:00,  4.28s/it]
100%|██████████| 6/6 [02:09<00:00, 21.59s/it]
```

```
_,_,std_a_predictions_dif = do_test(lambda x:sample_diffusion(diffusion_trainer.
 ↪diffuser, dif_network, x, num_diffusion_sample=500))
```

```
100%|██████████| 6/6 [03:40<00:00, 36.68s/it]
```

Now we have all the numbers, and the next cell produces a graph plotting the ground truth standard deviation per Re (blue line) next to the esimates from the different trained NNs.

```
x=[0.5 +2*i for i in range(6)]
plt.plot([0.5 +i for i in range(11)],std_value_gd,label="Ground truth")
plt.scatter(x,std_a_predictions_dif,label="Diffusion 200",marker="o")
for i in range(len(std_a_fms)):
    plt.scatter(x,std_a_fms[i],label=labels[i],marker="x")
plt.legend(bbox_to_anchor=(1.0,1.0))
plt.xlabel(r"$Re \times 10^{-6}$")
plt.ylabel(r"$\sigma_{y,a}$")
plt.show()
```



The graph shows that the single result visualized above was not an outlier: 1-step FM doesn't do well, but 5- or 20-step FM are already very good, and largely on-par with the DDPM variant. In general, the accuracy of the NNs is very good. Even the last and first dots, the extrapolation regions, are captured reasonably well. The networks over-estimate the variance for the low Re cases because they haven't really seen static cases in training data, but the potentially more difficult high-Re case on the right is handled very well.

Overall, this setup is a non-trivial case: the networks had to learn the posterior distributions from almost constant to strongly varying across the differen Reynolds numbers. The *flow matching* NNs really excel here: they yield excellent estimates and posterior samples with a very small number of integration steps. This is important for practical applications: these surrogate models have to compete with classical numerical simulations, and many fields have highly optimized simulation codes. Hence, it's important the trained NNs provide estimates quickly, with reasonable hardware requirements (i.e. not overly large parameter counts). These results indicate that *flow matching* is a highly interesting contender for probabilistic simulations.

26.10 Next steps

- For this setup it is interesting to try higher order integration methods. Can you observe any gains over Euler? (Make sure to count all NFEs within the integrator, e.g., the four NN calls of the RK4 method.)
- Improve the overall accuracy of the trained models by increasing the number of epochs, and the feature count of the U-net architecture.
- The implementation above uses *basic* denoising and flow matching. It's worth trying improvements, e.g., additional rectification steps from the paper by Liu et al. This could potentially reduce the number of required steps even further.

CHAPTER

TWENTYSEVEN

INCORPORATING PHYSICAL CONSTRAINTS

Despite the powerful capabilities of diffusion- and flow-based networks for generative modeling that we discussed in the previous sections, there is no direct feedback loop between the network, the observation and the sample at training time. This means there is no direct mechanism to include **physics-based constraints** such as priors from PDEs. As a consequence, it's very difficult to produce highly accurate samples based on learning alone: For scientific applications, we often want to make sure the errors go down to any chosen threshold.

In this chapter, we will outline strategies to remedy this shortcoming, and building on the content of previous chapters, the central goal of both methods is to get **differentiable simulations** back into the training and inference loop. The previous chapters have shown that they're very capable tools, so the main question is how to best employ them in the context of diffusion modeling.

Note

Below we'll focus on the inverse problem setting from *Introduction to Probabilistic Learning*. I.e., we have a system $y = f(x)$ (with numerical simulator $y = \mathcal{P}(x)$) and given an observation y, we'd like to obtain the posterior distribution for the distributional solution $x \sim p(x|y)$ of the inverse problem.

27.1 Guiding Diffusion Models

Having access to a physical model with a differentiable simulation $\mathcal{P}(x) = y$ means we can obtain gradients ∇_x through the simulation. As before, we aim for solving *inverse* problems where, given an output y we'd like to sample from the conditional posterior distribution $p(x|y)$ to obtain samples x that explain y. The previous chapter demonstrated learning such distributions with diffusion models, and given a physics prior $\mathcal{P}$, there's a first fundamental choice: should be use the gradient at *training time*, i.e., trying to improve the learned distribution p_θ, or at *inference time*, to improve sampling $x \sim p_\theta(x|y)$.

Training with physics priors: The hope of incorporating physics-based signals in the form of gradients at training time would be to improve the state of p_θ after training. While there's a certain hope this could, e.g., compensate for sparse training data, there is little hope for substantially improving the accuracy of the learned distribution. The training process for diffusion and flow matching models typically yields very capable neural networks, that are excellent at producing approximate samples from the posterior. They're typically limited in terms of their accuracy by model and training data size, but it's difficult to fundamentally improve the capabilities of a model at this stage. Rather, in this context it is more interesting to obtain higher accuracies at inference time.

Inference with physics priors: For scientific applications, classic simulations typically yield control knobs that allow for choosing a level of accuracy. E.g., iterative solvers for linear systems provide iteration counts and residual thresholds, and if a solution is not accurate enough, a user can simply reduce the residual threshold to obtain a more accurate output. In contrast, neural networks typically come without such controls, and even the iteration count of denoising or velocity integration (for flow matching) are bounded in terms of final accuracy. More steps typically reduce noise, and correspondingly the error, but will plateau at a level of accuracy given by the capabilities of the trained model. This is

exactly where the gradients of physics solver show promise: they provide an external process that can guide and improve the output of a diffusion model. As we'll show below, this makes it possible to push the levels of accuracy beyond those of pure learning, and can yield inverse problem solvers that really outperform traditional solvers.

Recall that for denoising, we train a noise estimator ϵ_θ, and at inference time iterate denoising steps of the form $x_{\text{new}} = x - \hat{\alpha}_t \epsilon_\theta(x,t) + \hat{\sigma}_t \mathcal{N}(0,I)$, where $\hat{\alpha}, \hat{\sigma}$ denote the merged scaling factors for both terms. The most straight-forward approach for including gradients is to additionally include a step in the direction of the gradient $\nabla_x ||\mathcal{P}(x) - y||_2$. For simplicity, we take an L^2 distance towards the observation y here. This was shown to direct sampling even when the posterior is not conditional, i.e., if we only have access to $x \sim p_\theta(x)$, and is known as *diffusion posterior sampling* [CKM+23].

While this approach manages to includes $\mathcal{P}$, there are two challenges: x is typically noisy, and the gradient step can distort the distributional sampling of the denoising process. The first point is handled quite easily with an *extrapolation step* (more details below), while the second one is more difficult to address: the gradient descent steps via $\nabla_x \mathcal{P}$ are akin to a classic optimization for the inverse problem and could strongly distort the outputs of the diffusion model. E.g., in the worst case they could pull the different points of the posterior distribution towards a single case favored by the simulator $\mathcal{P}$. Hence, the following paragraphs will outline a strategy that merges simulator and learning, while preserving the distribution of the posterior. We'll focus on flow matching as a state-of-the-art approach next, and afterwards discuss variant that treats the diffusion steps themselves as a physical process.

27.2 Physics-Guided Flow Matching

To reintroduce control signals using simulators into the flow matching algorithm we'll follow [HT23]. The goal is to transform an existing pretrained flow-based network, as outlined in *Introduction to Probabilistic Learning*, with a flexible control signal by aggregating the learned flow and control signals into a *controlled flow*. This is the task of a second neural network, the *control network*, in order to make sure that the posterior distribution is not negatively affected by the signals from the simulator. This second network is small compared to the pretrained flow network, and freezing the weights of the pretrained network works very well; thus, the refinement for control needs only a fairly small amount of additional parameters and computing resources.

The control signals can be based on gradients and a cost function, if the simulator is differentiable, but they can also be learned directly from the simulator output. Below, we'll show that performance gains due to simulator feedback are substantial and cannot be achieved by training on larger datasets alone. Specifically, we'll show that flow matching with simulator feedback is competitive with MCMC baselines for a problem from gravitational lensing in terms of accuracy, and it beats them significantly regarding inference time. This indicates that it provides a very attractive tool for practical applications.

Controlled flow v_θ^C First, it's a good idea to pretrain a regular, conditional flow network $v_\theta(x,y,t)$ without any control signals to make sure that we can realize the best achievable performance possible based on learning alone.

Then, in a second training phase, a control network $v_\theta^C(v,c,t)$ is introduced. It receives the pretrained flow v and control signal c as input. Based on these additional inputs, it can used, e.g., the gradient of a PDE to produce an improved flow matching velocity. At inference time, we integrate $dx/dt = v_\theta^C(v,c,t)$ just like before, only now this means evaluating $v_\theta(x,y,t)$ and then c beforehand. (We'll focus on the details of c in a moment.)

First, the control network is much smaller in size than the regular flow network, making up ca. 10% of the weights θ. The network weights of v_θ can be frozen, to train with the conditional flow matching loss (25.2) for a small number of additional steps. This reduces training time and compute since we do not need to backpropagate gradients through $v_\theta(x,y,t)$. Freezing the weights of v_θ typically does not negatively affects the performance, although a joint end-to-end training could provide some additional improvements.

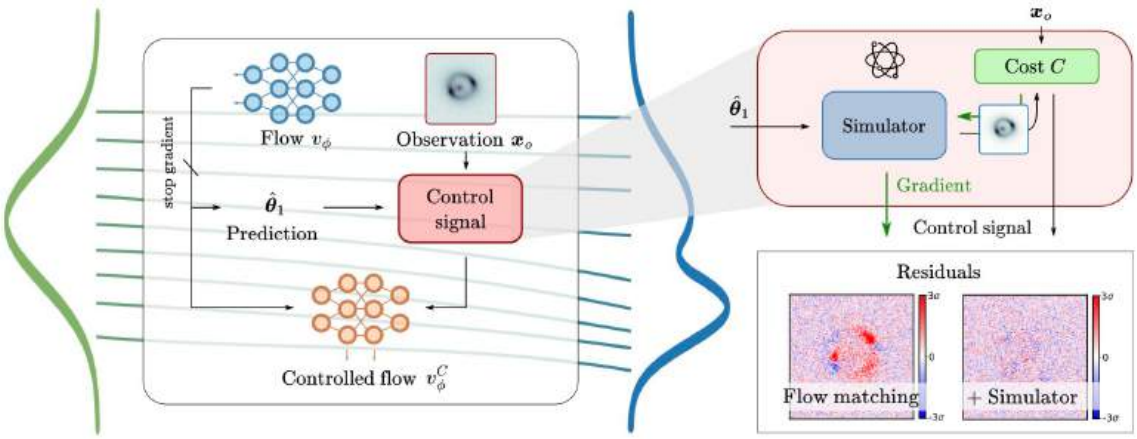


Fig. 27.1: An overview of the control framework. We will consider a pretrained flow network v_θ and use the predicted flow for the trajectory point x_t at time t to estimate $\hat{x}_1$. On the right, we show a gradient-based control signal with a differentiable simulator and cost function C for improving $\hat{x}_1$. An additional network learns to combine the predicted flow with feedback via the control signal to give a new controlled flow. By combining learning-based updates with suitable controls, we avoid local optima and obtain high-accuracy samples with low inference times.

1-step prediction The conditional flow matching networks $v_\theta(x, y, t)$ from *Introduction to Probabilistic Learning* gradually transform samples from p_0 to p_1 during inference via integrating the simple ODE $dx_t/dt = v_\theta(x_t, y, t)$ step by step. There is no direct feedback loop between the current point on the trajectory x_t, the observation y, and a physical model that we could bring into the picture. An important first issue is that the current trajectory point x_t is often not be close to a good estimate of a posterior sample x_1. This is especially severe at the beginning of inference, where x_0 is drawn from the source distribution (typically a Gaussian), and hence x_t will be very noisy. Most simulators really don't like very noisy inputs, and trying to compute gradients on top of it is clearly a very bad idea.

This issue is alleviated by extrapolating x_t forward in time to obtain an estimated $\hat{x}_1$

$$\hat{x}_1 = x_t + (1 - t)v_\theta(x_t, y, t). \tag{27.1}$$

and then performing subsequent operations for control and guidance on $\hat{x}_1$ instead of the current, potentially noisy x_1.

Note that this 1-step prediction is also conceptually related to diffusion sampling using *likelihood-guidance*. For inference in diffusion models, where sampling is based on the conditional score $\nabla_{x_t} \log p(x_t|y)$ and can be decomposed into

$$\nabla_{x_t} \log p(x_t|y) = \nabla_{x_t} \log p(x_t) + \nabla_{x_t} \log p(y|x_t). \tag{27.1}$$

The first expression can be estimated using a pretrained diffusion network, whereas the latter is usually intractable, but can be approximated using $p(y|x_t) \approx p_{y|x_0}(y|\hat{x}(x_t))$, where the denoising estimate $\hat{x}(x_t) = \mathbb{E}_q[x_0|x_t]$ is usually obtained via Tweedie's formula $(\mathbb{E}_q[x_0|x_t] - x_t)/t\sigma^2$. In practice, the estimate $\hat{x}(x_t)$ is very poor when x_t is still noisy, making inference difficult in the early stages. In contrast, flows based on linear conditional transportation paths have empirically been shown to have trajectories with less curvature compared to, for example, denoising-based networks. This property of flow matching enables inference in fewer steps and providing better estimates for $\hat{x}_1$.

27.2.1 Physics-based Controls

Now we focus on the content of the control signal c that was already used above. We extend the idea of self-conditioning via physics-based control signals to include an additional feedback loop between the network output and an underlying physics-based prior. We'll distinguish between two types of controls in the following: a gradient-based control from a differentiable simulator, and one from a learned estimator network.

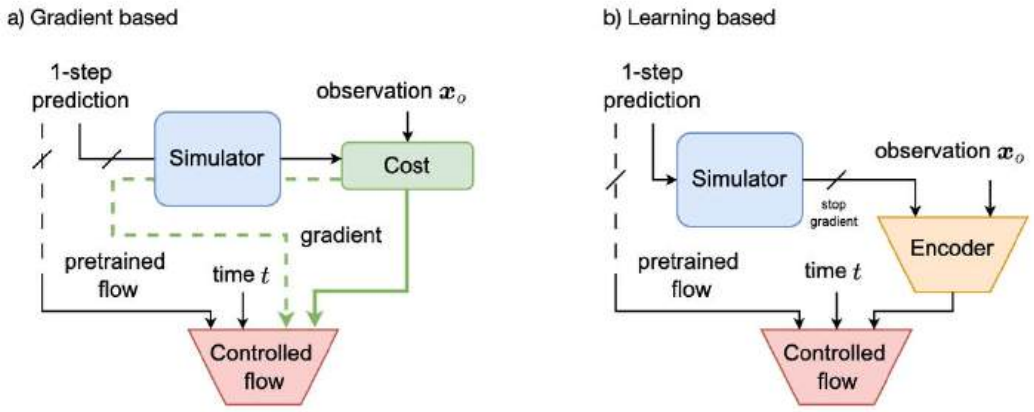


Fig. 27.2: Types of control signals. (a) From a differentiable simulator, and (b) from a learned encoder.

Gradient-based control signal In the first case, we make use of a differentiable simulator $\mathcal{P}$ to construct a cost function C. Naturally, C will likewise be differentiable such that we can compute a gradient for a predicted solution. Also, we will rely on the stochasticity of diffusion/flow matching, and as such the simulator can be deterministic.

Given an observation y and the estimated 1-step prediction $\hat{x}_1$, the control signal computes to how well $\hat{x}_1$ explains y via the cost function C. Good choices for the cost are, e.g., an L^2 loss or a likelihood $p(y|\hat{x}_1)$. We define the control signal c to consist of two components: the cost itself, and the gradient w.r.t. the cost function:

$$c(\hat{x}_1, y) := [C(\mathcal{P}(\hat{x}_1), y); \nabla_{\hat{x}_1} C(\mathcal{P}(\hat{x}_1), y)]. \tag{27.2}$$

As this information is passed to a network, the network can freely make use of the current distance to the target (the value of C) and the direction towards lowering it in the form of $\nabla_{\hat{x}_1} C$.

Learning-based control signal When the simulator is non-differentiable, the second variant of using a learned estimator comes in handy. To combine the simulator output with the observation y, a learnable encoder network *Enc* with parameters θ_E can be introduced to judge the similarity of the simulation and the observation. The output of the encoder is small and of size $O(\dim(x))$. The control signal is then defined as

$$c(\hat{x}_1, y) := Enc(\mathcal{P}(\hat{x}_1), y). \tag{27.3}$$

The gradient backpropagation is stopped at the output of the simulator $\mathcal{P}$, as shown in figure 27.2. Before showing some examples of the capabilities of these two types of control, we'll discuss some of their properties.

27.2.2 Additional Considerations

Stochastic simulators Many Bayesian inference problems have a stochastic simulator. For simplicity, we assume that all stochasticity within such a simulator can be controlled via a variable $z \sim \mathcal{N}(0, I)$, which is an additional input. Motivated by the equivalence of exchanging expectation and gradient

$$\nabla_{\hat{x}_1} \mathbb{E}_{z\sim\mathcal{N}(0,1)}[C(\mathcal{P}_z(\hat{x}_1), y)] = \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\nabla_{\hat{x}_1} C(\mathcal{P}_z(\hat{x}_1), y)], \tag{27.4}$$

when calling the simulator, we draw a random realization of z. During training, we randomly draw z for each sample and step while during inference we keep the value of z fixed for each trajectory.

Time-dependence

If the estimate $\hat{x}_1$ is bad and the corresponding cost $C(\hat{x}_1, y)$ is high, gradients and control signals can become unreliable. It turns out that the estimates $\hat{x}_1$ become more reliable for later times in the flow matching process.

In practice, $t \geq 0.8$ is a good threshold. Therefore, we only train the control network v_θ^C in this range, which allows for focusing on control signals containing more useful information to, e.g. fine tune the solutions with the accurate gradients of a differentiable simulator. For $t < 0.8$, we directly output the pretrained flow $v_\theta(t, x, y)$.

Theoretical correctness

In the formulation above, the approximation $\hat{x}_1$ only influences the control signal, which is an input to the controlled flow network v_θ^C. In the case of a deterministic simulator, this makes the control signal a function of x_t. The controlled flow network is trained with the same loss as vanilla flow matching. This has the nice consequence that the theoretical properties are preserved. This is in contrast to e.g. "likelihood-based guidance", which uses an approximation for $\nabla_{x_t} \log p(y|x_t)$ as a guidance term during inference, which is not covered by the original flow matching theory.

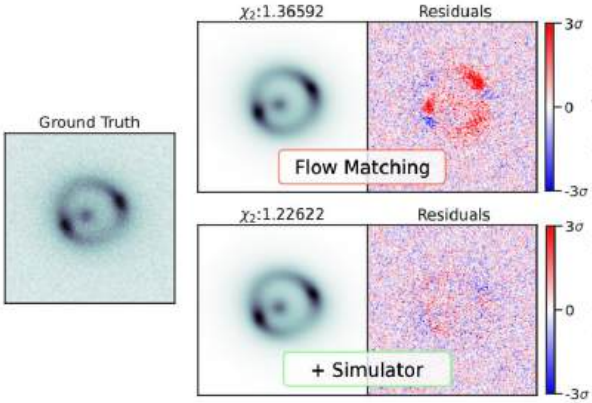
27.2.3 An Example from Astrophysics

To demonstrate how these guidance from a physics solver affect the accuracy of samples and the posterior, we show an example from strong gravitational lensing: an inverse problem in astrophysics that is challenging and requires precise posteriors for accurate modeling of observations. In galaxy-scale strong lenses, light from a source galaxy is deflected by the gravitational potential of a galaxy between the source and observer, causing multiple images of the source to be seen. Traditional computational approaches require several minutes to many hours or days to model a single lens system. Therefore, there is an urgent need to reduce the compute and inference with learning-based methods. In this experiment, it's shown that using flow matching and the control signals with feedback from a simulator gives posterior distributions for lens modeling that are competitive with the posteriors obtained by MCMC-based methods. At the same time, they are much faster at inference.

The image aboves shows an example reconstruction and the residual errors. While flow matching and the physics-based variant are both very accurate (it's hard to visually make out differences), the FM version is just on par with classic inverse solvers. The version with the simulator, however, provides a substantial boost in terms of accuracy that is very difficult to achieve even for classic solvers. The quantitative results are shown in the table on the right: the best classic baseline is AIES with an average χ_2 statistic of 1.74, while FM with simulator yields 1.48. Provided that the best possible result due to noisy observations is 1.17 for this scenario, the FM+simulation version is really highly accurate.

At the same time, the performance numbers for *modeling time* in the right column show that the FM variant clearly outperforms the classic solvers. While the simulator increases inference time compared to only the neural network (10s to 19s), the classic baselines require more than $50\times$ longer reconstruction times. Interestingly, this example also highlights the problems of "simpler" physics combinations in the form of DPS. The DPS version does not manage to keep up with the classic solvers in terms of accuracy. To conclude, the *FM+simulator* variant is not only substantially more accurate, but also ca. $35\times$ faster than the best classic solver above (AIES). (Source code for this approach will be available soon in this repository.)

A summary of the physics-based flow matching is given by the following bullet points:



Method	Avg. $\chi_2 \downarrow$	Modeling Time ↓
NUTS	1.83	∼ 56x (564s)
AIES	1.74	∼ 67x (672s)
DPS	9.98	∼ 42x (427s)
Flow-Mat.	1.83	1x (**10s**)
+ Simulator	**1.48**	∼ 2x (19s)

Fig. 27.3: Results from flow matching for reconstructing gravitational lenses. Left: flow matching with a differentiable simulator (bottom) clearly outperforms pure flow matching (top). Right: comparisons against classic baselines. The FM+simulator variant is more accurate while being faster.

✓ Pro:

- Improved accuracy over purely learned diffusion models
- Gives control over residual accuracy
- Reduced runtime compared to traditional inverse solvers

✗ Con:

- Requires differentiable physical process
- Increased computational resources

27.3 Score Matching with Differentiable Physics

So far we have treated the *diffusion time* of denoising and flow matching as a process that is purely virtual and orthogonal to the time of the physical process to be represented by the forward and inverse problems. This is the most generic viewpoint, and works nicely, as demonstrated above. However, it's interesting to think about the alternative: merging the two processes, i.e., treating the diffusion process as an inherent component of the physics system.

The following sections will explain such a combined approach, following the paper "Solving Inverse Physics Problems with Score Matching" [HVT23], which which code is available in this repository.

This approach solves inverse physics problems by leveraging the ideas of score matching. The system's current state is moved backward in time step by step by combining an approximate inverse physics simulator and a learned correction function. A central insight of this work is that training the learned correction with a single-step loss is equivalent to a score matching objective, while recursively predicting longer parts of the trajectory during training relates to maximum likelihood training of a corresponding probability flow. The resulting inverse solver exhibits good accuracy and temporal stability. In line with diffusion modeling and in contrast to classic learned solvers, it allows for sampling the posterior of the solutions. The method will be called *SMDP* (for *Score Matching with Differentiable Physics*) in the following.

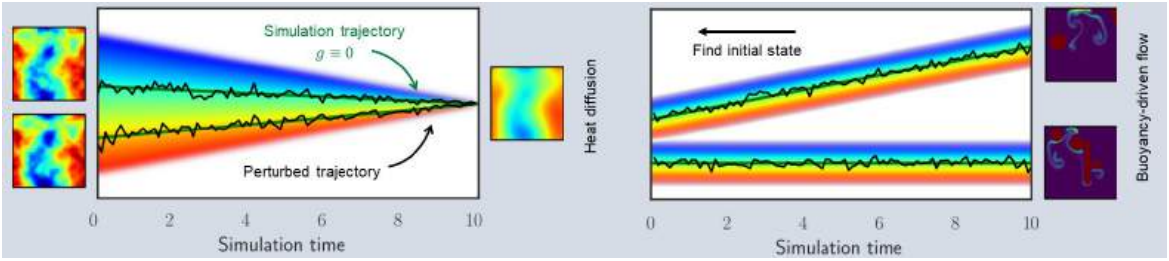


Fig. 27.4: The physics process (heat diffusion as an example, left) perturbs and "destroys" the initial state. At inference time (right, Buoyancy flow as an example), the solver is used to compute inverse steps and produce solutions by combining steps along the score and the gradient of the solver.

27.3.1 Training and Inference with SMDP

For training, SMDP fits a neural ODE, the probability flow, to the set of perturbed training trajectories. The probability flow is comprised of an approximate reverse physics simulator $\tilde{\mathcal{P}}^{-1}$ as well as a correction function s_θ. For inference, we simulate the system backward in time from $\mathbf{x}_T$ to $\mathbf{x}_0$ by combining $\tilde{\mathcal{P}}^{-1}$, the trained s_θ and Gaussian noise in each step. For optimizing s_θ, our approach moves a sliding window of size S along the training trajectories and reconstructs the current window. Gradients for θ are accumulated and backpropagated through all prediction steps. This process is illustrated in the following figure:

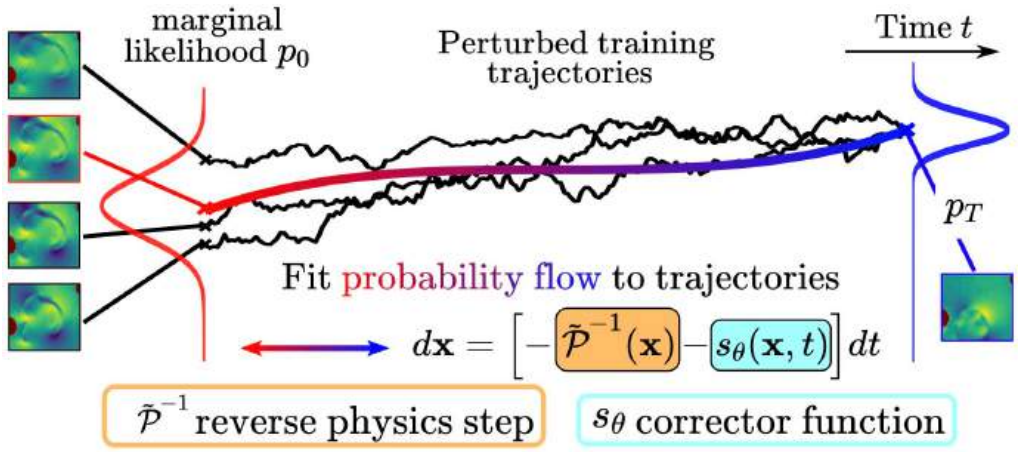


Fig. 27.5: Overview of the score matching training process while incorporating a physics solver $\mathcal{P}$ and it's approximate inverse solver $\matcal{P}^{-1}.

A differentiable solver or a learned surrogate model is employed for $\tilde{\mathcal{P}}^{-1}$. The neural network $s_\theta(\mathbf{x}, t)$ parameterized by θ is trained such that

$$\mathbf{x}_m \approx \mathbf{x}_{m+1} + \Delta t \left[\tilde{\mathcal{P}}^{-1}(\mathbf{x}_{m+1}) + s_\theta(\mathbf{x}_{m+1}, t_{m+1}) \right].$$

In this equation, the term $s_\theta(\mathbf{x}_{m+1}, t_{m+1})$ corrects approximation errors and resolves uncertainties from the stochastic forcing $F_{t_m}(z_m)$. Potentially, this process can be unrolled over multiple steps at training time to improve accuracy and stability. At inference, time the stochastic differential equation

$$d\mathbf{x} = \left[-\tilde{\mathcal{P}}^{-1}(\mathbf{x}) + C\, s_\theta(\mathbf{x}, t) \right] dt + g(t) dW$$

is integrated via the Euler-Maruyama method to obtain a solution for the inverse problem. Setting $C = 1$ and excluding

the noise gives the probability flow ODE: a unique, deterministic solution. This deterministic variant is not probablistic anymore, but has other interesting properties.

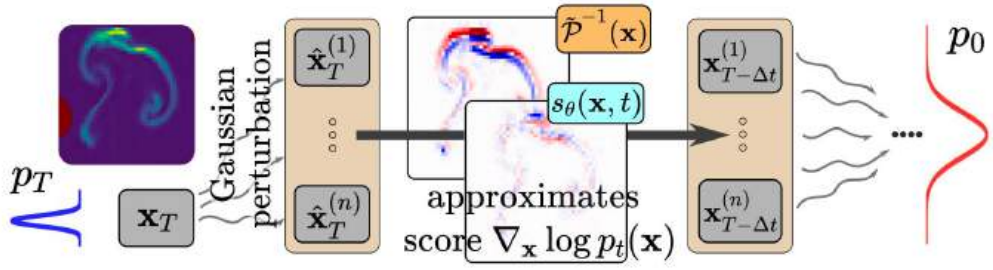


Fig. 27.6: An overview of SMDP at inference time.

27.3.2 SMDP in Action

This section shows experiments for the stochastic heat equation: $\frac{\partial u}{\partial t} = \alpha \Delta u$, which plays a fundamental role in many physical systems. It slightly perturbs the heat diffusion process and includes an additional term $g(t)\ \xi$, where ξ is space-time white noise. For the experiments, we fix the diffusivity constant to $\alpha = 1$ and sample initial conditions at $t = 0$ from Gaussian random fields with $n = 4$ at resolution 32×32. We simulate the heat diffusion with noise from $t = 0$ until $t = 0.2$ using the Euler-Maruyama method and a spectral solver $\mathcal{P}_h$ with a fixed step size and $g \equiv 0.1$. Given a simulation end state $\mathbf{x}_T$, we want to recover a possible initial state $\mathbf{x}_0$.

In this experiment, the forward solver cannot be used to infer $\mathbf{x}_0$ directly since high frequencies due to noise are amplified, leading to physically implausible solutions. Instead, the reverse physics step $\tilde{P}^{-1}$ is implemented by using the forward step of the solver $\mathcal{P}_h(\mathbf{x})$, i.e. $\tilde{\mathcal{P}}^{-1}(\mathbf{x}) \approx -\mathcal{P}_h(\mathbf{x})$.

A small ResNet-like architecture is used based on an encoder and decoder part as representation for the score function $s_\theta(\mathbf{x}, t)$. The spectral solver is implemented via differentiable programming in *JAX*. As baseline methods, a supervised training of the same architecture as $s_\theta(\mathbf{x}, t)$, a Bayesian neural network (BNN), as well as a FNO network are considered. An L_2 loss is used for all these methods, i.e., the training data consists of pairs of initial state $\mathbf{x}_0$ and end state $\mathbf{x}_T$. Additionally, a variant of the SMDP method is included for which the reverse physics step $\tilde{\mathcal{P}}^{-1}$ is reomved, such that the inversion of the dynamics has to be learned entirely by s_θ, denoted by "s_θ~only".

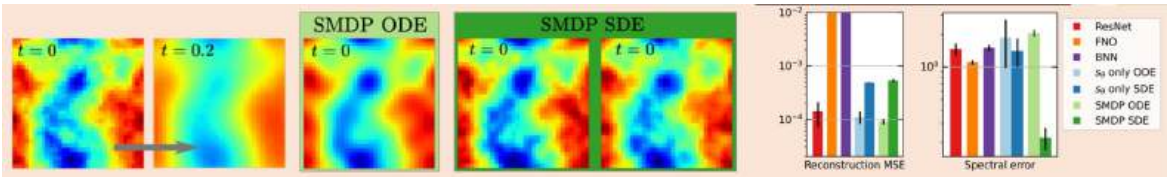


Fig. 27.7: While the ODE trajectories provide smooth solutions with the lowest reconstruction MSE, the SDE solutions synthesize high-frequency content, significantly improving spectral error.
The "s_θ only" version without the reverse physics step exhibits a significantly larger spectral error. Metrics (right) are averaged over three runs.

SMDP and the baselines are evaluated by considering the *reconstruction MSE* on a test set of 500 initial conditions and end states. For the reconstruction MSE, the prediction of the network is simulated forward in time with the solver $\mathcal{P}_h$ to obtain a corresponding end state, which is compared to the ground truth via the L_2 distance. This metric has the disadvantage that it does not measure how well the prediction matches the training data manifold. I.e., for this case, whether the prediction resembles the properties of the initial Gaussian random field. For that reason, the power spectral density of the states is shown as a *spectral loss*. An evaluation and visualization of the reconstructions are given in figure

\ref{fig:stochastic_heat_eq_overview}, which shows that the ODE inference performs best regarding the reconstruction MSE. However, its solutions are smooth and do not contain the necessary small-scale structures. This is reflected in a high spectral error. The SDE variant, on the other hand, performs very well in terms of spectral error and yields visually convincing solutions with only a slight increase in the reconstruction MSE.

This highlights the role of noise as a source of entropy in the inference process for diffusion models, such as the SDE in SMDP, which is essential for synthesizing small-scale structures. Note that there is a natural tradeoff between both metrics, and the ODE and SDE inference perform best for each of the cases while using an identical set of weights. This heat diffusion example highlights the advantages and properties of treating the physical process as part of the diffusion process. This, of course, extends to other physics. E.g., the SMDP repository additionally shows a case with an inverse Navier-Stokes solve.

27.4 Summary of Physics-based Diffusion Models

Overall, the sections above have explained two methods to incorporate physics-based constraints and models in the form of PDEs into diffusion modeling. Interestingly, the inclusion is largely in line with *Introduction to Differentiable Physics*, i.e. gradients of the physics solver are a central quantity, and concepts like unrolling play an important role. On the other hand, the probabilistic modeling introduces additional complexity on the training and inference sides. It provides powerful tools and access to distribiutions of solutions (we haven't even touched follow up applications such as uncertainty quantification above), but this comes at a cost.

As a rule of thumb ?, diffusion modeling should only be used if the solution is a distribution that is *not* well represented by the mean of the solutions. If the mean is accetable, "regular" neural networks offer substantial advantages in terms of reduced complexity for training and inference.

However, if the solutions are a distribution ?, diffusion models are powerful tools to work with complex and varied solutions. Given its capabilties, deep learning with diffusion models arguably introduces surprisingly *little* additional complexity. E.g., training flow matching models is suprisingly robust, can be build on top of deterministic training, and introduces only a mild computational overhead.

To show how the combination of physics solvers and diffusion models turns out in terms of an implementation, the next section shows source code for an SMDP use case.

CHAPTER

TWENTYEIGHT

PROBABILISTIC INVERSE PROBLEM WITH DIFFERENTIABLE SIMULATIONS

This notebook will illustrate some of the concepts introduced in *Introduction to Probabilistic Learning*, such as the training of score functions via log likelihoods, and what they look like in a clear and reduced problem. At the same time, the setup provides integration of a simple *differentiable simulator* to illustrate the concept of physics-based diffusion modeling with the SMDP method from *Incorporating Physical Constraints* (full paper). This approach combines physics and score matching along a merged time dimension to solve inverse problems. [run in colab]

28.1 Toy Problem setup

We'll consider a toy problem with quadratically decaying trajectories. The trajectories start at 1 or -1 and approach 0 as t increases. The corresponding SDE is given by

$$dx = -\left[\lambda_1 \cdot \text{sign}(x)x^2\right] dt + \lambda_2 dw, \tag{28.1}$$

with $\lambda_1 = 7$ and $\lambda_2 = 0.03$. The corresponding reverse-time SDE is

$$dx = -\left[\lambda_1 \cdot \text{sign}(x)x^2 - \lambda_2^2 \cdot \nabla_x \log p_t(x)\right] dt + \lambda_2 dw. \tag{28.1}$$

Throughout the experiments, p_0 is a categorical distribution, where we draw either 1 or -1 with the same probability.

28.2 Implementation Overview

The implementation below comprises the data generation, network definition, training using the sliding window method, visualization of the learned score and inference using the probability flow ODE and the reverse-time SDE. The core algorithm for training and inference can be modified easily for different variants that are discussed in the paper.

This implementation uses quite a few packages, which we'll import first. We'll make use of `JAX` with `haiku` and `optax` as neural network libraries. For data generation and inference, we use `diffrax`:

```
try:
    import google.colab  # only to ensure that we are inside colab
    %pip install diffrax jax jaxlib scipy optax dm-haiku
except ImportError:
    print("This notebook is running locally, please make sure the packages above are␣
↪installed")
    pass
```

```
import warnings
warnings.filterwarnings('ignore')

from diffrax import diffeqsolve, ControlTerm, Euler, MultiTerm, ODETerm, SaveAt,␣
↪VirtualBrownianTree, WeaklyDiagonalControlTerm
import diffrax as dfx

import jax
import jax.random as jr
import jax.numpy as jnp
import optax
import haiku as hk
```

In addition to several widely used numpy and matplotlib libraries:

```
import math
import numpy as np
from tqdm import tqdm
from scipy.interpolate import griddata
from typing import Any, Callable, Iterable, List, Optional, Tuple, Union

from matplotlib import pyplot as plt

from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.gridspec as gridspec
```

28.3 Physical System SDE

Next we set up a lambda function for the SDE above with constants $\lambda = 7$ and $g = 0.03$.

```
g = 0.03
lambda_ = 7
physics_operator = lambda x: - jnp.sign(x) * x * x * lambda_
```

We use diffrax to solve the SDE with Euler steps

```
def r_process(initial_value, noise_scaling, seed):

    initial_shape = (1,)
    y0 = jnp.ones(shape=initial_shape) * initial_value

    t0, t1 = 0.0, 10.0
    drift = lambda t, y, args: physics_operator(y)

    diffusion = lambda t, y, args: noise_scaling * jnp.ones(initial_shape)

    brownian_motion = VirtualBrownianTree(t0, t1, tol=1e-3, shape=initial_shape,␣
↪key=jr.PRNGKey(seed))
    terms = MultiTerm(ODETerm(drift), WeaklyDiagonalControlTerm(diffusion, brownian_
↪motion))
    solver = Euler()
    saveat = SaveAt(dense=True)

    sol = diffeqsolve(terms, solver, t0, t1, dt0=0.01, y0=y0, saveat=saveat)
```

(continues on next page)

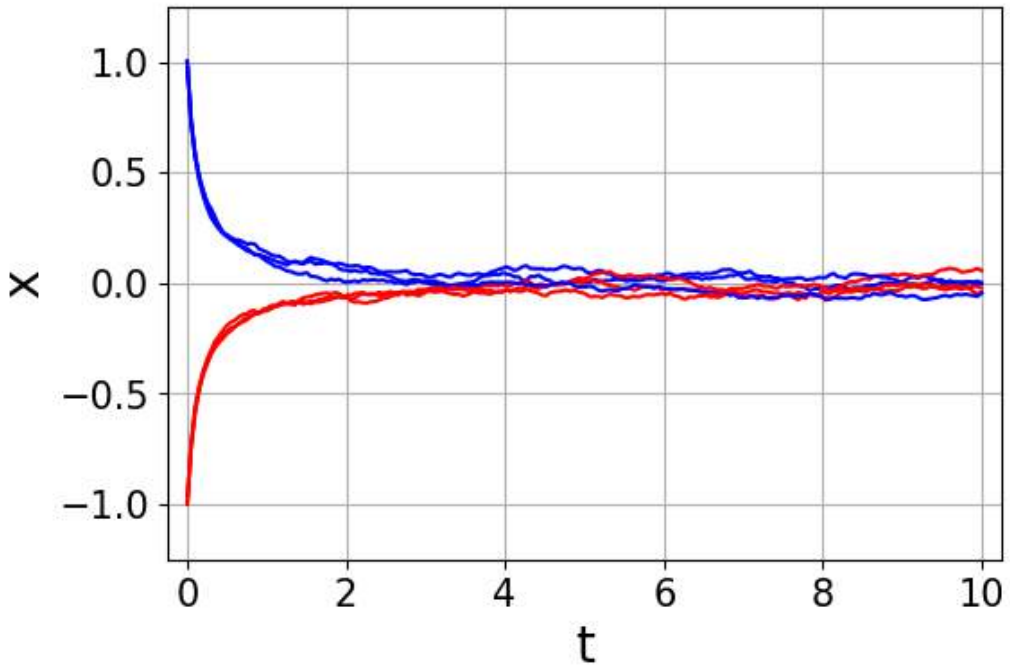
(continued from previous page)

```
    return sol
```

28.4 Visualization of SDE paths

We first set a *seed* for the SDE paths, and plot some example paths from the SDE.

```
seed = 2022

if 1:
    # paths starting in 1.0
    value_one = [r_process(1.0, g, seed+n) for n in range(0, 3)]

    # paths starting in -1.0
    value_minus_one = [r_process(-1.0, g, seed+n) for n in range(3, 6)]

    fig, ax = plt.subplots(figsize=(6,4))

    x = jnp.linspace(0,10,200)

    for i in range(3):

        sol = value_one[i]
        ax.plot(x, jnp.diag(sol.evaluate(x)), color='blue')

        sol = value_minus_one[i]
        ax.plot(x, jnp.diag(sol.evaluate(x)), color='red')

    ax.tick_params(axis='x', colors='black')
    ax.tick_params(axis='y', colors='black')
    ax.set_xlabel('t', size = 20)
    ax.set_ylabel('x', size = 20)
    ax.grid(True)
    ax.tick_params(labelsize=15)

    plt.xlim([-0.25, 10.25])
    plt.ylim([-1.25,1.25])

    plt.show()
```



As promised, the trajectories start on either side (blue and red), and approach the zero line in a stochastic fashion.

28.5 Generate Training Data

We generate a data set of 250 paths from $t = 0$ until $t = 10$ with $\Delta t = 0.02$

```
import os.path
import pickle

if os.path.isfile('toy_example_data.p'):

    with open('toy_example_data.p', 'rb') as file:

        dataset = pickle.load(file)

else:

    dataset = []
    x = jnp.linspace(0,10,500)

    for n in tqdm(range(250)):
        sol = r_process((-1) ** n, g, seed+n)
        dataset.append(jnp.diag(sol.evaluate(x)))

data = jnp.array(dataset)
```

```
100%|██████████| 250/250 [04:39<00:00,  1.12s/it]
```

```
import pickle
with open('toy_example_data.p', 'wb') as file:
    pickle.dump(dataset, file)
```

Next, we implement iterator functions for the dataset

```
def _prepare_batch(batch):

    batch_x = batch[:,::-1]
    batch_t = jnp.linspace(0,10, batch_x.shape[1])[::-1]

    return (batch_x, batch_t)

def iterbatches(X, batch_size, shuffle=False):


    def iterate(X, batch_size, shuffle=False):
        n_samples = X.shape[0]

        ids = np.arange(n_samples)
        sample_perm = np.arange(n_samples)
        if batch_size is None:
            batch_size = n_samples
        if shuffle:
            sample_perm = np.random.permutation(n_samples)
        batch_idx = 0
        num_batches = math.ceil(n_samples / batch_size)
        while batch_idx < num_batches:
            start = batch_idx * batch_size
            end = min(n_samples, (batch_idx + 1) * batch_size)
            indices = range(start, end)

            perm_indices = sample_perm[indices]
            X_batch = X[perm_indices]
            ids_batch = ids[perm_indices]

            batch_idx += 1
            yield X_batch

    return iterate(X, batch_size, shuffle)
```

Additionally, we implement iterator functions required for the score field visualization

```
def _prepare_batch_grid(batch):

    inputs = [np.split(x, x.shape[1], 1) for x in batch]

    return inputs

def iterbatches_grid(X, batch_size, shuffle=False):

    def iterate(X, batch_size, shuffle=False):
        n_samples = X.shape[0]

        sample_perm = np.arange(n_samples)
        if batch_size is None:
            batch_size = n_samples
```

(continues on next page)

(continued from previous page)

```
        if shuffle:
            sample_perm = np.random.permutation(n_samples)
        batch_idx = 0
        num_batches = math.ceil(n_samples / batch_size)
        while batch_idx < num_batches:
            start = batch_idx * batch_size
            end = min(n_samples, (batch_idx + 1) * batch_size)
            indices = range(start, end)

            perm_indices = sample_perm[indices]
            X_batch = X[perm_indices]

            batch_idx += 1
            yield ([X_batch])

    return iterate(X, batch_size, shuffle)
```

28.5.1 Neural Network Setup

This example employs a neural network $s_\theta(x, t)$ parameterized by θ to approximate the score. It's a simple multilayer perceptron with elu activations and five hidden layers with 30, 30, 25, 20, and then 10 neurons for the last hidden layer.

The neural network architecture is realized with `haiku`, using a stack of MLPs:

```
EPSILON = 1e-5

def f(x, t):
    t = jnp.log(t + EPSILON)
    x = jnp.hstack([x, t])
    net = hk.nets.MLP(output_sizes = [30, 30, 25, 20, 10, 1],
                      activation = jax.nn.elu)
    return net(x)
```

The next cell initializes the parameters. The `forward_fn` will be our main handle to evaluate the network `f` later on.

```
init_params, forward_fn = hk.transform(f)
rng = jax.random.PRNGKey(0)

x_init = jnp.ones((10,1))
t_init = x_init
params = init_params(rng, x_init, t_init)
```

We also define a function to evaluate the model and a function that implements the backpropagation and updating of parameters given the optimizer and model loss.

```
def create_eval_fn(forward_fn, params):
    @jax.jit
    def eval_model(t, x, rng=None):

        res = forward_fn(params, rng, x, t)
        return res
    return eval_model

def create_default_update_fn(optimizer: optax.GradientTransformation,
                             model_loss: Callable):
```

(continues on next page)

(continued from previous page)

```
    @jax.jit
    def update(params, opt_state, batch, rng) -> Tuple[hk.Params, optax.OptState, jnp.
↪ndarray]:

        batch_loss, grads = jax.value_and_grad(model_loss)(params, rng, *batch)
        updates, opt_state = optimizer.update(grads, opt_state)
        new_params = optax.apply_updates(params, updates)
        return new_params, opt_state, batch_loss
    return update
```

Finally, we implement the ODE solver for the probability flow ODE using Euler steps. Given an initial state and ground truth (**x_train**) as well as time discretization (**t_train**), it computes the L2 loss:

```
def gradient_fn(forward_fn, physics_operator, g):

    @jax.jit
    def model_loss(model_weights, rng, x_train, t_train):
        x = x_train[:,0]
        i = 1
        loss = 0.0

        for t1, t0 in zip(t_train, t_train[1:]):
            delta_t = t1-t0
            physics_update = physics_operator(x)

            # note that we absorb g**2 (constant) in the definition of forward_fn here
            score_update = - 0.5 * forward_fn(model_weights, rng, jnp.expand_dims(x,␣
↪axis=1),
                                              jnp.repeat(jnp.tile(t1, 1)[None], x.
↪shape[0], axis=0))[:,0]

            x = x - delta_t * (physics_update + score_update)
            x_true = x_train[:,i]
            loss += jnp.mean(jnp.square(x - x_true))
            i += 1

        return loss
    return model_loss
```

28.5.2 Visualizing the Score Field

While the problem itself is very clear, it's actually much less obvious what the score function underneath should look like. Hence, we'll show different version of score functions below, and the next cell implements a helper function that takes care of the visualization of the score field. We'll sample the score function on a regular grid to show it as an image, and hence first set up the corresponding grids.

```
x_ = np.linspace(0., 10., 400)
y_ = np.linspace(-1.25, 1.25, 200)

X, Y = np.meshgrid(x_,y_)
full_domain = np.hstack((X.flatten()[:,None], Y.flatten()[:,None]))
meshgrid = (X,Y)
```

The following function `save_snapshot_score` plots a score function as an image.

```
dpi = 200
height = 6
width = 4
scaling_factor = 1.

def save_snapshot_score(params, forward_fn, meshgrid, step=None, savename=None):
    eval_fn = create_eval_fn(forward_fn, params)
    results = None
    X, Y = meshgrid
    full_domain = np.hstack((X.flatten()[:,None], Y.flatten()[:,None]))
    full_domain_scaled = full_domain * scaling_factor

    generator = iterbatches_grid(full_domain_scaled, 1000, shuffle=False)
    for batch in generator:
        inputs = _prepare_batch_grid(batch)

        output_values = eval_fn(*inputs[0], rng)
        if isinstance(output_values, jnp.ndarray):
            output_values = [output_values]
        output_values = [jax.device_get(t) for t in output_values]

        if results is None:
            results = [[] for i in range(len(output_values))]
        for i, t in enumerate(output_values):
            results[i].append(t)

    final_results = []
    if results is not None:
        for r in results:
            final_results.append(np.concatenate(r, axis=0))

    x_scaled = x_ * scaling_factor
    y_scaled = y_ * scaling_factor
    meshgrid_scaled = (meshgrid[0] * scaling_factor, meshgrid[1] * scaling_factor)

    fig, axes = plt.subplots(nrows = 1, ncols=1, figsize=(height,width))
    fig.set_dpi(dpi)

    ax = axes
    u_pred = final_results[0][:].flatten()
    U_pred = griddata(full_domain_scaled, u_pred.flatten(), meshgrid_scaled, method=
↪'cubic')

    vmax = 75
    h = ax.imshow(jnp.flip(U_pred / (g**2), axis=0), cmap='jet',
                  extent=[ x_scaled.min(), x_scaled.max(), y_scaled.min(), y_scaled.
↪max()],
                  aspect='auto', vmin = -vmax, vmax = vmax)
    cbar = fig.colorbar(h)
    ax.set_xlabel("t")
    ax.set_ylabel("x")

    plt.figure(figsize=(10, 6)) # NT_DEBUG , test
    if savename:
        plt.savefig(f'{savename}.svg', transparent=True)
    plt.show()
```

Let's take a loot at the untrained score function `forward_fn`:

```
save_snapshot_score(params, forward_fn, meshgrid)
```

Not too surprisingly, it contains random but smooth transitions. Let's revisit it after a first training run.

28.6 Training and Sliding Window Method

Next we implement functions for network training. We implement the *sliding window* method and corresponding training algorithm below.

The sliding window method starts with window size **ROLLOUT_start**, which is increased by **ROLLOUT_add** every **steps** epochs for **ROLLOUT_increases** times. To reduce the number of points on a trajectory, we use subsampling (defined by **subsample**)

```
def update_network(params_, forward_fn, dataset, rng, steps = 20, ROLLOUT_increases =␣
↪18, max_training_time=None,
                   ROLLOUT_start = 4, ROLLOUT_add = 2, lr=5e-4, bidirectional=False,␣
↪subsample=5):

    scheduler = optax.piecewise_constant_schedule( init_value=lr)

    # Optax otimizer using Adam
    opt = optax.chain(
        optax.scale_by_adam(b1=0.9, b2=0.99),
        optax.scale_by_schedule(scheduler),
        optax.scale(-1.0))
    opt_state = opt.init(params_)

    # Define model loss and update for network parameters
    model_loss_fn = gradient_fn(forward_fn, physics_operator, g)
    grad_update = create_default_update_fn(opt, model_loss_fn)

    avg_loss = 0
    grad_updates = 0
    ROLLOUT = ROLLOUT_start
    history = []

    # Iterate through all sliding window sizes
    for _ in range(ROLLOUT_increases+1):
        print('Window size: ', ROLLOUT)
        pbar = tqdm(range(steps))

        # Iterate through all epochs
        for n in pbar:

            # Iterate through data set
            generator = iterbatches(dataset, 256, shuffle=False)
            for batch in generator:
                x_train, t_train = _prepare_batch(batch)


                # Concentrate on specific parts of trajectory
                if max_training_time is not None:

                    x_train = x_train[:, -max_training_time:]
                    t_train = t_train[-max_training_time:]
```

(continues on next page)

(continued from previous page)

```
                # Use subsampling to reduce number of points on trajectory
                x_train = x_train[:, ::subsample]
                t_train = t_train[::subsample]

                # Iterate through trajectory
                for t in range(x_train.shape[1]):

                    # Select values based on position and window size
                    x_train_sub = x_train[:, t:t+ROLLOUT]
                    t_train_sub = t_train[t:t+ROLLOUT]

                    # Compute loss for current part of trajectory
                    params_, opt_state, batch_loss_backward = grad_update(params_,␣
↪opt_state, [x_train_sub, t_train_sub], rng=rng)

                    # Reverse values and time discretization for the forward time␣
↪direction
                    if bidirectional:
                        params_, opt_state, batch_loss_forward = grad_update(params_,␣
↪opt_state,
                                                                             [x_train_
↪sub[:,::-1], t_train_sub[::-1]], rng=rng)

                    grad_updates += 1

                    rng, _ = jax.random.split(rng)
                    avg_loss += jax.device_get(batch_loss_backward)
                    if bidirectional:
                        avg_loss += jax.device_get(batch_loss_forward)

                    history.append(batch_loss_backward+batch_loss_forward)

            pbar.set_description(f'loss: {avg_loss/(n+1):.5f} grad updates: {grad_
↪updates}')

        ROLLOUT += ROLLOUT_add

    return params_, history
```

28.6.1 Single step loss

We train the score network with the single step loss for 2500 epochs using the probability flow ODE (see Single steps, Section 3). Single step is realized here with a *sliding window* of size 2, i.e. going one step forward in time for each sample. Increases of the unrolling are disabled via `ROLLOUT_increases=0` by default below. This speeds up training for default runs, but feel free to enable the increases in subsequent tests. We also use subsampling with factor 5 to reduce the number of points on the trajectory. After training, the resulting score function is visualized again.

```
ROLLOUT_increases=0

key = jax.random.PRNGKey(seed)
params_single_step, history = update_network(params, forward_fn, data, key, steps =␣
↪2500,
                                             ROLLOUT_start=2, ROLLOUT_
↪increases=ROLLOUT_increases, bidirectional=True, lr=1e-3,
```

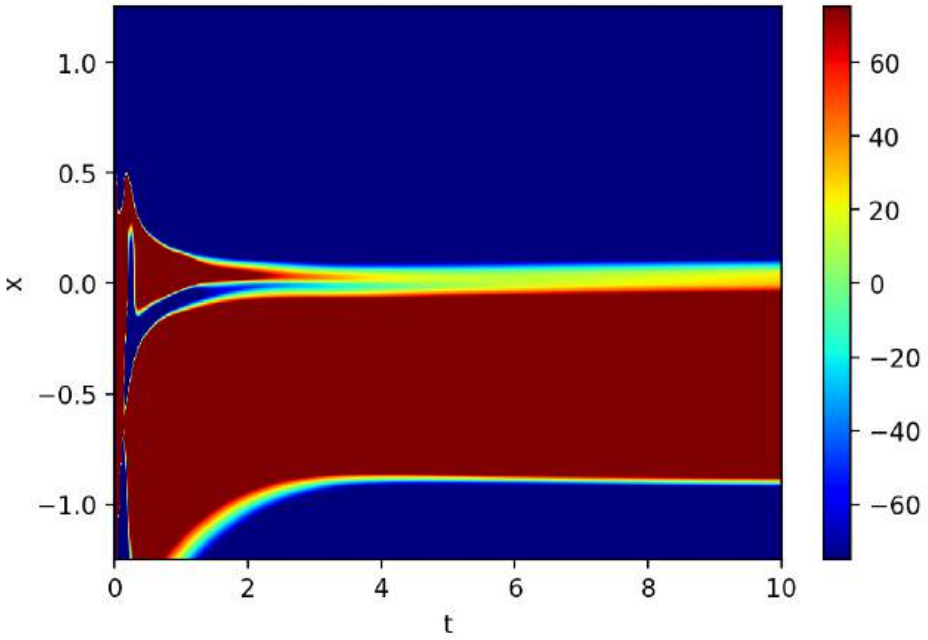
(continues on next page)

(continued from previous page)

```
                                            subsample=5)


save_snapshot_score(params_single_step, forward_fn, meshgrid)
```

```
Window size:  2
```

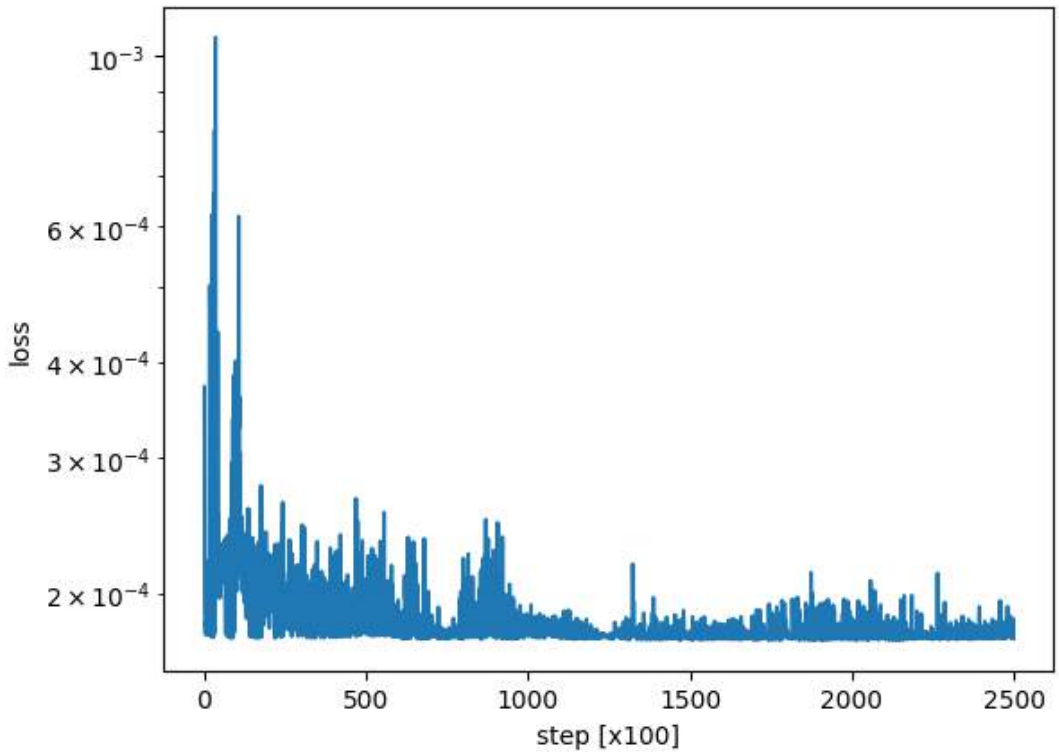
```
loss: 0.02322 grad updates: 250000: 100%|██████████| 2500/2500 [10:23<00:00,  4.
↪01it/s]
```



This should look more meaningful than the initial random state! The two bands of trajectories from -1 and 1 should be visible. E.g., the blue region at the top indicates a large negative gradient, driving samples ending up in this region down by a large distance, towards the correct trajectory in the positive region (as shown further above). Note that some parts, typically closer to the end time ($t = 10$ on the right), can show sign changes and mostly random content again. This is caused by the network primarily being trained in the vicinity of samples it has seen in the training data. Far away from it, it has "not learned" how to properly transform samples into admissible ones.

We also plot the training loss below, to show the training progressing for this first training step.

```
plt.plot(np.array(history[::100]))
plt.yscale('log')
plt.xlabel('step [x100]')
plt.ylabel('loss')
plt.show()
```

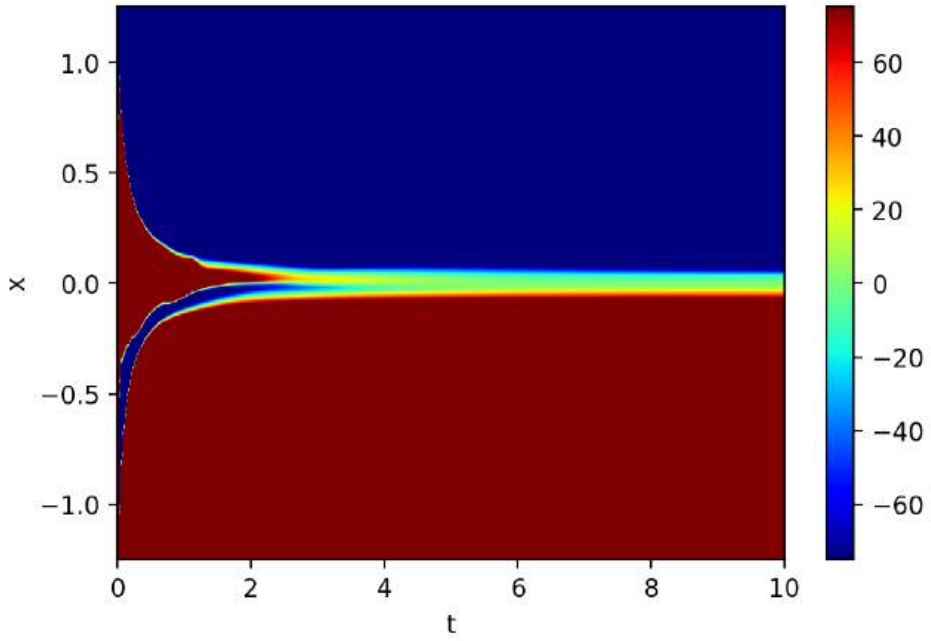


For the next training step, the goal is to improve the quality of the learned score: We decrease the learning rate and train without reducing the trajectories via "subsampling", meaning all samples in the training data are provided to the network.

```
params_single_step, history = update_network(params_single_step, forward_fn, data,␣
↪key, steps = 1000,
                                             ROLLOUT_start=2, ROLLOUT_
↪increases=ROLLOUT_increases, bidirectional=True, lr=1e-4,
                                             subsample=1)

save_snapshot_score(params_single_step, forward_fn, meshgrid)
```

```
Window size:  2
```

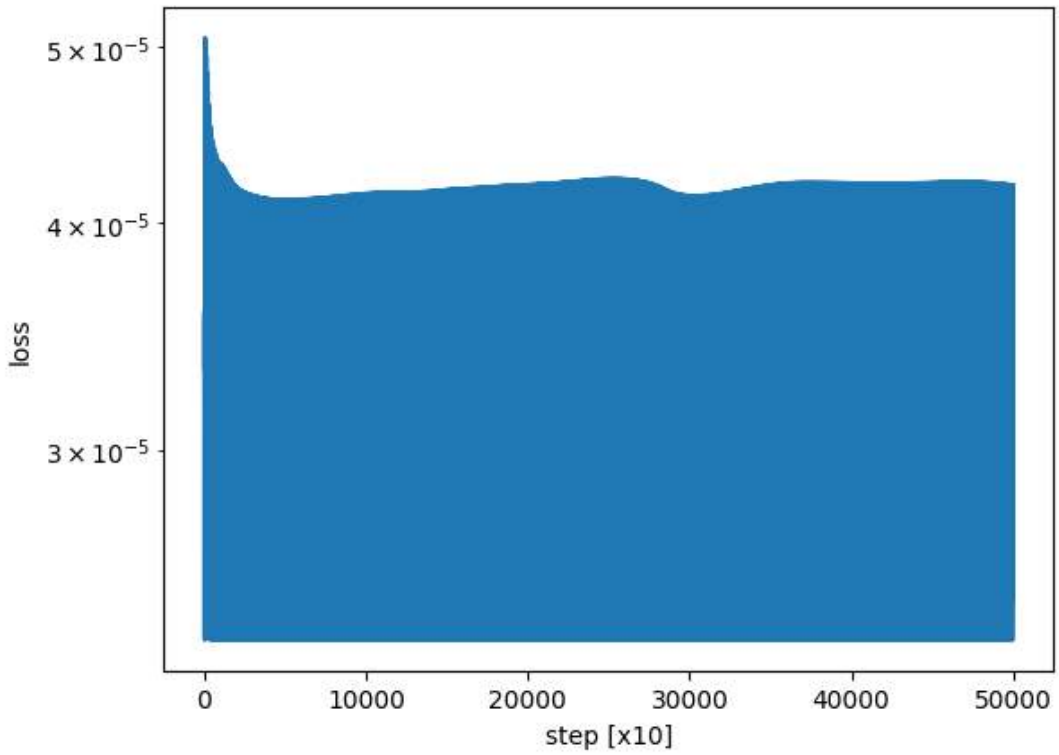
```
loss: 0.01599 grad updates: 500000: 100%|██████████| 1000/1000 [20:41<00:00,  1.
↪24s/it]
```



This typically results in an even *cleaner* score function, containing fewer sign changes outside the two bands of the ground truth trajectories. The runtime above was reduced via the `ROLLOUT_increases=0` setting. Once you enable it, e.g., with `ROLLOUT_increases=10` potentially with an increased number of iterations, you should see further improvements of the score field in the visualization.

The training in this stage is also more stable, as shown in the corresponding plot below.

```
plt.plot(np.array(history[::10]))
plt.yscale('log')
plt.xlabel('step [x10]')
plt.ylabel('loss')
plt.show()
```



28.7 Sampling SDE and ODE trajectories

Above we've only looked at the learned score function, and thus we'll evaluate the trajectories produces by integrating the two DE variants. We use the trained score with `diffrax` to solve the probability flow ODE and simulate paths from the reverse-time SDE.

28.7.1 Reverse-time SDE

We define a function to simulate paths from the reverse-time SDE with `diffrax`, where `WeaklyDiagonalControlTerm(diffusion, brownian_motion)` takes care of the noise:

```
def r_process_reverse(initial_value, params, noise_scaling, seed):
    key = jr.PRNGKey(seed)
    initial_shape = (1,)
    y1 = jnp.ones(shape=initial_shape) * initial_value
    t0, t1 = 0.0, 10.0
    dt0 = 0.01

    def drift(t, y, args):
        return physics_operator(y) - forward_fn(params, key, y, t)

    diffusion = lambda t, y, args: noise_scaling * jnp.ones(initial_shape)

    brownian_motion = VirtualBrownianTree(t0, t1, tol=1e-3, shape=initial_shape,␣
↪key=key)
```

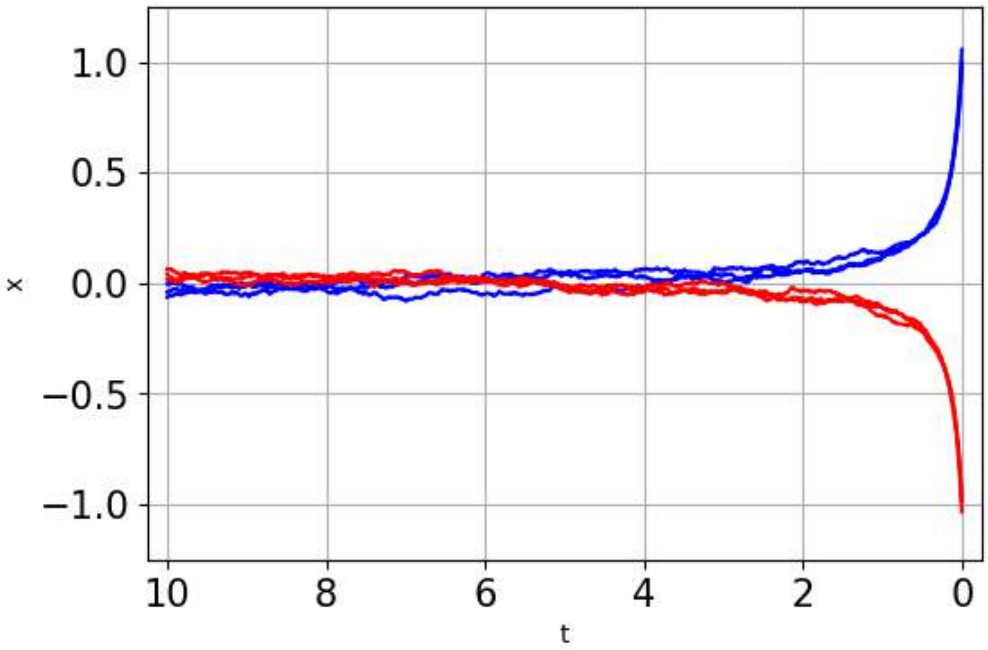
(continues on next page)

(continued from previous page)

```
    terms = MultiTerm(ODETerm(drift), WeaklyDiagonalControlTerm(diffusion, brownian_
↪motion))

    solver = dfx.Euler()
    t0 = jnp.array(0.0)
    args = None
    tprev = jnp.array(t1)
    tnext = jnp.array(t1 - dt0)
    y = y1

    state = solver.init(terms, tprev, tnext, y1, args)
    y_list = []
    for i in range(((t1-t0) / dt0).astype(int)):
        y, _, _, state, _ = solver.step(terms, tprev, tnext, y, args, state, made_
↪jump=False)
        tprev = tnext
        tnext = jnp.array(jnp.maximum(tprev - dt0, t0))
        y_list.append(y)

    return y_list
```

and plot ten different paths sampled from the reverse-time SDE:

```
if 1:
    combinations = zip(np.linspace(-0.065,0.065, 6), range(6))
    fig, ax = plt.subplots(figsize=(6,4))

    for init, seed_ in combinations:
        sol = r_process_reverse(init, params_single_step, g, seed_)
        if sol[-1] > 0:
            col = 'blue'
        else:
            col = 'red'
        ax.plot(np.linspace(10.0, 0.0, 1000), sol, color=col)

    ax.tick_params(axis='x')
    ax.tick_params(axis='y')
    ax.set_xlabel('t')
    ax.set_ylabel('x')
    ax.grid(True)
    ax.tick_params(labelsize=15)

    plt.xlim([10.25, -0.25])
    plt.ylim([-1.25,1.25])
    plt.show()
```



Note that the trajectories are shown along integration of the reverse-time SDE, and thus they appear mirrored along y in comparison to the ground truth ones above. With default settings, these trajectories should look quite good: exhibiting noisy motions during the initial phase, and reliably converging to 1 and -1.

If you reduce accuracy or training duration, you'll see that the score will contain erroneous regions, giving particles that shoot off before reaching $t = 0$.

28.7.2 Probability flow ODE

The probability flow ODE, as deterministic counterpart of the SDE, is an interesting variant that is obtained by removing the `brownian_motion` diffusion term. Analogously, we define a function to solve the probability flow ODE using only the drift term:

```
def r_process_reverse_ode(initial_value, params, noise_scaling, seed):
    key = jr.PRNGKey(seed)
    initial_shape = (1,)
    y1 = jnp.ones(shape=initial_shape) * initial_value
    t0, t1 = 0.0, 10.0
    dt0 = 0.01

    def drift(t, y, args):
        return physics_operator(y) - 0.5 * forward_fn(params, key, y, t)

    terms = ODETerm(drift)
    solver = dfx.Euler()
    t0 = jnp.array(0.0)

    args = None
    tprev = jnp.array(t1)
    tnext = jnp.array(t1 - dt0)
    y = y1
```

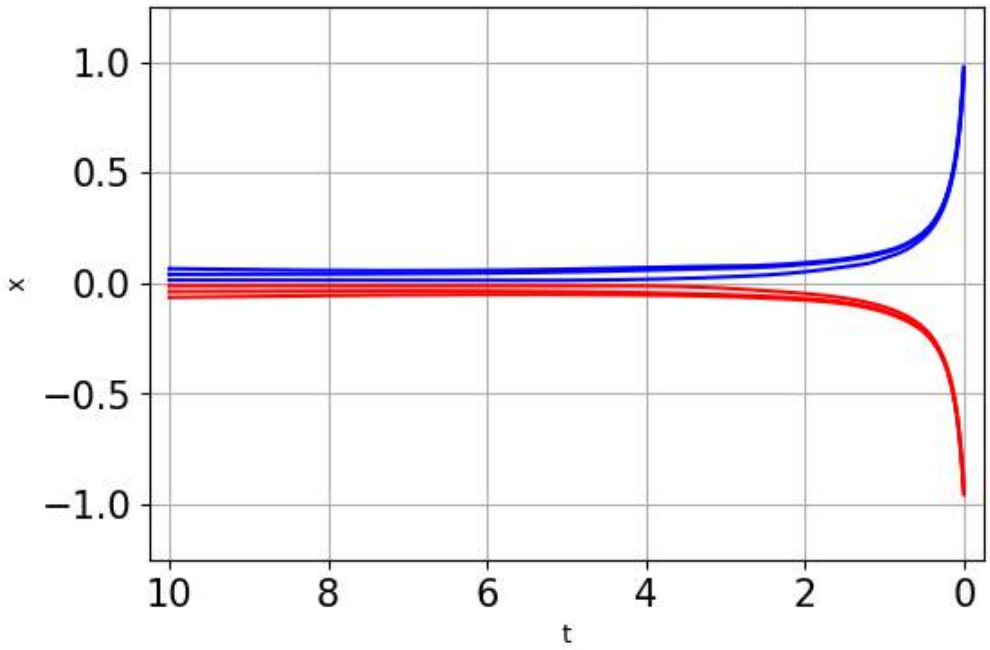
(continues on next page)

(continued from previous page)

```
    state = solver.init(terms, tprev, tnext, y1, args)
    y_list = []

    for i in range(((t1-t0) / dt0).astype(int)):
        y, _, _, state, _ = solver.step(terms, tprev, tnext, y, args, state, made_
↪jump=False)
        tprev = tnext
        tnext = jnp.array(jnp.maximum(tprev - dt0, t0))
        y_list.append(y)

    return y_list
```

Finally, we can plot the solutions to the probability flow ODE

```
#with plt.style.context("seaborn-white"):
if 1:
    combinations = zip(np.linspace(-0.065,0.065, 6), range(6))

    fig, ax = plt.subplots(figsize=(6,4))

    for init, seed_ in combinations:
        sol = r_process_reverse_ode(init, params_single_step, g, seed_)
        if sol[-1] > 0:
            col = 'blue'
        else:
            col = 'red'
        ax.plot(np.linspace(10.0, 0.0, 1000), sol, color=col)

    ax.tick_params(axis='x')
    ax.tick_params(axis='y')
    ax.set_xlabel('t')
    ax.set_ylabel('x')
    ax.grid(True)
    ax.tick_params(labelsize=15)

    plt.xlim([10.25, -0.25])
    plt.ylim([-1.25,1.25])
    plt.show()
```



They show the influence of the score without random perturbations, and in this case provide mostly horizontal motions with a bifurcation towards the two end points.

28.8 Next steps

- In this notebook it's worth experimenting with the unrolling steps (cf. `ROLLOUT_increases` above), to evaluate their effect on learning the score.
- In addition, it's interesting to intentionally deteriorate the quality of the score with short training runs. This should results in chaotic and erroneous trajectories.

CHAPTER

TWENTYNINE

DIFFUSION-BASED TIME PREDICTION

Simulating partial differential equations (PDEs), for example turbulent fluid flows, often requires resolving solutions over time. I.e., we're not interested in a time-averaged or long-term equilibrium state, but the actual changes of our system over time. This requires iterative solvers that are called *auto-regressively*, one step after the other, to produce a solution over time. Despite all advancements in this area, it is still a critical challenge to achieve stable and accurate predictions for extended temporal horizons. Many dynamical systems are inherently complex and chaotic, making it difficult to faithfully capture intricate physical phenomena over long timeframes.

At the same time, uncertainties also play a role for time series prediction: Even minor ambiguities in the spatially averaged states used for simulations can lead to very different outcomes over time. Moreover, most traditional solvers and learning-based methods process simulation trajectories in a deterministic way, treating them as being first-order Markovian (one state fully determines the next one). Instead a more realistic viewpoint of many systems is given by the Mori-Zwanzig formalism: we observe a part of our system, but at the same time an "unobserved" (or un-simulated) part of the state can influence its evolution over time. Deterministic simulators produce a single solution without accounting for a potentially probabilistic underlying processes. This motivates - as in the previous sections - to view the steps of a time series as a probabilistic distribution over time rather than a deterministic series of states. A probabilistic simulator can learn to take into account the influence of the un-observed state, and infer solutions from variations of this un-observed part of the system. Worst case, if this un-observed state has a negligible influence, we should see a mean state with an variance that's effectively zero. So there's nothing to loose!

The following notebook [run in colab] introduces an effective, distribution-based approach for temporal predictions:

- conditional diffusion models are used to compute autoregressive rollouts to obtain a "probabilistic simulator";
- it is of course highly interesting to compare this diffusion-based predictor to the deterministic baselines and neural operators from the previous chapters;
- in order to evaluate the results w.r.t. their accuracy, we'll employ a transonic fluid flow for which we can compute statistics from a simulated reference.

Problem formulation: while we've previously often focused on training networks for the task $f(x) = y$, we now focus on tasks of the form $f(x_t) = x_{t+1}$ to indicate that any subsequent step, e.g., $f(x_{t+1}) = x_{t+2}$, is a problem of the same importance as the first one. We still have ground truth values x^*_{t+1}, e.g., from an expensive high-fidelity simulation, and aim for a minimization problem

$$\arg\min_\theta |f(x_t;\theta) - x^*_{t+1}|_2^2. \tag{29.1}$$

Note

Unconditional stability: One of the most interesting aspects of using diffusion-based time predictors is their temporal stability. It seems that the diffusion process forces the networks to learn handling perturbations and accumulated errors in the states without being easily thrown off track. This is crucial for *unconditional stability*, i.e., neural networks that can be called autoregressively any number of times without blowing up. The training process below yields unconditionally stable networks with a surprisingly simple approach for training (we'll use DDPM below, but flow matching would likewise work).

For a more detailed evaluation of the long term stability of diffusion-based predictions can be found here.

29.1 Conditioning

Previously, for the inverse problem setting we only briefly mentioned that the inference task of producing the posterior distribution depends on a set of hyperparameters such as a chosen set of boundary conditions. Let's consider $x = (c, d)$, i.e. a data point x is made up of a component for conditioning c and the target data d. For time predictions, we additionally have a strong conditioning on the current time step, i.e. c will contain x_t in addition to, e.g., a Reynolds number. Hence, this is a good occasion to explain some of the subtleties of implementing the conditioning. The central take-away message here is: all inputs for conditioning should be treated in the same way as the outputs of the diffusion process.

This seems somewhat counter-intuitive at first: after all, the conditioning is more similar to an input than an output. However, it was shown that "forcing" the network to denoise the conditioning alongside the target at training time forces it to fully consider the conditioning variables. This leads to a tight entangling of features learned for the output with the conditioning. Removing the conditioning information at high noise levels in this way has the additional benefit of reducing error accumulation: Since the initial steps of the reverse process p_θ are mostly unconditional due to the very noisy conditioning, accumulated errors in c from the previous denoising steps are not immediately included in the prediction d.

Thus for training we consider both parts c and d in the same way. This is illustrated on the left side of the following picture. Conditioning c and data components d are treated the same at training time. This illustration denotes denoising time by r, to distinguish it from the time of the physical process t.

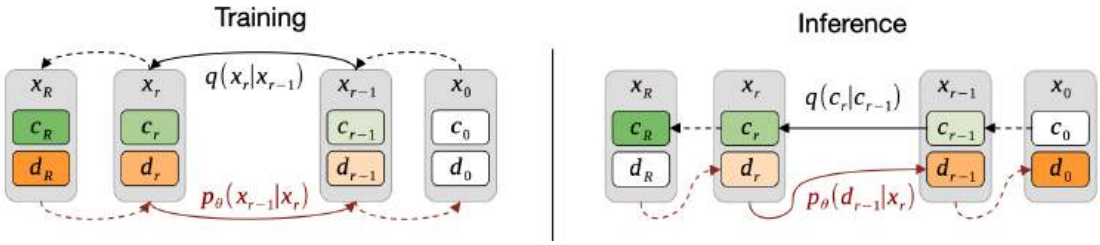


Fig. 29.1: An illustration of forward and backward noising/denoising chains with conditioning c and data d.

Once the model is trained, we make use of the fact that we know the exact value of c. As the noise ϵ is likewise an input to the model that we have under full control, we can ensure that the conditioning c_r at denoising time r has exactly the right content according to the chosen noise field and noising schedule. Hence we invoke $p_\theta(x_{r-1}|x_r)$ yielding c_{r-1} as well as d_{r-1}, both contained in x_{r-1}. The predicted conditioning c_{r-1} will be good if the model p_θ was trained well, but to make sure there is zero drift we simply recompute c_{r-1} from the known ground truth c and the right noise amount ϵ_{r-1}. We then invoke $p_\theta(x_{r-2}|x_{r-1})$ with x_{r-1} containing the re-computed c and the d_{r-1} component previously inferred in the previous denoising step.

For time prediction tasks, the situation is not anymore complicated, but slightly more confusing in terms of notation: here, the conditioning is the previous time step in *physical* time x^t. (There could be additional global parameters in c, but we'll

ignore those for simplicity; they can simply be appended to c.) The d component of each denoising chain will yield the next state of our physical system x^{t+1}. Thus, at denoising time, the task for our network is to denoise both time steps, while we prescribe the correctly noised known state at inference time: $c = x^t$, $d = x^{t+1}$. In the schematic from before, this yields:

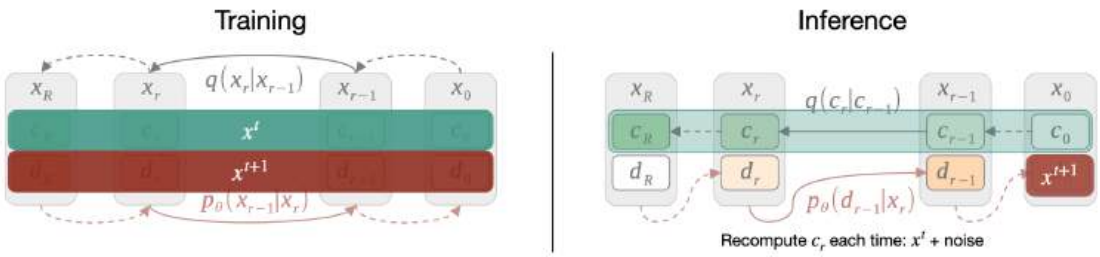


Fig. 29.2: Forward and backward chains for time prediction: At training time, $c = x^t$ and data $d = x^{t+1}$ are treated jointly. For inference, c is overwritten with ground truth values at each iteration of the diffusion model.

Note that in both cases, the physics time t is completely orthogonal to r, i.e., the denoising process does not interact or use t in any other way apart from being given the task to produce an output that obeys the dynamics of our system. A nice variation of this approach is to use a variable time step Δt as conditioning parameter in c. Below, we'll fix Δt and normalize it to 1 for simplicity.

29.2 Implementation

Specifically, this notebook will explain *autoregressive conditional diffusion models* (ACDM), following an existing benchmark and paper. The goal is the creation of a diffusion-based architecture, that can probabilistically and accurately predict the next simulation step of a turbulent flow simulation.

As a first step, we'll download a pre-trained model checkpoint to shorten the training time later on. This might take a few minutes. (Note: the command will not re-download files, if already downloaded successfully)

```
!mkdir -p models/
!wget -nc "https://dataserv.ub.tum.de/s/m1734798.001/download?path=/&
 ↪files=checkpoints_acdm_tra.zip" -O models/checkpoints_acdm_tra.zip --no-check-
 ↪certificate
!unzip -nq models/checkpoints_acdm_tra.zip -d models
```

```
File 'models/checkpoints_acdm_tra.zip' already there; not retrieving.
```

Colab already comes with a range of pre-installed packages and we only need to additionally install the *einops* package and the PBDL-Dataloader. If you are running this file locally and not inside colab, follow the installation instructions instead. Afterwards, make sure that the PBDL-Dataloader is also installed via `pip install git+https://github.com/tum-pbs/pbdl-dataset@main`.

```
try:
    import google.colab  # only to ensure that we are inside colab
    %pip install einops
    %pip install git+https://github.com/tum-pbs/pbdl-dataset@main
except ImportError:
    print("This notebook is running locally, please follow the ACDM installation␣
 ↪instructions and install the PBDL loader")
    pass
```

```
This notebook is running locally, please follow the ACDM installation instructions␣
↪and install the PBDL loader
```

Lets import some packages that we will need in the follwing.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, SequentialSampler, RandomSampler

from einops import rearrange
from functools import partial

import matplotlib.pyplot as plt
import numpy as np

import math
import os, json
from typing import List,Tuple,Dict

from pbdl.torch.loader import Dataloader
```

29.3 Backbone Network Definition

Of course, we also need a neural network architecture. Here, we will rely on a "modern" U-Net. In contrast to classic U-Net from *Supervised training for RANS flows around airfoils*, this modernized version differs in a few important places: the skip connection are replaced by attention mechanisms, *GELU* activations replace *ReLU*, and group normalizations are employed at each layer of the U-Net. This architecture is the backbone of many popular diffusion models, and typically yields at least a few percent improvements over simpler architectures (in some cases also much more).

We start with a residual block that defines the skip connections, as well as the up- and downsampling operations of the U-Net. We will also make use of sinusoidal position embeddings from the transformer architectures, to integrate the diffusion step with a time embedding MLP throughout the U-Net layers.

```
class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, *args, **kwargs):
        return self.fn(x, *args, **kwargs) + x

def Upsample(dim):
    return nn.ConvTranspose2d(dim, dim, 4, 2, 1)

def Downsample(dim):
    return nn.Conv2d(dim, dim, 4, 2, 1)


class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim
```

(continues on next page)

(continued from previous page)

```
    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings
```

Next, this class is the core convolution block in every level of the U-Net architecture, based on the ConvNeXtBlock.

```
class ConvNextBlock(nn.Module):

    def __init__(self, dim, dim_out, *, time_emb_dim=None, mult=2, norm=True):
        super().__init__()
        self.mlp = (
            nn.Sequential(nn.GELU(), nn.Linear(time_emb_dim, dim))
            if time_emb_dim is not None else None
        )

        self.ds_conv = nn.Conv2d(dim, dim, 7, padding=3, groups=dim)

        self.net = nn.Sequential(
            nn.GroupNorm(1, dim) if norm else nn.Identity(),
            nn.Conv2d(dim, dim_out * mult, 3, padding=1),
            nn.GELU(),
            nn.GroupNorm(1, dim_out * mult),
            nn.Conv2d(dim_out * mult, dim_out, 3, padding=1),
        )

        self.res_conv = nn.Conv2d(dim, dim_out, 1) if dim != dim_out else nn.
↪Identity()

    def forward(self, x, time_emb=None):
        h = self.ds_conv(x)

        if self.mlp is not None and time_emb is not None:
            assert time_emb is not None, "time embedding must be passed in"
            condition = self.mlp(time_emb)
            h = h + rearrange(condition, "b c -> b c 1 1")

        h = self.net(h)
        return h + self.res_conv(x)
```

We'll also define classes for the *attention* mechanisms. As regular attention (also known as "softmax attention") has a quadratic scaling w.r.t. the input size, it is only applied in the bottleneck layer of the U-Net. Here we have the smallest size of the latent space, and hence the dense connectivity of the attention should not lead to resource explosions.

For the skip connections, *linear* attention is used instead. As the name implies, it is linear in terms of input size, but still quadratic w.r.t. its internal kernel dimension. That one, however, can be chosen freely, while we're restricted to given (and potentially) large input sizes in the U-Net structure.

```
class Attention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=32):
        super().__init__()
        self.scale = dim_head**-0.5
        self.heads = heads
```

(continues on next page)

(continued from previous page)

```
        hidden_dim = dim_head * heads
        self.to_qkv = nn.Conv2d(dim, hidden_dim * 3, 1, bias=False)
        self.to_out = nn.Conv2d(hidden_dim, dim, 1)

    def forward(self, x):
        b, c, h, w = x.shape
        qkv = self.to_qkv(x).chunk(3, dim=1)
        q, k, v = map(
            lambda t: rearrange(t, "b (h c) x y -> b h c (x y)", h=self.heads), qkv
        )
        q = q * self.scale

        sim = torch.einsum("b h d i, b h d j -> b h i j", q, k)
        sim = sim - sim.amax(dim=-1, keepdim=True).detach()
        attn = sim.softmax(dim=-1)

        out = torch.einsum("b h i j, b h d j -> b h i d", attn, v)
        out = rearrange(out, "b h (x y) d -> b (h d) x y", x=h, y=w)
        return self.to_out(out)


class LinearAttention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=32):
        super().__init__()
        self.scale = dim_head**-0.5
        self.heads = heads
        hidden_dim = dim_head * heads
        self.to_qkv = nn.Conv2d(dim, hidden_dim * 3, 1, bias=False)

        self.to_out = nn.Sequential(nn.Conv2d(hidden_dim, dim, 1),
                                    nn.GroupNorm(1, dim))

    def forward(self, x):
        b, c, h, w = x.shape
        qkv = self.to_qkv(x).chunk(3, dim=1)
        q, k, v = map(
            lambda t: rearrange(t, "b (h c) x y -> b h c (x y)", h=self.heads), qkv
        )

        q = q.softmax(dim=-2)
        k = k.softmax(dim=-1)

        q = q * self.scale
        context = torch.einsum("b h d n, b h e n -> b h d e", k, v)

        out = torch.einsum("b h d e, b h d n -> b h e n", context, q)
        out = rearrange(out, "b h c (x y) -> b (h c) x y", h=self.heads, x=h, y=w)
        return self.to_out(out)
```

The next helper provides a convenient way to add group normalization, which we'll add before up- and down-sampling the intermediate states.

```
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.fn = fn
        self.norm = nn.GroupNorm(1, dim)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        x = self.norm(x)
        return self.fn(x)
```

Putting the introduced layers together, the next cell defines the full, attention-based U-Net architecture:

```
class Unet(nn.Module):
    def __init__(
        self,
        dim,
        init_dim=None,
        out_dim=None,
        dim_mults=(1, 2, 4, 8),
        channels=3,
        with_time_emb=True,
        resnet_block_groups=8,
        use_convnext=True,
        convnext_mult=2,
    ):
        super().__init__()

        # determine dimensions
        self.channels = channels

        init_dim = init_dim if init_dim is not None else dim // 3 * 2
        self.init_conv = nn.Conv2d(channels, init_dim, 7, padding=3)

        dims = [init_dim, *map(lambda m: dim * m, dim_mults)]
        in_out = list(zip(dims[:-1], dims[1:]))

        if use_convnext:
            block_klass = partial(ConvNextBlock, mult=convnext_mult)
        else:
            raise NotImplementedError()

        # time embeddings
        if with_time_emb:
            time_dim = dim * 4
            self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(dim),
                nn.Linear(dim, time_dim),
                nn.GELU(),
                nn.Linear(time_dim, time_dim),
            )

        else:
            time_dim = None
            self.time_mlp = None
            self.cond_mlp = None
            self.sim_mlp = None

        # layers
        self.downs = nn.ModuleList([])
        self.ups = nn.ModuleList([])
        num_resolutions = len(in_out)
```

(continues on next page)

(continued from previous page)

```
        for ind, (dim_in, dim_out) in enumerate(in_out):
            is_last = ind >= (num_resolutions - 1)

            self.downs.append(
                nn.ModuleList(
                    [
                        block_klass(dim_in, dim_out, time_emb_dim=time_dim),
                        block_klass(dim_out, dim_out, time_emb_dim=time_dim),
                        Residual(PreNorm(dim_out, LinearAttention(dim_out))),
                        Downsample(dim_out) if not is_last else nn.Identity(),
                    ]
                )
            )

        mid_dim = dims[-1]
        self.mid_block1 = block_klass(mid_dim, mid_dim, time_emb_dim=time_dim)
        self.mid_attn = Residual(PreNorm(mid_dim, Attention(mid_dim)))
        self.mid_block2 = block_klass(mid_dim, mid_dim, time_emb_dim=time_dim)

        for ind, (dim_in, dim_out) in enumerate(reversed(in_out[1:])):
            is_last = ind >= (num_resolutions - 1)

            self.ups.append(
                nn.ModuleList(
                    [
                        block_klass(dim_out * 2, dim_in, time_emb_dim=time_dim),
                        block_klass(dim_in, dim_in, time_emb_dim=time_dim),
                        Residual(PreNorm(dim_in, LinearAttention(dim_in))),
                        Upsample(dim_in) if not is_last else nn.Identity(),
                    ]
                )
            )

        out_dim = out_dim if out_dim is not None else channels
        self.final_conv = nn.Sequential(
            block_klass(dim, dim), nn.Conv2d(dim, out_dim, 1)
        )

    def forward(self, x, time):
        x = self.init_conv(x)

        t = self.time_mlp(time) if self.time_mlp is not None else None

        h = []

        # downsample
        for block1, block2, attn, downsample in self.downs:
            x = block1(x, t)
            x = block2(x, t)
            x = attn(x)
            h.append(x)
            x = downsample(x)

        # bottleneck
        x = self.mid_block1(x, t)
        x = self.mid_attn(x)
        x = self.mid_block2(x, t)
```

(continues on next page)

(continued from previous page)

```
        # upsample
        for block1, block2, attn, upsample in self.ups:
            x = torch.cat((x, h.pop()), dim=1)
            x = block1(x, t)
            x = block2(x, t)
            x = attn(x)
            x = upsample(x)

        return self.final_conv(x)
```

29.4 Variance Schedule

To determine the noise levels over the course of the diffusion process, a noise schedule is required. Here, we will employ the simple linear schedule proposed by Ho et al., with adjusted variance parameters, such that any number of diffusion steps larger than 10 should work well:

```
def linear_beta_schedule(timesteps):
    if timesteps < 10:
        raise ValueError("Warning: Less than 10 timesteps require adjustments to this␣
↪schedule!")

    beta_start = 0.0001 * (500/timesteps) # adjust reference values determined for␣
↪500 steps
    beta_end = 0.02 * (500/timesteps)
    betas = torch.linspace(beta_start, beta_end, timesteps)
    return torch.clip(betas, 0.0001, 0.9999)
```

29.5 Diffusion Model Definition

Finally, we have collected all pieces to define a class containing the actual diffusion model.

We first compute the noising schedule for DDPM, and make sure the hyperparameters of the diffusion process are not treated as variables during learning (declaring them as *buffers* in pytorch via `register_buffer`).

```
class DiffusionModel(nn.Module):
    def __init__(self, diffusionSteps:int, condChannels:int, dataChannels:int):
        super(DiffusionModel, self).__init__()

        self.timesteps = diffusionSteps
        betas = linear_beta_schedule(timesteps=self.timesteps)

        betas = betas.unsqueeze(1).unsqueeze(2).unsqueeze(3)
        alphas = 1.0 - betas
        alphasCumprod = torch.cumprod(alphas, axis=0)
        alphasCumprodPrev = F.pad(alphasCumprod[:-1], (0,0,0,0,0,0,1,0), value=1.0)
        sqrtRecipAlphas = torch.sqrt(1.0 / alphas)

        # calculations for diffusion q(x_t | x_{t-1}) and others
        sqrtAlphasCumprod = torch.sqrt(alphasCumprod)
        sqrtOneMinusAlphasCumprod = torch.sqrt(1. - alphasCumprod)
```

(continues on next page)

(continued from previous page)

```
        # calculations for posterior q(x_{t-1} | x_t, x_0)
        posteriorVariance = betas * (1. - alphasCumprodPrev) / (1. - alphasCumprod)
        sqrtPosteriorVariance = torch.sqrt(posteriorVariance)

        self.register_buffer("betas", betas)
        self.register_buffer("sqrtRecipAlphas", sqrtRecipAlphas)
        self.register_buffer("sqrtAlphasCumprod", sqrtAlphasCumprod)
        self.register_buffer("sqrtOneMinusAlphasCumprod", sqrtOneMinusAlphasCumprod)
        self.register_buffer("sqrtPosteriorVariance", sqrtPosteriorVariance)

        # backbone model
        self.unet = Unet(
            dim=128,
            channels= condChannels + dataChannels,
            dim_mults=(1,1,1),
            use_convnext=True,
            convnext_mult=1,
        )


    # input shape (both inputs): B S C W H (D) -> output shape (both outputs): B S nC␣
↪W H (D)
    def forward(self, conditioning:torch.Tensor, data:torch.Tensor) -> torch.Tensor:
        device = "cuda" if data.is_cuda else "cpu"
        seqLen = data.shape[1]

        # combine batch and sequence dimension for decoder processing
        d = torch.reshape(data, (-1, data.shape[2], data.shape[3], data.shape[4]))
        cond = torch.reshape(conditioning, (-1, conditioning.shape[2], conditioning.
↪shape[3], conditioning.shape[4]))

        # TRAINING
        if self.training:

            # forward diffusion process that adds noise to data
            d = torch.concat((cond, d), dim=1)
            noise = torch.randn_like(d, device=device)
            t = torch.randint(0, self.timesteps, (d.shape[0],), device=device).long()
            dNoisy = self.sqrtAlphasCumprod[t] * d + self.
↪sqrtOneMinusAlphasCumprod[t] * noise

            # noise prediction with network
            predictedNoise = self.unet(dNoisy, t)

            # unstack batch and sequence dimension again
            noise = torch.reshape(noise, (-1, seqLen, conditioning.shape[2] + data.
↪shape[2], data.shape[3], data.shape[4]))
            predictedNoise = torch.reshape(predictedNoise, (-1, seqLen, conditioning.
↪shape[2] + data.shape[2], data.shape[3], data.shape[4]))

            return noise, predictedNoise


        # INFERENCE
        else:
            # conditioned reverse diffusion process
```

(continues on next page)

(continued from previous page)

```
            dNoise = torch.randn_like(d, device=device)
            cNoise = torch.randn_like(cond, device=device)

            for i in reversed(range(0, self.timesteps)):
                t = i * torch.ones(cond.shape[0], device=device).long()

                # compute conditioned part with normal forward diffusion
                condNoisy = self.sqrtAlphasCumprod[t] * cond + self.
↪sqrtOneMinusAlphasCumprod[t] * cNoise

                dNoiseCond = torch.concat((condNoisy, dNoise), dim=1)

                # backward diffusion process that removes noise to create data
                predictedNoiseCond = self.unet(dNoiseCond, t)

                # use model (noise predictor) to predict mean
                modelMean = self.sqrtRecipAlphas[t] * (dNoiseCond - self.betas[t] *␣
↪predictedNoiseCond / self.sqrtOneMinusAlphasCumprod[t])

                dNoise = modelMean[:, cond.shape[1]:modelMean.shape[1]] # discard␣
↪prediction of conditioning
                if i != 0:
                    # sample randomly (only for non-final prediction), use mean␣
↪directly for final prediction
                    dNoise = dNoise + self.sqrtPosteriorVariance[t] * torch.randn_
↪like(dNoise)

            # unstack batch and sequence dimension again
            dNoise = torch.reshape(dNoise, (-1, seqLen, data.shape[2], data.shape[3],␣
↪data.shape[4]))

            return dNoise
```

The most important function above is the `forward` step of the `DiffusionModel` class. It switches between training and inference via the `self.training` flag, and correspondingly either evaluates a single step in the Markov chain for backpropagation, or the full chain to obtain a sample at inference time. The `c` and `d` prefixes of the variables indicate the distinction between *conditioning* and *data* components in x, as explained above. E.g., an important line in the inference code is `dNoiseCond=concat(condNoisy, dNoise)` this gives an x by concatenating conditioning and data. Both parts of the x are jointly denoised, and the conditioning part is removed after network execution via `modelMean[:, cond.shape[1]:modelMean.shape[1]]`. It's overwritten with the ground truth conditioning, such that the diffusion model can focus on producing an accurate `d` part.

In the cell below we'll use the `PBDL-Dataloader` class to obtain the training (and testing) dataset. The code below first cleans up (`del`-eting old loaders that might stick around, this is useful for working in Jupyter to avoid HDF5 file lock errors).

A slight complication here is that we'll only use a small part of the full dataset (3 simulations with IDs 20,21 for training and 22 for testing). To make sure we have the right normalization constants for the full dataset (and to spare you the time to download all the full 5GB), the `norm_X_mean` and `norm_X_std` fields of the dataloader are manually initialized with the constants from the full dataset below.

```
# delete loaders if existing to ensure the dataset file is acessible
try:
    del trainLoader
except:
```

(continues on next page)

(continued from previous page)

```
    pass
try:
    del testLoader
except:
    pass

# adjust dataset normalization buffers to match data normalization of pretrained model
download = Dataloader("transonic-cylinder-flow", sel_sims=[20,21,22])
del download
import h5py
dataPath = "datasets/transonic-cylinder-flow.hdf5"
with h5py.File(dataPath, "r+") as f:
    if "norm_fields_sca_mean" in f:
        f.__delitem__("norm_fields_sca_mean")
    if "norm_fields_sca_std" in f:
        f.__delitem__("norm_fields_sca_std")
    if "norm_const_mean" in f:
        f.__delitem__("norm_const_mean")
    if "norm_const_std" in f:
        f.__delitem__("norm_const_std")

    f["norm_const_mean"] = np.array([0.70])
    f["norm_const_std"] = np.array([0.118322])
    f["norm_fields_sca_mean"] = np.array([[[0.560642]], [[-0.000129]], [[0.903352]],␣
↪[[0.637941]]])
    f["norm_fields_sca_std"] = np.array([[[0.216987]], [[0.216987]], [[0.145391]],␣
↪[[0.119944]]])
    f.close()


batch = 32 # training batch size

# training set configuration
trainLoader = Dataloader(
    "transonic-cylinder-flow",
    time_steps=5, # leads to six timesteps that are later strided by a factor of 2␣
↪(for dt compatibility with pre-trained model with two input steps and one target␣
↪step)
    intermediate_time_steps=True,
    step_size=6, # discard some datasamples close together
    sel_sims=[20, 21], # select simulations
    trim_start=250, # discard some initial timesteps from the simulation
    normalize_data="mean-std", # data normalization
    normalize_const="mean-std", # constants normalization
    batch_size=batch,
    shuffle=True,
    drop_last=True,
    num_workers=2,
)
```

```
download completed          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100
↪%6m
Success: Loaded transonic-cylinder-flow with 41 simulations (3 selected) and 1␣
↪samples each.
download completed          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100
↪%6m
```

(continues on next page)

(continued from previous page)

```
Success: Loaded transonic-cylinder-flow with 41 simulations (2 selected) and 124␣
 ↪samples each.
```

29.6 Training

With all these building blocks and data available now, its time put them together and train the diffusion model. You can choose to either continue training for a few epochs from the provided model checkpoint (takes less than a minute for 10 epochs), or train the model from scratch on the small exemplary data set (this will take a few hours). Note that the prediction quality and diversity for training the model from scratch will be noticeably worse, due to limited amount of data available here.

Feel free to skip this step entirely, if you don't want to train the network. You can directly continue with the sampling below, by using the pre-trained checkpoint.

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Training device: %s" % device)

startFromCheckpoint = True

if startFromCheckpoint:
    epochs = 10 # finetune only for a small number of epochs
    lr = 0.00001 # since the model is already trained, a conservatively low learning␣
 ↪rate
else:
    epochs = 100000 # train from scratch for large number of epochs, this will take a␣
 ↪while
    lr = 0.0001 # larger learning rate for training from scratch

diffusionSteps = 20 # the provided model checkpoint was pretrained on 20 diffusion␣
 ↪steps

# model definition
condChannels = 2 * 5 # two timesteps with 5 channels each (vel_x, vel_y, dens, pres,␣
 ↪mach)
dataChannels = 5 # one timestep
model = DiffusionModel(diffusionSteps, condChannels, dataChannels)

if startFromCheckpoint:
    # load weights from checkpoint
    loaded = torch.load("models/models_tra/128_acdm-r20_02/Model.pth", map_
 ↪location=torch.device('cpu'))
    model.load_state_dict(loaded["stateDictDecoder"])
model.train()
model.to(device)

# print model info and trainable weigths
paramsTrainable = sum([np.prod(p.size()) for p in filter(lambda p: p.requires_grad,␣
 ↪model.parameters())])
params = sum([np.prod(p.size()) for p in model.parameters()])
#print(model)
print("Trainable Weights (All Weights): %d (%d)" % (paramsTrainable, params))

# training loop
print("\nStarting training...")
```

(continues on next page)

(continued from previous page)

```
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
for epoch in range(epochs):
    losses = []
    for s, sample in enumerate(trainLoader, 0):
        optimizer.zero_grad()

        input, target = sample
        input = input.unsqueeze(1) # add timestep dimension
        mach = input[:,:,-1:].expand(-1, target.shape[1], -1, -1, -1) # extract mach␣
↪number from input
        target = torch.concat([target, mach], dim=2) # add mach number to target
        d = torch.concat([input,target], dim=1).to(device).float() # combined [batch,␣
↪timesteps, channels, x, y] tensor
        d = d[:,::2] # temporal stride of 2 for dt compatiblity with pre-trained model

        inputSteps = 2
        cond = []
        for i in range(inputSteps):
            cond += [d[:,i:i+1]] # collect input steps
        conditioning = torch.concat(cond, dim=2) # combine along channel dimension
        data = d[:, inputSteps:inputSteps+1]

        noise, predictedNoise = model(conditioning=conditioning, data=data)

        loss = F.smooth_l1_loss(noise, predictedNoise)
        print("    [Epoch %2d, Batch %4d]: %1.7f" % (epoch, s, loss.detach().cpu().
↪item()))
        loss.backward()

        losses += [loss.detach().cpu().item()]

        optimizer.step()
    print("[Epoch %2d, FULL]: %1.7f" % (epoch, sum(losses)/len(losses)))

del trainLoader # delete to ensure file access
print("Training complete!")
```

```
Training device: cuda
```

```
/tmp/ipykernel_917142/3219208168.py:22: FutureWarning: You are using `torch.load`␣
↪with `weights_only=False` (the current default value), which uses the default␣
↪pickle module implicitly. It is possible to construct malicious pickle data␣
↪which will execute arbitrary code during unpickling (See https://github.com/
↪pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a␣
↪future release, the default value for `weights_only` will be flipped to `True`.␣
↪This limits the functions that could be executed during unpickling. Arbitrary␣
↪objects will no longer be allowed to be loaded via this mode unless they are␣
↪explicitly allowlisted by the user via `torch.serialization.add_safe_globals`.␣
↪We recommend you start setting `weights_only=True` for any use case where you don
↪'t have full control of the loaded file. Please open an issue on GitHub for any␣
↪issues related to this experimental feature.
  loaded = torch.load("models/models_tra/128_acdm-r20_02/Model.pth", map_
↪location=torch.device('cpu'))
```

```
Trainable Weights (All Weights): 6994035 (6994035)
```

(continues on next page)

(continued from previous page)

```
Starting training...
    [Epoch  0, Batch    0]: 0.0019515
    [Epoch  0, Batch    1]: 0.0019654
    [Epoch  0, Batch    2]: 0.0008050
    [Epoch  0, Batch    3]: 0.0006933
    [Epoch  0, Batch    4]: 0.0009360
    [Epoch  0, Batch    5]: 0.0018654
    [Epoch  0, Batch    6]: 0.0016832
[Epoch  0, FULL]: 0.0014143
    [Epoch  1, Batch    0]: 0.0011901
    [Epoch  1, Batch    1]: 0.0018340
    [Epoch  1, Batch    2]: 0.0007823
    [Epoch  1, Batch    3]: 0.0014465
    [Epoch  1, Batch    4]: 0.0011019
    [Epoch  1, Batch    5]: 0.0012040
    [Epoch  1, Batch    6]: 0.0007916
[Epoch  1, FULL]: 0.0011929
    [Epoch  2, Batch    0]: 0.0008890
    [Epoch  2, Batch    1]: 0.0013503
    [Epoch  2, Batch    2]: 0.0011220
    [Epoch  2, Batch    3]: 0.0009489
    [Epoch  2, Batch    4]: 0.0010360
    [Epoch  2, Batch    5]: 0.0012546
    [Epoch  2, Batch    6]: 0.0009530
[Epoch  2, FULL]: 0.0010791
    [Epoch  3, Batch    0]: 0.0009957
    [Epoch  3, Batch    1]: 0.0013200
    [Epoch  3, Batch    2]: 0.0025289
    [Epoch  3, Batch    3]: 0.0009580
    [Epoch  3, Batch    4]: 0.0010404
    [Epoch  3, Batch    5]: 0.0010360
    [Epoch  3, Batch    6]: 0.0009525
[Epoch  3, FULL]: 0.0012616
    [Epoch  4, Batch    0]: 0.0024588
    [Epoch  4, Batch    1]: 0.0013688
    [Epoch  4, Batch    2]: 0.0008885
    [Epoch  4, Batch    3]: 0.0007770
    [Epoch  4, Batch    4]: 0.0008546
    [Epoch  4, Batch    5]: 0.0012591
    [Epoch  4, Batch    6]: 0.0013123
[Epoch  4, FULL]: 0.0012742
    [Epoch  5, Batch    0]: 0.0013485
    [Epoch  5, Batch    1]: 0.0012143
    [Epoch  5, Batch    2]: 0.0013597
    [Epoch  5, Batch    3]: 0.0007690
    [Epoch  5, Batch    4]: 0.0014855
    [Epoch  5, Batch    5]: 0.0011670
    [Epoch  5, Batch    6]: 0.0015456
[Epoch  5, FULL]: 0.0012699
    [Epoch  6, Batch    0]: 0.0010924
    [Epoch  6, Batch    1]: 0.0023275
    [Epoch  6, Batch    2]: 0.0008677
    [Epoch  6, Batch    3]: 0.0008471
    [Epoch  6, Batch    4]: 0.0014519
    [Epoch  6, Batch    5]: 0.0011380
    [Epoch  6, Batch    6]: 0.0017657
[Epoch  6, FULL]: 0.0013558
```

(continues on next page)

(continued from previous page)

```
    [Epoch  7, Batch    0]: 0.0013877
    [Epoch  7, Batch    1]: 0.0011317
    [Epoch  7, Batch    2]: 0.0010489
    [Epoch  7, Batch    3]: 0.0018891
    [Epoch  7, Batch    4]: 0.0020820
    [Epoch  7, Batch    5]: 0.0007572
    [Epoch  7, Batch    6]: 0.0007788
[Epoch  7, FULL]: 0.0012965
    [Epoch  8, Batch    0]: 0.0017368
    [Epoch  8, Batch    1]: 0.0009061
    [Epoch  8, Batch    2]: 0.0009467
    [Epoch  8, Batch    3]: 0.0009662
    [Epoch  8, Batch    4]: 0.0013592
    [Epoch  8, Batch    5]: 0.0014016
    [Epoch  8, Batch    6]: 0.0010049
[Epoch  8, FULL]: 0.0011888
    [Epoch  9, Batch    0]: 0.0010400
    [Epoch  9, Batch    1]: 0.0014242
    [Epoch  9, Batch    2]: 0.0011576
    [Epoch  9, Batch    3]: 0.0008646
    [Epoch  9, Batch    4]: 0.0011597
    [Epoch  9, Batch    5]: 0.0011647
    [Epoch  9, Batch    6]: 0.0012795
[Epoch  9, FULL]: 0.0011557
Training complete!
```

29.7 Test Dataset

Next we download a test dataset to make sure we can evaluate the trained network on new data. Here, we only use sequences from the data set with a different mach number and physical time than the training data above (the simulation with ID 22, which should correspond to a larger Mach number than the ones used for training with Ma $= 0.72$).

```
# delete loaders if existing to ensure the dataset file is acessible
try:
    del trainLoader
except:
    pass
try:
    del testLoader
except:
    pass

# test set configuration
testLoader = Dataloader(
    "transonic-cylinder-flow",
    time_steps=119, # leads to 120 timesteps that are later strided by a factor of 2␣
↪(for dt compatibility with pre-trained model)
    intermediate_time_steps=True,
    step_size=120,
    sel_sims=[22], # select simulations
    trim_start=600, # discard some initial timesteps from the simulation
    normalize_data="mean-std", # data normalization
    normalize_const="mean-std", # constants normalization
    batch_size=1,
```

(continues on next page)

(continued from previous page)

```
    shuffle=False,
    drop_last=False,
    num_workers=0,
)
```

```
download completed          ———————————————————————————————————————— 100
 ↪%6m
Success: Loaded transonic-cylinder-flow with 41 simulations (1 selected) and 2␣
 ↪samples each.
```

29.8 Test Inference

We can now sample the trained diffusion model to create probabilistic predictions for a test set of flow trajectories. We store both predictions and ground truth in tensors with shape $(samples \times sequences \times sequenceLength \times channels \times sizeX \times sizeY)$, that are used for the evaluations and visualization below.

```
device = "cuda" if torch.cuda.is_available() else "cpu"

numSamples = 5
diffusionSteps = 20

try: # load model if not trained/finetuned above
    model
except NameError:
    condChannels = 2 * 5 # two timesteps with 5 channels each (vel_x, vel_y, dens,␣
 ↪pres, mach)
    dataChannels = 5 # one timestep
    model = DiffusionModel(diffusionSteps, condChannels, dataChannels)

    # load weights from checkpoint
    loaded = torch.load("models/models_tra/128_acdm-r20_02/Model.pth", map_
 ↪location=torch.device('cpu'))
    model.load_state_dict(loaded["stateDictDecoder"])
model.eval()
model.to(device)

# sampling loop
print("\nStarting sampling...")
gt = []
pred = []
with torch.no_grad():
    for s, sample in enumerate(testLoader, 0):

        input, target = sample
        input = input.unsqueeze(1) # add timestep dimension
        mach = input[:,:,-1:].expand(-1, target.shape[1], -1, -1, -1) # extract mach␣
 ↪number from input
        target = torch.concat([target, mach], dim=2) # add mach number to target
        data = torch.concat([input,target], dim=1).to(device).float() # combined␣
 ↪[batch, timesteps, channels, x, y] tensor
        data = data[:,::2] # temporal stride of 2 for dt compatiblity with pre-
 ↪trained model
```

(continues on next page)

(continued from previous page)

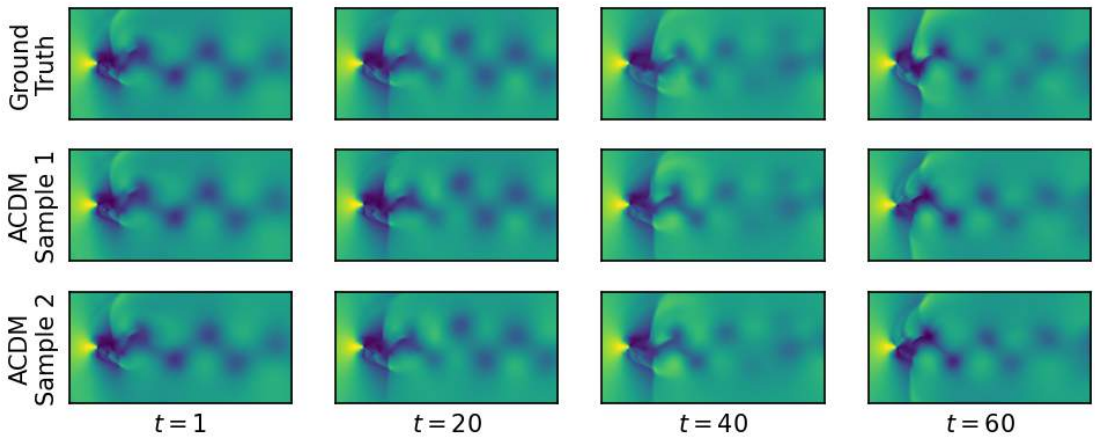
```
        gt += [data.unsqueeze(0).cpu().numpy()]
        d = data.to(device).repeat(numSamples,1,1,1,1) # reuse batch dim for samples

        prediction = torch.zeros_like(d, device=device)
        inputSteps = 2

        for i in range(inputSteps): # no prediction of first steps
            prediction[:,i] = d[:,i]

        for i in range(inputSteps, d.shape[1]):
            cond = []
            for j in range(inputSteps,0,-1):
                cond += [prediction[:, i-j : i-(j-1)]] # collect input steps
            cond = torch.concat(cond, dim=2) # combine along channel dimension

            result = model(conditioning=cond, data=d[:,i-1:i]) # auto-regressive␣
↪inference
            result[:,:,-1:] = d[:,i:i+1,-1:] # replace mach number prediction with␣
↪true values
            prediction[:,i:i+1] = result

        prediction = torch.reshape(prediction, (numSamples, -1, d.shape[1], d.
↪shape[2], d.shape[3], d.shape[4]))
        pred += [prediction.cpu().numpy()]
        print("  Sequence %d finished" % s)


print("Sampling complete!\n")

gt = np.concatenate(gt, axis=1)
pred = np.concatenate(pred, axis=1)

print("Ground truth and prediction tensor with shape:")
print("(samples, sequences, sequenceLength, channels, sizeX, sizeY)")
print("GT: %s" % str(gt.shape))
print("Prediction: %s" % str(pred.shape))
```

```
Starting sampling...
  Sequence 0 finished
  Sequence 1 finished
Sampling complete!

Ground truth and prediction tensor with shape:
(samples, sequences, sequenceLength, channels, sizeX, sizeY)
GT: (1, 2, 60, 5, 128, 64)
Prediction: (5, 2, 60, 5, 128, 64)
```

29.9 Accuracy of the Prediction

After the sampling, we can analyze the ground truth flow trajectory and the samples generated by ACDM. Let's start with a direct visualization of the predictions for a qualitative check.

```
sequence = 0
samples = [0,4]
timeSteps = [0,19,39,59]
field = 3 # velocity_x (0), velocity_y (1), density (2), or pressure (3)

predPart = pred[samples]
gtPred = np.concatenate([gt[:,sequence,timeSteps,field], predPart[:,sequence,
 ↪timeSteps,field]])

fig, axs = plt.subplots(nrows=gtPred.shape[0], ncols=gtPred.shape[1], figsize=(gtPred.
 ↪shape[1]*1.9, gtPred.shape[0]), dpi=150, squeeze=False)

for i in range(gtPred.shape[0]):
    for j in range(gtPred.shape[1]):
        if i == gtPred.shape[0]-1:
            axs[i,j].set_xlabel("$t=%s$" % (timeSteps[j]+1), fontsize=10)
        if j == 0:
            if i == 0:
              axs[i,j].set_ylabel("Ground\nTruth", fontsize=10)
            else:
              axs[i,j].set_ylabel("ACDM\nSample %d" % i, fontsize=10)
        axs[i,j].set_xticks([])
        axs[i,j].set_yticks([])
        im = axs[i,j].imshow(gtPred[i][j].transpose(), interpolation="catrom", cmap=
 ↪"viridis")

plt.show()
```



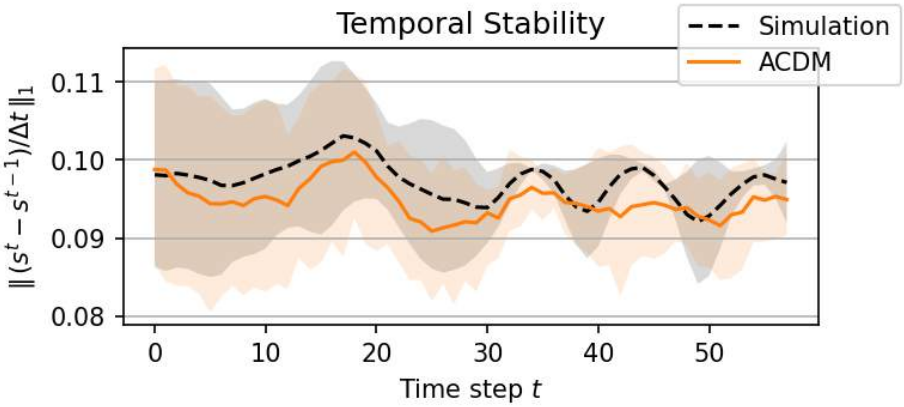
The trained time operator closely matches early states, but you should be able to see variations for the last states at $t = 60$. The shock waves for the cylinder are highly unstable, and hence give the network to create realistic but varying predictions. Re-running inference with different noise values will produce additional variations, and longer rollouts will further increase differences.

29.9.1 Temporal Stability

We also investigate the temporal stability of the samples by computing a temporal derivative, and comparing the result to the simulation. Note that the result will be smoother, the more sequences and samples are used. Furthermore, better results can be achieved with additional training data. Naturally, the ACDM posterior samples should exhibit a larger variance compared to the individual simulation trajectories.

```
gtTemp = gt[:,:,:,0:4] # ignore scalar Mach number here
predTemp = pred[:,:,:,0:4]

diffGt = np.abs( gtTemp[:,:,1:gtTemp.shape[2]-1] - gtTemp[:,:,2:gtTemp.shape[2]])
diffGt = np.mean(diffGt, axis=(3,4,5)) # channel-wise and spatial mean
minGt = np.min(diffGt, axis=(0,1)) # lower bound over sequences
maxGt = np.max(diffGt, axis=(0,1)) # upper bound over sequences
meanGt = np.mean(diffGt, axis=(0,1)) # sample- and sequence mean

diffPred = np.abs( predTemp[:,:,1:predTemp.shape[2]-1] - predTemp[:,:,2:predTemp.
↪shape[2]])
diffPred = np.mean(diffPred, axis=(3,4,5)) # channel-wise and spatial mean
minPred = np.min(diffPred, axis=(0,1)) # lower bound over samples and sequences
maxPred = np.max(diffPred, axis=(0,1)) # upper bound over samples and sequences
meanPred = np.mean(diffPred, axis=(0,1)) # sample- and sequence mean


fig, ax = plt.subplots(1, figsize=(5,2), dpi=150)
ax.set_title("Temporal Stability")
ax.set_ylabel("$\Vert \, (s^{t} - s^{t-1}) / \Delta t \, \Vert_1$")
ax.yaxis.grid(True)
ax.set_xlabel("Time step $t$")

ax.plot(np.arange(meanGt.shape[0]), meanGt, color="k", label="Simulation", linestyle=
↪"dashed")
ax.fill_between(np.arange(meanGt.shape[0]), minGt, maxGt, facecolor="k", alpha=0.15)

ax.plot(np.arange(meanPred.shape[0]), meanPred, color="tab:orange", label="ACDM")
ax.fill_between(np.arange(meanPred.shape[0]), minPred, maxPred, facecolor="tab:orange
↪", alpha=0.15)

fig.legend()
plt.show()
```

```
<>:19: SyntaxWarning: invalid escape sequence '\V'
<>:19: SyntaxWarning: invalid escape sequence '\V'
/tmp/ipykernel_917142/1134808229.py:19: SyntaxWarning: invalid escape sequence '\V'
  ax.set_ylabel("$\Vert \, (s^{t} - s^{t-1}) / \Delta t \, \Vert_1$")
```



29.9.2 Spectral Analysis

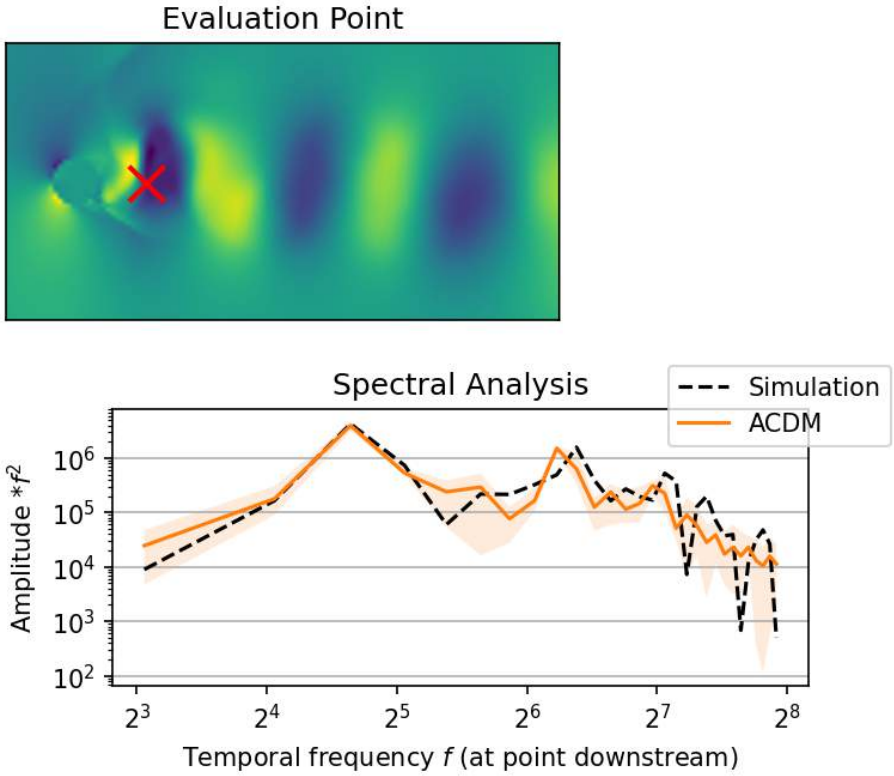
Finally, let's compute a frequency analysis on a point downstream of the cylinder and compare the spectra of ground truth and ACDM prediction.

```
sequence = 0
fracX = 0.25 # closely behing the cylinder
fracY = 0.5 # vertically centered
field = 1 # velocity_x (0), velocity_y (1), density (2), or pressure (3)

posX = int(fracX * gt.shape[4])
posY = int(fracY * gt.shape[5])

gtPred = np.concatenate([gt[:,sequence,:,field, posX, posY], pred[:,sequence,:,field,␣
↪posX, posY]])

fft = np.fft.fft(gtPred, axis=1)
fft = np.real(fft * np.conj(fft))
n = fft.shape[1]
gridSpacing = 0.002 # delta t between frames from simulation
freq = np.fft.fftfreq(n, d=gridSpacing)[1:int(n/2)]
fft = fft[:,1:int(n/2)] # only use positive fourier frequencies

gtFFT = fft[0]
minPredFFT = np.min(fft[1:], axis=0) # lower bound over samples
maxPredFFT = np.max(fft[1:], axis=0) # upper bound over samples
meanPredFFT = np.mean(fft[1:], axis=0) # sample mean


# plot eval point
fig, ax = plt.subplots(1, figsize=(5,2), dpi=150)
ax.set_title("Evaluation Point")
ax.imshow(gt[0,sequence,0,field].transpose(), interpolation="catrom", cmap="viridis")
ax.scatter(posX, posY, s=200, color="red", marker="x", linewidth=2)
ax.set_xticks([])
ax.set_yticks([])
plt.show()


# plot spectral analysis
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1, figsize=(5,2), dpi=150)
ax.set_title("Spectral Analysis")
ax.set_xlabel("Temporal frequency $f$ (at point downstream)")
ax.set_ylabel("Amplitude $*f^2$")
ax.set_xscale("log", base=2)
ax.set_yscale("log", base=10) # NOTE: y-axis values are not physical as data␣
↪normalization is not reversed
ax.yaxis.grid(True)

ax.plot(freq, gtFFT * (freq**2), color="k", label="Simulation", linestyle="dashed")

ax.plot(freq, meanPredFFT * (freq**2), color="tab:orange", label="ACDM")
ax.fill_between(freq, minPredFFT * (freq**2), maxPredFFT * (freq**2), facecolor=
↪"tab:orange", alpha=0.15)

fig.legend()
plt.show()
```



The plot of the spectrum shows how the trained ACDM network matches the ground truth simulation (dashed black line) both in the low- and high-frequency domain. The simulation has a few downward spikes at the end which are mostly smoothed out by the learned predictor, but it captures the statistics of the ground truth very well.

29.10 Summarizing Time Predictions with Diffusion Models

To conclude the results from above, this code has yielded a probabilistic model for time predictions of PDEs. The great thing about it is that it estimates the changes and uncertainties in the dataset in order to reproduce it at inference time. Hence it provides posterior sampling over time, and can be run multiple times to infer different possible solutions.

The flip side here is that diffusion models are generally not better at predicting the mean solution than classic methods (see the ACDM benchmark for detailed evaluations). Thus, if the input-output relationship in your data is unique, diffusion models will not pay off, and only incur higher inference computations. This holds for the networks above: they are more expensive, and are run repeatedly to produce a single sample. This could be sped up more (e.g. with flow matching, the model above uses denoising), but a certain (small) factor will remain.

Nonetheless, for most non-trivial datasets diffusion models will pay off: ambiguities in the data will **not be averaged out**, but treated (and reproduced) as a **distribution**. In addition, as hinted at mentioned above, a highly interesting aspect of the diffusion-based time prediction is its **unconditional stability**. Given an appropriate learning task, the trained models do not blow up over time or transform the input into trivial steady states. Both cases are common in models trained with other training methodologies. Rather, the diffusion-based networks can retain the statistics of the reference data over arbitrarily long rollouts. It's difficult to prove that they *never* diverge, but in our tests stable networks did not diverge over the course of several hundred thousand rollout steps. This is a highly attractive behavior, and indicates a fundamentally different behavior of diffusion-based models. In the next chapter we'll provide more details, and investigate it in comparison with temporal *unrolling*.

CHAPTER

THIRTY

UNCONDITIONAL STABILITY

The results of the previous section, for time predictions with diffusion models, and earlier ones (*Discussion of Differentiable Physics*) make it clear that unconditionally stable networks are definitely possible. This has also been reported various other works. However, there's still a fair amount of approaches that seem to have trouble with long term stability. This poses a very interesting question: which ingredients are necessary to obtain *unconditional stability*? Unconditional stability here means obtaining trained networks that are stable for arbitrarily long rollouts. Are inductive biases or special training methodologies necessary, or is it simply a matter of training enough different initializations? Our setup provides a very good starting point to shed light on this topic.

The "success stories" from earlier chapters, some with fairly simple setups, indicate that unconditional stability is "nothing special" for neural network based predictors. I.e., it does not require special loss functions or tricks beyond a proper learning setup (suitable hyperparameters, sufficiently large model plus enough data). As errors will accumulate over time, we can expect that network size and the total number of update steps in training are important. Interestingly, it seems that the neural network architecture doesn't really matter: we can obtain stable rollouts with pretty much "any" architecture once it's sufficiently large.

Note that we'll focus on time steps with a **fixed length** in the following. The "unconditional stability" refers to being stable over an arbitrary number of iterative steps. The following networks could potentially trained for variable time step sizes as well, but we will focus on the "dimension" of stability of multiple, iterative network calls below.

30.1 Main Considerations for an Evaluation

As shown in the previous chapter, diffusion models perform extremely well. This can be attributed to the underlying task of working with pure noise as input (e.g., for denoising or flow matching tasks). Likewise, the network architecture has only a minor influence: the network simply needs to be large enough to provide a converging iteration. For supervised or unrolled training, we can leverage a variety of discrete and continuous neural operators. CNNs, Unets, FNOs and Transformers are popular approaches here. Interestingly, FNOs, due to their architecture *project* the solution onto a subspace of the frequencies in the discretization. This inherently removes high frequencies that primarily drive instabilities. As such, they're influenced by unrolling to a lesser extent (details can be found, e.g., here). Operators that better preserve small-scale details, such as convolutions, can strongly benefit from unrolling. This will be a focus of the following ablations.

Interestingly, it turns out that the batch size and the length of the unrolling horizon play a crucial but conflicting role: small batches are preferable, but in the worst case under-utilize the hardware and require long training runs. Unrolling on the other hand significantly stabilizes the rollout, but leads to increased resource usage due to the longer computational graph for each NN update. Thus, our experiments show that a "sweet spot" along the Pareto-front of batch size vs unrolling horizon can be obtained by aiming for as-long-as-possible rollouts at training time in combination with a batch size that sufficiently utilizes the available GPU memory.

Learning Task: To analyze the temporal stability of autoregressive networks on long rollouts, two flow prediction tasks from the ACDM benchmark are considered: an easier incompressible cylinder flow (denoted by *Inc*), and a complex transonic wake flow (denoted by *Tra*) at Reynolds number 10 000. For Inc, the networks are trained on flows with Reynolds number 200 – 900 and required to extrapolate to Reynolds numbers of 960, 980, and 1000 during inference (*Inc-high*). For Tra, the training data consists of flows with Mach numbers between 0.53 and 0.9, and networks are tested on the Mach numbers 0.50, 0.51, and 0.52 (denoted by *Tra-ext*). This Mach number is tough as it contains a substantial amounts of shocks that interact with the flow. For each sequences in both data sets, three training runs of each architecture are unrolled over 200.000 steps. This unrolling length is no proof that these networks yield infinitely long stable rollouts, but they feature an extremely small probability for blowups.

30.2 Comparing Architectures

As a first comparison, we'll train three network architectures with an identical U-Net architecture, that use different stabilization techniques. This comparison shows that it is possible to successfully achieve the task "unconditional stability" in different ways:

- Unrolled training (*U-Net-ut*) where gradients are backpropagated through multiple time steps during training.
- Networks trained on a single prediction step with added training noise (*U-Net-tn*). This technique is known to improve stability by reducing data shift, as the added noise emulates errors that accumulate during inference.
- Autoregressive conditional diffusion models (ACDM). A denoising diffusion model is conditioned on the previous time step and iteratively refines noise to create a prediction for the next step, as shown in *Diffusion-based Time Prediction*.

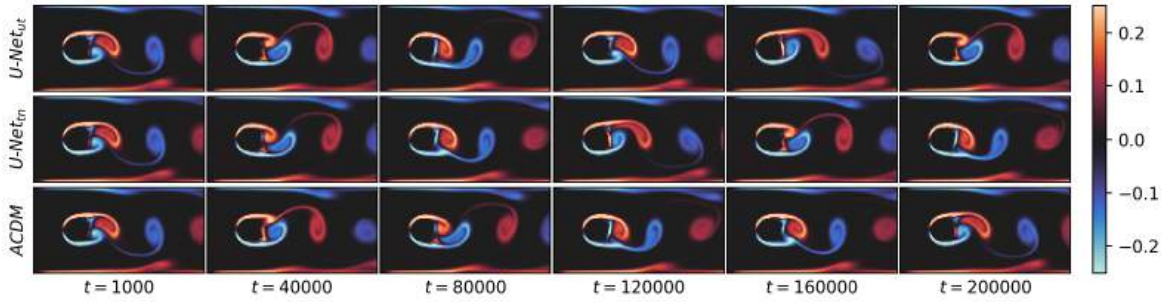


Fig. 30.1: Vorticity predictions for an incompressible flow with a Reynolds number of 1000 over 200 000 time steps (Inc-high).

The figure above illustrates the resulting predictions. All methods and training runs remain unconditionally stable over the entire rollout on Inc-high. Since this flow is unsteady but fully periodic, the results of all networks are simple, periodic trajectories that prevent error accumulation. This example serves to show that for simpler tasks, long term stability is less of an issue. Networks have a relatively easy time to keep their predictions within the manifold of the solutions. Let's consider a tougher example: the transonic flows with shock waves in Tra.

For the test sequences from Tra-ext, one from the three trained U-Net-tn networks has stability issues within the first few thousand steps. This network deteriorates to a simple, mean flow prediction without vortices. Unrolled training (U-Net-ut) and diffusion models (ACDM), on the other hand, are fully stable across sequences and training runs for this case, indicating a higher resistance to rollout errors which normally cause instabilities. The autoregressive diffusion models turn out to be unconditionally stable across the board (details here), so we'll drop them in the following evaluations and focus on models where stability is more difficult to achieve: the U-Nets, as representatives of convolutional, discrete neural operators.

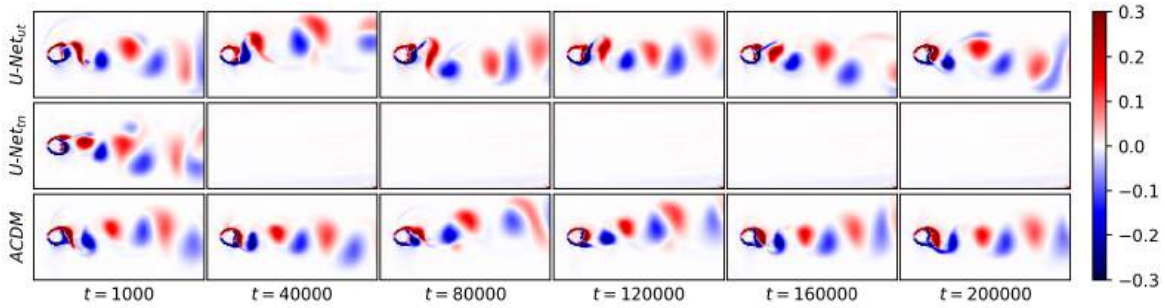


Fig. 30.2: Vorticity predictions for transonic flows with a Mach number 0.52 (Tra-ext, outside the trainig data range) over 200 000 time steps.

30.3 Stability Criteria

Focusing on the U-Net networks with unrolled training, we will next focus on training multiple models (3 each time), and measure the percentage of stable runs they achieve. This provides more thorough statistics compared to the single, qualitative examples above. We'll investigate the first key criteria rollout length, to show how it influences fully stable rollouts over extremely long horizons. Figure 2 lists the percentage of stable runs for a range of ablation networks on the Tra-ext data set with rollouts over 200 000 time steps. Results on the individual Mach numbers, as well as an average (top row) are shown.

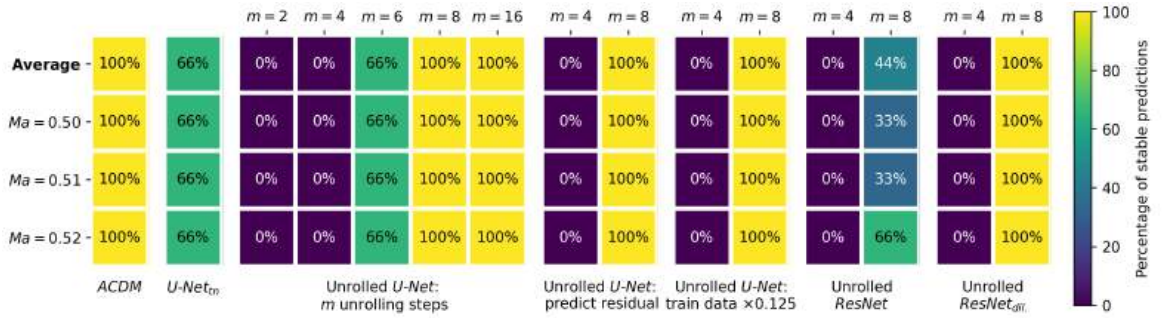


Fig. 30.3: Percentage of stable runs on the Tra-ext data set for different ablations of unrolled training.

The different generalization test over Mach numbers make no difference. The most important criterion for stability is the number of unrolling steps m: while networks with m <= 4 consistently do not achieve stable rollouts, using m >= 8 is sufficient for stability across different Mach numbers.

Negligible Aspects: Three factors that did not substantially impact rollout stability in experiments are the prediction strategy, the amount of training data, and the backbone architecture. We'll only briefly summarize the results here. First, using residual predictions, i.e., predicting the difference to the previous time step instead of the full time steps itself, did not impact stability. Second, the stability is not affected when reducing the amount of available training data by a factor of 8, from 1000 time steps per Mach number to 125 steps (while training with 8× more epochs to ensure a fair comparison). This training data reduction still retains the full physical behavior, i.e., complete vortex shedding periods. Third, it possible to train other backbone architectures with unrolling to achieve fully stable rollouts as well, such as dilated ResNets. For ResNets without dilations only one trained network is stable, most likely due to the reduced receptive field. However, we expect achieving full stability is also possible with longer training rollout horizons.

30.4 Batch Size vs Rollout

Interestingly, the batch size turns out to be an important factor: it can substantially impact the stability of autoregressive networks. This is similar to the image domain, where smaller batches are know to improve generalization (this is the motivation for using mini-batching instead of gradients over the full data set). The impact of the batch size on the stability and training time is shown in the figure below, for both investigated data sets. Networks that only come close to the ideal rollout length at a large batch size, can be stabilized with smaller batches. However, this effect does not completely remove the need for unrolled training, as networks without unrolling were unstable across all tested batch sizes. For the Inc case, the U-Net width was reduced by a factor of 8 across layers (in comparison to above), to artificially increase the difficulty of this task. Otherwise all parameter configurations would already be stable and show the effect of varying the batchsize.

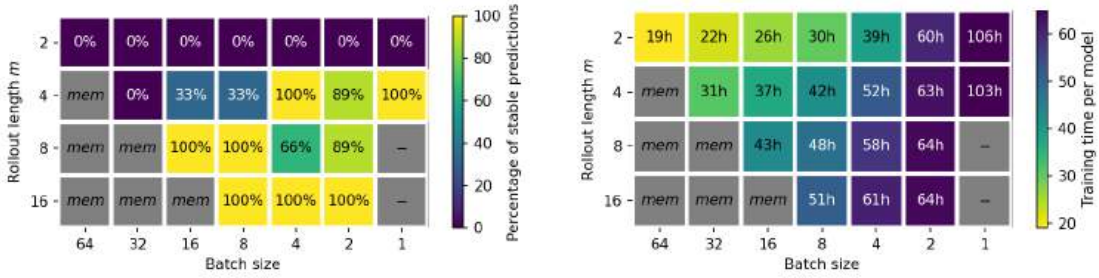


Fig. 30.4: Percentage of stable runs and training time for different combinations of rollout length and batch size for the Tra-ext data set. Grey configurations are omitted due to memory limitations (mem) or due to high computational demands (-).

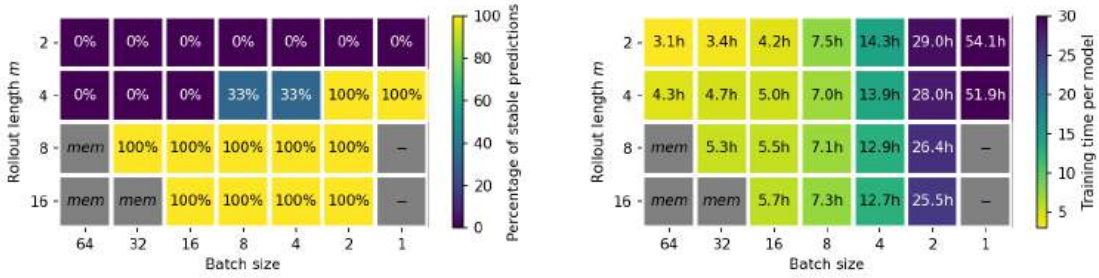


Fig. 30.5: Percentage of stable runs and training time for rollout length and batch size for the Inc-high data set. Grey again indicates out-of-memory (mem) or overly high computational demands (-).

This shows that increasing the batch size is more expensive in terms of training time on both data sets, due to less memory efficient computations. Using longer rollouts during training does not necessarily induce longer training times, as we compensate for longer rollouts with a smaller number of updates per epoch. E.g., we use either 250 batches with a rollout of 4, or 125 batches with a rollout of 8. Thus the number of simulation states that each network sees over the course of training remains constant. However, we did in practice observe additional computational costs for training the larger U-Net network on Tra-ext. This leads to the "central" question in these ablations: which combination of rollout length and batch size is most efficient?

The figure above answers this question by showing the central tradeoff between rollout length and batch size (only stable versions are included here). To achieve *unconditionally stable* networks and neural operators, it is consistently beneficial to choose configurations where large rollout lengths are paired with a batch size that is big enough the sufficiently utilize the available GPU memory. This means, improved stability is achieved more efficiently with longer training rollouts rather than smaller batches, as indicated by the green dots with the lowest training times.

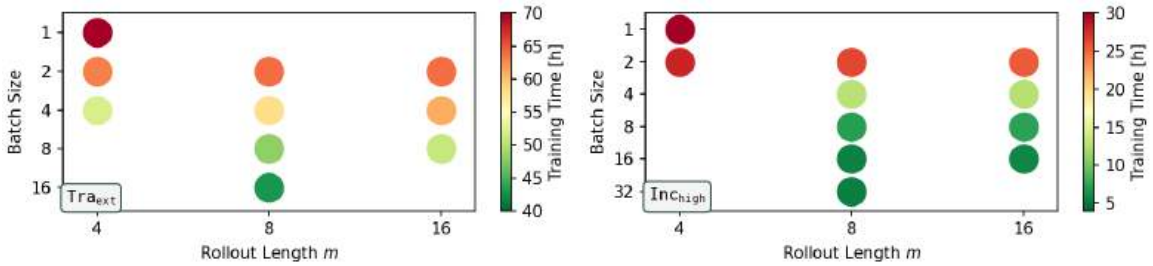


Fig. 30.6: Training time for different combinations of rollout length and batch size to on the Tra-ext data set (left) and the Inc-high data set (right). Only configurations that to lead to highly stable networks (stable run percentage >= 89%) are shown.

30.5 Summary

To conclude the results above: With a suitable training setup, unconditionally stable predictions with extremely long rollout are clearly possible, even for complex flows. According to the experiments, the most important factors that impact stability are the decision for or against diffusion-based training

Without diffusion, several factors need to be considered:

- Long rollouts at training time
- Small batch sizes
- Comparing these two factors: longer rollouts are preferable, and result in faster training times than smaller batch sizes
- At the same time, sufficiently large networks are necessary (this depends on the complexity of the learning task).

Factors that did not substantially impact long-term stability are:

- Prediction paradigm during training, i.e., residual and direct prediction are viable
- Additional training data without new physical behavior
- Different network architectures, although the ideal number of unrolling steps might vary for each architecture

This concludes the topic of “unconditional stability”. Further details of these experiments can be found in the ACDM paper

CHAPTER

THIRTYONE

GRAPH-BASED DIFFUSION MODELS

Similar to classical numerics, regular grids are ideal for certain situations, but sub-optimal for others. Diffusion models are no different, but luckily the concepts of the previous sections do carry over when replacing the regular grids with graphs. Importantly, denoising and flow matching work similarly well on unstrucuted Eulerian meshes, as will be demonstrated below. This test case will illustrate another important aspect: diffusion models excel at *completing* data distributions. I.e., even when the training data has an incomplete distribution for a single example (defined by the geometry of the physical domain, boundary conditions and physical parameters), the "global" view of learning from different examples let's the networks *complete* the posterior distribution over the course of seeing partial data for many different examples.

Most simulation problems like fluid flows are often poorly represented by a single mean solution. E.g., for many practical applications involving turbulence, it is crucial to **access the full distribution of possible flow states**, from which relevant statistics (e.g., RMS and two-point correlations) can be derived. This is where diffusion models can leverage their strengths: instead of having to simulate a lengthy transient phase to converge towards an equilibrium state, diffusion models can completely skip the transient warm-up, and directly produce the desired samples. Hence, this allows for computing the relevant flow statistics very efficiently compared to classic solvers.

31.1 Diffusion Graph Net (DGN)

In the following, we'll demonstrate these capabilities based on the *diffusion graph net* (DGN) approach [LPT25], the full source code for which can be found here.

To learn the probability distribution of dynamical states of physical systems, defined by their discretization mesh and their physical parameters, the DDPM and flow matching frameworks can directly be applied to the mesh nodes. Additionally, DGN introduces a second model variant, which operates in a pre-trained semantic *latent space* rather than directly in the physical space (this variant will be called LDGN).

In contrast to relying on regular grid discretizations as in previous sections, the system's geometry is now represented using a mesh with nodes $\mathcal{V}_M$ and edges $\mathcal{E}_M$, where each node i is located at x_i. The system's state at time t, $Y(t)$, is defined by F continuous fields sampled at the mesh nodes: $Y(t) \coloneqq \{y_i(t) \in \mathbb{R}^F \mid i \in \mathcal{V}_M\}$, with the short form $y_i(t) \equiv y(x_i, t)$. Simulators evolve the system through a sequence of states, $\mathcal{Y} = \{Y(t_0), Y(t_1), \dots, Y(t_n), \dots\}$, starting from an initial state $Y(t_0)$. We assume that after an initial transient phase, the system reaches a statistical equilibrium. In this stage, statistical measures of Y, computed over sufficiently long time intervals, are time-invariant, even if the dynamics display oscillatory or chaotic behavior. The states in the equilibrium stage, $\mathcal{Z} \subset \mathcal{Y}$, depend only on the system's geometry and physical parameters, and not on its initial state. This is illustrated in the following picture.

In many engineering applications, such as aerodynamics and structural vibrations, the primary focus is not on each individual state along the trajectory, but rather on the statistics that characterize the system's dynamics. However, simulating a trajectory of converged states $\mathcal{Z}$ long enough to accurately capture these statistics can be very expensive, especially for real-world problems involving 3D chaotic systems. The following DGN approachs aims for directly sampling converged states $Z(t) \in \mathcal{Z}$ without simulating the initial transient phase. Subsequently, we can analyze the system's dynamics by drawing enough samples.

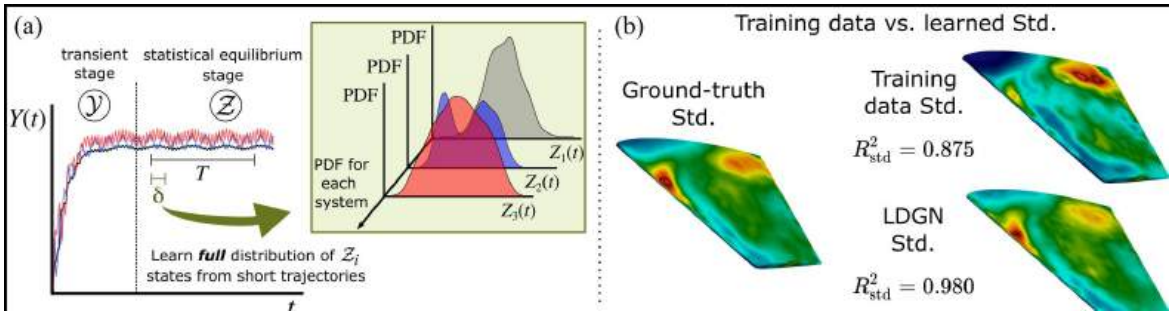


Fig. 31.1: (a) DGN learns the probability distribution of the systems' converged states provided only a short trajectory of length $\delta << T$ per system. (b) An example with a turbulent wing experiment. The distribution learned by the LDGN model accurately captures the variance of all states (bottom right), despite seeing only an incomplete distribution for each wing during training (top right).

Given a dataset of short trajectories from N systems, $\mathfrak{Z} = \{\mathcal{Z}_1, \mathcal{Z}_2, ..., \mathcal{Z}_N\}$, the goal in the following is to learn a probabilistic model of $\mathfrak{Z}$ that enables sampling of a converged state $Z(t) \in \mathcal{Z}$, conditioned on the system's mesh, boundary conditions, and physical parameters. This model must capture the underlying probability distributions even when trained on trajectories that are too short to fully characterize their individual statistics. Although this is an ill-posed problem, given sufficient training trajectories, diffusion models on graphs manage to uncover the statistical correlations and shared patterns, enabling interpolation across the condition space.

31.2 Diffusion on Graphs

We'll use DDPM (and later flow matching) to generate states $Z(t)$ by denoising a sample $Z^R \in \mathbb{R}^{|\mathcal{V}_M| \times F}$ drawn from an isotropic Gaussian distribution. The system's conditional information is encoded in a directed graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} \equiv \mathcal{V}_M$ and the mesh edges $\mathcal{E}_M$ are represented as bi-directional graph edges $\mathcal{E}$. Node attributes $V_c = \{v_i^c \mid i \in \mathcal{V}\}$ and edge attributes $E_c = \{e_{ij}^c \mid (i,j) \in \mathcal{E}\}$ encode the conditional features, including the relative positions between adjacent node, $x_j - x_i$.

In the *diffusion* (or *forward*) process, node features from $Z^1 \in \mathbb{R}^{|\mathcal{V}| \times F}$ to $Z^R \in \mathbb{R}^{|\mathcal{V}| \times F}$ are generated by sequentially adding Gaussian noise: $q(Z^r | Z^{r-1}) = \mathcal{N}(Z^r; \sqrt{1-\beta_r} Z^{r-1}, \beta_r \mathbf{I})$, where $\beta_r \in (0,1)$, and $Z^0 \equiv Z(t)$. Any Z^r can be sampled directly via:

$$Z^r = \sqrt{\bar{\alpha}_r} Z^0 + \sqrt{1-\bar{\alpha}_r}\epsilon,$$

with $\alpha_r := 1 - \beta_r$, $\bar{\alpha}_r := \prod_{s=1}^{r} \alpha_s$ and $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The denoising process removes noise through learned Gaussian transitions: $p_\theta(Z^{r-1} | Z^r) = \mathcal{N}(Z^{r-1}; \mu_\theta^r, \Sigma_\theta^r)$, where the mean and variance are parameterized as:

$$\mu_\theta^r = \frac{1}{\sqrt{\alpha_r}} \left(Z^r - \frac{\beta_r}{\sqrt{1-\bar{\alpha}_r}} \epsilon_\theta^r \right), \qquad \Sigma_\theta^r = \exp\left(\mathbf{v}_\theta^r \log \beta_r + (1 - \mathbf{v}_\theta^r) \log \tilde{\beta}_r \right),$$

with $\tilde{\beta}_r := (1-\bar{\alpha}_{r-1})/(1-\bar{\alpha}_r)\beta_r$. Here, $\epsilon_\theta^r \in \mathbb{R}^{|\mathcal{V}| \times F}$ predicts the noise ϵ in equation (1), and $\mathbf{v}_\theta^r \in \mathbb{R}^{|\mathcal{V}| \times F}$ interpolates between the two bounds of the process' entropy, β_r and $\tilde{\beta}_r$.

DGNs predict ϵ_θ^r and $\mathbf{v}_\theta^r$ using a regular message-passing-based GNN [SGGP+20]. This takes Z^{r-1} as input, and it is conditioned on the graph $\mathcal{G}$, its node and edge features, and the diffusion step r:

$$[\epsilon_\theta^r, \mathbf{v}_\theta^r] \leftarrow \text{DGN}_\theta(Z^{r-1}, \mathcal{G}, V_c, E_c, r).$$

The *DGN* network is trained using the hybrid loss function proposed in *"Improved Denoising Diffusion Probabilistic Models"* by Nichol et al. The full denoising process requires R evaluations of the DGN to transition from Z^R to Z^0.

DGN follows the widely used encoder-processor-decoder GNN architecture. In addition to the node and edge encoders, the encoder includes a diffusion-step encoder, which generates a vector $r_{\text{emb}} \in \mathbb{R}^{F_{\text{emb}}}$ that embeds the diffusion step r. The node encoder processes the conditional node features v_i^c, alongside r_{emb}. Specifically, the diffusion-step encoder and the node encoder operate as follows:

$$r_{\text{emb}} \leftarrow \phi \circ \text{Linear} \circ \text{SinEmb}(r), \quad v_i \leftarrow \text{Linear}\left([\phi \circ \text{Linear}(v_i^c) \mid r_{\text{emb}}]\right), \quad \forall i \in \mathcal{V},$$

where ϕ denotes the activation function and SinEmb is the sinusoidal embedding function. The edge encoder applies a linear layer to the conditional edge features e_{ij}^c. The encoded node and edge features are $\mathbb{R}^{F_h}$-dimensional vectors ($F_{\text{emb}} = 4 \times F_h$). We condition each message-passing layer on r by projecting r_{emb} to an F_h-dimensional space and adding the result to the node features before each of these layers — i.e., $v_i \leftarrow v_i + \text{Linear}(r_{\text{emb}})$. Each message-passing layer follows:

$$\begin{aligned}
\mathbf{e}_{ij} &\leftarrow W_e \mathbf{e}_{ij} + \text{MLP}^e\left(\text{LN}\left([\mathbf{e}_{ij}|\mathbf{v}_i|\mathbf{v}_j]\right)\right), \qquad \forall (i,j) \in \mathcal{E}, \\
\bar{\mathbf{e}}_j &\leftarrow \sum_{i \in \mathcal{N}_j^-} \mathbf{e}_{ij}, \qquad \forall j \in \mathcal{V}, \\
\mathbf{v}_j &\leftarrow W_v \mathbf{v}_j + \text{MLP}^v\left(\text{LN}\left([\bar{\mathbf{e}}_j|\mathbf{v}_j]\right)\right), \qquad \forall j \in \mathcal{V}.
\end{aligned}$$

Previous work on graph-based diffusion models has used sequential message passing to propagate node features across the graph. However, this approach fails for large-scale phenomena, such as the flows studied in the context of DGN, as denoising of global features becomes bottlenecked by the limited reach of message passing. To address this, a multi-scale GNN is adopted for the processor, applying message passing on $\mathcal{G}$ and multiple coarsened versions of it in a U-Net fashion. This design leverages the U-Net's effectiveness in removing both high- and low-frequency noise. To obtain each lower-resolution graph from its higher-resolution counterpart, we use Guillard's coarsening algorithm, originally developed for fast mesh coarsening in CFD applications. As in the conventional U-Net, pooling and unpooling operations, now based on message passing, are used to transition between higher- and lower-resolution graphs.

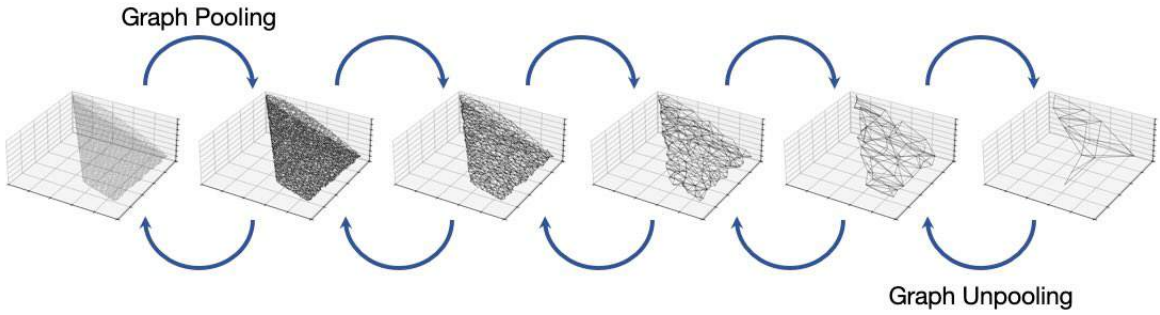


Fig. 31.2: Message passing is applied on $\mathcal{G}$ and multiple coarsened versions of it in a U-Net fashion. The lower-resolution graphs are obtained using a mesh coarsening algorithm popularised in CFD applications.

31.3 Diffusion in Latent Space

Diffusion models can also operate in a lower-dimensional graph-based representation that is perceptually equivalent to $\mathfrak{Z}$. This space is defined as the latent space of a Variational Graph Auto-Encoder (VGAE) trained to reconstruct $Z(t)$. We'll refer to a DGN trained on this latent space as a Latent DGN (LDGN).

In this configuration, the VGAE captures high-frequency information (e.g., spatial gradients and small vortices), while the LDGN focuses on modeling mid- to large-scale patterns (e.g., the wake and vortex street). By decoupling these two tasks, the generative learning process is simplified, allowing the LDGN to concentrate on more meaningful latent representations that are less sensitive to small-scale fluctuations. Additionally, during inference, the VGAE's decoder

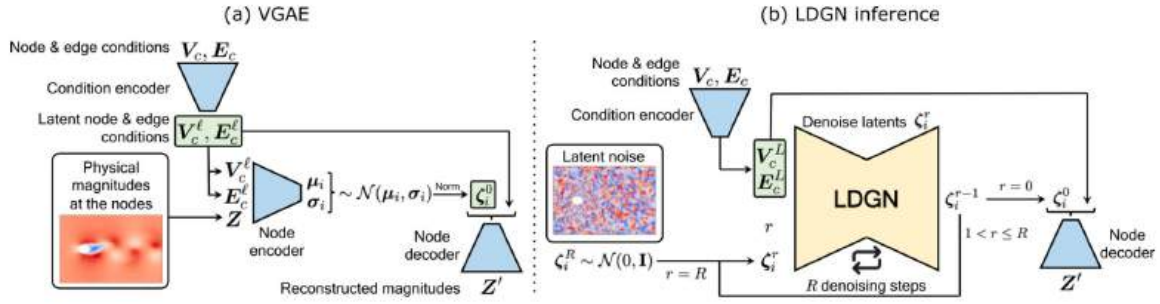


Fig. 31.3: (a) The VGAE consists of a condition encoder, a (node) encoder, and a (node) decoder. The multi-scale latent features from the condition encoder serve as conditioning inputs to both the encoder and the decoder. (b) During LDGN inference, Gaussian noise is sampled in the VGAE latent space and, after multiple denoising steps conditioned on the low-resolution outputs from the VGAE's condition encoder, transformed into the physical space by the VGAE's decoder.

helps remove residual noise from the samples generated by the LDGN. This approach significantly reduces sampling costs since the LDGN operates on a smaller graph rather than directly on $\mathcal{G}$.

For the VGAE, an encoder-decoder architecture is used with an additional condition encoder to handle conditioning inputs. The condition encoder processes V_c and E_c, encoding these into latent node features V_c^ℓ and edge features E_c^ℓ across L graphs $\{\mathcal{G}^\ell := (\mathcal{V}^\ell, \mathcal{E}^\ell) Id1 \leq \ell \leq L\}$, where $\mathcal{G}^1 \equiv \mathcal{G}$ and the size of the graphs decreases progressively, i.e., $|\mathcal{V}^1| > |\mathcal{V}^2| > \cdots > |\mathcal{V}^L|$. This transformation begins by linearly projecting V_c and E_c to a F_{ae}-dimensional space and applying two message-passing layers to yield V_c^1 and E_c^1. Then, $L-1$ encoding blocks are applied sequentially:

$$\left[V_c^{\ell+1}, E_c^{\ell+1}\right] \leftarrow MP \circ MP \circ \text{GraphPool}\left(V_c^\ell, E_c^\ell\right), \quad \text{for } l = 1, 2, \ldots, L-1,$$

where *MP* denotes a message-passing layer and *GraphPool* denotes a graph-pooling layer.

The encoder produces two F_L-dimensional vectors for each node $i \in \mathcal{V}^L$, the mean μ_i and standard deviation σ_i that parametrize a Gaussian distribution over the latent space. It takes as input a state $Z(t)$, which is linearly projected to a F_{ae}-dimensional vector space and then passed through $L-1$ sequential down-sampling blocks (message passing + graph pooling), each conditioned on the outputs of the condition encoder:

$$V \leftarrow \text{GraphPool} \circ MP \circ MP\left(V + \text{Linear}\left(V_c^\ell\right), \text{Linear}\left(E_c^\ell\right)\right), \text{ for } l = 1, 2, \ldots, L-1;$$

and a bottleneck block:

$$V \leftarrow MP \circ MP\left(V + \text{Linear}\left(V_c^L\right), \text{Linear}\left(E_c^L\right)\right).$$

The output features are passed through a node-wise MLP that returns μ_i and σ_i for each node $i \in \mathcal{V}^L$. The latent variables are then computed as $\zeta_i = \text{BatchNorm}(\mu_i + \sigma_i \epsilon_i)$, where $\epsilon_i \sim \mathcal{N}(0, I)$. Finally, the decoder mirrors the encoder, employing a symmetric architecture (replacing graph pooling by graph unpooling layers) to upsample the latent features back to the original graph $\mathcal{G}$. Its blocks are also conditioned on the outputs of the condition encoder. The message passing and the graph pooling and unpooling layers in the VGAE are the same as in the (L)DGN.

The VGAE is trained to reconstruct states $Z(t) \in \mathfrak{Z}$ with a KL-penalty towards a standard normal distribution on the learned latent space. Once trained, the LDGN can be trained following the approach in Section~\ref{sec:DGN}. However, the objective is now to learn the distribution of the latent states ζ, defined on the coarse graph $\mathcal{G}^L$, conditioned on the outputs V_c^L and E_c^L from the condition encoder. During inference, the condition encoder generates the conditioning features V_c^ℓ and E_c^ℓ (for $l = 1, 2, \ldots, L$), and after the LDGN completes its denoising steps, the decoder transforms the generated ζ_0 back into the physical feature-space defined on $\mathcal{G}$.

Unlike in conventional VGAEs, the condition encoder is necessary because, at inference time, an encoding of V_c and E_c is needed on graph $\mathcal{G}^L$, where the LDGN operates. This encoding cannot be directly generated by the encoder, as

it also requires $Z(t)$ as input, which is unavailable during inference. An alternative approach would be to define the conditions directly in the coarse representation of the system provided by $\mathcal{G}^L$, but this representation lacks fine-grained details, leading to sub-optimal results.

31.4 Turbulent Flows around Wings in 3D

Let's directly turn to a complex case to illustrate the capabilities of DGN. (A more basic case will be studied in the Jupyter notebook on the following page.)

The Wing experiments of the DGN project target wings in 3D turbulent flow, characterized by detailed vortices that form and dissipate on the wing surface. This task is particularly challenging due to the high-dimensional, chaotic nature of turbulence and its inherent multi-scale interactions across a wide range of scales. The geometry of the wings varies in terms of relative thickness, taper ratio, sweep angle, and twist angle. These simulations are computationally expensive, and using GNNs allows us to concentrate computational effort on the wing's surface, avoiding the need for costly volumetric fields. A regular grid around the wing would require over 10^5 cells, in contrast to approximately 7,000 nodes for the surface mesh representation. The surface pressure can be used to determine both the aerodynamic performance of the wing and its structural requirements. Fast access to the probabilistic distribution of these quantities would be highly valuable for aerodynamic modeling tasks. The training dataset for this task was generated using Detached Eddy Simulation (DES) with OpenFOAM's PISO solver, using 250 consecutive states shortly after the data-generating simulator reached statistical equilibrium. This represents about **10%** of the states needed to achieve statistically stationary variance, thus the models are trained with a very partial view on each case.

31.5 Distributional accuracy

A high accuracy for each sample does not necessarily imply that a model is learning the true distribution. In fact, these properties often conflict. For instance, in VGAEs, the KL-divergence penalty allows control over whether to prioritize sample quality or mode coverage. To evaluate how well models capture the probability distribution of system states, we use the Wasserstein-2 distance. This metric can be computed in two ways: (i) by treating the distribution at each node independently and averaging the result across all nodes, or (ii) by considering the joint distribution across all nodes in the graph. These metrics are denoted as W_2^{node} and W_2^{graph}, respectively. The node-level measure (W_2^{node}) provides insights into how accurately the model estimates point-wise statistics, such as the mean and standard deviation at each node. However, it does not penalize inaccurate spatial correlations, whereas the graph-wise measure (W_2^{graph}) does.

To ensure stable results when computing these metrics, the target distribution is represented by 2,500 consecutive states, and the predicted one by 3,000 samples. While the trajectories in the training data are long enough to capture the mean flow, they fall short of capturing the standard deviation, spatial correlations, or higher-order statistics. Despite these challenges, the DGN, and especially the LDGN, are capable of accurately learning the complete probability distributions of the training trajectories and accurately generating new distribution for both in- and out-of-distribution physical settings. The figure below shows a qualitative evaluation together with correlation measurements. Both DGN variants also fare much better than the *Gaussian-Mixture model* baseline denoted as GM-GNN.

In terms of Wasserstein distance W_2^{graph}, the latent-space diffusion model also outperforms the others, with a distance of **1.95 ± 0.89**, while DGN follows with 2.12 ± 0.90, and the gaussian mixture model gives 4.32 ± 0.86.

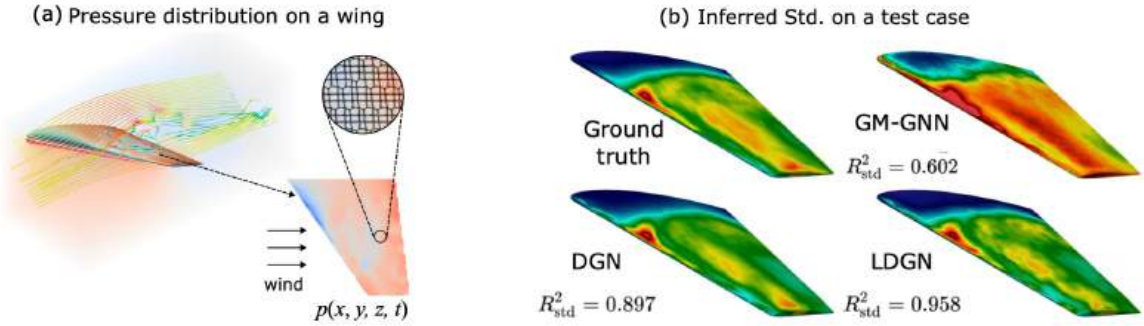


Fig. 31.4: (a) The *Wing* task targets pressure distributions on a wing in 3D turbulent flow. (b) The standard deviation of the distribution generated by the LDGN is the closest to the ground-truth (shown here in terms or correlation).

31.6 Computational Performance

Comparisons between runtimes of different implementations always should be taken with a grain of salt. Nonetheless, for the Wing experiments, the ground-truth simulator, running on 8 CPU threads, required 2,989 minutes to simulate the initial transient phase plus 2,500 equilibrium states. This duration is just enough to obtain a well converged variance. In contrast, the LDGN model took only 49 minutes on 8 CPU threads and 2.43 minutes on a single GPU to generate 3,000 samples. If we consider the generation of a single converged state (for use as an initial condition in another simulator, for example), the speedup is four orders of magnitude on the CPU, and five orders of magnitude on the GPU. Thanks to its latent space, the LDGN model is not only more accurate, but also $8\times$ faster than the DGN model, while requiring only about 55% more training time. These significant efficiency advantages suggest that graph-based diffusion models can be particularly valuable in scenarios where computational costs are otherwise prohibitive.

These results indicate that diffusion modeling in the context of unstructured simulations represent a significant step towards leveraging probabilistic methods in real-world engineering applications. To highlight the aspects of DGN and its implementation, we now turn to a simpler test case that can be analyzed in detail within a Jupyter notebook.

CHAPTER

THIRTYTWO

DISTRIBUTIONAL ACCURACY OF DIFFUSION GRAPH NETS

The following notebook shows how to run and evaluate trained diffusion graph net models (DGNs). It also evaluates the accuracy of the inferred distributions in comparison to popular baselines. In the experiments here, the training trajectories are intentionally short. Specifically, the trajectories are too short to cover one full oscillation period, meaning they do not explicitly provide full statistical information about the systems. [run in colab]

Learning with Partial Statistics

The key point to demonstrate below is the following: the diffusion training manages to learn complete statistics by combining the information *across multiple* initial conditions and geometries. This is a very powerful capability that sets it apart from other probabilistic learning methods.

The `Ellipse` task used below involves a canonical fluid dynamics problem: predicting the pressure p field around an elliptical cylinder. It makes use of the graph-based representation by focusing solely on the surface of the immersed object, as shown below. This helps to reduce dimensionality, and as we'll focus on using pretrained models, it makes this notebook easy to run.

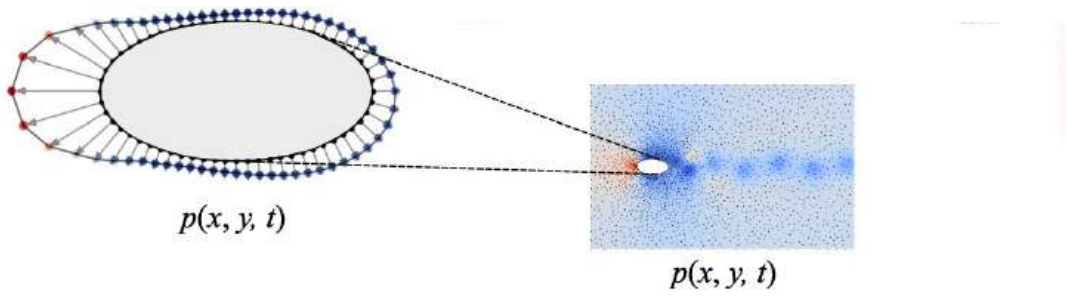


Fig. 32.1: The `Ellipse` tasks focuses on predicting samples from the pressure distribution on the surface of a elliptical obstacle immersed in a fluid. The trained models should directly predict samples on the surface, without resolving the far field or the initial transient phase of the flow.

It's important to keep in mind that we don't resolve the full flow of time with this method (this was the task in *Diffusion-based Time Prediction*). Rather, the goal is to very efficiently obtain samples from the equilibrium state of the simulation, as prescribed by the trianing data, not to resolve the evolution of an initial condition over time. A consequence is that trained models work on a graph that describes the geometry of the ellipse, but they are agnostic to time. They simply produce *one* sample of the distribution of states through time.

This in turn makes it more difficult to evaluate how closely a computed sample aligns with the ground truth: we don't know which sample out of the full distribution was generated! Thus, for a good test evaluation, it's important to have a densely sampled ground truth distribution, and for each sample inferred by a trained network, we'll search for the closest one in the test data set for the chosen input geometry. This can leave a small "discretization error" due to the discrete

samples in the test set. It will nonetheless provide a good estimate of the distributional accuracy when enough samples are used on both sides, in terms of the test set itself, and the model outputs.

32.1 Implementation

The following notebook uses the `dgn4cfd` code, and hence we'll import some basic libraries and clone the `dgn4cfd` repo below.

```
import numpy as np
import torch, tqdm
from torchvision import transforms
import matplotlib.pyplot as plt
device = torch.device('cuda:0')

try:
    import google.colab  # to ensure that we are inside colab
    !pip install --upgrade --quiet torch_geometric bayesian_torch pot
except ImportError:
    print('This notebook is running locally, please make sure the necessary pip␣
↪packages are installed.')
    pass
```

```
?25l   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/897.5 kB ? eta -:--:--
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 890.9/897.5 kB 44.2 MB/s eta 0:00:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 897.5/897.5 kB 25.2 MB/s eta 0:00:00
?25h
```

```
# clone respository, note - deactivate for local installs
!git clone https://github.com/tum-pbs/dgn4cfd
%cd dgn4cfd/
import dgn4cfd as dgn
```

```
/content/dgn4cfd
```

To keep runtimes of this notebook low, we'll simply load pre-trained models from the repository. We'll compare the following variants:

- Diffusion Graph Net (DGN)
- Latent Diffusion Graph Net (LDGN)
- DGN (same as above) trained with Flow-Matching (FM-DGN)
- LDGN with Flow-Matching (FM-LDGN) The last two show the advantages of the faster flow matching-based inference for the DGNs. Otherwise they're identical to the regular DGN/LDGN versions which use DDPM as underlying diffusion model.

We'll also load and evaluate a few baseline models, first of all a regular graph net with the same architecture as the DGN versions above. It already contains all the architectural tweaks, such as a hierarchy, but can show how much we gain (or loose) just by switching the regular, supervised training with a diffusion modeling process. Naturally, the vanilla Grap Net is fully deterministic, and can't provide a full distribution.

- Vanilla Graph Net
- Bayesian Graph Net
- Gaussian Mixure Graph Net

- Variational Graph Autoencoder (VGAE)

The last three baselines are probabilistic models, and as such they compete more directly with DGN: one would hope that they can likewise learn the posterior from the training data. As we'll see, DGN outperforms these baselines quite clearly.

```
# Diffusion Graph Net
DGN = dgn.nn.DiffusionGraphNet(
    checkpoint = "./examples/Ellipse/checkpoints/dgn-nt10.chk",
    device     = device,
)

# Latent Diffusion Graph Net
LDGN = dgn.nn.LatentDiffusionGraphNet(
    autoencoder_checkpoint = "./examples/Ellipse/checkpoints/ae-nt10.chk",
    checkpoint             = "./examples/Ellipse/checkpoints/ldgn-nt10.chk",
    device                 = device,
)

# Vanilla Graph Net
VanillaGN = dgn.nn.VanillaGnn(
    checkpoint = "./examples/Ellipse/checkpoints/vanilla-nt10.chk",
    device     = device,
)

# Bayesian Graph Net
BayesianGN = dgn.nn.BayesianGnn(
    checkpoint = "./examples/Ellipse/checkpoints/bayesian-nt10.chk",
    device     = device,
)

# Gassian Mixure Graph Net
GaussianMixGN = dgn.nn.GaussianMixtureGnn(
    checkpoint = "./examples/Ellipse/checkpoints/gaussian-nt10.chk",
    device     = device,
)

# Variational Graph Autoencoder
VGAE = dgn.nn.VGAE(
    checkpoint = "./examples/Ellipse/checkpoints/vgae-nt10.chk",
    device     = device,
)

# Flow-Matching Graph Net
FMGN = dgn.nn.FlowMatchingGraphNet(
    checkpoint = "./examples/Ellipse/checkpoints/fmgn-nt10.chk",
    device     = device,
)

# Latent Flow-Matching Graph Net
LFMGN = dgn.nn.LatentFlowMatchingGraphNet(
    autoencoder_checkpoint = "./examples/Ellipse/checkpoints/ae-nt10.chk",
    checkpoint             = "./examples/Ellipse/checkpoints/lfmgn-nt10.chk",
    device                 = device,
)
```

As a next step, we need a test dataset to evaluate our models with. The `dgn4cfd` codebase contains several test and training data sets, but we'll use the `pOnEllipseInDist` dataset which contains simulation parameters within the training range of values, although unseen. We'll also load the `TimeEllipseInDist` dataset, which contains the lenght for each simulation (two or three oscilation periods).

The `transform` object configures the input geometries and meshes. In this example we'll use a hierarchy with 3 coarsened graph levels.

```
DATASET = dgn.datasets.DatasetUrl.pOnEllipseInDist
TIME_DATASET = dgn.datasets.DatasetUrl.TimeEllipseInDist

# Training dataset
transform = transforms.Compose([
    dgn.transforms.MeshEllipse(),                                  # Create a mesh on␣
↪the ellipse
    dgn.transforms.ScaleEdgeAttr(0.02),                            # Scale the relative␣
↪position stored as `edge_attr`
    dgn.transforms.EdgeCondFreeStreamProjection(),                 # Add the projection␣
↪of the free stream velocity along the edges as `edge_cond`
    dgn.transforms.ScaleAttr('target', vmin=-1.05, vmax=0.84),     # Scale the target␣
↪field (pressure)
    dgn.transforms.ScaleAttr('glob',   vmin=500,   vmax=1000),     # Scale the global␣
↪feature (Re)
    dgn.transforms.ScaleAttr('loc',    vmin=2,     vmax=3.5),      # Scale the local␣
↪feature (distances to the walls)
    dgn.transforms.MeshCoarsening(                                 # Create 3 lower-
↪resolution graphs and normalise the relative position betwen the inter-graph nodes.
        num_scales      =  4,
        rel_pos_scaling = [0.02, 0.06, 0.15, 0.3],
        scalar_rel_pos  = True,
    ),
])
dataset = dgn.datasets.pOnEllipse(
    path          = dgn.datasets.DatasetDownloader(DATASET).file_path,
    T             = 101,
    transform     = transform,
)
print('Number of samples:', len(dataset))

# Load the length of each simulation to compute statistics
T = dgn.datasets.DatasetDownloader(TIME_DATASET).numpy()
```

```
Downloading dataset from https://huggingface.co/datasets/mariolinov/Ellipse/
↪resolve/main/pOnEllipseInDist.h5...
Dataset downloaded.
Number of samples: 50
Downloading dataset from https://huggingface.co/datasets/mariolinov/Ellipse/
↪resolve/main/TimeEllipseInDist.npy...
Dataset downloaded.
```

32.2 Sample-wise Accuracy

The next cell defines a plotting function that shows the closest ground truth pressure distribution that was found in the reference data set in black next to the neural network outputs, shown in light red.

The `SIM_IDX` variable chooses a specific flow condition from the test dataset (feel free to try others).

```
SIM_IDX = 25

def plot(ax, pos, target, pred, r2, title):
    pos = pos.cpu()
    target = target.cpu()
    pred = pred.cpu()
    # Plots
    top = pos[:, 1] >= 0.
    bottom = torch.logical_not(top)
    ax.plot(pos[top   , 0].cpu(), target[top   ].cpu(), 'k^', label='g.t.  top')
    ax.plot(pos[bottom, 0].cpu(), target[bottom].cpu(), 'kv', label='g.t.  bottom')
    ax.plot(pos[top,    0].cpu(), pred  [top   ].cpu(), '^', color="red", label='pred.
 ↪ top', alpha=0.6)
    ax.plot(pos[bottom, 0].cpu(), pred  [bottom].cpu(), 'v', color="red", label='pred.
 ↪ bottom', alpha=0.6)
    ax.set_title(title+r' ($R^2$ = ' + f"{r2:.4f}" + r')', fontsize=16)
    ax.set_ylabel(r'$p$', fontsize=16)
    ax.set_xlabel(r'$x$', fontsize=16)
    ax.grid()
    ax.legend(fontsize=16)
```

The next cell creates a 3x3 grid of graphs with the `plot()` function. The first row will contain the DGN models, and the vanilla GN. The next row will show flow-matching in direct comparison to the DDPM versions in the first row, while the last one will contain the outputs of the remaining three baseline.

```
# denoising steps
NUM_DENOISING_STEPS = 50
# flow matching steps
NUM_FM_STEPS = int(NUM_DENOISING_STEPS/10)+1

fig, ax = plt.subplots(3, 3, figsize=(14, 14))
ax = ax.flatten()
curr_ax = 0

graph = dataset.get_sequence(SIM_IDX, n_in=T[SIM_IDX])

# DGN inference
steps = dgn.nn.diffusion.DiffusionStepsGenerator('linear', DGN.diffusion_process.num_
 ↪steps)(NUM_DENOISING_STEPS)
pred = DGN.sample(graph, steps=steps)
# Compute the accuracy
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
 ↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"DGN R2 = {r2:.4f} at t={t}")
# Plot the results
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "DGN")
curr_ax += 1

# LDGN inference
```

(continues on next page)

(continued from previous page)

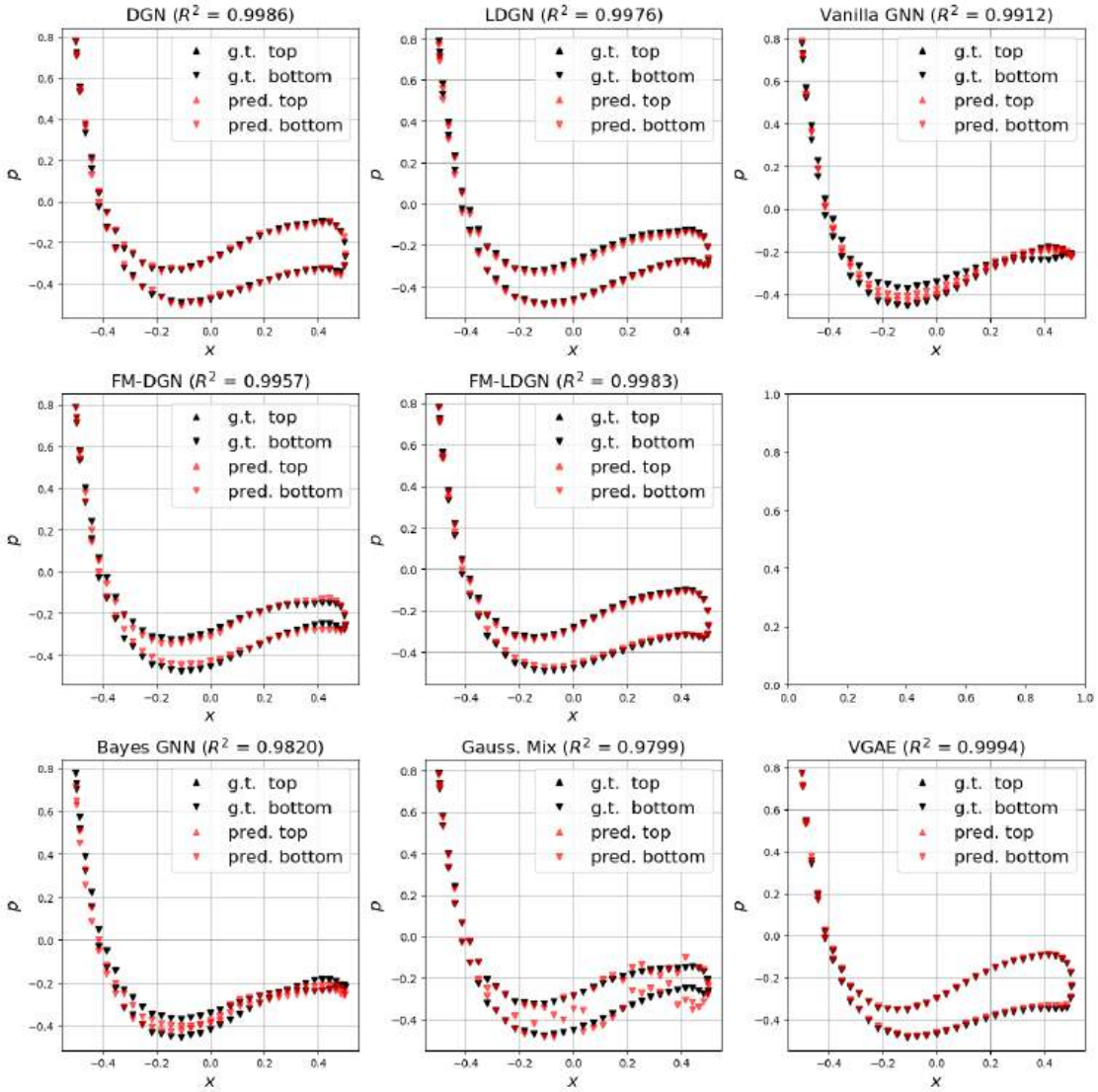
```
steps = dgn.nn.diffusion.DiffusionStepsGenerator('linear', DGN.diffusion_process.num_
 ↪steps)(NUM_DENOISING_STEPS)
pred = LDGN.sample(graph, steps=steps)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
 ↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"LDGN R2 = {r2:.4f} at t={t}")
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "LDGN")
curr_ax += 1

# Vanilla Graph Net inference
with torch.no_grad():
    VanillaGN.eval()
    pred = VanillaGN(graph)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
 ↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"Vanilla Graph Net R2 = {r2:.4f} at t={t}")
plot(ax[curr_ax], graph.pos, graph.target[:,t], pred, r2, "Vanilla GNN")
curr_ax += 1

# Flow matching versions

# Flow-Matching Graph Net inference
steps = np.linspace(0, 1, NUM_FM_STEPS)
pred = FMGN.sample(graph, steps=steps)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
 ↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"Flow-Matching Graph Net R2 = {r2:.4f} at t={t}")
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "FM-DGN")
curr_ax += 1

# Latent Flow-Matching Graph Net inference
steps = np.linspace(0, 1, NUM_FM_STEPS)
pred = LFMGN.sample(graph, steps=steps)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
 ↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"Latent Flow-Matching Graph Net R2 = {r2:.4f} at t={t}")
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "FM-LDGN")
curr_ax += 1

# skip one cell
curr_ax += 1

# Other baselines following

# Bayesian Graph Net inference
pred = BayesianGN.sample(graph)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
 ↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"Bayesian Graph Net R2 = {r2:.4f} at t={t}")
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "Bayes GNN")
curr_ax += 1
```

(continues on next page)

(continued from previous page)

```
# Gaussian Mixture Graph Net inference
pred = GaussianMixGN.sample(graph)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"Gaussian Mixture Graph Net R2 = {r2:.4f} at t={t}")
# Plot the results
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "Gauss. Mix")
curr_ax += 1

# VGAE inference
pred = VGAE.sample(graph)
r2 = [dgn.metrics.r2_accuracy(pred, target) for target in graph.target.split(1,␣
↪dim=1)]
r2, t = np.max(r2), np.argmax(r2)
print(f"VGAE R2 = {r2:.4f} at t={t}")
# Plot the results
plot(ax[curr_ax],graph.pos, graph.target[:,t], pred, r2, "VGAE")
curr_ax += 1


# Show the combined figure
plt.tight_layout()
plt.show()
```

```
DGN R2 = 0.9986 at t=74
LDGN R2 = 0.9976 at t=15
Vanilla Graph Net R2 = 0.9912 at t=6
Flow-Matching Graph Net R2 = 0.9957 at t=28
Latent Flow-Matching Graph Net R2 = 0.9983 at t=50
Bayesian Graph Net R2 = 0.9820 at t=67
Gaussian Mixture Graph Net R2 = 0.9799 at t=28
VGAE R2 = 0.9994 at t=36
```



There are a lot of graphs here, but they show several interesting aspects:

- First of all, the DGN and its FM variants produce very accurate pressure distributions; typically only the DGN version is slightly noise with the default settings (the latent space model LDGN already performs better).
- Flow matching uses one tenths of the steps (i.e. is 5x faster), but gives the same quality as denoising.
- The Bayes GNN has problems, and the Gaussian Mixture model produces clearly suboptimal outputs. We actually tuned this GM model quite a bit, but to no avail.
- The VGAE outputs often look surprisingly good here, but this holds only for a single sample. As we'll see, this model has trouble learning the proper distribution across many samples. This is more apparent in more chaotic and higher-dimensional tasks.

One difficulty here is that we're only seeing a single output. You'll see that re-running the cell will produce variants, but naturally, we'll need a thorough evaluation of the distributional accuracy of the models to properly draw conclusions about their performance.

32.3 Evaluating Distributional Accuracy

To evaluate a large number of samples, and compute their node wise and graph-based Wasserstein distances. These quantified metrics are a good start, but it's still interesting to visualize the distributions to provide more intuition for how well or badly certain methods do. For this, we'll plot stacks of Gaussian kernel density estimates that show the distribution of pressure values along the length of the ellipses.

These plots can directly be compared in terms of how much density they contain in different regions of the plot. For reference, the distribution of the ground truth data is shown in the first cell below.

```
from scipy.stats import gaussian_kde

def pdf(ax, pos, pred, title, vmin=None, vmax=None, w2_distance_1d=None, w2_distance_
↪nd=None):
    pos = pos.cpu()
    ang = torch.atan2(pos[:,1], pos[:,0])
    idx = torch.argsort(ang)
    idx = idx[ang[idx] > 0]
    y = pred[idx].cpu().numpy()
    if vmin is None:
        vmin = y.min() - 0.1
    if vmax is None:
        vmax = y.max() + 0.1
    x = np.linspace(vmin, vmax, 1000)
    f = np.stack([gaussian_kde(y[i])(x) for i in range(y.shape[0])])
    f = f / f.max(axis=1)[:,None]

    ax.imshow(np.flip(f.T), aspect='auto', cmap='binary')
    ax.set_yticks(np.linspace(100, 900, 5), np.flip(np.round(x[np.linspace(100, 900,␣
↪5).astype(int)], 2)), fontsize=16)
    ax.set_xticks([])
    #ax.set_yticks(fontsize=16)
    ax.set_title(title, fontsize=28)
    # Add a label with the Wasserstein-2 distance
    if w2_distance_1d is not None and w2_distance_nd is not None:
        ax.text(0.2, 0.76,  r'$W_2^\mathregular{node}$' + f'  = {w2_distance_1d:.4f} \
↪n' + r'$W_2^\mathregular{graph}$' + f' = {w2_distance_nd:.4f}', fontsize=18,␣
↪bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.2'),␣
↪transform=ax.transAxes)
```

We'll use `NUM_SAMPLES = 200` below, and plot the distributions for ground truth and the 8 models below.

```
NUM_SAMPLES = 200
BATCH_SIZE  = 200 # Number of samples generated in parallel. Reduce this number if␣
↪you run out of memory.

fig, ax = plt.subplots(3, 3, figsize=(14, 14))
ax = ax.flatten()
curr_ax = 0

graph = dataset.get_sequence(SIM_IDX, n_in=T[SIM_IDX])
```

(continues on next page)

(continued from previous page)

```
gt_mean = graph.target.mean(dim=1)
gt_std  = graph.target.std (dim=1)
# Plot the ground-truth PDF on the upper half of the ellipse
vmin, vmax = graph.target.min().item() - 0.1, graph.target.max().item() + 0.1
pdf(ax[curr_ax], graph.pos, graph.target, '*Ground truth*', vmin=vmin, vmax=vmax)
curr_ax += 1

# DGN inference
steps = dgn.nn.diffusion.DiffusionStepsGenerator('linear', DGN.diffusion_process.num_
 ↪steps)(NUM_DENOISING_STEPS)
pred = DGN.sample_n(NUM_SAMPLES, graph, steps=steps, batch_size=BATCH_SIZE).cpu().
 ↪squeeze(-1)
mean = pred.mean(dim=1)
std  = pred.std (dim=1)
# Compute the accuracy of the mean and std
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
print('DGN')
print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std: {std_r2:.4f}")
# Compute the Wasserstein-2 distance
w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
# Plot the PDF on the upper half of the ellipse
pdf(ax[curr_ax], graph.pos, pred, 'DGN', w2_distance_1d=w2_distance_1d, w2_distance_
 ↪nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1

# LDGN inference
steps = dgn.nn.diffusion.DiffusionStepsGenerator('linear', DGN.diffusion_process.num_
 ↪steps)(NUM_DENOISING_STEPS)
pred = LDGN.sample_n(NUM_SAMPLES, graph,  steps=steps, batch_size=BATCH_SIZE).cpu().
 ↪squeeze(-1)
mean = pred.mean(dim=1); std  = pred.std (dim=1)
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
print('LDGN'); print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std: {std_r2:.4f}")

w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
pdf(ax[curr_ax], graph.pos, pred, 'LDGN', w2_distance_1d=w2_distance_1d, w2_distance_
 ↪nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1

# skip one
curr_ax += 1

# FMGN inference
steps = np.linspace(0, 1, NUM_FM_STEPS )
pred = FMGN.sample_n(NUM_SAMPLES, graph, steps=steps, batch_size=BATCH_SIZE).cpu().
 ↪squeeze(-1)
mean = pred.mean(dim=1); std  = pred.std (dim=1)
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
```

(continues on next page)

(continued from previous page)

```
print('Flow-Matching DGN'); print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std: {std_r2:.
 ↪4f}")
w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
pdf(ax[curr_ax], graph.pos, pred, 'FM DGN', w2_distance_1d=w2_distance_1d, w2_
 ↪distance_nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1

# LFMGN inference
steps = np.linspace(0, 1, NUM_FM_STEPS )
pred = LFMGN.sample_n(NUM_SAMPLES, graph, steps=steps, batch_size=BATCH_SIZE).cpu().
 ↪squeeze(-1)
mean = pred.mean(dim=1); std  = pred.std (dim=1)
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
print('Latent Flow-Matching DGN'); print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std:
 ↪{std_r2:.4f}")
w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
pdf(ax[curr_ax], graph.pos, pred, 'FM LDGN', w2_distance_1d=w2_distance_1d, w2_
 ↪distance_nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1

# Bayesian Graph Net inference
pred_list = []
for _ in tqdm.tqdm(range(NUM_SAMPLES)):
    pred_list.append(
        BayesianGN.sample(graph).cpu()
    )
pred = torch.concatenate(pred_list, dim=1)
mean = pred.mean(dim=1); std  = pred.std (dim=1)
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
print('Bayesian Graph Net'); print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std: {std_
 ↪r2:.4f}")
w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
pdf(ax[curr_ax], graph.pos, pred, 'Bayesian GNN', w2_distance_1d=w2_distance_1d, w2_
 ↪distance_nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1

# Gaussian Mixture Graph Net inference
pred = GaussianMixGN.sample_n(NUM_SAMPLES, graph, batch_size=BATCH_SIZE).cpu().
 ↪squeeze(-1)
mean = pred.mean(dim=1); std  = pred.std (dim=1)
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
print('Gaussian Mixture Graph Net'); print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std:
 ↪{std_r2:.4f}")
# Compute the Wasserstein-2 distance
w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
```

(continues on next page)

(continued from previous page)

```
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
pdf(ax[curr_ax], graph.pos, pred, 'GM-GNN', w2_distance_1d=w2_distance_1d, w2_
 ↪distance_nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1

# VGAE inference
pred = VGAE.sample_n(NUM_SAMPLES, graph, batch_size=BATCH_SIZE).cpu().squeeze(-1)
mean = pred.mean(dim=1); std  = pred.std (dim=1)
mean_r2 = dgn.metrics.r2_accuracy(mean, gt_mean)
std_r2  = dgn.metrics.r2_accuracy(std , gt_std )
print('VGAE'); print(f"R2 of mean: {mean_r2:.4f}", f"R2 of std: {std_r2:.4f}")
w2_distance_1d = dgn.metrics.w2_distance_1d(pred, graph.target)
w2_distance_nd = dgn.metrics.w2_distance_nd(pred, graph.target)
print(f"Wasserstein-2 distance 1d: {w2_distance_1d:.4f}")
print(f"Wasserstein-2 distance nd: {w2_distance_nd:.4f}")
pdf(ax[curr_ax], graph.pos, pred, 'VGAE', w2_distance_1d=w2_distance_1d, w2_distance_
 ↪nd=w2_distance_nd, vmin=vmin, vmax=vmax)
curr_ax += 1
```

```
DGN
R2 of mean: 0.9989 R2 of std: 0.9814
Wasserstein-2 distance 1d: 0.0116
Wasserstein-2 distance nd: 0.1536
```

```
LDGN
R2 of mean: 0.9986 R2 of std: 0.9614
Wasserstein-2 distance 1d: 0.0143
Wasserstein-2 distance nd: 0.1787
```

```
Flow-Matching DGN
R2 of mean: 0.9976 R2 of std: 0.4237
Wasserstein-2 distance 1d: 0.0235
Wasserstein-2 distance nd: 0.2256
```

```
Latent Flow-Matching DGN
R2 of mean: 0.9985 R2 of std: 0.8425
Wasserstein-2 distance 1d: 0.0154
Wasserstein-2 distance nd: 0.1635
```

```
100%|██████████| 200/200 [00:04<00:00, 43.76it/s]
```

```
Bayesian Graph Net
R2 of mean: 0.9980 R2 of std: -0.6514
```

(continues on next page)

(continued from previous page)

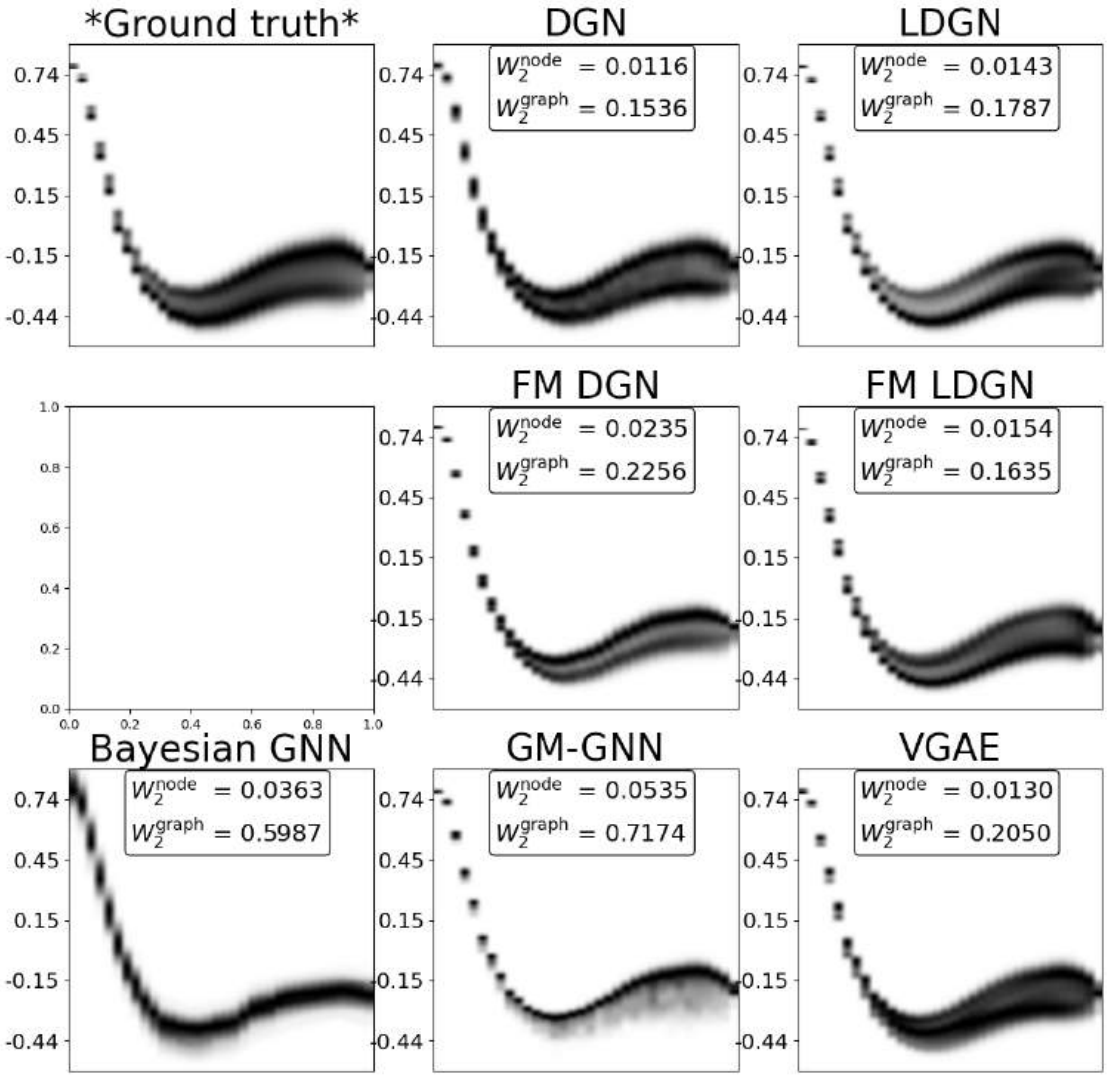
```
Wasserstein-2 distance 1d: 0.0363
Wasserstein-2 distance nd: 0.5987
```

```

```

```
Gaussian Mixture Graph Net
R2 of mean: 0.9708 R2 of std: 0.1079
Wasserstein-2 distance 1d: 0.0535
Wasserstein-2 distance nd: 0.7174
```

```

```

```
VGAE
R2 of mean: 0.9994 R2 of std: 0.8818
Wasserstein-2 distance 1d: 0.0130
Wasserstein-2 distance nd: 0.2050
```



First, the ground truth distribution is worth a closer look: it's a bi-modal distribution, caused by the recurring vortex shedding behind the ellipse. Hence, it's important for models to capture the two extremes (regions with higher density), as well as the less dense inner part. Comparing the GT version with the distributions of the different approaches shows that the DGN methods faithfully reproduce the distribution. The noise of the regular DGN averages out to some extent here, it can even exhibit a performance that surpasses the LDGN version. As before, flow matching can produce these distributions at a fraction of the cost, and hence is generally preferable.

The problems of the Baysian NN and the Gaussian mixture model very clearly show up in their distributions. The problems of the VGAE approach show up more clearly here: it has mode-collapse issues, which are even more severe in more complex scenarios.

The graph-based Wasserstein distance largely captures the intuition behind the observations above, but fails to illustrate how well outer (and inner) parts of the distribution are matched. Nonetheless, it's of course crucial as a tool for higher-dimensional distributions such as those of the 3D wing case from the previous page.

CHAPTER

THIRTYTHREE

DISCUSSION OF PROBABILISTIC LEARNING

As the previous sections have demonstrated, probabilistic learning offers a wide range of very exciting possibilities in the context of physics-based learning. First, these methods come with a highly interesting and well developed theory. Surprisingly, some parts are actually more developed than basic questions about simpler learning approaches.

At the same time, they enable a fundamentally different way to work with simulations: they provide a simple way to work with complex distributions of solutions. This is of huge importance for inverse problems, e.g. in the context of obtaining likelihood-based estimates for *simulation-based inference*.

That being said, diffusion based approaches will not show relatively few advantages for deterministic settings: they are not more accurate, and typically induce slightly larger computational costs. An interesting exception is the long-term stability, as discussed in *Unconditional Stability*. To summarize the key aspects of probabilistic deep learning approaches:

✓ Pro:

- Enable training and inference for distributions
- Well developed theory
- Stable training

✗ Con:

- (Slightly) increased inference cost
- No real advantage for deterministic settings

One more concluding recommendation: if your problems contains ambiguities, diffusion modeling in the form of *flow matching* is the method of choice. If your data contains reliable input-output pairs, go with simpler *deterministic training* instead.

Next, we can turn to a new viewpoint on learning problems, the field of *reinforcement learning*. As the next sections will point out, it is actually not so different from the topics of the previous chapters despite the new viewpoint.

Part VI

Reinforcement Learning

CHAPTER

THIRTYFOUR

INTRODUCTION TO REINFORCEMENT LEARNING

Deep reinforcement learning, which we'll just call *reinforcement learning* (RL) from now on, is a class of methods in the larger field of deep learning that takes a different viewpoint from classic "train with data" one: RL effectively lets an AI agent learn from interactions with an environment. While performing actions, the agent receives reward signals and tries to discern which actions contribute to higher rewards, to adapt its behavior accordingly. RL has been very successful at playing games such as Go [SSS+17], and it bears promise for engineering applications such as robotics.

The setup for RL generally consists of two parts: the environment and the agent. The environment receives actions a from the agent while supplying it with observations in the form of states s, and rewards r. The observations represent the fraction of the information from the respective environment state that the agent is able to perceive. The rewards are given by a predefined function, usually tailored to the environment and might contain, e.g., a game score, a penalty for wrong actions or a bounty for successfully finished tasks.

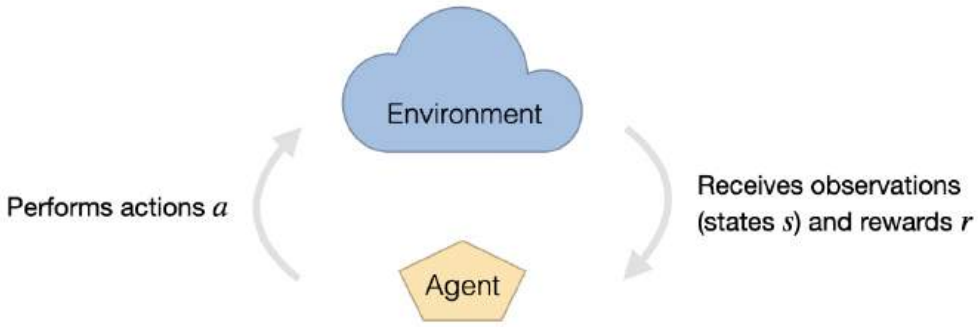


Fig. 34.1: Reinforcement learning is formulated in terms of an environment that gives observations in the form of states and rewards to an agent. The agent interacts with the environment by performing actions.

In its simplest form, the learning goal for reinforcement learning tasks can be formulated as

$$\arg\max_\theta \mathbb{E}_{a\sim\pi(;s,\theta_p)}[\sum_t r_t], \tag{34.1}$$

where the reward at time t (denoted by r_t above) is the result of an action a performed by an agent. The agents choose their actions based on a neural network policy which decides via a set of given observations. The policy $\pi(a; s, \theta)$ returns the probability for the action, and is conditioned on the state s of the environment and the weights θ.

During the learning process the central aim of RL is to uses the combined information of state, action and corresponding rewards to increase the cumulative intensity of reward signals over each trajectory. To achieve this goal, multiple algorithms have been proposed, which can be roughly divided into two larger classes: *policy gradient* and *value-based* methods [SB18].

34.1 Algorithms

In vanilla policy gradient methods, the trained neural networks directly select actions a from environment observations. In the learning process, an NN is trained to infer the probability of actions. Here, probabilities for actions leading to higher rewards in the rest of the respective trajectories are increased, while actions with smaller return are made less likely.

Value-based methods, such as *Q-Learning*, on the other hand work by optimizing a state-action value function, the so-called *Q-Function*. The network in this case receives state s and action a to predict the average cumulative reward resulting from this input for the remainder of the trajectory, i.e. $Q(s, a)$. Actions are then chosen to maximize Q given the state.

In addition, *actor-critic* methods combine elements from both approaches. Here, the actions generated by a policy network are rated based on a corresponding change in state potential. These values are given by another neural network and approximate the expected cumulative reward from the given state. *Proximal policy optimization* (PPO) [SWD+17] is one example from this class of algorithms and is our choice for the example task of this chapter, which is controlling Burgers' equation as a physical environment.

34.2 Proximal policy optimization

As PPO methods are an actor-critic approach, we need to train two interdependent networks: the actor, and the critic. The objective of the actor inherently depends on the output of the critic network (it provides feedback which actions are worth performing), and likewise the critic depends on the actions generated by the actor network (this determines which states to explore).

This interdependence can promote instabilities, e.g., as strongly over- or underestimated state values can give wrong impulses during learning. Actions yielding higher rewards often also contribute to reaching states with higher informational value. As a consequence, when the - possibly incorrect - value estimate of individual samples are allowed to unrestrictedly affect the agent's behavior, the learning progress can collapse.

PPO was introduced as a method to specifically counteract this problem. The idea is to restrict the influence that individual state value estimates can have on the change of the actor's behavior during learning. PPO is a popular choice especially when working on continuous action spaces. This can be attributed to the fact that it tends to achieve good results with a stable learning progress, while still being comparatively easy to implement.

34.2.1 PPO-clip

More specifically, we will use the algorithm *PPO-clip* [SWD+17]. This PPO variant sets a hard limit for the change in behavior caused by singular update steps. As such, the algorithm uses a previous network state (denoted by a subscript $_p$ below) to limit the change per step of the learning process. In the following, we will denote the network parameters of the actor network as θ and those of the critic as ϕ.

34.2.2 Actor

The actor computes a policy function returning the probability distribution for the actions conditioned by the current network parameters θ and a state s. In the following we'll denote the probability of choosing a specific action a from the distribution with $\pi(a; s, \theta)$. As mentioned above, the training procedure computes a certain number of weight updates using policy evaluations with a fixed previous network state $\pi(a; s, \theta_p)$, and in intervals re-initializes the previous weights θ_p from θ. To limit the changes, the objective function makes use of a clip(a, b, c) function, which simply returns a clamped to the interval $[b, c]$.

ϵ defines the bound for the deviation from the previous policy. In combination, the objective for the actor is given by the following expression:

$$\arg\max_\theta \mathbb{E}_{a\sim\pi(;s,\theta_p)} \left[\min\left(\frac{\pi(a;s,\theta)}{\pi(a;s,\theta_p)} A(s,a;\phi), \text{clip}\left(\frac{\pi(a;s,\theta)}{\pi(a;s,\theta_p)}, 1-\epsilon, 1+\epsilon \right) A(s,a;\phi) \right) \right]$$

As the actor network is trained to provide the expected value, at training time an additional standard deviation is used to sample values from a Gaussian distribution around this mean. It is decreased over the course of the training, and at inference time we only evaluate the mean (i.e. a distribution with variance 0).

34.2.3 Critic and advantage

The critic is represented by a value function $V(s; \phi)$ that predicts the expected cumulative reward to be received from state s. Its objective is to minimize the squared advantage A:

$$\arg\min_\phi \mathbb{E}_{a\sim\pi(;s,\theta_p)} [A(s,a;\phi)^2] \,,$$

where the advantage function $A(s, a; \phi)$ builds upon V: it's goal is to evaluate the deviation from an average cumulative reward. I.e., we're interested in estimating how much the decision made via $\pi(; s, \theta_p)$ improves upon making random decisions (again, evaluated via the unchanging, previous network state θ_p). We use the so-called Generalized Advantage Estimation (GAE) [SML+15] to compute A as:

$$\begin{aligned} A(s_t, a_t; \phi) &= \sum_{i=0}^{n-t-1} (\gamma\lambda)^i \delta_{t+i} \\ \delta_t &= r_t + \gamma(V(s_{t+1}; \phi) - V(s_t; \phi)) \end{aligned}$$

Here r_t describes the reward obtained in time step t, while n denotes the total length of the trajectory. γ and λ are two hyperparameters which influence rewards and state value predictions from the distant futures have on the advantage calculation. They are typically set to values smaller than one.

The δ_t in the formulation above represent a biased approximation of the true advantage. Hence the GAE can be understood as a discounted cumulative sum of these estimates, from the current time step until the end of the trajectory.

34.3 Application to inverse problems

Reinforcement learning is widely used for trajectory optimization with multiple decision problems building upon one another. However, in the context of physical systems and PDEs, reinforcement learning algorithms are likewise attractive. In this setting, they can operate in a fashion that's similar to supervised single shooting approaches by generating full trajectories and learning by comparing the final approximation to the target.

Still, the approaches differ in terms of how this optimization is performed. For example, reinforcement learning algorithms like PPO try to explore the action space during training by adding a random offset to the actions selected by the actor. This way, the algorithm can discover new behavioral patterns that are more refined than the previous ones.

The way how long term effects of generated forces are taken into account can also differ for physical systems. In a control force estimator setup with differentiable physics (DP) loss, as discussed e.g. in *Burgers Optimization with a Differentiable Physics Gradient*, these dependencies are handled by passing the loss gradient through the simulation step back into previous time steps. Contrary to that, reinforcement learning usually treats the environment as a black box without gradient information. When using PPO, the value estimator network is instead used to track the long term dependencies by predicting the influence any action has for the future system evolution.

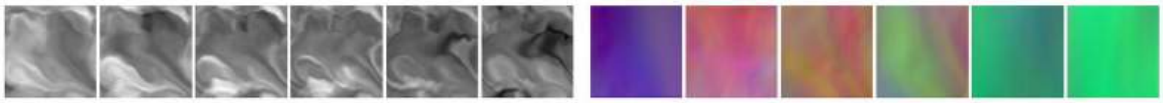
Working with Burgers' equation as physical environment, the trajectory generation process can be summarized as follows. It shows how the simulation steps of the environment and the neural network evaluations of the agent are interleaved:

$$\mathbf{u}_{t+1} = \mathcal{P}(\mathbf{u}_t + \pi(\mathbf{u}_t; \mathbf{u}^*, t, \theta)\Delta t)$$

The $*$ superscript (as usual) denotes a reference or target quantity, and hence here $\mathbf{u}^*$ denotes a velocity target. For the continuous action space of the PDE, π directly computes an action in terms of a force, rather than probabilities for a discrete set of different actions.

The reward is calculated in a similar fashion as the Loss in the DP approach: it consists of two parts, one of which amounts to the negative square norm of the applied forces and is given at every time step. The other part adds a punishment proportional to the L^2 distance between the final approximation and the target state at the end of each trajectory.

$$\begin{aligned} r_t &= r_t^f + r_t^o \\ r_t^f &= -\left\|\mathbf{a}_t\right\|_2^2 \\ r_t^o &= \begin{cases} -\left\|\mathbf{u}^* - \mathbf{u}_t\right\|_2^2, & \text{if } t = n-1 \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$



34.4 Implementation

In the following, we'll describe a way to implement a PPO-based RL training for physical systems. This implementation is also the basis for the notebook of the next section, i.e., *Controlling Burgers' Equation with Reinforcement Learning*. While this notebook provides a practical example, and an evaluation in comparison to DP training, we'll first give a more generic overview below.

To train a reinforcement learning agent to control a PDE-governed system, the physical model has to be formalized as an RL environment. The stable-baselines3 framework, which we use in the following to implement a PPO training, uses a vectorized version of the OpenAI gym environment. This way, rollout collection can be performed on multiple trajectories in parallel for better resource utilization and wall time efficiency.

Vectorized environments require a definition of observation and action spaces, meaning the in- and output spaces of the agent policy. In our case, the former consists of the current physical states and the goal states, e.g., velocity fields, stacked along their channel dimension. Another channel is added for the elapsed time since the start of the simulation divided by the total trajectory length. The action space (the output) encompasses one force value for each cell of the velocity field.

The most relevant methods of vectorized environments are `reset`, `step_async`, `step_wait` and `render`. The first of these is used to start a new trajectory by computing initial and goal states and returning the first observation for each vectorized instance. As these instances in other applications are not bound to finish trajectories synchronously, `reset` has to be called from within the environment itself when entering a terminal state. `step_async` and `step_wait` are the two main parts of the `step` method, which takes actions, applies them to the velocity fields and performs one iteration of the physics models. The split into async and wait enables supporting vectorized environments that run each instance on separate threads. However, this is not required in our approach, as phiflow handles the simulation of batches

internally. The `render` method is called to display training results, showing reconstructed trajectories in real time or rendering them to files.

Because of the strongly differing output spaces of the actor and critic networks, we use different architectures for each of them. The network yielding the actions uses a variant of the network architecture from Holl et al. [HKT19], in line with the CFE function performing actions there. The other network consists of a series of convolutions with kernel size 3, each followed by a max pooling layer with kernel size 2 and stride 2. After the feature maps have been downsampled to one value in this way, a final fully connected layer merges all channels, generating the predicted state value.

In the example implementation of the next chapter, the `BurgersTraining` class manages all aspects of this training internally, including the setup of agent and environment and storing trained models and monitor logs to disk. It also includes a variant of the Burgers' equation environment described above, which, instead of computing random trajectories, uses data from a predefined set. During training, the agent is evaluated in this environment in regular intervals to be able to compare the training progress to the DP method more accurately.

The next chapter will use this `BurgersTraining` class to run a full PPO scenario, evaluate its performance, and compare it to an approach that uses more domain knowledge of the physical system, i.e., the gradient-based control training with a DP approach.

CHAPTER

THIRTYFIVE

CONTROLLING BURGERS' EQUATION WITH REINFORCEMENT LEARNING

In the following, we will target inverse problems with Burgers equation as a testbed for reinforcement learning (RL). The setup is similar to the inverse problems previously targeted with differentiable physics (DP) training (cf. *Solving Inverse Problems with NNs*), and hence we'll also directly compare to these approaches below. Similar to before, Burgers equation is simple but non-linear with interesting dynamics, and hence a good starting point for RL experiments. In the following, the goal is to train a control force estimator network that should predict the forces needed to generate smooth transitions between two given states. [run in colab]

35.1 Overview

Reinforcement learning describes an agent perceiving an environment and taking actions inside it. It aims at maximizing an accumulated sum of rewards, which it receives for those actions by the environment. Thus, the agent learns empirically which actions to take in different situations. *Proximal policy optimization* (PPO) is a widely used reinforcement learning algorithm describing two neural networks: a policy NN selecting actions for given observations and a value estimator network rating the reward potential of those states. These value estimates form the loss of the policy network, given by the change in reward potential by the chosen action.

This notebook illustrates how PPO reinforcement learning can be applied to the described control problem of Burgers' equation. In comparison to the DP approach, the RL method does not have access to a differentiable physics solver, it is *model-free*.

However, the goal of the value estimator NN is to compensate for this lack of a solver, as it tries to capture the long term effect of individual actions. Thus, an interesting question the following code example should answer is: can the model-free PPO reinforcement learning match the performance of the model-based DP training. We will compare this in terms of learning speed and the amount of required forces.

35.2 Software installation

This example uses the reinforcement learning framework stable_baselines3 with PPO as reinforcement learning algorithm. For the physical simulation, version 1.5.1 of the differentiable PDE solver ΦFlow is used.

After the RL training is completed, we'll additionally compare it to a differentiable physics approach using a "control force estimator" (CFE) network from *Solving Inverse Problems with NNs* (as introduced by [HKT19]).

```
!pip install stable-baselines3==1.1 phiflow==1.5.1
!git clone https://github.com/Sh0cktr4p/PDE-Control-RL.git
!git clone https://github.com/holl-/PDE-Control.git
```

Now we can import the necessary modules. Due to the scope of this example, there are quite a few modules to load.

```
import sys; sys.path.append('PDE-Control/src'); sys.path.append('PDE-Control-RL/src')
import time, csv, os, shutil
from tensorboard.backend.event_processing.event_accumulator import EventAccumulator
from phi.flow import *
import burgers_plots as bplt
import matplotlib.pyplot as plt
from envs.burgers_util import GaussianClash, GaussianForce
```

35.3 Data generation

First we generate a dataset which we will use to train the differentiable physics model on. We'll also use it to evaluate the performance of both approaches during and after training. The code below simulates 1000 cases (i.e. phiflow "scenes"), and keeps 100 of them as validation and test cases, respectively. The remaining 800 are used for training.

```
DOMAIN = Domain([32], box=box[0:1])      # Size and shape of the fields
VISCOSITY = 0.003
STEP_COUNT = 32                          # Trajectory length
DT = 0.03
DIFFUSION_SUBSTEPS = 1

DATA_PATH = 'forced-burgers-clash'
SCENE_COUNT = 1000
BATCH_SIZE = 100

TRAIN_RANGE = range(200, 1000)
VAL_RANGE = range(100, 200)
TEST_RANGE = range(0, 100)
```

```
for batch_index in range(SCENE_COUNT // BATCH_SIZE):
    scene = Scene.create(DATA_PATH, count=BATCH_SIZE)
    print(scene)
    world = World()
    u0 = BurgersVelocity(
        DOMAIN,
        velocity=GaussianClash(BATCH_SIZE),
        viscosity=VISCOSITY,
        batch_size=BATCH_SIZE,
        name='burgers'
    )
    u = world.add(u0, physics=Burgers(diffusion_substeps=DIFFUSION_SUBSTEPS))
    force = world.add(FieldEffect(GaussianForce(BATCH_SIZE), ['velocity']))
    scene.write(world.state, frame=0)
    for frame in range(1, STEP_COUNT + 1):
        world.step(dt=DT)
        scene.write(world.state, frame=frame)
```

```
forced-burgers-clash/sim_000000
forced-burgers-clash/sim_000100
forced-burgers-clash/sim_000200
forced-burgers-clash/sim_000300
forced-burgers-clash/sim_000400
forced-burgers-clash/sim_000500
forced-burgers-clash/sim_000600
forced-burgers-clash/sim_000700
```

(continues on next page)

(continued from previous page)

```
forced-burgers-clash/sim_000800
forced-burgers-clash/sim_000900
```

35.4 Training via reinforcement learning

Next we set up the RL environment. The PPO approach uses a dedicated value estimator network (the "critic") to predict the sum of rewards generated from a certain state. These predicted rewards are then used to update a policy network (the "actor") which, analogously to the CFE network of *Solving Inverse Problems with NNs*, predicts the forces to control the simulation.

```
from experiment import BurgersTraining

N_ENVS = 10                           # On how many environments to train in parallel, load balancing
FINAL_REWARD_FACTOR = STEP_COUNT      # Penalty for not reaching the goal state
STEPS_PER_ROLLOUT = STEP_COUNT * 10 # How many steps to collect per environment between agent updates
N_EPOCHS = 10                         # How many epochs to perform during each agent update
RL_LEARNING_RATE = 1e-4               # Learning rate for agent updates
RL_BATCH_SIZE = 128                   # Batch size for agent updates
RL_ROLLOUTS = 500                    # Number of iterations for RL training
```

To start training, we create a trainer object which manages the environment and the agent internally. Additionally, a directory for storing models, logs, and hyperparameters is created. This way, training can be continued at any later point using the same configuration. If the model folder specified in `exp_name` already exists, the agent within is loaded; otherwise, a new agent is created. For the PPO reinforcement learning algorithm, the implementation of `stable_baselines3` is used. The trainer class acts as a wrapper for this system. Under the hood, an instance of a `BurgersEnv` gym environment is created, which is loaded into the PPO algorithm. It generates random initial states, precomputes corresponding ground truth simulations and handles the system evolution influenced by the agent's actions. Furthermore, the trainer regularly evaluates the performance on the validation set by loading a different environment that uses the initial and target states of the validation set.

35.4.1 Gym environment

The gym environment specification provides an interface leveraging the interaction with the agent. Environments implementing it must specify observation and action spaces, which represent the in- and output spaces of the agent. Further, they have to define a set of methods, the most important ones being `reset`, `step`, and `render`.

- `reset` is called after a trajectory has ended, to revert the environment to an initial state, and returns the corresponding observation.
- `step` takes an action given by the agent and iterates the environment to the next state. It returns the resulting observation, the received reward, a flag determining whether a terminal state has been reached and a dictionary for debugging and logging information.
- `render` is called to display the current environment state in a way the creator of the environment specifies. This function can be used to inspect the training results.

`stable-baselines3` expands on the default gym environment by providing an interface for vectorized environments. This makes it possible to compute the forward pass for multiple trajectories simultaneously which can in turn increase time efficiency because of better resource utilization. In practice, this means that the methods now work on vectors of

observations, actions, rewards, terminal state flags and info dictionaries. The step method is split into `step_async` and `step_wait`, making it possible to run individual instances of the environment on different threads.

35.4.2 Physics simulation

The environment for Burgers' equation contains a `Burgers` physics object provided by `phiflow`. The states are internally stored as `BurgersVelocity` objects. To create the initial states, the environment generates batches of random fields in the same fashion as in the data set generation process shown above. The observation space consists of the velocity fields of the current and target states stacked in the channel dimension with another channel specifying the current time step. Actions are taken in the form of a one dimensional array covering every velocity value. The `step` method calls the physics object to advance the internal state by one time step, also applying the actions as a `FieldEffect`.

The rewards encompass a penalty equal to the square norm of the generated forces at every time step. Additionally, the L^2 distance to the target field, scaled by a predefined factor (`FINAL_REWARD_FACTOR`) is subtracted at the end of each trajectory. The rewards are then normalized with a running estimate for the reward mean and standard deviation.

35.4.3 Neural network setup

We use two different neural network architectures for the actor and critic respectively. The former uses the U-Net variant from [HKT19], while the latter consists of a series of 1D convolutional and pooling layers reducing the feature map size to one. The final operation is a convolution with kernel size one to combine the feature maps and retain one output value. The `CustomActorCriticPolicy` class then makes it possible to use these two separate network architectures for the reinforcement learning agent.

By default, an agent is stored at `PDE-Control-RL/networks/rl-models/bench`, and loaded if it exists. (If necessary, replace the `path` below with a new one to start with a new model.) As the training takes quite long, we're starting with a pre-trained agent here. It is already trained for 3500 iterations, and hence we're only doing a "fine-tuning" below with another `RL_ROLLOUTS=500` iterations. These typically take around 2 hours, and hence the total training time of almost 18 hours would be too long for interactive tests. (However, the code provided here contains everything to train a model from scratch if you have the resources.)

```
rl_trainer = BurgersTraining(
    path='PDE-Control-RL/networks/rl-models/bench', # Replace path to train a new␣
↪model
    domain=DOMAIN,
    viscosity=VISCOSITY,
    step_count=STEP_COUNT,
    dt=DT,
    diffusion_substeps=DIFFUSION_SUBSTEPS,
    n_envs=N_ENVS,
    final_reward_factor=FINAL_REWARD_FACTOR,
    steps_per_rollout=STEPS_PER_ROLLOUT,
    n_epochs=N_EPOCHS,
    learning_rate=RL_LEARNING_RATE,
    batch_size=RL_BATCH_SIZE,
    data_path=DATA_PATH,
    val_range=VAL_RANGE,
    test_range=TEST_RANGE,
)
```

```
Tensorboard log path: PDE-Control-RL/networks/rl-models/bench/tensorboard-log
Loading existing agent from PDE-Control-RL/networks/rl-models/bench/agent.zip
```

The following cell is optional but very useful for debugging: it opens *tensorboard* inside the notebook to display the progress of the training. If a new model was created at a different location, please change the path accordingly. When

resuming the learning process of a pre-trained agent, the new run is shown separately in tensorboard (enable reload via the cogwheel button).

The graph titled "forces" shows how the overall amount of forces generated by the network evolves during training. "rew_unnormalized" shows the reward values without the normalization step described above. The corresponding values with normalization are shown under "rollout/ep_rew_mean". "val_set_forces" outlines the performance of the agent on the validation set.

```
%load_ext tensorboard
%tensorboard --logdir PDE-Control-RL/networks/rl-models/bench/tensorboard-log
```

Now we are set up to start training the agent. The RL approach requires many iterations to explore the environment. Hence, the next cell typically takes multiple hours to execute (around 2h for 500 rollouts).

```
rl_trainer.train(n_rollouts=RL_ROLLOUTS, save_freq=50)
```

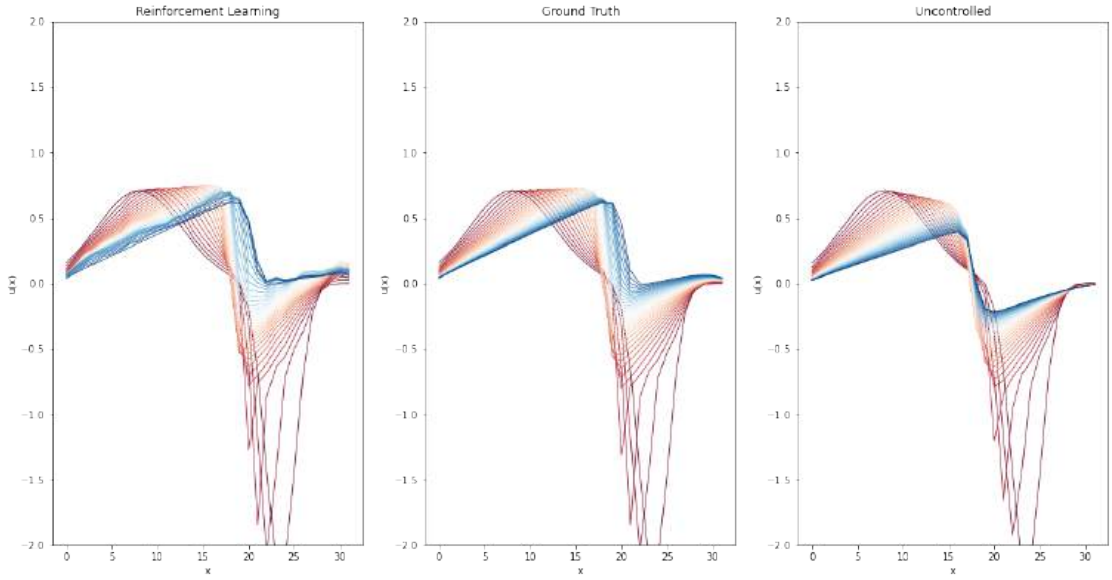
```
Storing agent and hyperparameters to disk...
```

```
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
Storing agent and hyperparameters to disk...
```

35.5 RL evaluation

Now that we have a trained model, let's take a look at the results. The leftmost plot shows the results of the reinforcement learning agent. As reference, next to it are shown the ground truth, i.e. the trajectory the agent should reconstruct, and the uncontrolled simulation where the system follows its natural evolution.

```
TEST_SAMPLE = 0    # Change this to display a reconstruction of another scene
rl_frames, gt_frames, unc_frames = rl_trainer.infer_test_set_frames()

fig, axs = plt.subplots(1, 3, figsize=(18.9, 9.6))
axs[0].set_title("Reinforcement Learning"); axs[1].set_title("Ground Truth"); axs[2].
 ↪set_title("Uncontrolled")
for plot in axs:
    plot.set_ylim(-2, 2); plot.set_xlabel('x'); plot.set_ylabel('u(x)')

for frame in range(0, STEP_COUNT + 1):
    frame_color = bplt.gradient_color(frame, STEP_COUNT+1);
    axs[0].plot(rl_frames[frame][TEST_SAMPLE,:], color=frame_color, linewidth=0.8)
    axs[1].plot(gt_frames[frame][TEST_SAMPLE,:], color=frame_color, linewidth=0.8)
    axs[2].plot(unc_frames[frame][TEST_SAMPLE,:], color=frame_color, linewidth=0.8)
```



As we can see, a trained reinforcement learning agent is able to reconstruct the trajectories fairly well. However, they still appear noticeably less smooth than the ground truth.

35.6 Differentiable physics training

To classify the results of the reinforcement learning method, we now compare them to an approach using differentiable physics training. In contrast to the full approach from *Solving Inverse Problems with NNs* which includes a second *OP* network, we aim for a direct control here. The OP network represents a separate "physics-predictor", which is omitted here for fairness when comparing with the RL version.

The DP approach has access to the gradient data provided by the differentiable solver, making it possible to trace the loss over multiple time steps and enabling the model to comprehend long term effects of generated forces better. The reinforcement learning algorithm, on the other hand, is not limited by training set size like the DP algorithm, as new training samples are generated on policy. However, this also introduces additional simulation overhead during training, which can increase the time needed for convergence.

```
from control.pde.burgers import BurgersPDE
from control.control_training import ControlTraining
from control.sequences import StaggeredSequence
```

```
Could not load resample cuda libraries: CUDA binaries not found at /usr/local/lib/
↪python3.7/dist-packages/phi/tf/cuda/build/resample.so. Run "python setup.py cuda
↪" to compile them
```

The cell below sets up a model for training or to load an existing model checkpoint.

```
dp_app = ControlTraining(
    STEP_COUNT,
    BurgersPDE(DOMAIN, VISCOSITY, DT),
    datapath=DATA_PATH,
    val_range=VAL_RANGE,
    train_range=TRAIN_RANGE,
    trace_to_channel=lambda trace: 'burgers_velocity',
```

(continues on next page)

(continued from previous page)

```
    obs_loss_frames=[],
    trainable_networks=['CFE'],
    sequence_class=StaggeredSequence,
    batch_size=100,
    view_size=20,
    learning_rate=1e-3,
    learning_rate_half_life=1000,
    dt=DT
).prepare()
```

```
App created. Scene directory is /root/phi/model/control-training/sim_000000 (INFO),
 ↪ 2021-08-04 10:11:58,466n

Sequence class: <class 'control.sequences.StaggeredSequence'> (INFO), 2021-08-04␣
 ↪10:12:01,449n

Partition length 32 sequence (from 0 to 32) at frame 16
```

```
Partition length 16 sequence (from 0 to 16) at frame 8
Partition length 8 sequence (from 0 to 8) at frame 4
Partition length 4 sequence (from 0 to 4) at frame 2
Partition length 2 sequence (from 0 to 2) at frame 1
Execute -> 1
Execute -> 2
Partition length 2 sequence (from 2 to 4) at frame 3
Execute -> 3
Execute -> 4
Partition length 4 sequence (from 4 to 8) at frame 6
Partition length 2 sequence (from 4 to 6) at frame 5
Execute -> 5
Execute -> 6
Partition length 2 sequence (from 6 to 8) at frame 7
Execute -> 7
Execute -> 8
Partition length 8 sequence (from 8 to 16) at frame 12
Partition length 4 sequence (from 8 to 12) at frame 10
Partition length 2 sequence (from 8 to 10) at frame 9
Execute -> 9
Execute -> 10
Partition length 2 sequence (from 10 to 12) at frame 11
Execute -> 11
Execute -> 12
Partition length 4 sequence (from 12 to 16) at frame 14
Partition length 2 sequence (from 12 to 14) at frame 13
Execute -> 13
Execute -> 14
Partition length 2 sequence (from 14 to 16) at frame 15
Execute -> 15
Execute -> 16
Partition length 16 sequence (from 16 to 32) at frame 24
Partition length 8 sequence (from 16 to 24) at frame 20
Partition length 4 sequence (from 16 to 20) at frame 18
Partition length 2 sequence (from 16 to 18) at frame 17
Execute -> 17
Execute -> 18
Partition length 2 sequence (from 18 to 20) at frame 19
```

(continues on next page)

(continued from previous page)

```
Execute -> 19
Execute -> 20
Partition length 4 sequence (from 20 to 24) at frame 22
Partition length 2 sequence (from 20 to 22) at frame 21
Execute -> 21
Execute -> 22
Partition length 2 sequence (from 22 to 24) at frame 23
Execute -> 23
Execute -> 24
Partition length 8 sequence (from 24 to 32) at frame 28
Partition length 4 sequence (from 24 to 28) at frame 26
Partition length 2 sequence (from 24 to 26) at frame 25
Execute -> 25
Execute -> 26
Partition length 2 sequence (from 26 to 28) at frame 27
Execute -> 27
Execute -> 28
Partition length 4 sequence (from 28 to 32) at frame 30
Partition length 2 sequence (from 28 to 30) at frame 29
Execute -> 29
Execute -> 30
Partition length 2 sequence (from 30 to 32) at frame 31
Execute -> 31
Execute -> 32
Target loss: Tensor("truediv_1:0", shape=(), dtype=float32) (INFO), 2021-08-04␣
 ↪10:13:10,701n

Force loss: Tensor("Sum_97:0", shape=(), dtype=float32) (INFO), 2021-08-04␣
 ↪10:13:14,221n

Setting up loss (INFO), 2021-08-04 10:13:14,223n

Preparing data (INFO), 2021-08-04 10:13:51,128n

INFO:tensorflow:Summary name Total Force is illegal; using Total_Force instead.
Initializing variables (INFO), 2021-08-04 10:13:51,156n

Model variables contain 0 total parameters. (INFO), 2021-08-04 10:13:55,961n

Validation (000000): Learning_Rate: 0.001, Loss_reg_unscaled: 205.98526, Loss_reg_
 ↪scale: 1.0, Loss: 0.0, Total Force: 393.8109 (INFO), 2021-08-04 10:14:32,455n
```

Now we can execute the model training. This cell typically also takes a while to execute (ca. 2h for 1000 iterations).

```
DP_TRAINING_ITERATIONS = 10000  # Change this to change training duration

dp_training_eval_data = []
start_time = time.time()

for epoch in range(DP_TRAINING_ITERATIONS):
    dp_app.progress()
    # Evaluate validation set at regular intervals to track learning progress
    # Size of intervals determined by RL epoch count per iteration for accurate␣
 ↪comparison
    if epoch % N_EPOCHS == 0:
        f = dp_app.infer_scalars(VAL_RANGE)['Total Force'] / DT
        dp_training_eval_data.append((time.time() - start_time, epoch, f))
```

The trained model and the validation performance `val_forces.csv` with respect to iterations and wall time are saved on disk:

```
DP_STORE_PATH = 'networks/dp-models/bench'
if not os.path.exists(DP_STORE_PATH):
    os.makedirs(DP_STORE_PATH)

# store training progress information
with open(os.path.join(DP_STORE_PATH, 'val_forces.csv'), 'at') as log_file:
    logger = csv.DictWriter(log_file, ('time', 'epoch', 'forces'))
    logger.writeheader()
    for (t, e, f) in dp_training_eval_data:
        logger.writerow({'time': t, 'epoch': e, 'forces': f})

dp_checkpoint = dp_app.save_model()
shutil.move(dp_checkpoint, DP_STORE_PATH)
```

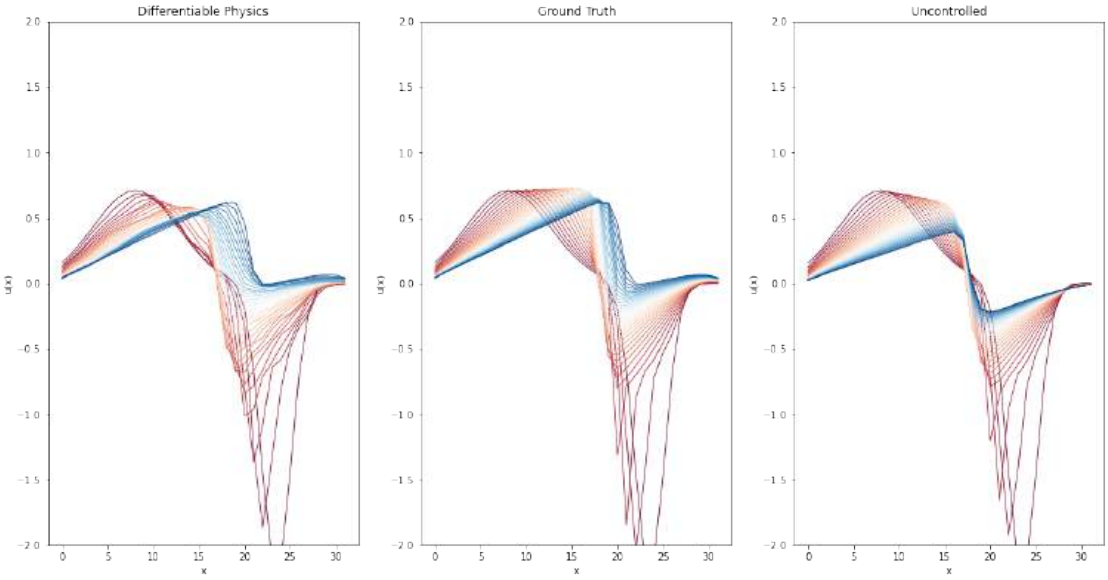
```
'networks/dp-models/bench/checkpoint_00010000'
```

Alternatively, uncomment the code in the cell below to load an existing network model.

```
# dp_path = 'PDE-Control-RL/networks/dp-models/bench/checkpoint_00020000/'
# networks_to_load = ['OP2', 'OP4', 'OP8', 'OP16', 'OP32']

# dp_app.load_checkpoints({net: dp_path for net in networks_to_load})
```

Similar to the RL version, the next cell plots an example to visually show how well the DP-based model does. The leftmost plot again shows the learned results, this time of the DP-based model. Like above, the other two show the ground truth and the natural evolution.

```
dp_frames = dp_app.infer_all_frames(TEST_RANGE)
dp_frames = [s.burgers.velocity.data for s in dp_frames]
_, gt_frames, unc_frames = rl_trainer.infer_test_set_frames()

TEST_SAMPLE = 0    # Change this to display a reconstruction of another scene
fig, axs = plt.subplots(1, 3, figsize=(18.9, 9.6))

axs[0].set_title("Differentiable Physics")
axs[1].set_title("Ground Truth")
axs[2].set_title("Uncontrolled")

for plot in axs:
    plot.set_ylim(-2, 2)
    plot.set_xlabel('x')
    plot.set_ylabel('u(x)')

for frame in range(0, STEP_COUNT + 1):
    frame_color = bplt.gradient_color(frame, STEP_COUNT+1)
    axs[0].plot(dp_frames[frame][TEST_SAMPLE,:], color=frame_color, linewidth=0.8)
    axs[1].plot(gt_frames[frame][TEST_SAMPLE,:], color=frame_color, linewidth=0.8)
    axs[2].plot(unc_frames[frame][TEST_SAMPLE,:], color=frame_color, linewidth=0.8)
```



The trained DP model also reconstructs the original trajectories closely. Furthermore, the generated results seem less noisy than using the RL agent.

With this, we have an RL and a DP version, which we can compare in more detail in the next section.

35.7 Comparison between RL and DP

Next, the results of both methods are compared in terms of visual quality of the resulting trajectories as well as quantitatively via the amount of generated forces. The latter provides insights about the performance of either approaches as both methods aspire to minimize this metric during training. This is also important as the task is trivially solved by applying a huge force at the last time step. Therefore, an ideal solution takes into account the dynamics of the PDE to apply as little forces as possible. Hence, this metric is a very good one to measure how well the network has learned about the underlying physical environment (Burgers equation in this example).

```
import utils
import pandas as pd
```

35.7.1 Trajectory comparison

To compare the resulting trajectories, we generate trajectories from the test set with either method. Also, we collect the ground truth simulations and the natural evolution of the test set fields.

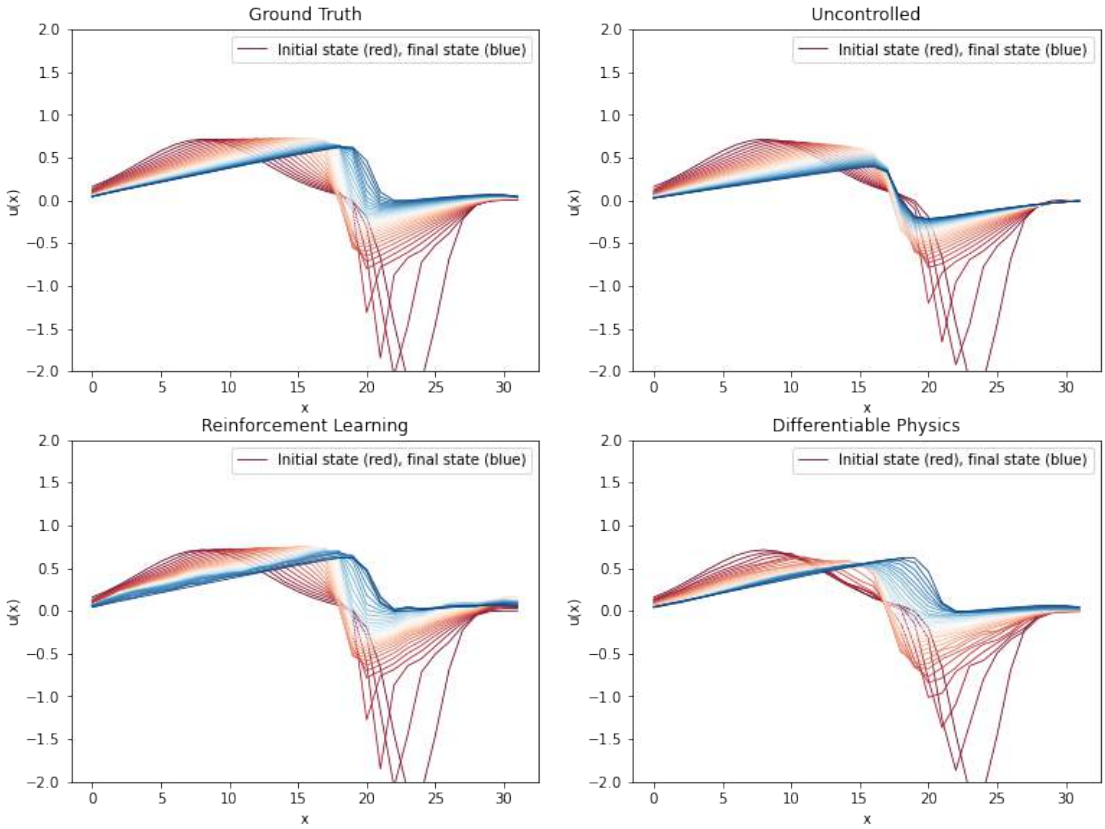
```
rl_frames, gt_frames, unc_frames = rl_trainer.infer_test_set_frames()

dp_frames = dp_app.infer_all_frames(TEST_RANGE)
dp_frames = [s.burgers.velocity.data for s in dp_frames]

frames = {
    (0, 0): ('Ground Truth', gt_frames),
    (0, 1): ('Uncontrolled', unc_frames),
```

(continues on next page)

(continued from previous page)

```
    (1, 0): ('Reinforcement Learning', rl_frames),
    (1, 1): ('Differentiable Physics', dp_frames),
}
```

```
TEST_SAMPLE = 0  # Specifies which sample of the test set should be displayed

def plot(axs, xy, title, field):
    axs[xy].set_ylim(-2, 2); axs[xy].set_title(title)
    axs[xy].set_xlabel('x'); axs[xy].set_ylabel('u(x)')
    label = 'Initial state (red), final state (blue)'
    for step_idx in range(0, STEP_COUNT + 1):
        color = bplt.gradient_color(step_idx, STEP_COUNT+1)
        axs[xy].plot(
            field[step_idx][TEST_SAMPLE].squeeze(), color=color, linewidth=0.8,␣
↪label=label)
        label = None
    axs[xy].legend()

fig, axs = plt.subplots(2, 2, figsize=(12.8, 9.6))
for xy in frames:
    plot(axs, xy, *frames[xy])
```



This diagram connects the two plots shown above after each training. Here we again see that the differentiable physics approach seems to generate less noisy trajectories than the RL agent, while both manage to approximate the ground truth.

35.7.2 Comparison of exerted forces

Next, we compute the forces the approaches have generated and applied for the test set trajectories.

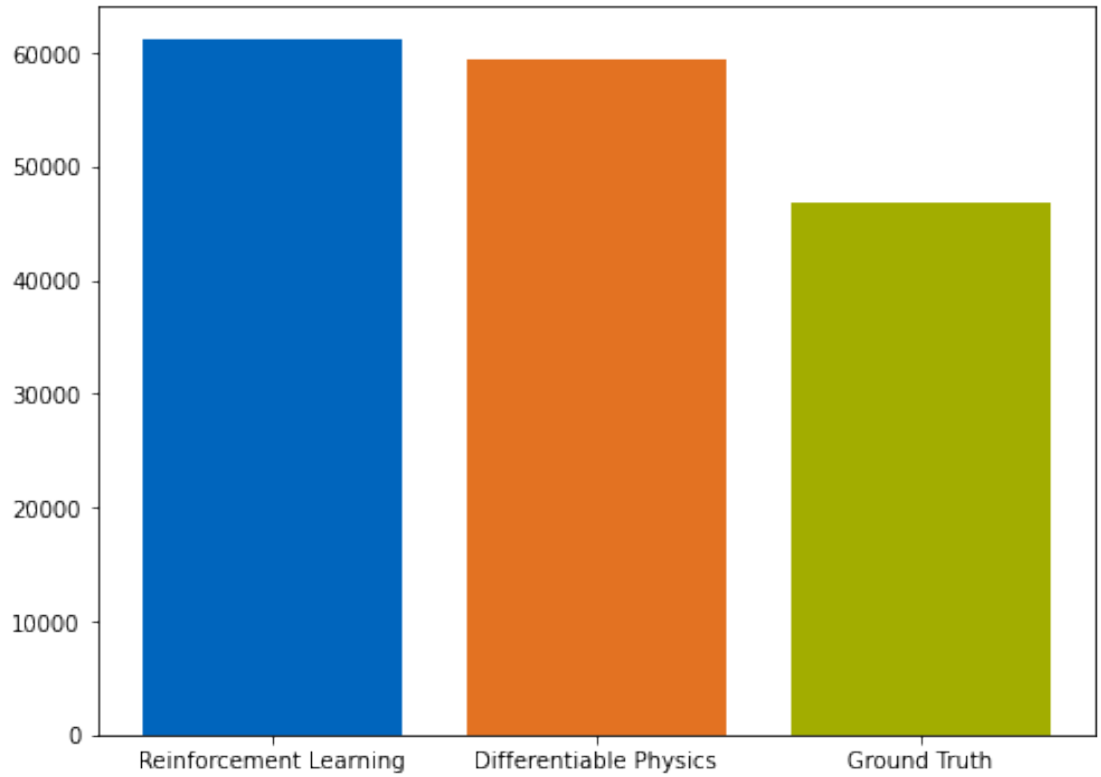
```
gt_forces = utils.infer_forces_sum_from_frames(
    gt_frames, DOMAIN, DIFFUSION_SUBSTEPS, VISCOSITY, DT
)
dp_forces = utils.infer_forces_sum_from_frames(
    dp_frames, DOMAIN, DIFFUSION_SUBSTEPS, VISCOSITY, DT
)
rl_forces = rl_trainer.infer_test_set_forces()
```

```
Sanity check - maximum deviation from target state: 0.000000
Sanity check - maximum deviation from target state: 0.000011
```

At first, we will compare the total sum of the forces that are generated by the RL and DP approaches and compare them to the ground truth.

```
plt.figure(figsize=(8, 6))
plt.bar(
    ["Reinforcement Learning", "Differentiable Physics", "Ground Truth"],
    [np.sum(rl_forces), np.sum(dp_forces), np.sum(gt_forces)],
    color = ["#0065bd", "#e37222", "#a2ad00"],
    align='center', label='Absolute forces comparison' )
```

```
<BarContainer object of 3 artists>
```



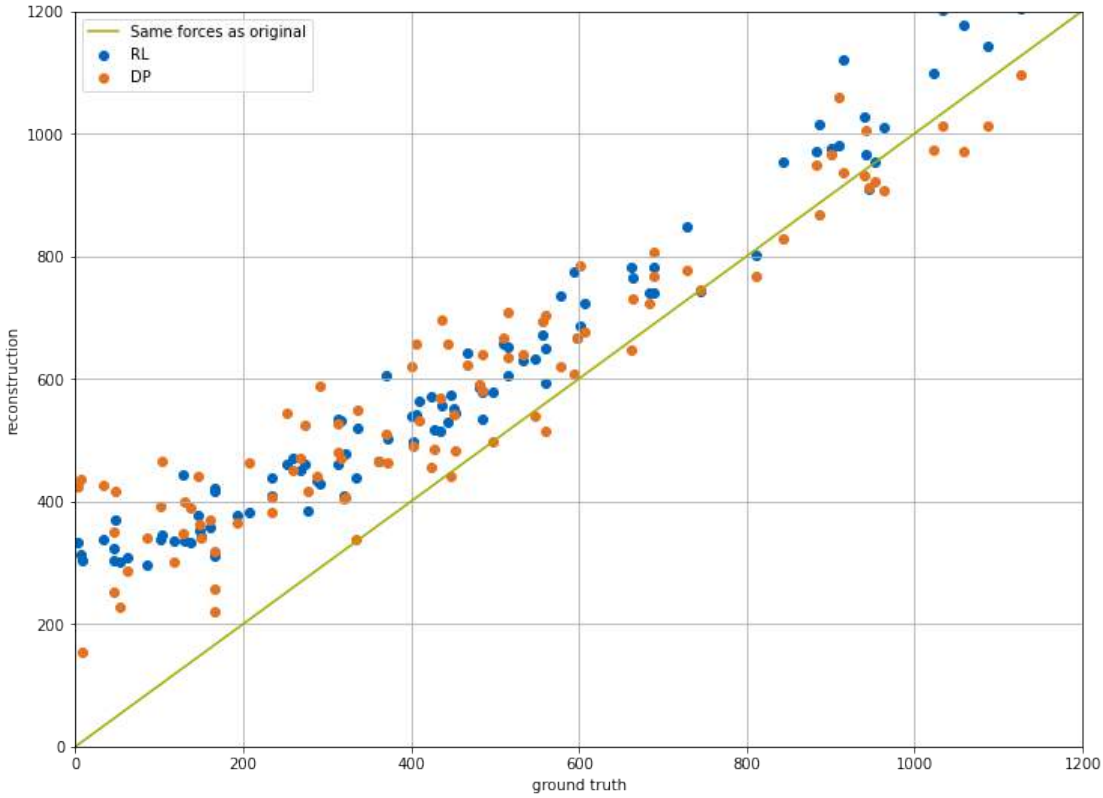
As visualized in these bar plots, the DP approach learns to apply slightly lower forces than the RL model. As both

methods are on-par in terms of how well they reach the final target states, this is the main quantity we use to compare the performance of both methods.

In the following, the forces generated by the methods are also visually compared to the ground truth of the respective sample. Dots placed above the blue line denote stronger forces in the analyzed deep learning approach than in the ground truth and vice versa.

```
plt.figure(figsize=(12, 9))
plt.scatter(gt_forces, rl_forces, color="#0065bd", label='RL')
plt.scatter(gt_forces, dp_forces, color="#e37222", label='DP')
plt.plot([x * 100 for x in range(15)], [x * 100 for x in range(15)], color="#a2ad00",␣
 ↪label='Same forces as original')
plt.xlabel('ground truth'); plt.ylabel('reconstruction')
plt.xlim(0, 1200); plt.ylim(0, 1200); plt.grid(); plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f4cbc6d5090>
```



The graph shows that the orange dots of the DP training run are in general closer to the diagonal - i.e., this network learned to generate forces that are closer to the ground truth values.

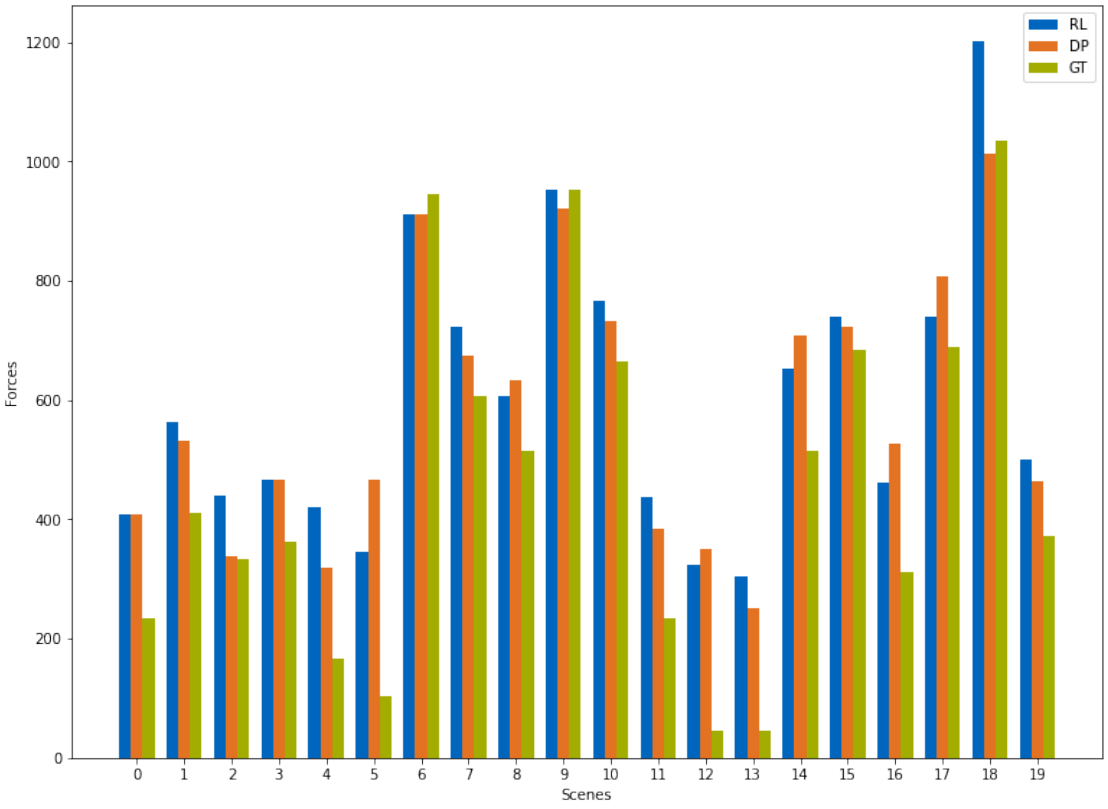
The following plot displays the performance of all reinforcement learning, differentiable physics and ground truth with respect to individual samples.

```
w=0.25; plot_count=20   # How many scenes to show
plt.figure(figsize=(12.8, 9.6))
plt.bar( [i - w for i in range(plot_count)], rl_forces[:plot_count], color="#0065bd",␣
 ↪width=w, align='center', label='RL' )
plt.bar( [i     for i in range(plot_count)], dp_forces[:plot_count], color="#e37222",␣
```

(continues on next page)

(continued from previous page)

```
 ↪width=w, align='center', label='DP' )
plt.bar( [i + w for i in range(plot_count)], gt_forces[:plot_count], color="#a2ad00",␣
 ↪width=w, align='center', label='GT' )
plt.xlabel('Scenes'); plt.xticks(range(plot_count))
plt.ylabel('Forces'); plt.legend(); plt.show()
```



35.8 Training progress comparison

Although the quality of the control in terms of force magnitudes is the primary goal of the setup above, there are interesting differences in terms of how both methods behave at training time. The main difference of the physics-unaware RL training and the DP approach with its tightly coupled solver is that the latter results in a significantly faster convergence. I.e., the gradients provided by the numerical solver give a much better learning signal than the undirected exploration of the RL process. The behavior of the RL training, on the other hand, can in part be ascribed to the on-policy nature of training data collection and to the "brute-force" exploration of the reinforcement learning technique.

The next cell visualizes the training progress of both methods with respect to wall time.

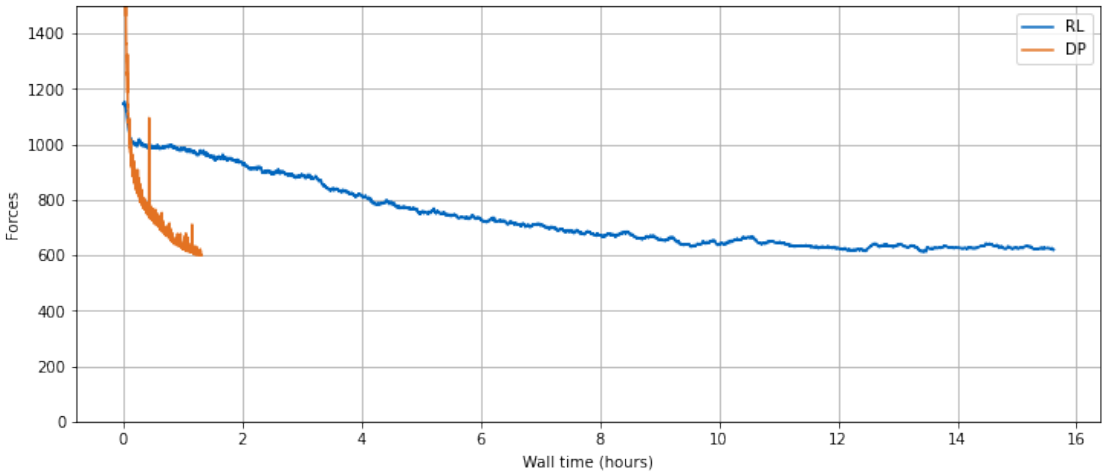
```
def get_dp_val_set_forces(experiment_path):
    path = os.path.join(experiment_path, 'val_forces.csv')
    table = pd.read_csv(path)
    return list(table['time']), list(table['epoch']), list(table['forces'])

rl_w_times, rl_step_nums, rl_val_forces = rl_trainer.get_val_set_forces_data()
```

(continues on next page)

(continued from previous page)

```
dp_w_times, dp_epochs, dp_val_forces = get_dp_val_set_forces(DP_STORE_PATH)

plt.figure(figsize=(12, 5))
plt.plot(np.array(rl_w_times) / 3600, rl_val_forces, color="#0065bd", label='RL')
plt.plot(np.array(dp_w_times) / 3600, dp_val_forces, color="#e37222", label='DP')
plt.xlabel('Wall time (hours)'); plt.ylabel('Forces')
plt.ylim(0, 1500); plt.grid(); plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f4cbc595c50>
```



To conclude, the PPO reinforcement learning exerts higher forces in comparison to the differentiable physics approach. Hence, PPO yields a learned solution with slightly inferior quality. Additionally, the time needed for convergence is significantly higher in the RL case (both in terms of wall time and training iterations).

35.9 Next steps

- See how different values for hyperparameters, such as learning rate, influence the training process
- Work with fields of different resolution and see how the two approaches then compare to each other. Larger resolutions make the physical dynamics more complex, and hence harder to control
- Use trained models in settings with different environment parameters (e.g. viscosity, dt) and test how well they generalize

Part VII

Improved Gradients

CHAPTER

THIRTYSIX

SCALE-INVARIANCE AND INVERSION

In the following we will question some fundamental aspects of the formulations so far, namely the update step computed via gradients. To re-cap, the approaches explained in the previous chapters either dealt with purely *supervised* training, integrated the physical model as a *physical loss term* or included it via *differentiable physics* (DP) operators embedded into the training graph. The latter two methods are more relevant in the context of this book. They share similarities, but in the loss term case, the physics evaluations are only required at training time. For DP approaches, the solver itself is usually also employed at inference time, which enables an end-to-end training of NNs and numerical solvers. All three approaches employ *first-order* derivatives to drive optimizations and learning processes, and the latter two also using them for the physics terms. This is a natural choice from a deep learning perspective, but we haven't questioned at all whether this is actually the best choice.

Not too surprising after this introduction: A central insight of the following chapter will be that regular gradients can be a *sub-optimal choice* for learning problems involving physical quantities. It turns out that both supervised and DP gradients have their pros and cons, and leave room for custom methods that are aware of the physics operators. In particular, we'll show how scaling problems of DP gradients affect NN training (as outlined in [HKT22]), and revisit the problems of multi-modal solutions. Finally, we'll explain several alternatives to prevent these issues.

A preview of this chapter

Below, we'll proceed in the following steps:

- Show how the properties of different optimizers and the associated scaling issues can negatively affect NN training.
- Identify the problem with our GD or Adam training runs so far. Spoiler: they're missing an *inversion* process to make the training scale-invariant.
- We'll then explain two alternatives to alleviate these problems: an analytical full-, and a numerical half-inversion scheme.

36.1 The crux of the matter

Before diving into the details of different optimizers, the following paragraphs should provide some intuition for why this inversion is important. As mentioned above, all methods discussed so far use gradients, which come with fundamental scaling issues: even for relatively simple linear cases, the direction of the gradient can be negatively distorted, thus preventing effective progress towards the minimum. (In non-linear settings, the length of the gradient anticorrelates with the distance from the minimum point, making it even more difficult to converge.)

In 1D, this problem can alleviated by tweaking the learning rate, but it becomes very clear in higher dimensions. Let's consider a very simple toy "physics" function in two dimensions that simply applies a factor α to the second component,

followed by an L^2 loss:

$$\mathcal{P}(x_1, x_2) = \begin{bmatrix} x_1 \\ \alpha\, x_2 \end{bmatrix} \text{ with } L(\mathcal{P}) = |\mathcal{P}|^2$$

For $\alpha = 1$ everything is very simple: we're faced with a radially symmetric loss landscape, and x_1 and x_2 behave in the same way. The gradient $\nabla_x = (\partial L / \partial x)^T$ is perpendicular to the isolines of the loss landscape, and hence an update with $-\eta \nabla_x$ points directly to the minimum at 0. This is a setting we're dealing with for classical deep learning scenarios, like most supervised learning cases or classification problems. This example is visualized on the left of the following figure.

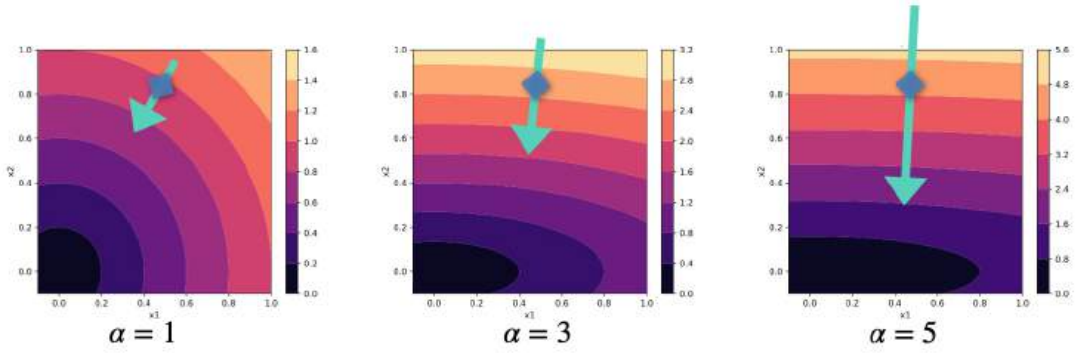


Fig. 36.1: Loss landscapes in x for different α of the 2D example problem. The green arrows visualize an example update step $-\nabla_x$ (not exactly to scale) for each case.

However, within this book we're targeting *physical* learning problems, and hence we have physical functions integrated into the learning process, as discussed at length for differentiable physics approaches. This is fundamentally different! Physical processes pretty much always introduce different scaling behavior for different components: some changes in the physical state are sensitive and produce massive responses, others have barely any effect. In our toy problem we can mimic this by choosing different values for α, as shown in the middle and right graphs of the figure above.

For larger α, the loss landscape away from the minimum steepens along x_2. x_1 will have an increasingly different scale than x_2. As a consequence, the gradients grow along this x_2. If we don't want our optimization to blow up, we'll need to choose a smaller learning rate η, reducing progress along x_1. The gradient of course stays perpendicular to the loss. In this example we'll move quickly along x_2 until we're close to the x axis, and then only very slowly creep left towards the minimum. Even worse, as we'll show below, regular updates actually apply the square of the scaling! And in settings with many dimensions, it will be extremely difficult to find a good learning rate. Thus, to make proper progress, we somehow need to account for the different scaling of the components of multi-dimensional functions. This requires some form of *inversion*, as we'll outline in detail below.

Note that inversion, naturally, does not mean negation ($g^{-1} \neq -g$?). A negated gradient would definitely move in the wrong direction. We need an update that still points towards a decreasing loss, but accounts for differently scaled dimensions. Hence, a central aim in the following will be *scale-invariance*.

Definition of *scale-invariance*

A scale-invariant optimization for a given function yields the same result for different parametrizations (i.e. scalings) of the function.

E.g., for our toy problem above this means that optimization trajectories are identical no matter what value we choose for α.

36.2 Traditional optimization methods

We'll now evaluate and discuss how different optimizers perform in comparison. As before, let $L(x)$ be a scalar loss function, subject to minimization. The goal is to compute a step in terms of the input parameters x , denoted by Δx. Below, we'll compute different versions of Δx that will be distinguished by a subscript.

All NNs of the previous chapters were trained with gradient descent (GD) via backpropagation. GD with backprop was also employed for the PDE solver (*simulator*) $\mathcal{P}$, resulting in the DP training approach. When we simplify the setting, and leave out the NN for a moment, this gives the minimization problem $\arg\min_x L(x)$ with $L(x) = 1/2 \left\| \mathcal{P}(x) - y^* \right\|_2^2$. As a central quantity, we have the composite gradient $(\partial L / \partial x)^T$ of the loss function L:

$$\left(\frac{\partial L}{\partial x} \right)^T = \left(\frac{\partial \mathcal{P}}{\partial x} \right)^T \left(\frac{\partial L}{\partial \mathcal{P}} \right)^T \tag{36.1}$$

As the $(\cdots)^T$ notation makes things difficult to read, and we're effectively only dealing with transposed Jacobians, we'll omit the ^{T} in the following.

We've shown in previous chapters that using $\partial L / \partial x$ works, but in the field of classical optimization, other algorithms are more widely used than GD: popular are so-called Quasi-Newton methods, which use fundamentally different updates. Hence, in the following we'll revisit GD along with Quasi-Newton methods and Inverse Jacobians as a third alternative. We'll focus on the pros and cons of the different methods on a theoretical level. Among others, it's interesting to discuss why classical optimization algorithms aren't widely used for NN training despite having some obvious advantages.

Note that we exclusively consider multivariate functions, and hence all symbols represent vector-valued expressions unless noted otherwise.

36.3 Gradient descent

The optimization updates Δx_{GD} of GD scale with the derivative of the objective w.r.t. the inputs,

$$\Delta x_{\text{GD}} = -\eta \cdot \frac{\partial L}{\partial x} \tag{36.2}$$

where η is the scalar learning rate. The Jacobian $\frac{\partial L}{\partial x}$ describes how the loss reacts to small changes of the input. Surprisingly, this very widely used update has a number of undesirable properties that we'll highlight in the following. Note that we've naturally applied this update in supervised settings such as *Supervised training for RANS flows around airfoils*, but we've also used it in the differentiable physics approaches. E.g., in *Reducing Numerical Errors with Neural Operators* we've computed the derivative of the fluid solver. In the latter case, we've still only updated the NN parameters, but the fluid solver Jacobian was part of equation (36.2), as shown in (36.1).

We'll jointly evaluate GD and several other methods with respect to a range of categories: their handling of units, function sensitivity, and behavior near optima. While these topics are related, they illustrate differences and similarities of the approaches.

Units ?

A first indicator that something is amiss with GD is that it inherently misrepresents dimensions. Assume two parameters x_1 and x_2 have different physical units. Then the GD parameter updates scale with the inverse of these units because the parameters appear in the denominator for the GD update above ($\cdots / \partial x$). The learning rate η could compensate for

this discrepancy but since x_1 and x_2 have different units, there exists no single η to produce the correct units for both parameters.

One could argue that units aren't very important for the parameters of NNs, but nonetheless it's unnerving from a physics perspective that they're wrong, and it hints at some more fundamental problems.

Function sensitivity ?

As illustrated above, GD has also inherent problems when functions are not *normalized*. Consider a simplified version of the toy example above, consisting only of the function $L(x) = \alpha \cdot x$. Then the parameter updates of GD scale with α, i.e. $\Delta x_{\text{GD}} = -\eta \cdot \alpha$, and $L(x + \Delta x_{\text{GD}})$ will even have terms on the order of α^2. If L is normalized via $\alpha = 1$, everything's fine. But in practice, we'll often have $\alpha \ll 1$, or even worse $\alpha \gg 1$, and then our optimization will be in trouble.

More specifically, if we look at how the loss changes, the expansion around x for the update step of GD gives: $L(x + \Delta x_{\text{GD}}) = L(x) + \Delta x_{\text{GD}} \frac{\partial L}{\partial x} + \cdots$. This first-order step causes a change in the loss of $(L(x) - L(x + \Delta x_{\text{GD}})) = -\eta \cdot (\frac{\partial L}{\partial x})^2 + \mathcal{O}(\Delta x^2)$. Hence the loss changes by the squared derivative, which leads to the α^2 factor mentioned above. Even worse, in practice we'd like to have a normalized quantity here. For a scaling of the gradients by α, we'd like our optimizer to compute a quantity like $1/\alpha^2$, in order to get a reliable update from the gradient.

This demonstrates that for sensitive functions, i.e. functions where *small changes* in x cause *large* changes in L, GD counter-intuitively produces large Δx_{GD}. This causes even larger steps in L, and leads to exploding gradients. For insensitive functions where *large changes* in the input don't change the output L much, GD produces *small* updates, which can lead to the optimization coming to a halt. That's the classic *vanishing gradients* problem.

Such sensitivity problems can occur easily in complex functions such as deep neural networks where the layers are typically not fully normalized. Normalization in combination with correct setting of the learning rate η can be used to counteract this behavior in NNs to some extent, but these tools are not available when optimizing physics simulations. Applying normalization to a simulation anywhere but after the last solver step would destroy the state of the simulation. Adjusting the learning rate is also difficult in practice, e.g., when simulation parameters at different time steps are optimized simultaneously or when the magnitude of the simulation output varies w.r.t. the initial state.

Convergence near optimum ?

Finally, the loss landscape of any differentiable function necessarily becomes flat close to an optimum, as the gradient approaches zero upon convergence. Therefore $\Delta x_{\text{GD}} \to 0$ as the optimum is approached, resulting in slow convergence.

This is an important point, and we will revisit it below. It's also somewhat surprising at first, but it can actually stabilize the training. On the other hand, it makes the learning process difficult to control.

36.4 Quasi-Newton methods

Newton's method employs the gradient $\frac{\partial L}{\partial x}$ and the inverse of the Hessian $\frac{\partial^2 L}{\partial x^2}$ for the update

$$\Delta x_{\text{QN}} = -\eta \cdot \left(\frac{\partial^2 L}{\partial x^2} \right)^{-1} \frac{\partial L}{\partial x}. \tag{36.3}$$

More widely used in practice are Quasi-Newton methods, such as BFGS and its variants, which approximate the Hessian matrix. However, the resulting update Δx_{QN} stays the same. As a further improvement, the step size η is often determined via a line search (we'll leave out this step for now). This construction solves some of the problems of gradient descent from above, but has other drawbacks.

Units and Sensitivity ?

Quasi-Newton methods definitely provide a much better handling of physical units than GD. The quasi-Newton update from equation (36.3) produces the correct units for all parameters to be optimized. As a consequence, η can stay dimensionless.

If we now consider how the loss changes via $L(x + \Delta x_{\text{QN}}) = L(x) + -\eta \cdot \left(\frac{\partial^2 L}{\partial x^2}\right)^{-1} \frac{\partial L}{\partial x} \frac{\partial L}{\partial x} + \cdots$, the second term correctly cancels out the x quantities, and leaves us with a scalar update in terms of L. Thinking back to the example with a scaling factor α from the GD section, the inverse Hessian in Newton's methods successfully gives us a factor of $1/\alpha^2$ to counteract the undesirable scaling of our updates.

Convergence near optimum ?

Quasi-Newton methods also exhibit much faster convergence when the loss landscape is relatively flat. Instead of slowing down, they take larger steps, even when η is fixed. This is thanks to the eigenvalues of the inverse Hessian, which scale inversely with the eigenvalues of the Hessian, and hence increase with the flatness of the loss landscape.

Consistency in function compositions

So far, quasi-Newton methods address both shortcomings of GD. However, similar to GD, the update of an intermediate space still depends on all functions before that. This behavior stems from the fact that the Hessian of a composite function carries non-linear terms of the gradient.

Consider a function composition $L(y(x))$, with L as above, and an additional function $y(x)$. Then the Hessian $\frac{d^2 L}{dx^2} = \frac{\partial^2 L}{\partial y^2} \left(\frac{\partial y}{\partial x}\right)^2 + \frac{\partial L}{\partial y} \cdot \frac{\partial^2 y}{\partial x^2}$ depends on the square of the inner Jacobian $\frac{\partial y}{\partial x}$. This means that if we'd use this update in a backpropagation step, the Hessian is influenced by the *later* functions of the backprop chain. As a consequence, the update of any intermediate latent space is unknown during the computation of the gradients.

Dependence on Hessian ?

In addition, a fundamental disadvantage of quasi-Newton methods that becomes apparent from the discussion above is their dependence on the Hessian. It plays a crucial role for all the improvements discussed so far.

The first obvious drawback is the *computational cost*. While evaluating the exact Hessian only adds one extra pass to every optimization step, this pass involves higher-dimensional tensors than the computation of the gradient. As $\frac{\partial^2 L}{\partial x^2}$ grows with the square of the parameter count, both its evaluation and its inversion become very expensive for large systems. This is where Quasi-Newton methods spend significant efforts to compute approximations with a reasonable amount of resources, but it's nonetheless a central problem.

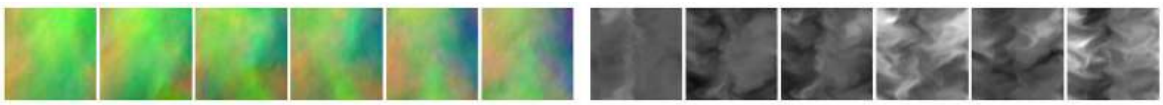
The quasi-Newton update above additionally requires the *inverse* Hessian matrix. Thus, a Hessian that is close to being non-invertible typically causes numerical stability problems, while inherently non-invertible Hessians require a fallback to a first order GD update.

Another related limitation of quasi-Newton methods is that the objective function needs to be *twice-differentiable*. While this may not seem like a big restriction, note that many common neural network architectures use ReLU activation functions of which the second-order derivative is zero.

Related to this is the problem that higher-order derivatives tend to change more quickly when traversing the parameter space, making them more prone to high-frequency noise in the loss landscape.

Note

Quasi-Newton Methods are still a very active research topic, and hence many extensions have been proposed that can alleviate some of these problems in certain settings. E.g., the memory requirement problem can be sidestepped by storing only lower-dimensional vectors that can be used to approximate the Hessian. However, these difficulties illustrate the problems that often arise when applying methods like BFGS.



36.5 Inverse gradients

As a first step towards fixing the aforementioned issues, we'll consider what we'll call *inverse* gradients (IGs). These methods actually use an inverse of the Jacobian, but as we always have a scalar loss at the end of the computational chain, this results in a gradient vector. Unfortunately, they come with their own set of problems, which is why they only represent an intermediate step (we'll revisit them in a more practical form later on).

Instead of L (which is scalar), let's consider optimization problems for a generic, potentially non-scalar function $y(x)$. This will typically be the physical simulator $\mathcal{P}$ later on, but to keep things general and readable, we'll call it y for now. This setup implies an inverse problem: for $y = \mathcal{P}(x)$ we want to find an x given a target y^*. We define the update

$$\Delta x_{\text{IG}} = \frac{\partial x}{\partial y} \cdot \Delta y. \tag{36.4}$$

to be the IG update. Here, the Jacobian $\frac{\partial x}{\partial y}$, which is similar to the inverse of the GD update above, encodes with first-order accuracy how the inputs must change in order to obtain a small change Δy in the output. The crucial step is the inversion, which of course requires the Jacobian matrix to be invertible. This is a problem somewhat similar to the inversion of the Hessian, and we'll revisit this issue below. However, if we can invert the Jacobian, this has some very nice properties.

Note that instead of using a learning rate, here the step size is determined by the desired increase or decrease of the value of the output, Δy. Thus, we need to choose a Δy instead of an η, but effectively has the same role: it controls the step size of the optimization. It the simplest case, we can compute it as a step towards the ground truth via $\Delta y = \eta\,(y^* - y)$. This Δy will show up frequently in the following equations, and make them look quite different to the ones above at first sight.

Units ?

IGs scale with the inverse derivative. Hence the updates are automatically of the same units as the parameters without requiring an arbitrary learning rate: $\frac{\partial x}{\partial y}$ times Δy has the units of x.

Function sensitivity ?

They also don't have problems with normalization as the parameter updates from the example $L(x) = \alpha \cdot x$ above now scale with α^{-1}. Sensitive functions thus receive small updates while insensitive functions get large (or exploding) updates.

Convergence near optimum and function compositions ?

Like Newton's method, IGs show the opposite behavior of GD close to an optimum: they produce updates that still progress the optimization, which usually improves convergence.

Additionally, IGs are consistent in function composition. The change in x is $\Delta x_{\text{IG}} = \Delta L \cdot \frac{\partial x}{\partial y} \frac{\partial y}{\partial L}$ and the approximate change in y is $\Delta y = \Delta L \cdot \frac{\partial y}{\partial x} \frac{\partial x}{\partial y} \frac{\partial y}{\partial L} = \Delta L \frac{\partial y}{\partial L}$. The change in intermediate spaces is independent of their respective dependencies, at least up to first order. Consequently, the change to these spaces can be estimated during backprop, before all gradients have been computed.

Note that even Newton's method with its inverse Hessian didn't fully get this right. The key here is that if the Jacobian is invertible, we'll directly get the correctly scaled direction at a given layer, without helper quantities such as the inverse Hessian.

Dependence on the inverse Jacobian ?

So far so good. The above properties are clearly advantageous, but unfortunately IGs require the inverse of the Jacobian, $\frac{\partial x}{\partial y}$. It is only well-defined for square Jacobians, meaning for functions y with the same inputs and output dimensions. In optimization, however, the input is typically high-dimensional while the output is a scalar objective function. And, somewhat similar to the Hessians of quasi-Newton methods, even when the $\frac{\partial y}{\partial x}$ is square, it may not be invertible.

Thus, we now consider the fact that inverse gradients are linearizations of inverse functions and show that using inverse functions provides additional advantages while retaining the same benefits.

36.6 Inverse simulators

So far we've discussed the problems of existing methods, and a common theme among the methods that do better, Newton and IGs, is that the regular gradient is not sufficient. We somehow need to address it's problems with some form of *inversion* to arrive at scale invariance. Before going into details of NN training and numerical methods to perform this inversion, we will consider one additional "special" case that will further illustrate the need for inversion: if we can make use of an *inverse simulator*, this likewise addresses many of the inherent issues of GD. It actually represents the ideal setting for computing update steps for the physics simulation part.

Let $y = \mathcal{P}(x)$ be a forward simulation, and $\mathcal{P}(y)^{-1} = x$ denote its inverse. In contrast to the inversion of Jacobian or Hessian matrices from before, $\mathcal{P}^{-1}$ denotes a full inverse of all functions of $\mathcal{P}$.

Trying to this employ inverse solver in the minimization problem from the top, somewhat surprisingly, makes the whole minimization obsolete (at least if we consider single cases with one x, y^* pair). We just need to evaluate $\mathcal{P}^{-1}(y^*)$ to solve the inverse problem and obtain x. As we plan to bring back NNs and more complex scenarios soon, let's assume that we are still dealing with a collection of y^* targets, and non-obvious solutions x. One example could be that we're looking for an x that yields multiple y^* targets with minimal distortions in terms of L^2.

Now, instead of evaluating $\mathcal{P}^{-1}$ once to obtain the solution, we can iteratively update a current approximation of the solution x_0 with an update that we'll call Δx_{PG} when employing the inverse physical simulator.

It also turns out to be a good idea to employ a *local* inverse that is conditioned on an initial guess for the solution x. We'll denote this local inverse with $\mathcal{P}^{-1}(y^*; x)$. As there are potentially very different x-space locations that result in very similar y^*, we'd like to find the one closest to the current guess. This is important to obtain well behaved solutions in multi-modal settings, where we'd like to avoid the solution manifold to consist of a set of very scattered points.

Equipped with these changes, we can formulate an optimization problem where a current state of the optimization x_0, with $y_0 = \mathcal{P}(x_0)$, is updated with

$$\Delta x_{\text{PG}} = \frac{(\mathcal{P}^{-1}(y_0 + \Delta y; x_0) - x_0)}{\Delta y} \cdot \Delta y. \tag{36.5}$$

Here the step in y-space, Δy, is either the full distance $y^* - y_0$ or a part of it, in line with the y-step used for IGs. When applying the update $\mathcal{P}^{-1}(y_0 + \Delta y; x_0) - x_0$ it will produce $\mathcal{P}(x_0 + \Delta x) = y_0 + \Delta y$ exactly, despite $\mathcal{P}$ being a potentially highly nonlinear function. Note that the Δy in equation (36.5) effectively cancels out to give a step in terms of x. However, this notation serves to show the similarities with the IG step from equation (36.4). The update Δx_{PG} gives us a first iterative method that makes use of $\mathcal{P}^{-1}$, and as such leverages all its information, such as higher-order terms.

36.7 Summary

The update obtained with a regular gradient descent method has surprising shortcomings due to scaling issues. Classical, inversion-based methods like IGs and Newton's method remove some of these shortcomings, with the somewhat theoretical construct of the update from inverse simulators (Δx_{PG}) including the most higher-order terms. Δx_{PG} can be seen as an "ideal" setting for improved (inverted) update steps. It gets all of the aspect above right: units ?, function sensitivity ?, compositions, and convergence near optima ?, and it provides a *scale-invariant* update. This comes at the cost of requiring an expression and discretization for a local inverse solver. ?

In contrast to the second- and first-order approximations from Newton's method and IGs, it can potentially take highly nonlinear effects into account. Due to the potentially difficult construct of the inverse simulator, the main goal of the following sections is to illustrate how much we can gain from including all the higher-order information. Note that all three methods successfully include a rescaling of the search direction via inversion, in contrast to the previously discussed GD training. All of these methods represent different forms of differentiable physics, though.

Before moving on to including improved updates in NN training processes, we will discuss some additional theoretical aspects, and then illustrate the differences between these approaches with a practical example.

Note

The following sections will provide an in-depth look ("deep-dive"), into optimizations with inverse solvers. If you're interested in practical examples and connections to NNs, feel free to skip ahead to *Simple Example comparing Different Optimizers* or *Scale Invariant Physics Training*, respectively.

36.8 Deep Dive into Inverse simulators

We'll now derive and discuss the Δx_{PG} update in more detail. Physical processes can be described as a trajectory in state space where each point represents one possible configuration of the system. A simulator typically takes one such state space vector and computes a new one at another time. The Jacobian of the simulator is, therefore, necessarily square. As long as the physical process does *not destroy* information, the Jacobian is non-singular. In fact, it is believed that information in our universe cannot be destroyed so any physical process could in theory be inverted as long as we have perfect knowledge of the state. Hence, it's not unreasonable to expect that $\mathcal{P}^{-1}$ can be formulated in many settings.

Update steps computed as described above also have some nice theoretical properties, e.g., that the optimization converges given that $\mathcal{P}^{-1}$ consistently a fixed target x^*. Details of the corresponding proofs can be found in [HKT22].

36.8.1 Fundamental theorem of calculus

To more clearly illustrate the advantages in non-linear settings, we apply the fundamental theorem of calculus to rewrite the ratio $\Delta x_{\text{PG}} / \Delta y$ from above. This gives,

$$\frac{\Delta x_{\text{PG}}}{\Delta y} = \frac{\int_{y_0}^{y_0+\Delta y} \frac{\partial x}{\partial y} \, dy}{\Delta y}$$

Here the expressions inside the integral is the local gradient, and we assume it exists at all points between y_0 and $y_0 + \Delta y_0$. The local gradients are averaged along the path connecting the state before the update with the state after the update. The whole expression is therefore equal to the average gradient of $\mathcal{P}$ between the current x and the estimate for the next optimization step $x_0 + \Delta x_{\text{PG}}$. This effectively amounts to *smoothing the objective landscape* of an optimization by computing updates that can take nonlinearities of $\mathcal{P}$ into account.

The equations naturally generalize to higher dimensions by replacing the integral with a path integral along any differentiable path connecting x_0 and $x_0 + \Delta x_{\text{PG}}$ and replacing the local gradient by the local gradient in the direction of the path.

36.8.2 Global and local inverse simulators

Let $\mathcal{P}$ be a function with a square Jacobian and $y = \mathcal{P}(x)$. A global inverse function $\mathcal{P}^{-1}$ is defined only for bijective $\mathcal{P}$. If the inverse exists, it can find x for any y such that $y = \mathcal{P}(x)$.

Instead of using this "perfect" inverse $\mathcal{P}^{-1}$ directly, we'll in practice often use a local inverse $\mathcal{P}^{-1}(y; x_0)$, which is conditioned for the point x_0, and correspondingly on $y_0 = \mathcal{P}(x_0)$. This local inverse is easier to obtain, as it only needs to exist near a given y_0, and not for all y. For the generic $\mathcal{P}^{-1}$ to exist $\mathcal{P}$ would need to be globally invertible.

By contrast, a *local inverse* only needs to exist and be accurate in the vicinity of (x_0, y_0). If a global inverse $\mathcal{P}^{-1}(y)$ exists, the local inverse approximates it and matches it exactly as $y \to y_0$. More formally, $\lim_{y \to y_0} \frac{\mathcal{P}^{-1}(y;x_0) - P^{-1}(y)}{|y-y_0|} = 0$. Local inverse functions can exist, even when a global inverse does not.

Non-injective functions can be inverted, for example, by choosing the closest x to x_0 such that $\mathcal{P}(x) = y$. As an example, consider $\mathcal{P}(x) = x^2$. It doesn't have a global inverse as two solutions ($\pm$) exist for each y. However, we can easily construct a local inverse by choosing the closest solution taking from an initial guess.

For differentiable functions, a local inverse is guaranteed to exist by the inverse function theorem as long as the Jacobian is non-singular. That is because the inverse Jacobian $\frac{\partial x}{\partial y}$ itself is a local inverse function, albeit, with being first-order, not the most accurate one. Even when the Jacobian is singular (because the function is not injective, chaotic or noisy), we can usually find good local inverse functions.

36.8.3 Time reversal

The inverse function of a simulator is typically the time-reversed physical process. In some cases, inverting the time axis of the forward simulator, $t \to -t$, can yield an adequate global inverse simulator. Unless the simulator destroys information in practice, e.g., due to accumulated numerical errors or stiff linear systems, this approach can be a starting point for an inverse simulation, or to formulate a *local* inverse simulation.

However, the simulator itself needs to be of sufficient accuracy to provide the correct estimate. For more complex settings, e.g., fluid simulations over the course of many time steps, the first- and second-order schemes as employed in *Navier-Stokes Forward Simulation* would not be sufficient.

36.8.4 Integrating a loss function

Since introducing IGs, we've only considered a simulator with an output y. Now we can re-introduce the loss function L. As before, we consider minimization problems with a scalar objective function $L(y)$ that depends on the result of an invertible simulator $y = \mathcal{P}(x)$. In equation (36.4) we've introduced the inverse gradient (IG) update, which gives $\Delta x = \frac{\partial x}{\partial L} \cdot \Delta L$ when the loss function is included. Here, ΔL denotes a step to take in terms of the loss.

By applying the chain rule and substituting the IG $\frac{\partial x}{\partial L}$ for the update from the inverse physics simulator from equation (36.5), we obtain, up to first order:

$$\begin{aligned}
\Delta x_{\text{PG}} &= \frac{\partial x}{\partial L} \cdot \Delta L \\
&= \frac{\partial x}{\partial y} \left(\frac{\partial y}{\partial L} \cdot \Delta L \right) \\
&= \frac{\partial x}{\partial y} \cdot \Delta y \\
&= \mathcal{P}^{-1}(y_0 + \Delta y; x_0) - x_0 + \mathcal{O}(\Delta y^2)
\end{aligned}$$

These equations show that equation (36.5) is equal to the IG from the section above up to first order, but contains nonlinear terms, i.e. $\Delta x_{\text{PG}} / \Delta y = \frac{\partial x}{\partial y} + \mathcal{O}(\Delta y^2)$. The accuracy of the update depends on the fidelity of the inverse function $\mathcal{P}^{-1}$. We can define an upper limit to the error of the local inverse using the local gradient $\frac{\partial x}{\partial y}$. In the worst case, we can therefore fall back to the regular gradient.

Also, we have turned the step w.r.t. L into a step in y space: Δy. However, this does not prescribe a unique way to compute Δy since the derivative $\frac{\partial y}{\partial L}$ as the right-inverse of the row-vector $\frac{\partial L}{\partial y}$ puts almost no restrictions on Δy. Instead, we use a Newton step from equation (36.3) to determine Δy where η controls the step size of the optimization steps. We will explain this in more detail in connection with the introduction of NNs after the following code example.

CHAPTER

THIRTYSEVEN

SIMPLE EXAMPLE COMPARING DIFFERENT OPTIMIZERS

The previous section has made many comments about the advantages and disadvantages of different optimization methods. Below we'll show with a practical example how much differences these properties actually make. [run in colab]

37.1 Problem formulation

We'll consider a very simple setup to clearly illustrate what's happening: we have a two-dimensional input space **x**, a mock "physical model" likewise with two dimensions **y**, and a scalar loss L, i.e. $\mathbf{x} \in \mathbb{R}^2$, $\mathbf{y} : \mathbb{R}^2 \to \mathbb{R}^2$, and $L : \mathbb{R}^2 \to \mathbb{R}$. The components of a vector like **x** are denoted with x_i, and to be in sync with python arrays the indices start at 0.

Specifically, we'll use the following **y** and L:

$$\mathbf{y}(\mathbf{x}) = \mathbf{y}(x_0, x_1) = \begin{bmatrix} x_0 \\ x_1^2 \end{bmatrix},$$

i.e. **y** only squares the second component of its input, and $L(\mathbf{y}) = |\mathbf{y}|^2 = y_0^2 + y_1^2$ represents a simple squared L^2 loss. As starting point for some example optimizations we'll use $\mathbf{x} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$ as initial guess for solving the following simple minimization problem: $\arg\min_{\mathbf{x}} L(\mathbf{x})$.

For us as humans it's quite obvious that $[0\ 0]^T$ is the right answer, but let's see how quickly the different optimization algorithms discussed in the previous section can find that solution. And while **y** is a very simple function, it is nonlinear due to its x_1^2.

37.2 3 Spaces

In order to understand the following examples, it's important to keep in mind that we're dealing with mappings between the three *spaces* we've introduced here: **x**, **y** and L. A regular forward pass maps an **x** via **y** to L, while for the optimization we'll need to associate values and changes in L with positions in **x**. While doing this, it will be interesting how this influences the positions in **y** that develop while searching for the right position in **x**.

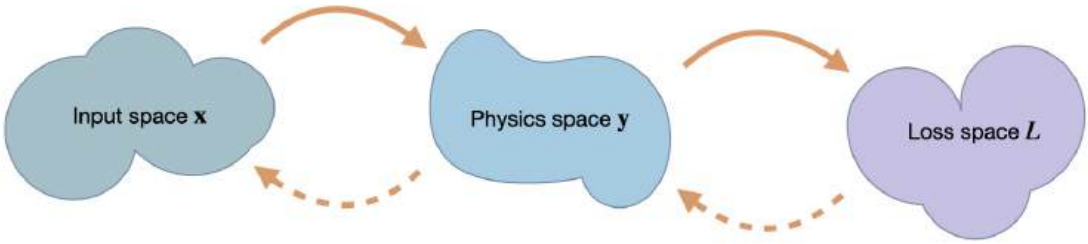


Fig. 37.1: We're targeting inverse problems to retrieve an entry in **x** from a loss computed in terms of output from a physics simulator **y**. Hence in a forward pass, we transform from **x** to **y**, and then compute a loss L. The backwards pass transforms back to **x**. Thus, the accuracy in terms of **x** is the most crucial one, but we can likewise track progress of an optimization in terms of **y** and L.

37.3 Implementation

For this example we'll use the JAX framework, which represents a nice alternative for efficiently working with differentiable functions. JAX also has a nice numpy wrapper that implements most of numpy's functions. Below we'll use this wrapper as `np`, and the *original* numpy as `onp`.

```
import jax
import jax.numpy as np
import numpy as onp
```

We'll start by defining the **y** and L functions, together with a single composite function `fun` which calls L and y. Having a single native python function is necessary for many of the JAX operations.

```
# "physics" function y
def physics_y(x):
    return np.array( [x[0], x[1]*x[1]] )

# simple L2 loss
def loss_y(y):
    #return y[0]*y[0] + y[1]*y[1] # "manual version"
    return np.sum( np.square(y) )

# composite function with L & y , evaluating the loss for x
def loss_x(x):
    return loss_y(physics_y(x))


x = np.asarray([3,3], dtype=np.float32)
print("Starting point x = "+format(x) +"\n")

print("Some test calls of the functions we defined so far, from top to bottom, y,␣
 ↪manual L(y), L(y):")
physics_y(x) , loss_y( physics_y(x) ), loss_x(x)
```

```
Starting point x = [3. 3.]

Some test calls of the functions we defined so far, from top to bottom, y, manual␣
 ↪L(y), L(y):
```

```
(DeviceArray([3., 9.], dtype=float32),
 DeviceArray(90., dtype=float32),
 DeviceArray(90., dtype=float32))
```

Now we can evaluate the derivatives of our function via `jax.grad`. E.g., `jax.grad(loss_y)(physics_y(x))` evaluates the Jacobian $\partial L / \partial \mathbf{y}$. The cell below evaluates this and a few variants, together with a sanity check for the inverse of the Jacobian of $\mathbf{y}$:

```
# this works:
print("Jacobian L(y): " + format(jax.grad(loss_y)(physics_y(x))) +"\n")

# the following would give an error as y (and hence physics_y) is not scalar
#jax.grad(physics_y)(x)

# computing the jacobian of y is a valid operation:
J = jax.jacobian(physics_y)(x)
print( "Jacobian y(x): \n" + format(J) )

# the code below also gives error, JAX grad needs a single function object
#jax.grad( loss_y(physics_y) )(x)

print( "\nSanity check with inverse Jacobian of y, this should give x again: " +␣
↪format(np.linalg.solve(J, np.matmul(J,x) )) +"\n")

# instead use composite 'fun' from above
print("Gradient for full L(x): " + format( jax.grad(loss_x)(x) )  +"\n")
```

```
Jacobian L(y): [ 6. 18.]

Jacobian y(x):
[[1. 0.]
 [0. 6.]]

Sanity check with inverse Jacobian of y, this should give x again: [3. 3.]

Gradient for full L(x): [  6. 108.]
```

The last line is worth a closer look: here we print the gradient $\partial L / \partial \mathbf{x}$ at our initial position. And while we know that we should just move diagonally towards the origin (with the zero vector being the minimizer), this gradient is not very diagonal - it has a strongly dominant component along x_1 with an entry of 108.

Let's see how the different methods cope with this situation. We'll compare

- the first order method *gradient descent* (i.e., regular, non-stochastic, "steepest gradient descent"),
- *Newton's method* as a representative of the second order methods,
- and scale-invariant updates from *inverse simulators*.

37.4 Gradient descent

For gradient descent, the simple gradient based update from equation (36.2) in our setting gives the following update step in $\mathbf{x}$:

$$\begin{aligned}\Delta \mathbf{x}_{\text{GD}} &= -\eta (J_L J_{\mathbf{y}})^T \\ &= -\eta (\frac{\partial L}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T\end{aligned}$$

where η denotes the step size parameter .

Let's start the optimization via gradient descent at $x = [3, 3]$, and update our solution ten times with $\eta = 0.01$:

```
x = np.asarray([3.,3.])
eta = 0.01
historyGD = [x]; updatesGD = []

for i in range(10):
    G = jax.grad(loss_x)(x)
    x += -eta * G
    historyGD.append(x); updatesGD.append(G)
    print( "GD iter %d: "%i + format(x) )
```

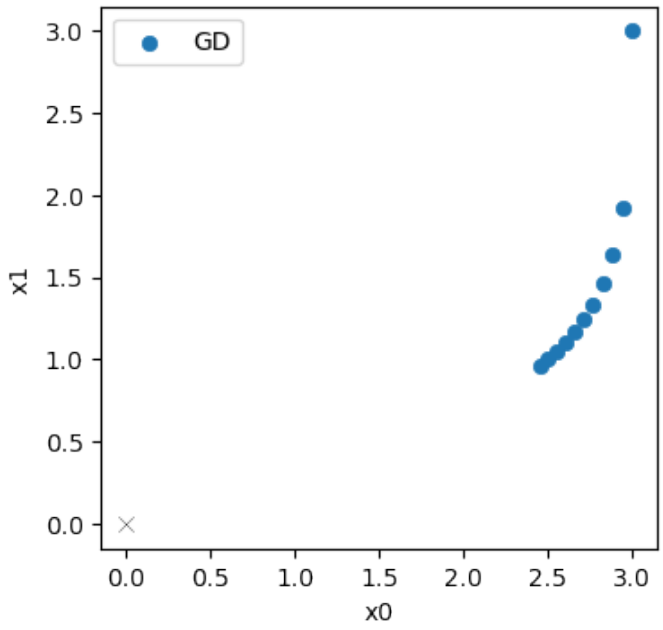
```
GD iter 0: [2.94      1.9200001]
GD iter 1: [2.8812    1.6368846]
GD iter 2: [2.823576  1.4614503]
GD iter 3: [2.7671044 1.3365935]
GD iter 4: [2.7117622 1.2410815]
GD iter 5: [2.657527  1.1646168]
GD iter 6: [2.6043763 1.1014326]
GD iter 7: [2.5522888 1.0479842]
GD iter 8: [2.501243  1.0019454]
GD iter 9: [2.4512184  0.96171147]
```

Here we've already printed the resulting positions in $\mathbf{x}$, and they seem to be going down, i.e. moving in the right direction. The last point, $[2.451\ 0.962]$ still has a fair distance of 2.63 to the origin.

Let's take a look at the progression over the course of the iterations (the evolution was stored in the `history` list above). The blue points denote the positions in $\mathbf{x}$ from the GD iterations, with the target at the origin shown with a thin black cross.

```
import matplotlib.pyplot as plt
axes = plt.figure(figsize=(4, 4), dpi=100).gca()
historyGD = onp.asarray(historyGD)
updatesGD = onp.asarray(updatesGD) # for later
axes.scatter(historyGD[:,0], historyGD[:,1], lw=0.5, color='#1F77B4', label='GD')
axes.scatter([0], [0], lw=0.25, color='black', marker='x') # target at 0,0
axes.set_xlabel('x0'); axes.set_ylabel('x1'); axes.legend()
```

```
<matplotlib.legend.Legend at 0x7fc742a69430>
```



No surprise here: the initial step mostly moves downwards along x_1 (in top right corner), and the updates afterwards curve towards the origin. But they don't get very far. It's still quite a distance to the solution in the bottom left corner.

37.5 Newton

For Newton's method, the update step is given by

$$\begin{aligned}\Delta \mathbf{x}_{\text{QN}} &= -\eta \left(\frac{\partial^2 L}{\partial \mathbf{x}^2} \right)^{-1} \frac{\partial L}{\partial \mathbf{x}} \\ &= -\eta \, H_L^{-1} \, (J_L J_{\mathbf{y}})^T\end{aligned}$$

Hence, in addition to the same gradient as for GD, we now need to evaluate and invert the Hessian of $\frac{\partial^2 L}{\partial \mathbf{x}^2}$.

This is quite straightforward in JAX: we can call `jax.jacobian` two times, and then use the JAX version of `linalg.inv` to invert the resulting matrix.

For the optimization with Newton's method we'll use a larger step size of $\eta = 1/3$. For this example and the following one, we've chosen the step size such that the magnitude of the first update step is roughly the same as the one of GD. In this way, we can compare the trajectories of all three methods relative to each other. Note that this is by no means meant to illustrate or compare the stability of the methods here. Stability and upper limits for η are separate topics. Here we're focusing on convergence properties.

In the next cell, we apply the Newton updates ten times starting from the same initial guess:

```
x = np.asarray([3.,3.])
eta = 1./3.
historyNt = [x]; updatesNt = []
```

(continues on next page)

(continued from previous page)

```
Gx = jax.grad(loss_x)
Hx = jax.jacobian(jax.jacobian(loss_x))
for i in range(10):
    g = Gx(x)
    h = Hx(x)
    hinv = np.linalg.inv(h)

    x += -eta * np.matmul( hinv , g )
    historyNt.append(x); updatesNt.append( np.matmul( hinv , g) )
    print( "Newton iter %d: "%i + format(x) )
```

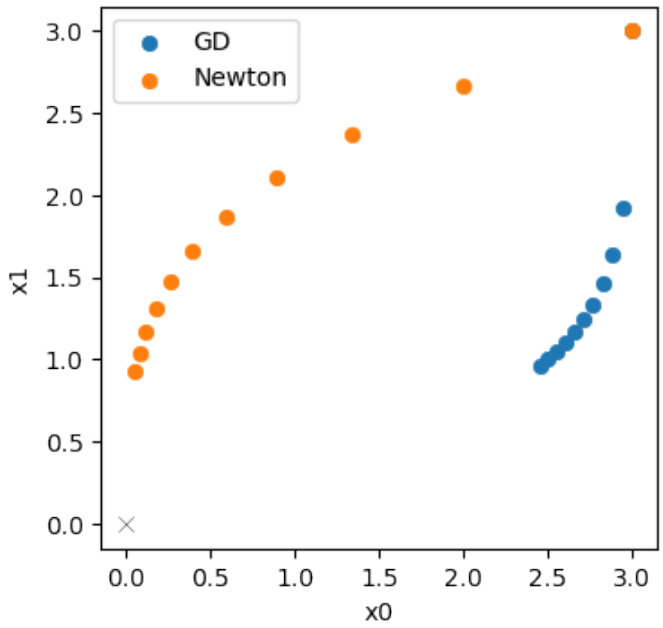
```
Newton iter 0: [2.        2.6666667]
Newton iter 1: [1.3333333 2.3703704]
Newton iter 2: [0.88888884 2.1069958 ]
Newton iter 3: [0.59259254 1.8728852 ]
Newton iter 4: [0.39506167 1.6647868 ]
Newton iter 5: [0.26337445 1.4798105 ]
Newton iter 6: [0.17558296 1.315387  ]
Newton iter 7: [0.1170553 1.1692328]
Newton iter 8: [0.07803687 1.0393181 ]
Newton iter 9: [0.05202458 0.92383826]
```

The last line already indicates: Newton's method does quite a bit better. The last point $[0.052\ 0.924]$ only has a distance of 0.925 to the origin (compared to 2.63 for GD)

Below, we plot the Newton trajectory in orange next to the GD version in blue.

```
axes = plt.figure(figsize=(4, 4), dpi=100).gca()
historyNt = onp.asarray(historyNt)
updatesNt = onp.asarray(updatesNt)
axes.scatter(historyGD[:,0], historyGD[:,1], lw=0.5, color='#1F77B4', label='GD')
axes.scatter(historyNt[:,0], historyNt[:,1], lw=0.5, color='#FF7F0E', label='Newton')
axes.scatter([0], [0], lw=0.25, color='black', marker='x') # target at 0,0
axes.set_xlabel('x0'); axes.set_ylabel('x1'); axes.legend()
```

```
<matplotlib.legend.Legend at 0x7fc7428c5bb0>
```



Not completely surprising: for this simple example we can reliably evaluate the Hessian, and Newtons's method profits from the second order information. It's trajectory is much more diagonal (that would be the ideal, shortest path to the solution), and does not slow down as much as GD.

37.6 Inverse simulators

Now we also use an analytical inverse of $\mathbf{y}$ for the optimization. It represents our inverse simulator $\mathcal{P}^{-1}$ from the previous sections: $\mathbf{y}^{-1}(\mathbf{x}) = [x_0 \; x_1^{1/2}]^T$, to compute the scale-invariant update denoted by PG below. As a slight look-ahead to the next section, we'll use a Newton's step for L, and combine it with the inverse physics function to get an overall update. This gives an update step:

$$\Delta \mathbf{x}_{\text{PG}} = \mathbf{y}^{-1} \left(\mathbf{y}(\mathbf{x}) - \eta \left(\frac{\partial^2 L}{\partial \mathbf{y}^2} \right)^{-1} \frac{\partial L}{\partial \mathbf{y}} \right) - \mathbf{x}$$

Below, we define our inverse function `physics_y_inv_analytic`, and then evaluate an optimization with the PG update for ten steps:

```
x = np.asarray([3.,3.])
eta = 0.3
historyPG = [x]; historyPGy = []; updatesPG = []

def physics_y_inv(y):
    return np.array( [y[0], np.power(y[1],0.5)] )

Gy = jax.grad(loss_y)
Hy = jax.jacobian(jax.jacobian(loss_y))
for i in range(10):

    # Newton step for L(y)
    zForw = physics_y(x)
```

(continues on next page)

(continued from previous page)

```
    g = Gy(zForw)
    h = Hy(zForw)
    hinv = np.linalg.inv(h)

    # step in y space
    zBack = zForw -eta * np.matmul( hinv , g)
    historyPGy.append(zBack)

    # "inverse physics" step via y-inverse
    x = physics_y_inv(zBack)
    historyPG.append(x)
    updatesPG.append( historyPG[-2] - historyPG[-1] )
    print( "PG iter %d: "%i + format(x) )
```

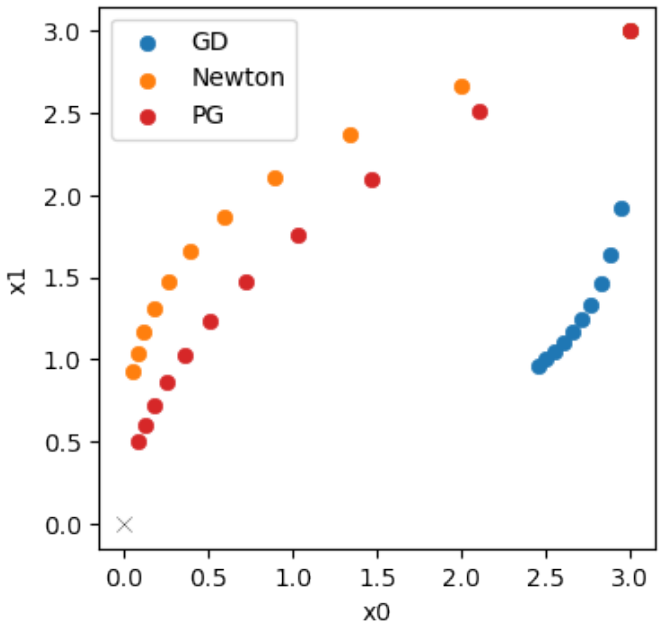
```
PG iter 0: [2.1       2.5099802]
PG iter 1: [1.4699999 2.1000001]
PG iter 2: [1.0289999 1.7569861]
PG iter 3: [0.72029996 1.47      ]
PG iter 4: [0.50421   1.2298902]
PG iter 5: [0.352947 1.029    ]
PG iter 6: [0.24706289 0.86092323]
PG iter 7: [0.17294402 0.7203    ]
PG iter 8: [0.12106082 0.60264623]
PG iter 9: [0.08474258 0.50421   ]
```

Now we obtain $[0.084\ 0.504]$ as the final position, with a distance of only 0.51! This is clearly better than both Newton and GD.

Let's directly visualize how the PGs (in red) fare in comparison to Newton's method (orange) and GD (blue).

```
historyPG = onp.asarray(historyPG)
updatesPG = onp.asarray(updatesPG)

axes = plt.figure(figsize=(4, 4), dpi=100).gca()
axes.scatter(historyGD[:,0], historyGD[:,1], lw=0.5, color='#1F77B4', label='GD')
axes.scatter(historyNt[:,0], historyNt[:,1], lw=0.5, color='#FF7F0E', label='Newton')
axes.scatter(historyPG[:,0], historyPG[:,1], lw=0.5, color='#D62728', label='PG')
axes.scatter([0], [0], lw=0.25, color='black', marker='x') # target at 0,0
axes.set_xlabel('x0'); axes.set_ylabel('x1'); axes.legend()
```

```
<matplotlib.legend.Legend at 0x7fc742ddba60>
```



This illustrates that the inverse simulator variant, PG in red, does even better than Newton's method in orange. It yields a trajectory that is better aligned with the ideal *diagonal* trajectory, and its final state is closer to the origin. A key ingredient here is the inverse function for $\mathbf{y}$, which provided higher order terms than the second-order approximation for Newton's method. This improves the scale-invariance of the optimization. Despite the simplicity of the problem, Newton's method has problems finding the right search direction. For the inverse simulator update, on the other hand, the higher order information yields an improved direction for the optimization.

This difference also shows in first update step for each method: below we measure how well it is aligned with the diagonal.

```
def mag(x):
    return np.sqrt(np.sum(np.square(x)))

def one_len(x):
    return np.dot( x/mag(x), np.array([1,1]))

print("Diagonal lengths (larger is better): GD %f, Nt %f, PG %f " %
      (one_len(updatesGD[0]) , one_len(updatesNt[0]) , one_len(updatesPG[0])) )
```

```
Diagonal lengths (larger is better): GD 1.053930, Nt 1.264911, PG 1.356443
```

The largest value of 1.356 for PG confirms what we've seen above: the PG gradient was the closest one to the diagonal direction from our starting point to the origin.

37.7 y Space

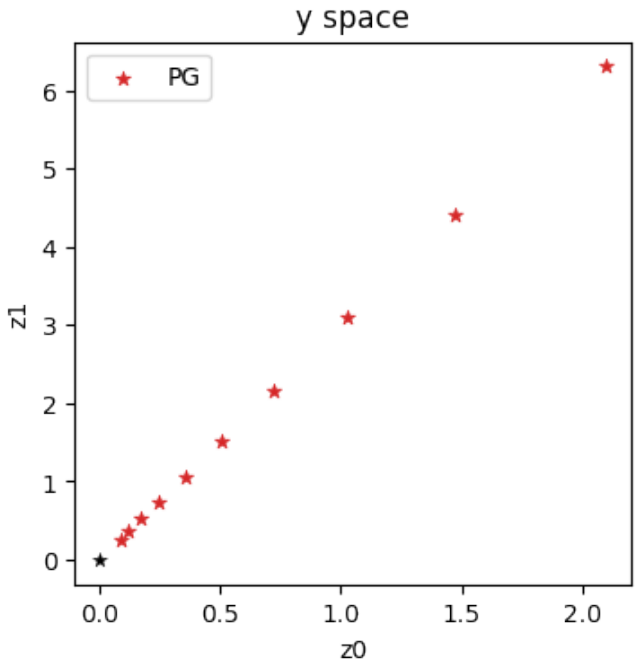
To understand the behavior and differences of the methods here, it's important to keep in mind that we're not dealing with a black box that maps between **x** and L, but rather there are spaces in between that matter. In our case, we only have a single **y** space, but for DL settings, we might have a large number of latent spaces, over which we have a certain amount of control. We will return to NNs soon, but for now let's focus on **y**.

A first thing to note is that for PG, we explicitly map from L to **y**, and then continue with a mapping to **x**. Thus we already obtained the trajectory in **y** space, and not coincidentally, we already stored it in the `historyPGy` list above.

Let's directly take a look what the inverse simulator did in **y** space:

```
historyPGy = onp.asarray(historyPGy)

axes = plt.figure(figsize=(4, 4), dpi=100).gca()
axes.set_title('y space')
axes.scatter(historyPGy[:,0], historyPGy[:,1], lw=0.5, color='#D62728', marker='*',␣
↪label='PG')
axes.scatter([0], [0], lw=0.25, color='black', marker='*')
axes.set_xlabel('z0'); axes.set_ylabel('z1'); axes.legend()
```

```
<matplotlib.legend.Legend at 0x7fc742e83a90>
```



With this variant, we're making explicit steps in **y** space, which progress in a straight diagonal line to the origin (which is likewise the solution in **y** space).

Interestingly, neither GD nor Newton's method give us information about progress in intermediate spaces (like the **y** space).

For GD we're concatenating the Jacobians, so we're moving in directions that locally should decrease the loss. However, the **y** position is influenced by **x**, and hence we don't know where we end up in **y** space until we have the definite point

there. (For NNs in general we won't know at which latent-space points we end up after a GD update until we've actually computed all updated weights.)

More specifically, we have an update $-\eta \frac{\partial L}{\partial \mathbf{x}}$ for GD, which means we arrive at $\mathbf{y}(\mathbf{x}-\eta \frac{\partial L}{\partial \mathbf{x}})$ in $\mathbf{y}$ space. A Taylor expansion with $h = \eta \frac{\partial L}{\partial \mathbf{x}}$ yields

$$\mathbf{y}(\mathbf{x}-h) = \mathbf{y}(\mathbf{x}) - h \frac{\partial \mathbf{y}}{\partial \mathbf{x}} + \mathcal{O}(h^2) = \mathbf{y}(x) - \eta \frac{\partial L}{\partial \mathbf{y}} (\frac{\partial \mathbf{y}}{\partial x})^2 + \mathcal{O}(h^2).$$

And $\frac{\partial L}{\partial \mathbf{y}} (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^2$ clearly differs from $\frac{\partial L}{\partial \mathbf{y}}$, which we would apply with GD when optimizing for $\mathbf{y}$ directly.

Newton's method does not fare much better: we compute first-order derivatives like for GD, and the second-order derivatives for the Hessian for the full process. But since both are approximations, the actual intermediate states resulting from an update step are unknown until the full chain is evaluated. In the *Consistency in function compositions* paragraph for Newton's method in *Scale-Invariance and Inversion* the squared $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ term for the Hessian already indicated this dependency.

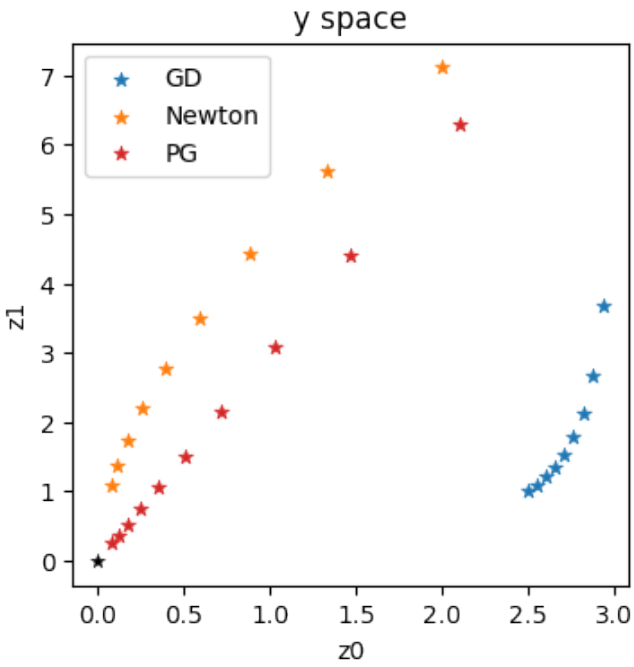
With **inverse simulators** we do not have this problem: they can directly map points in $\mathbf{y}$ to $\mathbf{x}$. Hence we know eactly where we started in $\mathbf{y}$ space, as this position is crucial for evaluating the inverse.

In the simple setting of this section, we only have a single latent space, and we already stored all values in $\mathbf{x}$ space during the optimization (in the `history` lists). Hence, now we can go back and re-evaluate `physics_y` to obtain the positions in $\mathbf{y}$ space.

```
x = np.asarray([3.,3.])
eta = 0.01
historyGDy = []
historyNty = []

for i in range(1,10):
    historyGDy.append(physics_y(historyGD[i]))
    historyNty.append(physics_y(historyNt[i]))

historyGDy = onp.asarray(historyGDy)
historyNty = onp.asarray(historyNty)
```

```
axes = plt.figure(figsize=(4, 4), dpi=100).gca()
axes.set_title('y space')
axes.scatter(historyGDy[:,0], historyGDy[:,1], lw=0.5, marker='*', color='#1F77B4',␣
 ↪label='GD')
axes.scatter(historyNty[:,0], historyNty[:,1], lw=0.5, marker='*', color='#FF7F0E',␣
 ↪label='Newton')
axes.scatter(historyPGy[:,0], historyPGy[:,1], lw=0.5, marker='*', color='#D62728',␣
 ↪label='PG')
axes.scatter([0], [0], lw=0.25, color='black', marker='*')
axes.set_xlabel('z0'); axes.set_ylabel('z1'); axes.legend()
```

```
<matplotlib.legend.Legend at 0x7fc7430c4b20>
```



These trajectories confirm the intuition outlined in the previous sections: GD in blue gives a very sub-optimal trajectory in **y**. Newton (in orange) does better, but is still clearly curved. It can't approximate the higher order terms of this example well enough. This is in contrast to the straight, and diagonal red trajectory for the optimization using the inverse simulator.

The behavior in intermediate spaces becomes especially important when they're not only abstract latent spaces as in this example, but when they have actual physical meanings.

37.8 Conclusions

Despite its simplicity, this example already shows surprisingly large differences between gradient descent, Newton's method, and using the *inverse simulator*.

The main takeaways of this section are the following.

- GD easily yields "unbalanced" updates, and gets stuck.
- Newtons method does better, but is far from optimal.
- the higher-order information of the inverse simulator outperform both, even if it is applied only partially (we still used Newton's method for L above).
- Also, the methods (and in general the choice of optimizer) strongly affects progress in latent spaces, as shown for **y** above.

In the next sections we can build on these observations to use PGs for training NNs via invertible physical models.

37.9 Approximate inversions

If an analytic inverse like the `physics_y_inv_analytic` above is not readily available, we can actually resort to optimization schemes like Newton's method or BFGS to obtain a local inverse numerically. This is a topic that is orthogonal to the comparison of different optimization methods, but it can be easily illustrated based on the inverse simulator variant from above.

Below, we'll use the BFGS variant `fmin_l_bfgs_b` from `scipy` to compute the inverse. It's not very complicated, but we'll use numpy and scipy directly here, which makes the code a bit messier than it should be.

```
def physics_y_inv_opt(target_y, x_ini):
    # a bit ugly, we switch to pure scipy here inside each iteration for BFGS
    import numpy as np
    from scipy.optimize import fmin_l_bfgs_b
    target_y = onp.array(target_y)
    x_ini    = onp.array(x_ini)

    def physics_y_opt(x,target_y=[2,2]):
        y = onp.array( [x[0], x[1]*x[1]] ) # we cant use physics_y from JAX here
        ret = onp.sum( onp.square(y-target_y) )
        return ret

    ret = fmin_l_bfgs_b(lambda x: physics_y_opt(x,target_y), x_ini, approx_grad=True )
    #print( ret ) # return full BFGS details
    return ret[0]

print("BFGS optimization test run, find x such that y=[2,2]:")
physics_y_inv_opt([2,2], [3,3])
```

```
BFGS optimization test run, find x such that y=[2,2]:
```

```
array([2.00000003, 1.41421353])
```

Nonetheless, we can now use this numerically inverted **y** function to perform the inverse simulator optimization. Apart from calling `physics_y_inv_opt`, the rest of the code is unchanged.

```
x = np.asarray([3.,3.])
eta = 0.3
history = [x]; updates = []

Gy = jax.grad(loss_y)
Hy = jax.jacobian(jax.jacobian(loss_y))
for i in range(10):
    # same as before, Newton step for L(y)
    y = physics_y(x)
    g = Gy(y)
    y += -eta * np.matmul( np.linalg.inv( Hy(y) ) , g)

    # optimize for inverse physics, assuming we dont have access to an inverse for␣
 ↪physics_y
    x = physics_y_inv_opt(y,x)
    history.append(x)
    updates.append( history[-2] - history[-1] )
    print( "PG iter %d: "%i + format(x) )
```

```
PG iter 0: [2.09999967 2.50998022]
PG iter 1: [1.46999859 2.10000011]
PG iter 2: [1.02899871 1.75698602]
PG iter 3: [0.72029824 1.4699998 ]
PG iter 4: [0.50420733 1.22988982]
PG iter 5: [0.35294448 1.02899957]
PG iter 6: [0.24705997 0.86092355]
PG iter 7: [0.17294205 0.72030026]
PG iter 8: [0.12106103 0.60264817]
PG iter 9: [0.08474171 0.50421247]
```

This confirms that the approximate inversion works, in line with the regular PG version above. There's not much point plotting this, as it's basically the same, but let's measure the difference. Below, we compute the MAE, which for this simple example turns out to be on the order of our floating point accuracy.

```
historyPGa = onp.asarray(history)
updatesPGa = onp.asarray(updates)

print("MAE difference between analytic PG and approximate inversion: %f" % (np.
 ↪average(np.abs(historyPGa-historyPG))) )
```

```
MAE difference between analytic PG and approximate inversion: 0.000001
```

37.10 Next steps

Based on this code example you can try the following modifications:

- Instead of the simple L(y(x)) function above, try other, more complicated functions.
- Replace the simple "regular" gradient descent with another optimizer, e.g., commonly used DL optimizers such as AdaGrad, RmsProp or Adam. Compare the existing versions above with the new trajectories.

CHAPTER

THIRTYEIGHT

SCALE INVARIANT PHYSICS TRAINING

The discussion in the previous two sections already hints at inversion of gradients being a important step for optimization and learning. We will now integrate the update step Δx_{PG} into NN training, and give details of the two way process of inverse simulator and Newton step for the loss that was already used in the previous code from the *Simple Example*.

As outlined in the IG section of *Scale-Invariance and Inversion*, we're focusing on NN solutions of *inverse problems* below. That means we have $y = \mathcal{P}(x)$, and our goal is to train an NN representation f such that $f(y;\theta) = x$. This is a slightly more constrained setting than what we've considered for the differentiable physics (DP) training. Also, as we're targeting optimization algorithms now, we won't explicitly denote DP approaches: all of the following variants involve physics simulators, and the gradient descent (GD) versions as well as its variants (such as Adam) use DP training.

Note

Important to keep in mind: In contrast to the previous sections and *Models and Equations*, we are targeting inverse problems, and hence y is the input to the network: $f(y;\theta)$. Correspondingly, it outputs x.

This gives the following minimization problem with i denoting the indices of a mini-batch:

$$\text{arg min}_\theta \sum_i \frac{1}{2} |\mathcal{P}(f(y_i^*;\theta)) - y_i^*|_2^2 \tag{38.1}$$

38.1 NN training

To integrate the update step from equation (36.5) into the training process for an NN, we consider three components: the NN itself, the physics simulator, and the loss function:

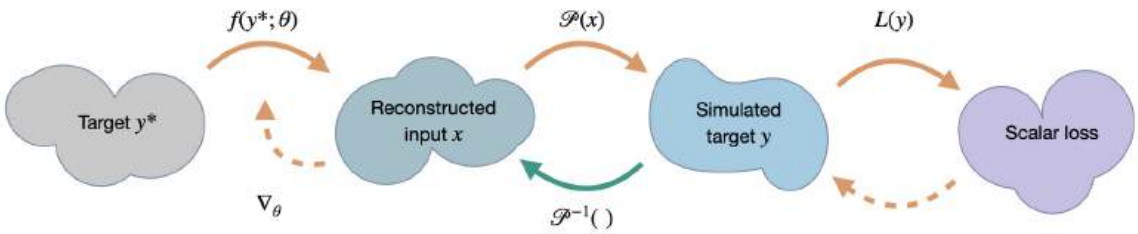


Fig. 38.1: A visual overview of the different spaces involved in SIP training.

To join these three pieces together, we use the following algorithm. As introduced by Holl et al. [HKT22], we'll denote this training process as *scale-invariant physics* (SIP) training.

Scale-Invariant Physics (SIP) Training

To update the weights θ of the NN f, we perform the following update step:

- Given a set of inputs y^*, evaluate the forward pass to compute the NN prediction $x = f(y^*; \theta)$
- Compute y via a forward simulation ($y = \mathcal{P}(x)$) and invoke the (local) inverse simulator $P^{-1}(y; x)$ to obtain the step $\Delta x_{\text{PG}} = \mathcal{P}^{-1}(y + \eta \Delta y; x)$ with $\Delta y = y^* - y$
- Evaluate the network loss, e.g., $L = \frac{1}{2}|x - \tilde{x}|_2^2$ with $\tilde{x} = x + \Delta x_{\text{PG}}$, and perform a Newton step treating $\tilde{x}$ as a constant
- Use GD (or a GD-based optimizer like Adam) to propagate the change in x to the network weights θ with a learning rate η_{NN}

This combined optimization algorithm depends on both the learning rate η_{NN} for the network as well as the step size η from above, which factors into Δy. To first order, the effective learning rate of the network weights is $\eta_{\text{eff}} = \eta \cdot \eta_{\text{NN}}$. We recommend setting η as large as the accuracy of the inverse simulator allows. In many cases $\eta = 1$ is possible, otherwise η_{NN} should be adjusted accordingly. This allows for nonlinearities of the simulator to be maximally helpful in adjusting the optimization direction.

This algorithm combines the inverse simulator to compute accurate, higher-order updates with traditional training schemes for NN representations. This is an attractive property, as we have a large collection of powerful methodologies for training NNs that stay relevant in this way. The treatment of the loss functions as "glue" between NN and physics component plays a central role here.

38.2 Loss functions

In the above algorithm, we have assumed an L^2 loss, and without further explanation introduced a Newton step to propagate the inverse simulator step to the NN. Below, we explain and justify this treatment in more detail.

The central reason for introducing a Newton step is the improved accuracy for the loss derivative. Unlike with regular Newton or the quasi-Newton methods from equation (36.3), we do not need the Hessian of the full system. Instead, the Hessian is only needed for $L(y)$. This makes Newton's method attractive again. Even better, for many typical L the analytical form of the Newton updates is known.

E.g., consider the most common supervised objective function, $L(y) = \frac{1}{2}|y - y^*|_2^2$ as already put to use above. y denotes the predicted, and y^* the target value. We then have $\frac{\partial L}{\partial y} = y - y^*$ and $\frac{\partial^2 L}{\partial y^2} = 1$. Using equation (36.3), we get $\Delta y = \eta \cdot (y^* - y)$ which can be computed right away, without evaluating any additional Hessian matrices.

Once Δy is determined, the gradient can be backpropagated to x, e.g. an earlier time, using the inverse simulator $\mathcal{P}^{-1}$. We've already used this combination of a Newton step for the loss and an inverse simulator for the PDE in *Simple Example comparing Different Optimizers*.

The loss in x here acts as a *proxy* to embed the update from the inverse simulator into the network training pipeline. It is not to be confused with a traditional supervised loss in x space. Due to the dependency of $\mathcal{P}^{-1}$ on the prediction y, it does not average multiple modes of solutions in x. To demonstrate this, consider the case that GD is being used as solver for the inverse simulation. Then the total loss is purely defined in y space, reducing to a regular first-order optimization.

Hence, to summarize with SIPs we employ a trivial Newton step for the loss in y, and a proxy L^2 loss in x that connects the computational graphs of inverse physics and NN for backpropagation. The following figure visualizes the different steps.

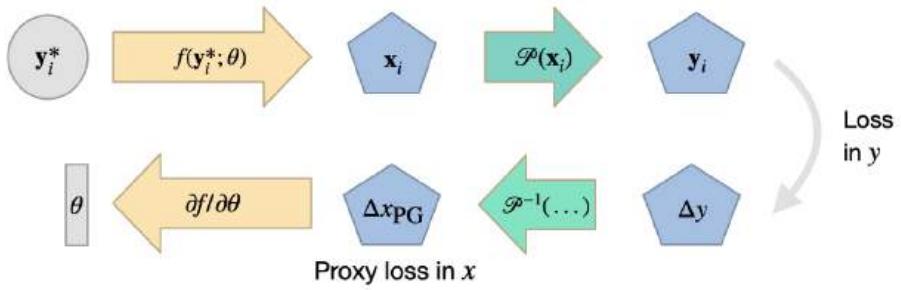


Fig. 38.2: A visual overview of SIP training for an entry i of a mini-batch, including the two loss computations in y and in x space (for the proxy loss).

38.3 Iterations and time dependence

The above procedure describes the optimization of neural networks that make a single prediction. This is suitable for scenarios to reconstruct the state of a system at t_0 given the state at a $t_e > t_0$ or to estimate an optimal initial state to match certain conditions at t_e.

However, the SIP method can also be applied to more complex setups involving multiple objectives and multiple network interactions at different times. Such scenarios arise e.g. in control tasks, where a network induces small forces at every time step to reach a certain physical state at t_e. It also occurs in correction tasks where a network tries to improve the simulation quality by performing corrections at every time step.

In these scenarios, the process above (Newton step for loss, inverse simulator step for physics, GD for the NN) is iteratively repeated, e.g., over the course of different time steps, leading to a series of additive terms in L. This typically makes the learning task more difficult, as we repeatedly backpropagate through the iterations of the physical solver and the NN, but the SIP algorithm above extends to these case just like a regular GD training.

38.4 SIP training in action

Let's illustrate the convergence behavior of SIP training and how it depends on characteristics of $\mathcal{P}$ with an example [HKT22]. We consider the synthetic two-dimensional function

$$\mathcal{P}(x) = (\sin(\hat{x}_1)/\xi,\ \hat{x}_2 \cdot \xi) \quad \text{with} \quad \hat{x} = R_\phi \cdot x,$$

where $R_\phi \in \mathrm{SO}(2)$ denotes a rotation matrix. The parameters ξ and ϕ allow us to continuously change the characteristics of the system. The value of ξ determines the conditioning of $\mathcal{P}$ with large ξ representing ill-conditioned problems while ϕ describes the coupling of x_1 and x_2. When $\phi = 0$, the off-diagonal elements of the Hessian vanish and the problem factors into two independent problems.

Here's an example of the resulting loss landscape for $y^* = (0.3, -0.5)$, $\xi = 1$, $\phi = 15°$ that shows the entangling of the sine function for x_1 and linear change for x_2:

Next we train a fully-connected neural network to invert this problem via equation (38.1). We'll compare SIP training using a saddle-free Newton solver to various state-of-the-art network optimizers. For fairness, the best learning rate is selected independently for each optimizer. When choosing $\xi = 1$ the problem is perfectly conditioned. In this case all network optimizers converge, with Adam having a slight advantage. This is shown in the left graph:

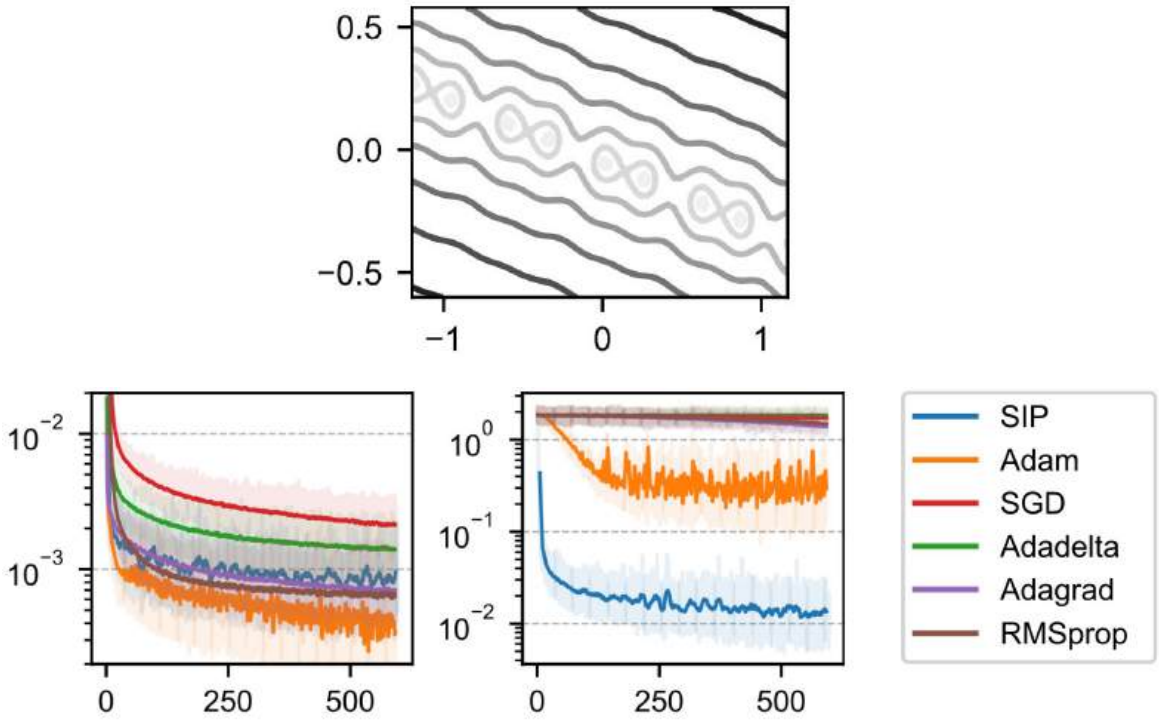


Fig. 38.3: Loss over time in seconds for a well-conditioned (left), and ill-conditioned case (right).

At $\xi = 32$, we have a fairly badly conditioned case, and only SIP and Adam succeed in optimizing the network to a significant degree, as shown on the right.

Note that the two graphs above show convergence over time. The relatively slow convergence of SIP mostly stems from it taking significantly more time per iteration than the other methods, on average 3 times as long as Adam. While the evaluation of the Hessian inherently requires more computations, the per-iteration time of SIP could likely be significantly reduced by optimizing the computations.

By increasing ξ while keeping $\phi = 0$ fixed we can show how the conditioning continually influences the different methods, as shown on the left here:

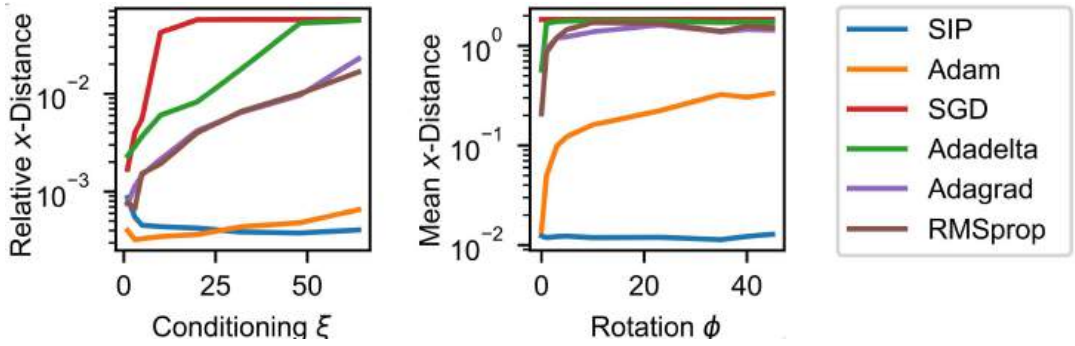


Fig. 38.4: Performance when varying the conditiong (left) and the entangling of dimensions via the rotation (right).

The accuracy of all traditional network optimizers decreases because the gradients scale with $(1/\xi, \xi)$ in x, becoming

longer in x_2, the direction that requires more precise values. SIP training avoids this using the Hessian, inverting the scaling behavior and producing updates that align with the flat direction in x. This allows SIP training to retain its relative accuracy over a wide range of ξ. Even for Adam, the accuracy becomes worse for larger ξ.

By varying only ϕ we can demonstrate how the entangling of the different components influences the behavior of the optimizers. The right graph of Fig. 38.4 varies ϕ with $\xi = 32$ fixed. This sheds light on how Adam manages to learn in ill-conditioned settings. Its diagonal approximation of the Hessian reduces the scaling effect when x_1 and x_2 lie on different scales, but when the parameters are coupled, the lack of off-diagonal terms prevents this. It's performance deteriorates by more than an order of magnitude in this case. SIP training has no problem with coupled parameters since its update steps for the optimization are using the full-rank Hessian $\frac{\partial^2 L}{\partial x}$. Thus, the SIP training yields the best results across the varying optimization problems posed by this example setup.

38.5 Discussion of SIP Training

Although we've only looked at smaller toy problems so far, we'll pull the discussion of SIP training forward. The next chapter will illustrate this with a more complex example, but as we'll directly switch to a new algorithm afterwards, below is a better place for a discussion of the properties of SIP.

Overall, the scale-invariance of SIP training allows it to find solutions exponentially faster than other learning methods for many physics problems, while keeping the computational cost relatively low. It provably converges when enough network updates $\Delta\theta$ are performed per solver evaluation and it can be shown that it converges with a single $\Delta\theta$ update for a wide range of physics experiments.

38.5.1 Limitations

While SIP training can find vastly more accurate solutions, there are some caveats to consider.

First, an approximately scale-invariant physics solver is required. While in low-dimensional x spaces, Newton's method is a good candidate, high-dimensional spaces require some other form of inversion. Some equations can locally be inverted analytically but for complex problems, domain-specific knowledge may be required, or we can employ to numerical methods (coming up).

Second, SIP focuses on an accurate inversion of the physics part, but uses traditional first-order optimizers to determine $\Delta\theta$. As discussed, these solvers behave poorly in ill-conditioned settings which can also affect SIP performance when the network outputs lie on very different scales. Thus, we should keep inversion for the NN in mind as a goal.

Third, while SIP training generally leads to more accurate solutions, measured in x space, the same is not always true for the loss $L = \sum_i L_i$. SIP training weighs all examples equally, independent of their loss values. This can be useful, but it can cause problems in examples where regions with overly small or large curvatures $|\frac{\partial^2 L}{\partial x^2}|$ distort the importance of samples. In these cases, or when the accuracy in x space is not important, like in control tasks, traditional training methods may perform better than SIP training.

38.5.2 Similarities to supervised training

Interestingly, the SIP training resembles the supervised approaches from *Supervised Training*. It effectively yields a method that provides reliable updates which are computed on-the-fly, at training time. The inverse simulator provides the desired inversion, possibly with a high-order method, and avoids the averaging of multi modal solutions (cf. *A Teaser Example*).

The latter is one of the main advantages of this setup: a pre-computed data set can not take multi-modality into account, and hence inevitably leads to suboptimal solutions being learned once the mapping from input to reference solutions is not unique.

At the same time this illustrates a difficulty of the DP training from *Introduction to Differentiable Physics*: the gradients it yields are not properly inverted, and are difficult to reliably normalize via pre-processing. Hence they can lead to the scaling problems discussing in *Scale-Invariance and Inversion*, and correspondingly give vanishing and exploding gradients at training time. These problems are what we're targeting in this chapter.

In the next section we'll show a more complex example of training physics-based NNs with SIP updates from inverse simulators, before explaining a second alternative for tackling the scaling problems.

CHAPTER

THIRTYNINE

LEARNING TO INVERT HEAT CONDUCTION WITH SCALE-INVARIANT UPDATES

We now turn to a practical example that use the scale-invariant physics (SIP) updates in a more complex example. Specifically, we'll consider the heat equation, which poses some particularly interesting challenges: the differentiable physics (DP) gradient just diffuses more, while the inversion is numerically challenging. Below, we'll explain how SIPs adress the former, while a special solver can address the latter issue.

The notebook below provides a full implementation via *phiflow* to generate data, run the DP version, and compute the improved SIP updates. [run in colab]

39.1 Problem Statement

We consider a two-dimensional system governed by the heat equation $\frac{\partial u}{\partial t} = \nu \cdot \nabla^2 u$. Given an initial state $x = u(t_0)$ at t_0, the simulator computes the state at a later time t_* via $y = u(t_*) = \mathcal{P}(x)$. Exactly inverting this system is only possible for $t \cdot \nu = 0$ and becomes increasingly unstable for larger $t \cdot \nu$ because initially distinct heat levels even out over time, drowning the original information in noise. Hence the Jacobian of the physics $\frac{\partial y}{\partial x}$ is near-singular.

We'll use periodic boundary conditions and compute the result in frequency space where the physics can be computed analytically as $\hat{y} = \hat{x} \cdot e^{-k^2(t_*-t_0)}$, where $\hat{y}_k \equiv \mathcal{F}(y)_k$ denotes the k-th element of the Fourier-transformed vector y. In the regular *forward* simulation, high frequencies are dampened exponentially. We'll need to revisit this aspect for the inverse simulator.

To summarize, the inverse problem we're targeting here can be written as minimizing:

$$L(x) = ||\mathcal{P}(x) - y^*||_2^2 = ||\mathcal{F}^{-1}\left(\mathcal{F}(x) \cdot e^{-k^2(t_*-t_0)}\right) - y^*||_2^2.$$

39.2 Implementation

Below, we'll set $t \cdot \nu = 8$ on a domain consisting of 64x64 cells of unit length. This level of diffusion is challenging, and diffuses most details while leaving only the large-scale structure intact.

We'll use phiflow with PyTorch as default backend, but this example code likewise runs with TensorFlow (just switch to `phi.tf.flow` below).

```
!pip install --upgrade --quiet phiflow==3.1
from phi.torch.flow import *    # switch to TF with "phi.tf.flow"
```

```
?25l     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/182.2 kB ? eta -:--:--
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 182.2/182.2 kB 9.2 MB/s eta 0:00:00
?25h  Preparing metadata (setup.py) ... ?25l?25hdone
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 306.1/306.1 kB 18.7 MB/s eta 0:00:00
?25h  Preparing metadata (setup.py) ... ?25l?25hdone
  Building wheel for phiflow (setup.py) ... ?25l?25hdone
  Building wheel for phiml (setup.py) ... ?25l?25hdone
```

39.3 Data generation

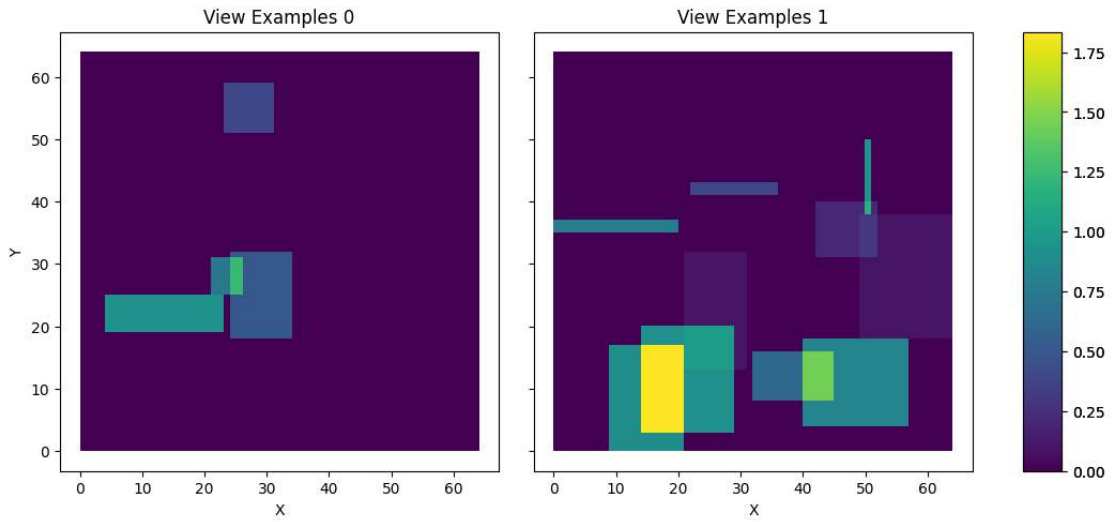
For training, we generate x^* by randomly placing between 4 and 10 "hot" rectangles of random size and shape in the domain. The `generate_heat_example()` function below generates a full mini batch (via `shape.batch`) of example positions. These will be x later on. They're never passed to the solver, but should be reconstructed by the neural network.

```
def generate_heat_example(*shape, bounds: Box = None):
    shape = math.merge_shapes(*shape)
    heat_t0 = CenteredGrid(0, extrapolation.PERIODIC, bounds, resolution=shape.
↪spatial)
    bounds = heat_t0.bounds
    component_counts = math.to_int32(4 + 7 * math.random_uniform(shape.batch))
    positions = (math.random_uniform(shape.batch, batch(components=10),␣
↪channel(vector=shape.spatial.names)) - 0.5) * bounds.size * 0.8 + bounds.size * 0.5
    for i in range(10):
        position = positions.components[i]
        half_size = math.random_uniform(shape.batch, channel(vector=shape.spatial.
↪names)) * 10
        strength = math.random_uniform(shape.batch) * math.to_float(i < component_
↪counts)
        position = math.clip(position, bounds.lower + half_size, bounds.upper - half_
↪size)
        component_box = Cuboid(position, half_size)
        component_mask = SoftGeometryMask(component_box)
        component_mask = component_mask.at(heat_t0)
        heat_t0 += component_mask * strength
    return heat_t0
```

The data is generated on-the-fly later during training, but let's look at two example x for now using phiflow's `vis.plot` function:

```
vis.plot(generate_heat_example(batch(view_examples=2), spatial(x=64, y=64)));
```

```
<ipython-input-2-350e9b9c59ed>:13: DeprecationWarning: HardGeometryMask and␣
↪SoftGeometryMask are deprecated. Use field.mask or field.resample instead.
  component_mask = SoftGeometryMask(component_box)
```



39.4 Differentiable physics and gradient descent

Nothing in this setup so far prevents us from using regular DP training, as described in *Introduction to Differentiable Physics*. For this diffusion case, we can write out gradient descent update from in an analytic fashion as: $\Delta x_{\text{GD}} = -\eta \cdot \mathcal{F}^{-1}\left(e^{-k^2(t_*-t_0)}\mathcal{F}(y-y^*)\right)$.

Looking at this expression, it means that gradient descent (GD) with the gradient from the differentiable simulator applies the forward physics to the gradient vector itself. This is surprising: the forward simulation performs diffusion, and now the backward pass performs even more of it, instead of somehow undoing the diffusion? Unfortunately, this is the inherent and "correct" behavior of DP, and it results in updates that are stable but lack high frequency spatial information.

Consequently, GD-based optimization methods converge slowly on this task after fitting the coarse structure and have severe problems in recovering high-frequency details, as will be demonstrated below. This is not because the information is fundamentally missing but because GD cannot adequately process high-frequency details.

For the implementation below, we will simply use `y = diffuse.fourier(x, 8., 1)` for the forward pass, and then similarly compute an L^2 loss for two y fields to which `diffuse.fourier( , 8., 1)` is applied.

39.5 Stable SIP gradients

What is more interesting in the context of this chapter is the improved update step computed via the inverse simulator, the *SIP* update. In line with the previous sections, we'll call this update Δx_{PG}.

The frequency formulation of the heat equation can be inverted analytically, yielding $\hat{x}_k = \hat{y}_k \cdot e^{k^2(t_*-t_0)}$. This allows us to define the update

$$\Delta x_{\text{PG}} = -\eta \cdot \mathcal{F}^{-1}\left(e^{k^2(t_*-t_0)}\mathcal{F}(y-y^*)\right).$$

Here, high frequencies are multiplied by exponentially large factors, resulting in numerical instabilities. When applying this formula directly to the gradients, it can lead to large oscillations in Δx_{PG}.

Note that these numerical instabilities also occur when computing the gradients in real space instead of frequency space. However, frequency space allows us to more easily quantify them.

Now we can leverage our knowledge of the physical simulation process to construct a stable inverse: the numerical instabilities can be avoided by taking a probabilistic viewpoint. The observed values y contain a certain amount of noise n, with the remainder constituting the signal $s = y - n$. For the noise, we assume a normal distribution $n \sim \mathcal{N}(0, \epsilon \cdot y)$ with $\epsilon > 0$ and for the signal, we assume that it arises from reasonable values of x so that $y \sim \mathcal{N}(0, \delta \cdot e^{-k^2})$ with $\delta > 0$. With this, we can estimate the probability of an observed value arising from the signal using Bayes' theorem $p(s|v) = \frac{p(v|s) \cdot p(s)}{p(v|s) \cdot p(s) + p(v|n) \cdot p(n)}$ where we assume the priors $p(s) = p(n) = \frac{1}{2}$. Based on this probability, we dampen the amplification of the inverse physics which yields a stable inverse.

Gradients computed in this way hold as much high-frequency information as can be extracted given the noise that is present. This leads to a much faster convergence and more precise solution than any generic optimization method. The cell below implements this probabilistic approach, with `probability_signal` in `apply_damping()` containing the parts of the signal to be dampened. The `inv_diffuse()` functions employs it to compute a stabilized inverse diffusion process.

```
def apply_damping(kernel, inv_kernel, amp, f_uncertainty, log_kernel):
    signal_prior = 0.5
    expected_amp = 1. * kernel.shape.get_size('x') * inv_kernel  # This can be␣
↪measured
    signal_likelihood = math.exp(-0.5 * (abs(amp) / expected_amp) ** 2) * signal_
↪prior  # this can be NaN
    signal_likelihood = math.where(math.isfinite(signal_likelihood), signal_
↪likelihood, math.zeros_like(signal_likelihood))
    noise_likelihood = math.exp(-0.5 * (abs(amp) / f_uncertainty) ** 2) * (1 - signal_
↪prior)
    probability_signal = math.divide_no_nan(signal_likelihood, (signal_likelihood +␣
↪noise_likelihood))
    action = math.where((0.5 >= probability_signal) | (probability_signal >= 0.68), 2␣
↪* (probability_signal - 0.5), 0.)  # 1 sigma required to take action
    prob_kernel = math.exp(log_kernel * action)
    return prob_kernel, probability_signal


def inv_diffuse(grid: Grid, amount: float, uncertainty: Grid):
    f_uncertainty: math.Tensor = math.sqrt(math.sum(uncertainty.values ** 2, dim='x,y
↪'))  # all frequencies have the same uncertainty, 1/N in iFFT
    k_squared: math.Tensor = math.sum(math.fftfreq(grid.shape, grid.dx) ** 2, 'vector
↪')
    fft_laplace: math.Tensor = -(2 * np.pi) ** 2 * k_squared
    # --- Compute sharpening kernel with damping ---
    log_kernel = fft_laplace * -amount
    log_kernel_clamped = math.minimum(log_kernel, math.to_float(math.floor(math.
↪log(math.wrap(np.finfo(np.float32).max))))  # avoid overflow
    raw_kernel = math.exp(log_kernel_clamped)  # inverse diffusion FFT kernel, all␣
↪values >= 1
    inv_kernel = math.exp(-log_kernel)
    amp = math.fft(grid.values)
    kernel, sig_prob = apply_damping(raw_kernel, inv_kernel, amp, f_uncertainty, log_
↪kernel)
    # --- Apply and compute uncertainty ---
    data = math.real(math.ifft(amp * math.to_complex(kernel)))
    uncertainty = math.sqrt(math.sum(((f_uncertainty * kernel) ** 2))) / grid.shape.
↪get_size('x')  # 1/N normalization in iFFT
    uncertainty = grid * 0 + uncertainty
    return grid.with_values(data), uncertainty, abs(amp), raw_kernel, kernel, sig_prob
```

39.6 Neural network and loss function

For the neural network, we use a simple U-net architecture for the SIP and the regular DP+Adam version (in line with the previous sections, we'll denote it as `GD`). We train with a batch size of 128 and a constant learning rate of $\eta = 10^{-3}$, using 64 bit precision for physics but 32 bits for the network. The network updates are computed with TensorFlow's or PyTorch's automatic differentiation.

```
math.set_global_precision(64)
BATCH = batch(batch=128)
STEPS = 50

math.seed(0)
net = u_net(1, 1)
optimizer = adam(net, 0.001)
```

Now we'll define loss function for phiflow. The gradients for the network weights are always computed as d(loss_function)/dθ.

For SIP, we invert the physics, and then define the proxy loss as L^2(prediction - correction), where correction is taken as constant. Below this is realized via `x = field.stop_gradient(prediction)`. This L2 loss triggers the backprop towards the neural network weights. For SIP, the `y_l2` is only computed for comparison, and is not crucial for training.

The Adam / GD version for `sip=False` simply computes the L2 difference of the predicted, diffused y field to the target, and backprops through the operations for the gradient.

We also compute the reference $y = \mathcal{P}(x^*)$ in the loss function on-the-fly.

```
# @math.jit_compile
def loss_function(net, x_gt: CenteredGrid, sip: bool):
    y_target = diffuse.fourier(x_gt, 8., 1)
    with math.precision(32):
        prediction = field.native_call(net, field.to_float(y_target)).vector[0]
        prediction += field.mean(x_gt) - field.mean(prediction)
    x = field.stop_gradient(prediction)
    if sip:
        y = diffuse.fourier(x, 8., 1)
        dx, _, amp, raw_kernel, kernel, sig_prob = inv_diffuse(y_target - y, 8.,␣
↪uncertainty=abs(y_target - y) * 1e-6)
        correction = x + dx
        y_l2 = field.l2_loss(y - y_target)  # not important, just for tracking
        loss = field.l2_loss(prediction - correction) # proxy L2 loss for network
    else:
        y = diffuse.fourier(prediction, 8., 1)
        loss = y_l2 = field.l2_loss(y - y_target) # for Adam we backprop through the␣
↪loss and y
    return loss, x, y, field.l2_loss(x_gt - x), y_l2
```

39.7 Training

In the training loop, we generate data on-the-fly via `generate_heat_example()` and use phiflow's `update_weights()` function to call the correct functions of the chosen backend. Note that we're only printing the loss of the first 5 steps below for clarity, the whole history is collected in the `loss_` lists. The following cell runs the SIP version of the training:

```
loss_sip_x=[]; loss_sip_y=[]
for training_step in range(STEPS):
    data = generate_heat_example(spatial(x=64, y=64), BATCH)
    loss_value, x_sip, y_sip, x_l2, y_l2 = update_weights(net, optimizer, loss_
↪function, net, data, sip=True)
    loss_sip_x.append(float(x_l2.mean))
    loss_sip_y.append(float(y_l2.mean))
    if(training_step<5): print("SIP L2 loss x ; y: "+format(float(x_l2.mean))+" ;
↪"+format(float(y_l2.mean)) )
```

```
<ipython-input-2-350e9b9c59ed>:13: DeprecationWarning: HardGeometryMask and␣
↪SoftGeometryMask are deprecated. Use field.mask or field.resample instead.
  component_mask = SoftGeometryMask(component_box)
```

```
SIP L2 loss x ; y: 185.01902418878524 ; 52.86203787432579
SIP L2 loss x ; y: 80.09229485521256 ; 13.743946340489366
SIP L2 loss x ; y: 51.198420978751095 ; 4.212126705173006
SIP L2 loss x ; y: 51.166816963598585 ; 3.871802844235444
SIP L2 loss x ; y: 41.97257479831386 ; 3.3333446973370604
```

And now we can repeat the training with the DP version using Adam, deactivating the SIP update via `sip=False`:

```
math.seed(0)
net_gd = u_net(1, 1)
optimizer_gd = adam(net_gd, 0.001)

loss_gd_x=[]; loss_gd_y=[]
for training_step in range(STEPS):
    data = generate_heat_example(spatial(x=64, y=64), BATCH)
    loss_value, x_gd, y_gd, x_l2, y_l2 = update_weights(net_gd, optimizer_gd, loss_
↪function, net_gd, data, sip=False)
    loss_gd_x.append(float(x_l2.mean))
    loss_gd_y.append(float(y_l2.mean))
    if(training_step<5): print("GD L2 loss x ; y: "+format(float(x_l2.mean))+" ;
↪"+format(float(y_l2.mean)) )
```

```
<ipython-input-2-350e9b9c59ed>:13: DeprecationWarning: HardGeometryMask and␣
↪SoftGeometryMask are deprecated. Use field.mask or field.resample instead.
  component_mask = SoftGeometryMask(component_box)
```

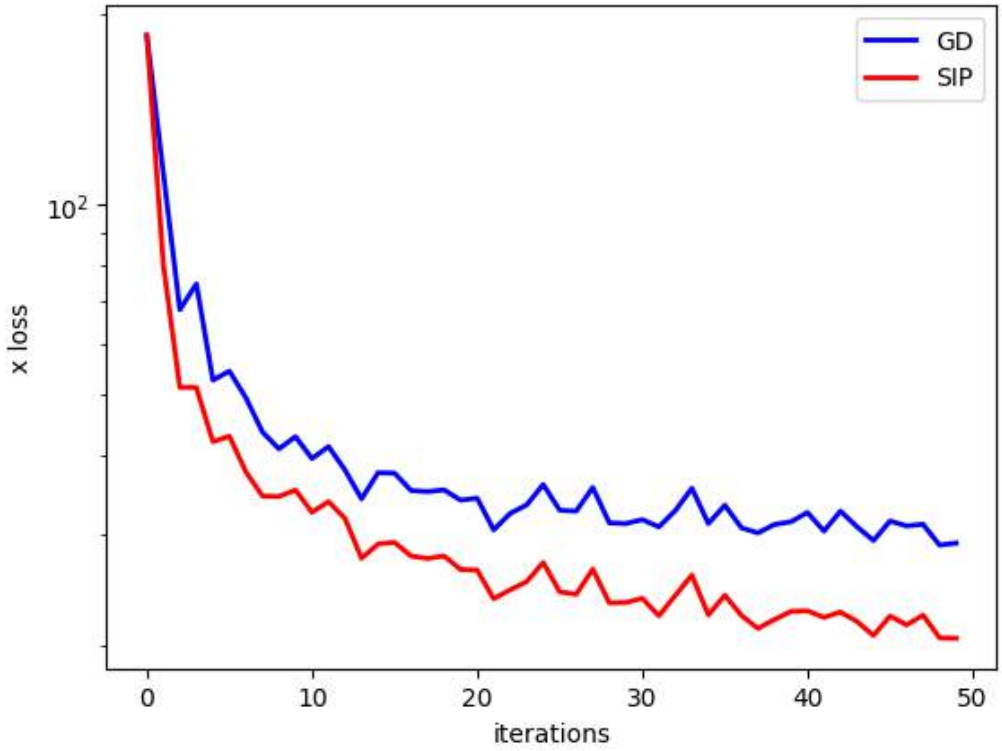
```
GD L2 loss x ; y: 185.01902418878524 ; 52.86203787432579
GD L2 loss x ; y: 111.90876514444714 ; 19.610744935770025
GD L2 loss x ; y: 67.98615290073411 ; 5.831835761821972
GD L2 loss x ; y: 74.69661928237372 ; 11.433268852185064
GD L2 loss x ; y: 52.55936273792499 ; 4.227996622501346
```

39.8 Evaluation

Now we can evaluate how the two variants behave in direct comparison. Note that due to the on-the-fly generation of randomized data, all samples are previously unseen, and hence we'll directly use the training curves here to draw conclusions about the performance of the two approaches.

The following graph shows the L^2 error over training in terms of the reconstructed x input, the main goal of our training.

```
import pylab as plt
fig = plt.figure().gca()
pltx = range(len(loss_gd_x))
fig.plot(pltx, loss_gd_x , lw=2, color='blue',      label="GD")
fig.plot(pltx, loss_sip_x  , lw=2, color='red', label="SIP")
plt.xlabel('iterations'); plt.ylabel('x loss'); plt.legend(); plt.yscale("log")
```



The log-scale for the loss in x nicely highlights that the SIP version does inherently better, and shows a significantly improved convergence for the NN training. This is purely caused by the better signal for the physics via the proxy loss. As shown in `loss_function()`, both variants use the same backprop-based update to change the weights of the NN. The improvements purely stem from the higher-order step in y space computed by the inverse simulator.

Just out of curiosity, we can also compare how the two versions compare in terms of difference in the output space y.

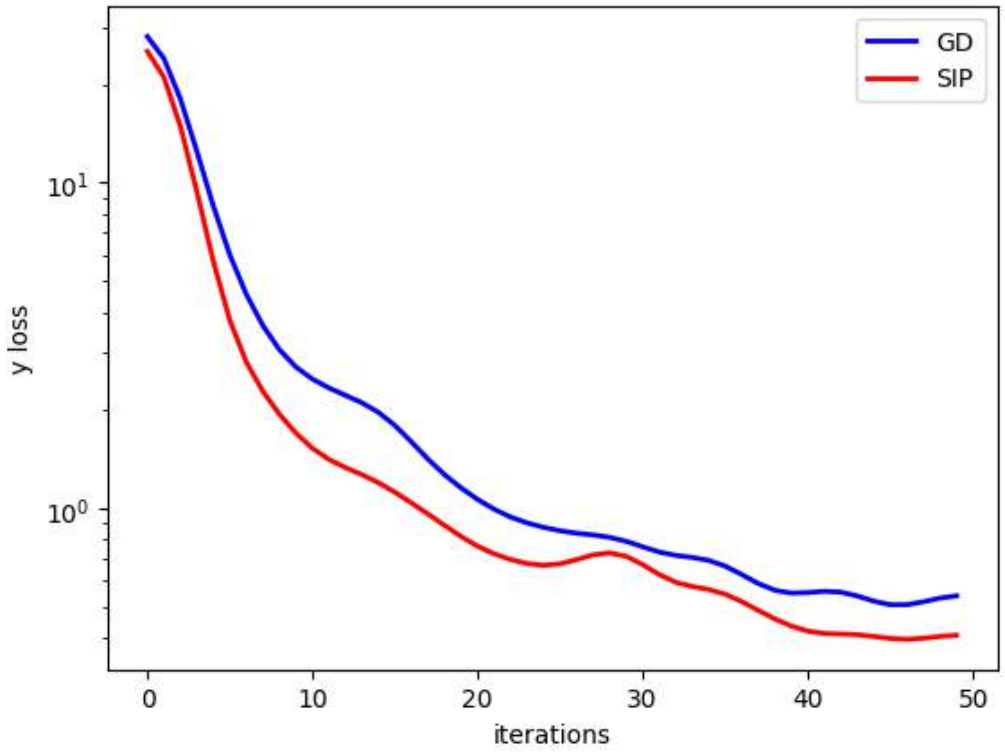
```
fig = plt.figure().gca()
pltx = range(len(loss_gd_y))
import scipy # for filtering
fig.plot(pltx, scipy.ndimage.filters.gaussian_filter1d(loss_gd_y,sigma=2)  , lw=2,␣
↪color='blue',      label="GD")
fig.plot(pltx, scipy.ndimage.filters.gaussian_filter1d(loss_sip_y ,sigma=2), lw=2,␣
```

(continues on next page)

(continued from previous page)

```
 ↪color='red', label="SIP")
plt.xlabel('iterations'); plt.ylabel('y loss'); plt.legend(); plt.yscale("log")
```

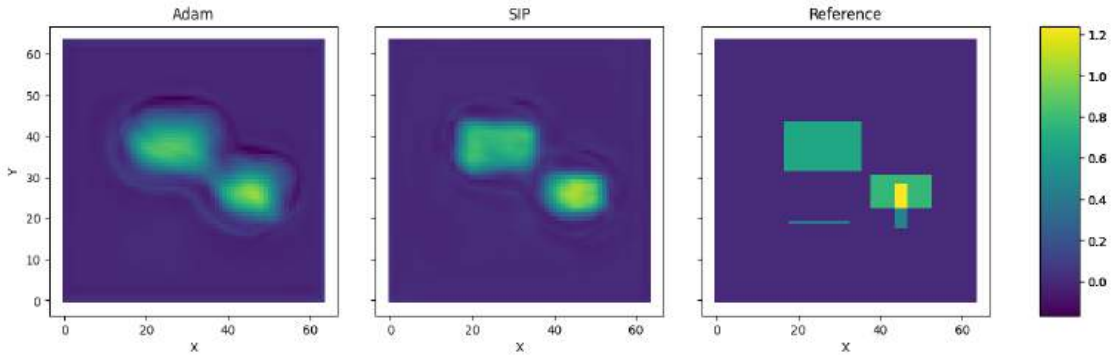
```
<ipython-input-10-63a3d36fc9b2>:4: DeprecationWarning: Please import `gaussian_
 ↪filter1d` from the `scipy.ndimage` namespace; the `scipy.ndimage.filters`␣
 ↪namespace is deprecated and will be removed in SciPy 2.0.0.
  fig.plot(pltx, scipy.ndimage.filters.gaussian_filter1d(loss_gd_y,sigma=2)  ,␣
 ↪lw=2, color='blue',      label="GD")
<ipython-input-10-63a3d36fc9b2>:5: DeprecationWarning: Please import `gaussian_
 ↪filter1d` from the `scipy.ndimage` namespace; the `scipy.ndimage.filters`␣
 ↪namespace is deprecated and will be removed in SciPy 2.0.0.
  fig.plot(pltx, scipy.ndimage.filters.gaussian_filter1d(loss_sip_y ,sigma=2),␣
 ↪lw=2, color='red', label="SIP")
```



There's likewise an improvements for SIPs, but it is not as pronounced as in x space. Luckily, the x reconstruction is the primary target, and hence of higher importance for the inverse problem at hand.

The differences in terms of L^2 error are also very obvious in direct comparison. The following cell plots a reconstruction from GD & Adam next to the SIP version, and the ground truth on the right. The difference is obvious: the SIP reconstruction is significantly sharper, and contains fewer halos than the GD version. This is a direct consequence of the undesirable behavior of GD applying diffusion to the backpropagated gradient, instead of working against it.

```
pltv = vis.plot(x_gd.values.batch[0], x_sip.values.batch[0], data.values.batch[0],␣
 ↪size=(15,4) )
pltv.axes[0].set_title("Adam"); pltv.axes[1].set_title("SIP"); pltv.axes[2].set_title(
 ↪"Reference");
```



39.9 Next steps

- For this example, it's worth experimenting with various training parameters: run the training longer, with varying learning rates, and different network sizes (or even different architectures).

CHAPTER

FORTY

HALF-INVERSE GRADIENTS

The scale-invariant physics updates (SIPs) of the previous chapters illustrated the importance of *inverting* the direction of the update step (in addition to making use of higher order terms). We'll now turn to an alternative for achieving the inversion, the so-called *Half-Inverse Gradients* (HIGs) [SHT22]. They come with their own set of pros and cons, and thus provide an interesting alternative for computing improved update steps for physics-based deep learning tasks.

Unlike the SIPs, they do not require an analytical inverse solver. The HIGs jointly invert the neural network part as well as the physical model. As a drawback, they require an SVD for a large Jacobian matrix.

Preview: HIGs versus SIPs (and versus Adam)

More specifically the HIGs:

- do not require an analytical inverse solver (in contrast to SIPs),
- and they jointly invert the neural network part as well as the physical model.

As a drawback, HIGs:

- require an SVD for a large Jacobian matrix,
- and are based on first-order information (similar to regular gradients).

Howver, in contrast to regular gradients, they use the full Jacobian matrix. So as we'll see below, they typically outperform regular GD and Adam significantly.

40.1 Derivation

As mentioned during the derivation of inverse simulator updates in (36.3), the update for regular Newton steps uses the inverse Hessian matrix. If we rewrite its update for the case of an L^2 loss, we arrive at the *Gauss-Newton* (GN) method:

$$\Delta\theta_{\text{GN}} = -\eta \left(\left(\frac{\partial y}{\partial \theta} \right)^T \cdot \left(\frac{\partial y}{\partial \theta} \right) \right)^{-1} \cdot \left(\frac{\partial y}{\partial \theta} \right)^T \cdot \left(\frac{\partial L}{\partial y} \right)^T . \tag{40.1}$$

For a full-rank Jacobian $\partial y / \partial \theta$, the transposed Jacobian cancels out, and the equation simplifies to

$$\Delta\theta_{\text{GN}} = -\eta \left(\frac{\partial y}{\partial \theta} \right)^{-1} \cdot \left(\frac{\partial L}{\partial y} \right)^T . \tag{40.2}$$

This looks much simpler, but still leaves us with a Jacobian matrix to invert. This Jacobian is typically non-square, and has small singular values which cause problems during inversion. Naively applying methods like Gauss-Newton can quickly explode. However, as we're dealing with cases where we have a physics solver in the training loop, the small singular

values are often relevant for the physics. Hence, we don't want to just discard these parts of the learning signal, but rather preserve as many of them as possible.

This motivates the HIG update, which employs a partial and truncated inversion of the form

$$\Delta\theta_{\text{HIG}} = -\eta \cdot \left(\frac{\partial y}{\partial \theta}\right)^{-1/2} \cdot \left(\frac{\partial L}{\partial y}\right)^{T}, \tag{40.3}$$

where the square-root for $^{-1/2}$ is computed via an SVD, and denotes the half-inverse. I.e., for a matrix A, we compute its half-inverse via a singular value decomposition as $A^{-1/2} = V\Lambda^{-1/2}U^T$, where Λ contains the singular values. During this step we can also take care of numerical noise in the form of small singular values. All entries of Λ smaller than a threshold τ are set to zero.

Note

Truncation versus Clamping: It might seem attractive at first to clamp singular values to a small value τ, instead of discarding them by setting them to zero. However, the singular vectors corresponding to these small singular values are exactly the ones which are potentially unreliable. A small τ yields a large contribution during the inversion, and thus these singular vectors would cause problems when clamping. Hence, it's a much better idea to discard their content by setting their singular values to zero.

The use of a partial inversion via $^{-1/2}$ instead of a full inversion with $^{-1}$ helps preventing that small eigenvalues lead to overly large contributions in the update step. This is inspired by Adam, which normalizes the search direction via $J/(\sqrt{(diag(J^TJ)})$ instead of inverting it via $J/(J^TJ)$, with J being the diagonal of the Jacobian matrix. For Adam, this compromise is necessary due to the rough approximation via the diagonal. For HIGs, we use the full Jacobian, and hence can do a proper inversion. Nonetheless, as outlined in the original paper [SHT22], the half-inversion regularizes the inverse and provides substantial improvements for the learning, while reducing the chance of gradient explosions.

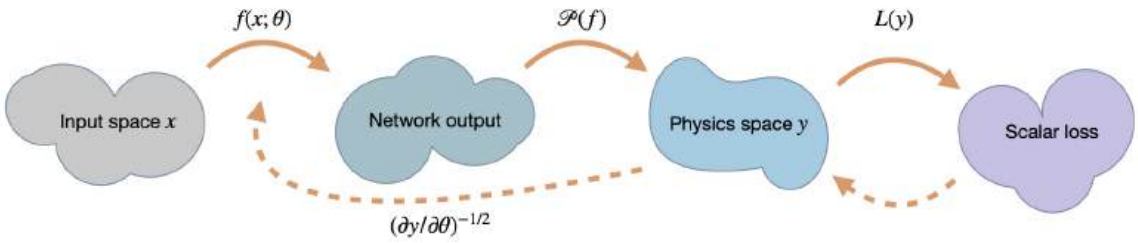


Fig. 40.1: A visual overview of the different spaces involved in HIG training. Most importantly, it makes use of the joint, inverse Jacobian for neural network and physics.

40.2 Constructing the Jacobian

The formulation above hides one important aspect of HIGs: the search direction we compute not only jointly takes into account the scaling of neural network and physics, but can also incorporate information from all the samples in a mini-batch. This has the advantage of finding the optimal direction (in an L^2 sense) to minimize the loss, instead of averaging directions as done with GD or Adam.

To achieve, this, the Jacobian matrix for $\partial y/\partial\theta$ is concatenated from the individual Jacobians of each sample in a mini-batch. Let x_i, y_i denote input and output of sample i in a mini-batch, respectively, then the final Jacobian is constructed

via all the $\frac{\partial y_i}{\partial \theta}\Big|_{x_i}$ as

$$\frac{\partial y}{\partial \theta} := \begin{pmatrix} \frac{\partial y_1}{\partial \theta}\Big|_{x_1} \\ \frac{\partial y_2}{\partial \theta}\Big|_{x_2} \\ \vdots \\ \frac{\partial y_b}{\partial \theta}\Big|_{x_b} \end{pmatrix} .$$

The notation with $\Big|_{x_i}$ also makes clear that all parts of the Jacobian are evaluated with the corresponding input states. In contrast to regular optimizations, where larger batches typically don't pay off too much due to the averaging effect, the HIGs have a stronger dependence on the batch size. They often profit from larger mini-batch sizes.

To summarize, compute the HIG update requires evaluating the individual Jacobians of a batch, doing an SVD of the combined Jacobian, truncating and half-inverting the singular values, and computing the update direction by re-assembling the half-inverted Jacobian matrix.

40.3 Properties Illustrated via a Toy Example

This is a good time to illustrate the properties mentioned in the previous paragraphs with a real example. As learning target, we'll consider a simple two-dimensional setting with the function

$$\hat{y}(x) = (\sin(6x), \cos(9x)) \text{ for } x \in [-1, 1]$$

and a scaled loss function

$$L(y, \hat{y}; \lambda) = \frac{1}{2}(y_1 - \hat{y}_1)^2 + \frac{1}{2}(\lambda \cdot y_2 - \hat{y}_2)^2 .$$

Here y_1 and y_2 denote the first, and second component of y (in contrast to the subscript i used for the entries of a mini-batch above). Note that the scaling via λ is intentionally only applied to the second component in the loss. This mimics an uneven scaling of the two components as commonly encountered in physical simulation settings, the amount of which can be chosen via λ.

We'll use a small neural network with a single hidden layer consisting of 7 neurons with *tanh()* activations and the objective to learn $\hat{y}$.

40.4 Well-conditioned

Let's first look at the well-conditioned case with $\lambda = 1$. In the following image, we'll compare Adam as the most popular GD-representative, Gauss-Newton (GN) as "classical" method, and the HIGs. These methods are evaluated w.r.t. three aspects: naturally, it's interesting to see how the loss evolves. In addition, we'll consider the distribution of neuron activations from the resulting neural network states (more on that below). Finally, it's also interesting to observe how the optimization influences the resulting target states (in y space) produced by the neural network. Note that the y-space graph below shows only a single, but fairly representative, x, y pair. The other two show quantities from a larger set of validation inputs.

As seen here, all three methods fare okay on all fronts for the well conditioned case: the loss decreases to around 10^{-2} and 10^{-3}.

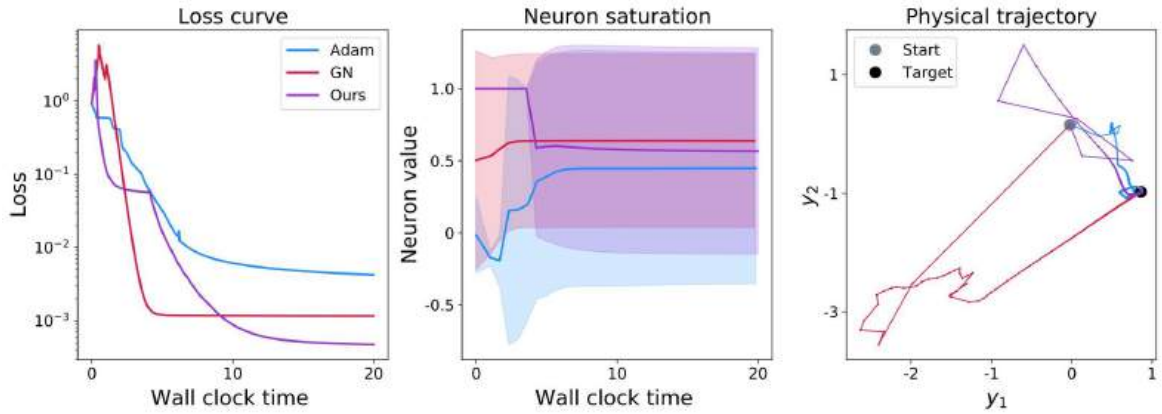


Fig. 40.2: The example problem for a well-conditioned case. Comparing Adam, GN, and HIGs.

In addition, the neuron activations, which are shown in terms of mean and standard deviation, all show a broad range of values (as indicated by the solid-shaded regions representing the standard deviation). This means that the neurons of all three networks produce a wide range of values. While it's difficult to interpret specific values here, it's a good sign that different values are produced by different inputs. If this was not the case, i.e., different inputs producing constant values (despite the obviously different targets in $\hat{y}$), this would be a very bad sign. This is typically caused by fully saturated neurons whose state was "destroyed" by an overly large update step. But for this well-conditioned toy example, this saturation is not showing up.

Finally, the third graph on the right shows the evolution in terms of a single input-output pair. The starting point from the initial network state is shown in light gray, while the ground truth target $\hat{y}$ is shown as a black dot. Most importantly, all three methods reach the black dot in the end. For this simple example, it's not overly impressive to see this. However, it's still interesting that both GN and HIG exhibit large jumps in the initial stages of the learning process (the first few segments leaving the gray dot). This is caused by the fairly bad initial state, and the inversion, which leads to significant changes of the NN state and its outputs. In contrast, the momentum terms of Adam reduce this jumpiness: the initial jumps in the light blue line are smaller than those of the other two.

Overall, the behavior of all three methods is largely in line with what we'd expect: while the loss surely could go down more, and some of the steps in y seem to momentarily do in the wrong direction, all three methods cope quite well with this case. Not surprisingly, this picture will change when making things harder with a more ill-conditioned Jacobian resulting from a small λ.

40.5 Ill-conditioned

Now we can consider a less well-conditioned case with $\lambda = 0.01$. The conditioning could be much worse in real-world PDE solvers, but interestingly, this factor of $1/100$ is sufficient to illustrate the problems that arise in practice. Here are the same 3 graphs for the ill-conditioned case:

The loss curves now show a different behavior: both Adam and GN do not manage to decrease the loss beyond a level of around 0.2 (compared to the 0.01 and better from before). Adam has significant problems with the bad scaling of the y^2 component, and fails to properly converge. For GN, the complete inversion of the Jacobians causes gradient explosions, which destroy the positive effects of the inversion. Even worse, they cause the neural network to effectively get stuck.

This becomes even clearer in the middle graph, showing the activations statistics. The red curve of GN very quickly saturates at 1, without showing any variance. Hence, all neurons have saturated, and do not produce meaningful signals anymore. This not only means that the target function isn't approximated well, it also means that future gradients will effectively be zero, and these neurons are lost to all future learning iterations. Hence, this is a highly undesirable case that

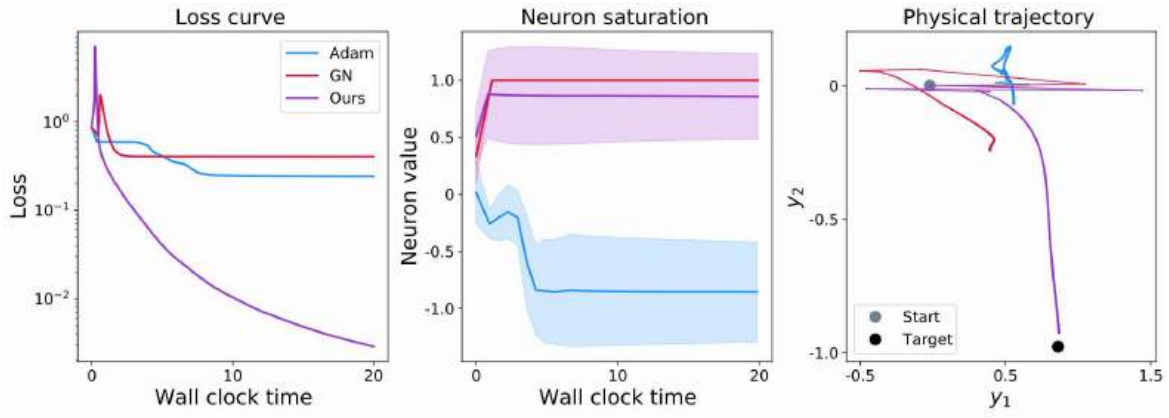


Fig. 40.3: The example problem for an ill-conditioned case. Comparing Adam, GN, and HIGs.

we want to avoid in practice. It's also worth pointing out that this doesn't always happen for GN. However, it regularly happens, e.g. when individual samples in a batch lead to vectors in the Jacobian that are linearly dependent (or very close to it), and thus makes GN a sub-optimal choice.

The third graph on the right side of figure Fig. 40.3 shows the resulting behavior in terms of the outputs. As already indicated by the loss values, both Adam and GN do not reach the target (the black dot). Interestingly, it's also apparent that both have much more problems along the y^2 direction, which we used to cause the bad conditioning: they both make some progress along the x-axis of the graph (y^1), but don't move much towards the y^2 target value. This is illustrating the discussions above: GN gets stuck due to its saturated neurons, while Adam struggles to undo the scaling of y^2.

40.6 Summary of Half-Inverse Gradients

Note that for all examples so far, we've improved upon the *differentiable physics* (DP) training from the previous chapters. I.e., we've focused on combinations of neural networks and PDE solving operators. The latter need to be differentiable for training with regular GD, as well as for HIG-based training.

In contrast, for training with SIPs (from *Scale Invariant Physics Training*), we even needed to provide a full inverse solver. As shown there, this has advantages, but differentiates SIPs from DP and HIGs. Thus, the HIGs share more similarities with, e.g., *Reducing Numerical Errors with Neural Operators* and *Solving Inverse Problems with NNs*, than with the example *Learning to Invert Heat Conduction with Scale-invariant Updates*.

This is a good time to give a specific code example of how to train physical NNs with HIGs: we'll look at a classic case, a system of coupled oscillators.

CHAPTER

FORTYONE

COUPLED OSCILLATORS WITH HALF-INVERSE GRADIENTS

In this notebook, we'll turn to a practical example comparing the half-inverse gradients (HIGs) to other methods for training neural networks with physical loss functions. Specifically, we'll compare:

1. Adam: as a standard gradient-descent (GD) based network optimizer,
2. Scale-Invariant Physics: the previously described algorithm that fully inverts the physics,
3. Half-Inverse Gradients: which locally and jointly inverting physics and network.

41.1 Inverse problem setup

The learning task is to find the control function steering the dynamics of a coupled oscillator system. This is a classical problem in physics, and a good case to evaluate the HIGs due to it's smaller size. We're using two mass points, and thus we'll only have four degrees of freedom for position and velocity of both points (compared to, e.g., the $32 \times 32 \times 2$ unknowns we'd get even for "only" a small fluid simulation with 32 cells along x and y).

Nonetheless, the oscillators are a highly-non trivial case: we aim for applying a control such that the initial state is reached again after a chosen time interval. We'll use 24 steps of a fourth-order Runge-Kutta scheme, and hence the NN has to learn how to best "nudge" the two mass points over the course of all time steps, so that they end up at the desired position with the right velocity at the right time.

A system of N coupled oscillators is described by the following Hamiltonian:

$$\mathcal{H}(x_i, p_i, t) = \sum_i \left(\frac{x_i^2}{2} + \frac{p_i^2}{2} + \alpha \cdot (x_i - x_{i+1})^4 + u(t) \cdot x_i \cdot c_i \right),$$

which provides the basis for the RK4 time integration below.

41.2 Problem statement

More concretely, we consider a set of different phyiscal inputs (x_i). Using a corresponding control function (u_i), we can influence the time evolution of our physical system $\mathcal{P}$ and receive an output state (y_i).

$$y_i = \mathcal{P}(x_i; u_i)$$

If we want to evolve a given initial state(x_i) into a given target state (y_i^*), we receive an inverse problem for the control function (u_i). The quality of between desired target state (y_i^*) and received target state (y_i) is measured by a loss function (L).

$$\arg\min_{u_i} L(y_i^*, \mathcal{P}(x_i; u_i))$$

If we use a neural network (f parameterized by θ) to learn the control function over a set of input/output pairs (x_i, y_i^*), we transform the above physics optimization task into a learning problem.

$$\arg\min_\theta \sum_i L(y_i^*, \mathcal{P}(x_i; f(x_i, y_i; \theta)))$$

Before we begin by setting up the physics solver $\mathcal{P}$, we import the necessary libraries (this example uses TensorFlow), and define the main global variable `MODE`, that switches between *Adam* (`'GD'`), *Scale-invariant physics* (`'SIP'`), and *Half-inverse gradients* (`'HIG'`).

```
import numpy as np
import tensorflow as tf
import time, os

# main switch for the three methods:
MODE = 'HIG'  # HIG | SIP | GD
```

41.3 Coupled linear oscillator simulation

For the physics simulation, we'll solve a differential equation for a system of coupled linear oscillators with a control term. The time integration, a fourth-order Runge-Kutta scheme is used.

Below, we're first defining a few global constants: `Nx`: the number of oscillators, `Nt`: the number of time evolution steps, and `DT`: the length of one time step.

We'll then define a helper function to set up a Laplacian stencil, the `coupled_oscillators_batch()` function which computes the simulation for a whole mini batch of values, and finally the `solver()` function, which runs the desired number of time steps with a given control signal.

```
Nx = 2
Nt = 24
DT = 0.5

def build_laplace(n,boundary='0'):
    if n==1:
        return np.zeros((1,1),dtype=np.float32)
    d1 = -2 * np.ones((n,),dtype=np.float32)
    d2 = 1 * np.ones((n-1,),dtype=np.float32)
    lap = np.zeros((n,n),dtype=np.float32)
    lap[range(n),range(n)]=d1
    lap[range(1,n),range(n-1)]=d2
    lap[range(n-1),range(1,n)]=d2
    if boundary=='0':
        lap[0,0]=lap[n-1,n-1]=-1

    return lap

@tf.function
def coupled_oscillators_batch( x, control):
    '''
    ODE of type:    x' = f(x)
    :param x_in:    position and velocities, shape: (batch, 2 * number of osc) order␣
↪second index: x_i , v_i
    :param control: control function, shape: (batch,)
    :return:
    '''
```

(continues on next page)

(continued from previous page)

```
    #print('coupled_oscillators_batch')
    n_osc = x.shape[1]//2

    # natural time evo
    a1 = np.array([[0,1],[-1,0]],dtype=np.float32)
    a2 = np.eye(n_osc,dtype=np.float32)
    A = np.kron(a1,a2)
    x_dot1 = tf.tensordot(x,A,axes = (1,1))

    # interaction term
    interaction_strength = 0.2
    b1 = np.array([[0,0],[1,0]],dtype=np.float32)
    b2 = build_laplace(n_osc)
    B = interaction_strength * np.kron(b1,b2)
    x_dot2 = tf.tensordot(x,B, axes=(1, 1))

    # control term
    control_vector = np.zeros((n_osc,),dtype=np.float32)
    control_vector[-1] = 1.0
    c1 = np.array([0,1],dtype=np.float32)
    c2 = control_vector
    C = np.kron(c1,c2)
    x_dot3 = tf.tensordot(control,C, axes=0)

    #all terms
    x_dot = x_dot1 + x_dot2 +x_dot3
    return x_dot

@tf.function
def runge_kutta_4_batch(x_0, dt, control, ODE_f_batch):

    f_0_0 = ODE_f_batch(x_0, control)
    x_14 = x_0 + 0.5 * dt * f_0_0

    f_12_14 = ODE_f_batch(x_14, control)
    x_12 = x_0 + 0.5 * dt * f_12_14

    f_12_12 = ODE_f_batch(x_12, control)
    x_34 = x_0 + dt * f_12_12

    terms = f_0_0 + 2 * f_12_14 + 2 * f_12_12 + ODE_f_batch(x_34, control)
    x1 = x_0 + dt * terms / 6

    return x1

@tf.function
def solver(x0, control):
    x = x0
    for i in range(Nt):
        x = runge_kutta_4_batch(x, DT, control[:,i], coupled_oscillators_batch)
    return x
```

41.4 Training setup

The neural network itself is quite simple: it consists of four dense layers (the intermediate ones with 20 neurons each), and `tanh` activation functions.

```
act = tf.keras.activations.tanh
model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(input_shape=(2*Nx)),
    tf.keras.layers.Dense(20, activation=act),
    tf.keras.layers.Dense(20, activation=act),
    tf.keras.layers.Dense(20, activation=act),
    tf.keras.layers.Dense(Nt, activation='linear')
    ])
```

As loss function, we'll use an L^2 loss:

```
@tf.function
def loss_function(a,b):
    diff = a-b
    loss_batch = tf.reduce_sum(diff**2,axis=1)
    loss = tf.reduce_sum(loss_batch)
    return loss
```

And as data set for training we simply create 4k of random position values which the oscillators start with (`X_TRAIN`), and which they should return to at the end of the simulation (`Y_TRAIN`). As they should return to their initial states, we have `X_TRAIN=Y_TRAIN`.

```
N = 2**12
X_TRAIN = np.random.rand(N, 2 * Nx).astype(np.float32)
Y_TRAIN = X_TRAIN # the target states are identical
```

41.5 Training

For the optimization procedure of the neural network training, we need to set up some global parameters. The next cell initializes some suitable values tailored to each of the three methods. These were determined heuristically to work best for each. If we try to use the same settings for all, this would inevitably make the comparison unfair for some of them.

1. *Adam*: This is the most widely used NN optimizer, and we're using it here as a representative of the GD family. Note that the truncation parameter has no meaning for Adam.
2. *SIP*: The specified optimizer is the one used for network optimization. The physics inversion is done via Gauss-Newton and corresponds to an exact inversion since the physical optimization landscape is quadratic. For the Jacobian inversion in Gauss-Newton, we can specify a truncation parameter.
3. *HIG*: To obtain the HIG algorithm, the optimizer has to be set to SGD. For the Jacobian half-inversion, we can specify a truncation parameter. Optimal batch sizes are typically lower than for the other two, and a learning rate of 1 typically works very well.

The maximal training time in seconds is set via `MAX_TIME` below.

```
if MODE=='HIG': # HIG training
  OPTIMIZER = tf.keras.optimizers.SGD
  BATCH_SIZE = 32 # larger batches make HIGs unnecessariy slow...
  LR = 1.0
  TRUNC = 10**-10
```

(continues on next page)

(continued from previous page)

```
elif MODE=='SIP': # SIP training
  OPTIMIZER = tf.keras.optimizers.Adam
  BATCH_SIZE = 256
  LR = 0.001  # for the internal step with Adam
  TRUNC = 0   # not used

else: #  Adam Training (as GD representative)
  MODE = 'GD'
  OPTIMIZER = tf.keras.optimizers.Adam
  BATCH_SIZE = 256
  LR = 0.001
  TRUNC = 0 # not used

# global parameters for all three methods
MAX_TIME = 100 # [s]
print("Running variant: "+MODE)
```

```
Running variant: HIG
```

The next function, `HIG_pinv()`, is a crucial one: it constructs the half-inverse of a given matrix for HIGs. It computes an SVD, takes the square-root of the singular values, and then re-assembles the matrix.

```
from tensorflow.python.framework import ops
from tensorflow.python.framework import tensor_shape
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.util import dispatch
from tensorflow.python.util.tf_export import tf_export
from tensorflow.python.ops.linalg.linalg_impl import _maybe_validate_matrix,svd

# partial inversion of the Jacobian via SVD:
# this function is adopted from tensorflow's SVD function, and published here under␣
 ↪its license: Apache-2.0 License , https://github.com/tensorflow/tensorflow/blob/
 ↪master/LICENSE
@tf_export('linalg.HIG_pinv')
@dispatch.add_dispatch_support
def HIG_pinv(a, rcond=None,beta=0.5, validate_args=False, name=None):

  with ops.name_scope(name or 'pinv'):
    a = ops.convert_to_tensor(a, name='a')

    assertions = _maybe_validate_matrix(a, validate_args)
    if assertions:
      with ops.control_dependencies(assertions):
        a = array_ops.identity(a)

    dtype = a.dtype.as_numpy_dtype

    if rcond is None:

      def get_dim_size(dim):
        dim_val = tensor_shape.dimension_value(a.shape[dim])
        if dim_val is not None:
          return dim_val
        return array_ops.shape(a)[dim]
```

(continues on next page)

(continued from previous page)

```
      num_rows = get_dim_size(-2)
      num_cols = get_dim_size(-1)
      if isinstance(num_rows, int) and isinstance(num_cols, int):
        max_rows_cols = float(max(num_rows, num_cols))
      else:
        max_rows_cols = math_ops.cast(
            math_ops.maximum(num_rows, num_cols), dtype)
      rcond = 10. * max_rows_cols * np.finfo(dtype).eps

    rcond = ops.convert_to_tensor(rcond, dtype=dtype, name='rcond')

    [ singular_values, left_singular_vectors, right_singular_vectors, ] = svd(
        a, full_matrices=False, compute_uv=True)

    cutoff = rcond * math_ops.reduce_max(singular_values, axis=-1)
    singular_values = array_ops.where_v2(
        singular_values > array_ops.expand_dims_v2(cutoff, -1), singular_values**beta,
        np.array(np.inf, dtype))

    a_pinv = math_ops.matmul(
        right_singular_vectors / array_ops.expand_dims_v2(singular_values, -2),
        left_singular_vectors,
        adjoint_b=True)

    if a.shape is not None and a.shape.rank is not None:
      a_pinv.set_shape(a.shape[:-2].concatenate([a.shape[-1], a.shape[-2]]))

    return a_pinv
```

Now we have all pieces in place to run the training. The next cell defines a Python class to organize the neural network optimization. It receives the physics solver, network model, loss function and a data set, and runs as many epochs as possible within the given time limit `MAX_TIME`.

Depending on the chosen optimization method, the mini batch updates differ:

1. Adam: Compute loss gradient, then apply the Adam update.
2. PG: Compute loss gradient und physics Jacobian, invert them data-point-wise, and compute network updates via the proxy loss and Adam.
3. HIG: Compute loss gradient and network-physics Jacobian, then jointly compute the half-inversion, and update the network parameters with the resulting step.

The `mini_batch_update()` method of the optimizer class realizes these three variants.

```
class Optimization():
    def __init__(self,model,solver,loss_function,x_train,y_train):
        self.model = model
        self.solver = solver
        self.loss_function = loss_function
        self.x_train = x_train
        self.y_train = y_train
        self.y_dim = y_train.shape[1]
        self.weight_shapes = [weight_tensor.shape for weight_tensor in self.model.
↪trainable_weights]

    def set_params(self,batch_size,learning_rate,optimizer,max_time,mode,trunc):
```

(continues on next page)

(continued from previous page)

```
        self.number_of_batches = N // batch_size
        self.max_time = max_time
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.optimizer = optimizer(learning_rate)
        self.mode = mode
        self.trunc = trunc


    def computation(self,x_batch, y_batch):
        control_batch = self.model(y_batch)
        y_prediction_batch = self.solver(x_batch,control_batch)
        loss = self.loss_function(y_batch,y_prediction_batch)
        return loss


    @tf.function
    def gd_get_derivatives(self,x_batch, y_batch):

        with tf.GradientTape(persistent=True) as tape:
            tape.watch(self.model.trainable_variables)
            loss = self.computation(x_batch,y_batch)
            loss_per_dp = loss / self.batch_size
        grad = tape.gradient(loss_per_dp, self.model.trainable_variables)
        return grad


    @tf.function
    def pg_get_physics_derivatives(self,x_batch, y_batch): # physics gradient for SIP

        with tf.GradientTape(persistent=True) as tape:
            control_batch = self.model(y_batch)
            tape.watch(control_batch)
            y_prediction_batch = self.solver(x_batch,control_batch)
            loss = self.loss_function(y_batch,y_prediction_batch)
            loss_per_dp = loss / self.batch_size

        jacy = tape.batch_jacobian(y_prediction_batch,control_batch)
        grad = tape.gradient(loss_per_dp, y_prediction_batch)
        return jacy,grad,control_batch

    @tf.function
    def pg_get_network_derivatives(self,x_batch, y_batch,new_control_batch):
↪#physical grads

        with tf.GradientTape(persistent=True) as tape:
            tape.watch(self.model.trainable_variables)
            control_batch = self.model(y_batch)
            loss = self.loss_function(new_control_batch,control_batch)
            #y_prediction_batch = self.solver(x_batch,control_batch)
            #loss = self.loss_function(y_batch,y_prediction_batch)
            loss_per_dp = loss / self.batch_size

        network_grad = tape.gradient(loss_per_dp, self.model.trainable_variables)
        return network_grad

    @tf.function
```

(continues on next page)

(continued from previous page)

```
    def hig_get_derivatives(self,x_batch,y_batch):

        with tf.GradientTape(persistent=True) as tape:
            tape.watch(self.model.trainable_variables)
            control_batch = self.model(y_batch)
            y_prediction_batch = self.solver(x_batch,control_batch)
            loss = self.loss_function(y_batch,y_prediction_batch)
            loss_per_dp = loss / self.batch_size

        jacy = tape.jacobian(y_prediction_batch, self.model.trainable_variables,␣
↪experimental_use_pfor=True)
        loss_grad = tape.gradient(loss_per_dp, y_prediction_batch)
        return jacy, loss_grad


    def mini_batch_update(self,x_batch, y_batch):
        if self.mode=="GD":
            grad = self.gd_get_derivatives(x_batch, y_batch)
            self.optimizer.apply_gradients(zip(grad, self.model.trainable_weights))

        elif self.mode=="SIP":
            jacy,grad,control_batch = self.pg_get_physics_derivatives(x_batch, y_
↪batch)
            grad_e = tf.expand_dims(grad,-1)
            pinv = tf.linalg.pinv(jacy,rcond=10**-5)
            delta_control_label_batch = (pinv@grad_e)[:,:,0]
            new_control_batch = control_batch - delta_control_label_batch
            network_grad = self.pg_get_network_derivatives(x_batch, y_batch,new_
↪control_batch)
            self.optimizer.apply_gradients(zip(network_grad, self.model.trainable_
↪weights))

        elif self.mode =='HIG':
            jacy, grad = self.hig_get_derivatives(x_batch, y_batch)
            flat_jacy_list = [tf.reshape(jac, (self.batch_size * self.y_dim, -1)) for␣
↪jac in jacy]
            flat_jacy = tf.concat(flat_jacy_list, axis=1)
            flat_grad = tf.reshape(grad, (-1,))
            inv = HIG_pinv(flat_jacy, rcond=self.trunc)
            processed_derivatives = tf.tensordot(inv, flat_grad, axes=(1, 0))
            #processed_derivatives = self.linear_solve(flat_jacy, flat_grad)
            update_list = []
            l1 = 0
            for k, shape in enumerate(self.weight_shapes):
                l2 = l1 + np.prod(shape)
                upd = processed_derivatives[l1:l2]
                upd = np.reshape(upd, shape)
                update_list.append(upd)
                l1 = l2
            self.optimizer.apply_gradients(zip(update_list, self.model.trainable_
↪weights))


    def epoch_update(self):
        for batch_index in range(self.number_of_batches):
            position = batch_index * self.batch_size
            x_batch = self.x_train[position:position + self.batch_size]
```

(continues on next page)

(continued from previous page)

```
            y_batch = self.y_train[position:position + self.batch_size]
            self.mini_batch_update(x_batch, y_batch)

    def eval(self,epoch,wc_time,ep_dur):
        train_loss = self.computation(self.x_train,self.y_train)
        train_loss_per_dp = train_loss / N
        if epoch<5 or epoch%20==0: print('Epoch: ', epoch,', wall clock time: ',wc_
↪time,', loss: ', float(train_loss_per_dp) )
        #print('TrainLoss:', train_loss_per_dp)
        #print('Epoch: ', epoch,' WallClockTime: ',wc_time,' EpochDuration: ',ep_dur )
        return train_loss_per_dp

    def start_training(self):
        init_loss = self.eval(0,0,0)
        init_time = time.time()
        time_list = [init_time]
        loss_list = [init_loss]

        epoch=0
        wc_time = 0

        while wc_time<self.max_time:

            duration = time.time()
            self.epoch_update()
            duration = time.time()-duration

            epoch += 1
            wc_time += duration

            loss = self.eval(epoch,wc_time,duration)
            time_list.append(duration)
            loss_list.append(loss)

        time_list = np.array(time_list)
        loss_list = np.array(loss_list)
        time_list[0] = 0
        return time_list, loss_list
```

All that's left to do is to start the training with the chosen global parameters, and collect the results in `time_list`, and `loss_list`.

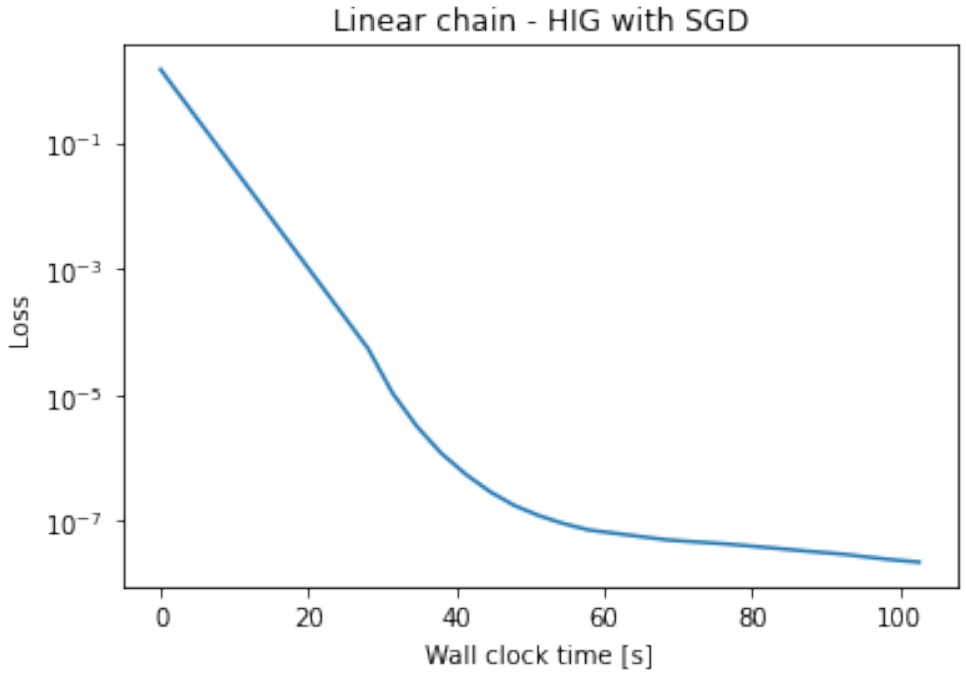
```
opt = Optimization(model, solver, loss_function, X_TRAIN, Y_TRAIN)
opt.set_params(BATCH_SIZE, LR, OPTIMIZER, MAX_TIME, MODE, TRUNC)
time_list, loss_list = opt.start_training()
```

```
Epoch:  0 , wall clock time:  0 , loss:  1.4401766061782837
Epoch:  1 , wall clock time:  28.06755018234253 , loss:  5.44398972124327e-05
Epoch:  2 , wall clock time:  31.38792371749878 , loss:  1.064436037268024e-05
Epoch:  3 , wall clock time:  34.690271854400635 , loss:  3.163525434501935e-06
Epoch:  4 , wall clock time:  37.9914448261261 , loss:  1.1857609933940694e-06
```

41.6 Evaluation

Now we can evaluate how our training converged over time. The following graph shows the loss evolution over time in seconds.

```
import matplotlib.pyplot as plt

plt.plot(np.cumsum(time_list),loss_list)
plt.yscale('log')
plt.xlabel('Wall clock time [s]'); plt.ylabel('Loss')
plt.title('Linear chain - '+MODE+' with '+str(OPTIMIZER.__name__))
plt.show()
```



For all three methods, you'll see a big linear step right at the start. As we're – for fairness – measuring the whole runtime, this first step includes all TensorFlow initialization steps, which are significantly more involved for HIG and SIP. Adam is much faster in terms of initialization, and likewise faster per training iteration.

All three methods by themselves manage to bring down the loss. What's more interesting is to see how they compare. For this, the next cell stores the training evolution, and this notebook needs to be run one time with each of the three methods to produce the final comparison graph.

```
path = '/home/'
namet = 'time'
namel = 'loss'
np.savetxt(path+MODE+namet+'.txt',time_list)
np.savetxt(path+MODE+namel+'.txt',loss_list)
```

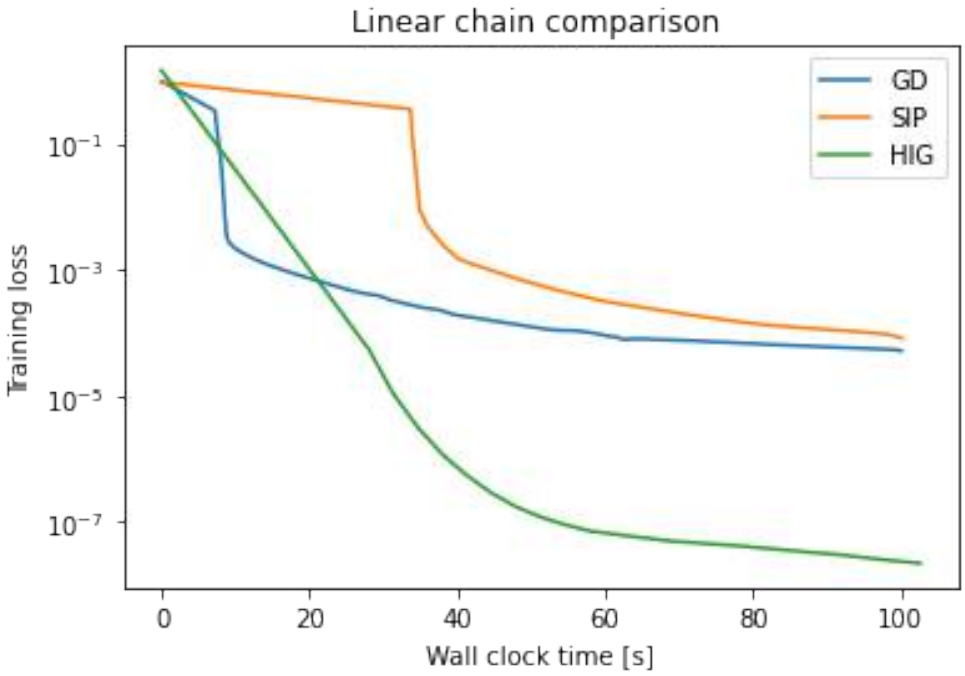
After runs with each of the methods, we can show them side by side:

```
from os.path import exists
```

(continues on next page)

(continued from previous page)

```
# if previous runs are available, compare all 3
if exists(path+'HIG'+namet+'.txt') and exists(path+'HIG'+namel+'.txt') and␣
↪exists(path+'SIP'+namet+'.txt') and exists(path+'SIP'+namel+'.txt') and exists(path+
↪'GD'+namet+'.txt') and exists(path+'GD'+namel+'.txt'):
  lt_hig = np.loadtxt(path+'HIG'+namet+'.txt')
  ll_hig = np.loadtxt(path+'HIG'+namel+'.txt')

  lt_sip = np.loadtxt(path+'SIP'+namet+'.txt')
  ll_sip = np.loadtxt(path+'SIP'+namel+'.txt')

  lt_gd = np.loadtxt(path+'GD'+namet+'.txt')
  ll_gd = np.loadtxt(path+'GD'+namel+'.txt')

  plt.plot(np.cumsum(lt_gd),ll_gd,   label="GD")
  plt.plot(np.cumsum(lt_sip),ll_sip, label="SIP")
  plt.plot(np.cumsum(lt_hig),ll_hig,   label="HIG")
  plt.yscale('log')
  plt.xlabel('Wall clock time [s]'); plt.ylabel('Training loss'); plt.legend()
  plt.title('Linear chain comparison ')
  plt.show()
else:
  print("Run this notebook three times with MODE='HIG|SIP|GD' to produce the final␣
↪graph")
```



This graph makes the significant differences in terms of convergence very clear: Adam (the blue `GD` curve), performs a large number of updates, but its rough approximation of the Hessian is not enough to converge to high accuracies. It stagnates at a high loss level.

The SIP updates don't outperform Adam in this scenario. This is caused by the relatively simple physics (the *linear* oscillators), and the higher runtime cost of SIP. If you run this example longer, SIP will actually overtake Adam, but start suffering from numerical issues with the full inversion.

The HIGs perform much better than the two others: despite being fairly slow per iteration, the half-inversion produces a very good update, that makes the training converge to very low loss values very quickly. The HIGs reach an accuracy that is around four order of magnitudes better than the other two methods.

41.7 Next steps

There's a variety of interesting directions for further tests and modifications with this notebook:

- Most importantly, we've actually only looked at training performance so far! This keeps the notebook reasonably short, but it's admittedly bad practice. While we claim that HIGs likewise work on *real* test data, this is a great next step with this notebook: allocate proper test samples, and re-run the evaluation for all three methods on the test data.
- Also, you can vary the physics behavior: use more oscillators for longer or shorter time spans, or even include a non-linear force (as employed in the HIG paper). Be warned: for the latter, the SIP version will require a new inverse solver, though.

CHAPTER

FORTYTWO

DISCUSSION OF IMPROVED GRADIENTS

At this point it's a good time to take another step back, and assess the different methods introduced so far. For deep learning applications, we can broadly distinguish three approaches: the *regular* differentiable physics (DP) training, the training with half-inverse gradients (HIGs), and using the scale-invariant physics updates (SIPs). Unfortunately, we can't simply discard two of them, and focus on a single approach for all future endeavours. However, discussing the pros and cons sheds light on some fundamental aspects of physics-based deep learning.

42.1 Addressing scaling issues

First and foremost, a central motivation for improved updates is the need to address the scaling issues of the learning problems. This is not a completely new problem: numerous deep learning algorithms were proposed to address these for training NNs. However, the combination of NNs with physical simulations brings new challenges that at the same time provide new angles to tackle this problem. On the negative side, we have additional, highly non-linear operators from the PDE models. On the positive side, these operators typically do not have free parameters during learning, and thus can be treated with different, tailored methods.

This is exactly where HIGs and SIPs come in: instead of treating the physical simulation like the rest of the NNs (this is the DP approach), they show how much can be achieved with custom inverse solvers (SIPs) or a custom numerical inversion (HIGs). Both methods make important steps towards *scale-invariant* training.

42.2 Computational Resources

Both cases usually lead to more complicated and resource intensive training. However, assuming that we can re-use a trained model many times after the training has been completed, there are many areas of application where this can quickly pay off: the trained NNs, despite being identical in runtime to those obtained from other training methods, often achieve significantly improved accuracies. Achieving similar levels of accuracy with regular Adam and DP-based training can be completely infeasible.

When such a trained NN is used, e.g., as a surrogate model for an inverse problem, it might be executed a large number of times, and the improved accuracy can save correspondingly large amounts of computational resources in such a follow up stage. A good potential example are shape optimizations for the drag reduction of bodies immersed in a fluid [CCHT21].

42.3 Summary

To summarize, this chapter demonstrated the importance of the inversion. An important takeaway message is that the regular gradients from NN training are not the best choice when PDEs are involved. In these situations we can get much better information about how to direct the optimization than the localized first-order information that regular gradients provide.

Even when the inversion is only done for the physics simulation component (as with SIPs), it can substantially improve the learning process. The custom inverse solvers allow us to employ higher-order information in the training.

✓ Pro SIP:

- Very accurate "gradient" information for physical simulations.
- Often strongly improved convergence and model performance.

✗ Con SIP:

- Require inverse simulators (at least local ones).
- Only makes the physics component scale-invariant.

The HIGs on the other hand, go back to first order information in the form of Jacobians. They show how useful the inversion can be even without any higher order terms. At the same time, they make use of a combined inversion of NN and physics, taking into account all samples of a mini-batch to compute an optimal first-order direction.

✓ Pro HIG:

- Robustly addresses scaling issues, jointly for physical models and NN.
- Improved convergence and model performance.

✗ Con HIG:

- Requires an SVD for a potentially large Jacobian matrix.
- This can be costly in terms of runtime and memory.

In both cases, the resulting neural networks can yield a performance that we simply can't obtain by, e.g., training longer with a simpler DP or supervised approach. So, if we plan to evaluate these models often, e.g., shipping them in an application, this increased one-time cost will pay off in the long run.

This concludes the chapter on improved learning methods for physics-based NNs. It's clearly an active topic of research, with plenty of room for new methods, but the algorithms here already indicate the potential of tailored learning algorithms for physical problems. This also concludes the focus on numerical simulations as DL components. In the next chapter, we'll instead focus on a different statistical viewpoint, namely the inclusion of uncertainty.

Part VIII

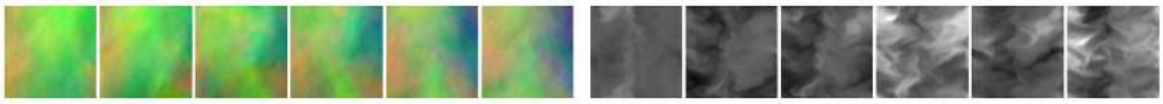
Fast Forward Topics

CHAPTER

FORTYTHREE

ADDITIONAL TOPICS

The next sections will give a shorter introduction to other classic topics that are interesting in the context of physics-based deep learning. These topics (for now) do not come with executable notebooks, but we will still point to existing open source implementations for each of them.



More specifically, we will look at:

- Model reduction and time series predictions, i.e., using to DL predict the evolution of a physical system in a latent space. This typically replaces a numerical solver, and we can make use of special techniques from the DL area that target time series.
- Generative models are likewise an own topic in DL, and here especially generative adversarial networks were shown to be powerful tools. They also represent a highly interesting training approach involving to separate NNs.
- Meshless methods and unstructured meshes are an important topic for classical simulations. Here, we'll look at a specific Lagrangian method that employs learning in the context of dynamic, particle-based representations.

CHAPTER

FORTYFOUR

MODEL REDUCTION AND TIME SERIES

An inherent challenge for many practical PDE solvers is the large dimensionality of the resulting problems. Our model $\mathcal{P}$ is typically discretized with $\mathcal{O}(n^3)$ samples for a 3 dimensional problem (with n denoting the number of samples along one axis), and for time-dependent phenomena we additionally have a discretization along time. The latter typically scales in accordance to the spatial dimensions. This gives an overall samples count on the order of $\mathcal{O}(n^4)$. Not surprisingly, the workload in these situations quickly explodes for larger n (and for all practical high-fidelity applications we want n to be as large as possible).

One popular way to reduce the complexity is to map a spatial state of our system $\mathbf{s_t} \in \mathbb{R}^{n^3}$ into a much lower dimensional state $\mathbf{c_t} \in \mathbb{R}^m$, with $m \ll n^3$. Within this latent space, we estimate the evolution of our system by inferring a new state $\mathbf{c_{t+1}}$, which we then decode to obtain $\mathbf{s_{t+1}}$. In order for this to work, it's crucial that we can choose m large enough that it captures all important structures in our solution manifold, and that the time prediction of $\mathbf{c_{t+1}}$ can be computed efficiently, such that we obtain a gain in performance despite the additional encoding and decoding steps. In practice, the explosion in terms of unknowns for regular simulations (the $\mathcal{O}(n^3)$ above) coupled with a super-linear complexity for computing a new state $\mathbf{s_t}$ directly makes this approach very expensive, while working with the latent space points $\mathbf{c}$ very quickly pays off for small m.

However, it's crucial that encoder and decoder do a good job at reducing the dimensionality of the problem. This is a very good task for DL approaches. Furthermore, we then need a time evolution of the latent space states $\mathbf{c}$, and for most practical model equations, we cannot find closed form solutions to evolve $\mathbf{c}$. Hence, this likewise poses a very good problem for DL. To summarize, we're facing two challenges: learning a good spatial encoding and decoding, together with learning an accurate time evolution. Below, we will describe an approach to solve this problem following Wiewel et al. [WBT19] & [WKA+20], which in turn employs the encoder/decoder of Kim et al. [KAT+19].

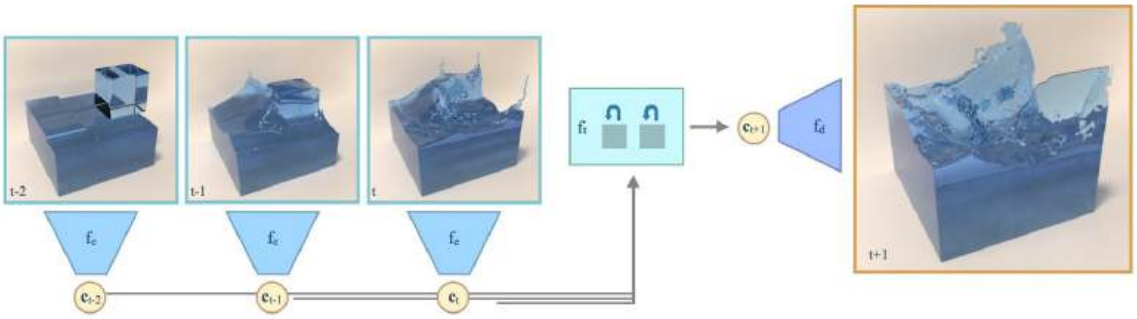


Fig. 44.1: For time series predictions with ROMs, we encode the state of our system with an encoder f_e, predict the time evolution with f_t, and then decode the full spatial information with a decoder f_d.

44.1 Reduced order models

Reducing the dimension and complexity of computational models, often called *reduced order modeling* (ROM) or *model reduction*, is a classic topic in the computational field. Traditional techniques often employ techniques such as principal component analysis to arrive at a basis for a chosen space of solution. However, being linear by construction, these approaches have inherent limitations when representing complex, non-linear solution manifolds. In practice, all "interesting" solutions are highly non-linear, and hence DL has received a substantial amount of interest as a way to learn non-linear representations. Due to the non-linearity, DL representations can potentially yield a high accuracy with fewer degrees of freedom in the reduced model compared to classic approaches.

The canonical NN for reduced models is an *autoencoder*. This denotes a network whose sole task is to reconstruct a given input x while passing it through a bottleneck that is typically located in or near the middle of the stack of layers of the NN. The data in the bottleneck then represents the compressed, latent space representation $\mathbf{c}$. The part of the network leading up to the bottleneck $\mathbf{c}$ is the encoder f_e, and the part after it the decoder f_d. In combination, the learning task can be written as

$$\arg\min_{\theta_e,\theta_d} |f_d(f_e(\mathbf{s};\theta_e);\theta_d) - \mathbf{s}|_2^2$$

with the encoder $f_e : \mathbb{R}^{n^3} \to \mathbb{R}^m$ with weights θ_e, and the decoder $f_d : \mathbb{R}^m \to \mathbb{R}^{n^3}$ with weights θ_d. For this learning objective we do not require any other data than the $\mathbf{s}$, as these represent inputs as well as the reference outputs.

Autoencoder networks are typically realized as stacks of convolutional layers. While the details of these layers can be chosen flexibly, a key property of all autoencoder architectures is that no connection between encoder and decoder part may exist. Hence, the network has to be separable for encoder and decoder. This is natural, as any connections (or information) shared between encoder and decoder would prevent using the encoder or decoder in a standalone manner. E.g., the decoder has to be able to decode a full state $\mathbf{s}$ purely from a latent space point $\mathbf{c}$.

44.1.1 Autoencoder variants

One popular variant of autoencoders is worth a mention here: the so-called *variational autoencoders*, or VAEs. These autoencoders follow the structure above, but additionally employ a loss term to shape the latent space of $\mathbf{c}$. Its goal is to let the latent space follow a known distribution. This makes it possible to draw samples in latent space without workarounds such as having to project samples into the latent space.

Typically we use a normal distribution as target, which makes the latent space an m dimensional unit cube: each dimension should have a zero mean and unit standard deviation. This approach is especially useful if the decoder should be used as a generative model. E.g., we can then produce $\mathbf{c}$ samples directly, and decode them to obtain full states. While this is very useful for applications such as constructing generative models for faces or other types of natural images, it is less crucial in a simulation setting. Here we want to obtain a latent space that facilitates the temporal prediction, rather than being able to easily produce samples from it.

44.2 Time series

The goal of the temporal prediction is to compute a latent space state at time $t + 1$ given one or more previous latent space states. The most straight-forward way to formulate the corresponding minimization problem is

$$\arg\min_{\theta_p} |f_p(\mathbf{c}_t;\theta_p) - \mathbf{c}_{t+1}|_2^2$$

where the prediction network is denoted by f_p to distinguish it from encoder and decoder, above. This already implies that we're facing a recurrent task: any ith step is the result of i evaluations of f_p, i.e. $\mathbf{c}_{t+i} = f_p^{(i)}(\mathbf{c}_t;\theta_p)$. As there is an inherent per-evaluation error, it is typically important to train this process for more than a single step, such that the f_p network "sees" the drift it produces in terms of the latent space states over time.

Koopman operators

In classical dynamical systems literature, a data-driven prediction of future states is typically formulated in terms of the so-called *Koopman operator*, which usually takes the form of a matrix, i.e. uses a linear approach.

Traditional works have focused on obtaining good *Koopman operators* that yield a high accuracy in combination with a basis to span the space of solutions. In the approach outlined above the f_p network can be seen as a non-linear Koopman operator.

In order for this approach to work, we either need an appropriate history of previous states to uniquely identify the right next state, or our network has to internally store the previous history of states it has seen.

For the former variant, the prediction network f_p receives more than a single $\mathbf{c}_t$. For the latter variant, we can turn to algorithms from the subfield of *recurrent neural networks* (RNNs). A variety of architectures have been proposed to encode and store temporal states of a system, the most popular ones being *long short-term memory* (LSTM) networks, *gated recurrent units* (GRUs), or lately attention-based *transformer* networks. No matter which variant is used, these approaches always work with fully-connected layers as the latent space vectors do not exhibit any spatial structure, but typically represent a seemingly random collection of values. Due to the fully-connected layers, the prediction networks quickly grow in terms of their parameter count, and thus require a relatively small latent-space dimension m. Luckily, this is in line with our main goals, as outlined at the top.

44.3 End-to-end training

In the formulation above we have clearly split the en- / decoding and the time prediction parts. However, in practice an *end-to-end* training of all networks involved in a certain task is usually preferable, as the networks can adjust their behavior in accordance with the other components involved in the task.

For the time prediction, we can formulate the objective in terms of $\mathbf{s}$, and use en- and decoder in the time prediction to compute the loss:

$$\arg\min_{\theta_e,\theta_p,\theta_d} | f_d(f_p(f_e(\mathbf{s}_t;\theta_e);\theta_p);\theta_d) - \mathbf{s}_{t+1}|_2^2$$

Ideally, this step is furthermore unrolled over time to stabilize the evolution over time. The resulting training will be significantly more expensive, as more weights need to be trained at once, and a much larger number of intermediate states needs to be processed. However, the increased cost typically pays off with a reduced overall inference error. The following images show several time frames of an example prediction of [WKA+20], which additionally couples the learned time evolution with a numerically solved advection step.

To summarize, DL allows us to move from linear subspaces to non-linear manifolds, and provides a basis for performing complex steps (such as time evolutions) in the resulting latent space.

44.4 Source code

In order to make practical experiments in this area of deep learning, we can recommend this latent space simulation code, which realizes an end-to-end training for encoding and prediction. Alternatively, this learned model reduction code focuses on the encoding and decoding aspects.

Both are available as open source and use a combination of TensorFlow and mantaflow as DL and fluid simulation frameworks.

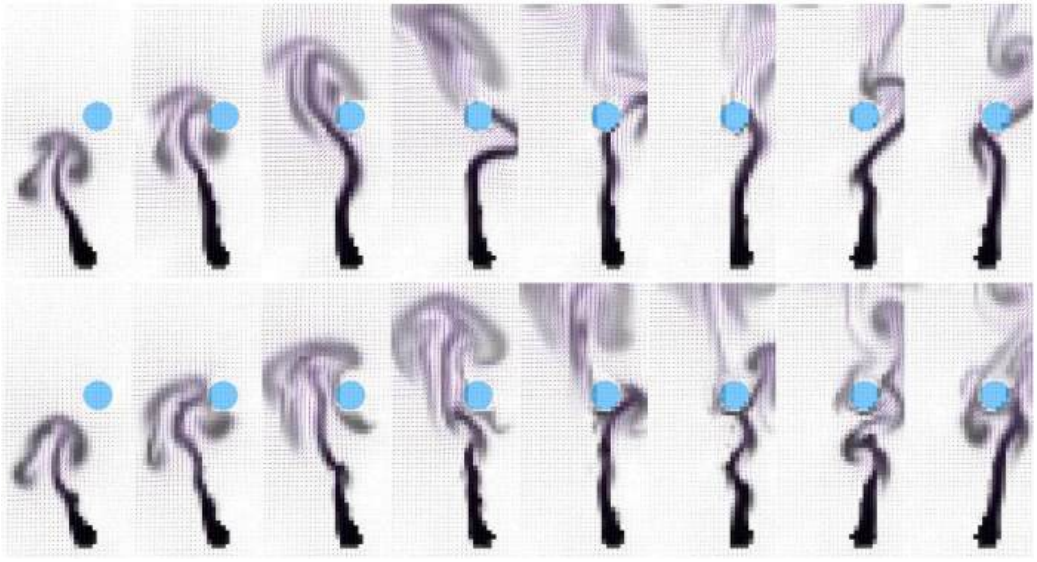


Fig. 44.2: The learned prediction is shown at the top, the reference simulation at the bottom.

CHAPTER

FORTYFIVE

UNSTRUCTURED MESHES AND MESHLESS METHODS

For all computer-based methods we need to find a suitable *discrete* representation. While this is straight-forward for cases such as data consisting only of integers, it is more challenging for continuously changing quantities such as the temperature in a room. While the previous examples have focused on aspects beyond discretization (and used Cartesian grids as a placeholder), the following chapter will target scenarios where learning Neural operators with dynamically changing and adaptive discretizations have a benefit.

45.1 Types of computational meshes

As outlined in *Neural Network Architectures*, we can distinguish three types of computational meshes (or "grids") with which discretizations are typically performed:

- **structured** meshes: Structured meshes have a regular arrangement of the sample points, and an implicitly defined connectivity. In the simplest case it's a dense Cartesian grid.
- **unstructured** meshes: On the other hand can have an arbitrary connectivity and arrangement. The flexibility gained from this typically also leads to an increased computational cost.
- **meshless** or particle-based finally share arbitrary arrangements of the sample points with unstructured meshes, but in contrast implicitly define connectivity via neighborhoods, i.e. a suitable distance metric.

Structured meshes are currently very well supported within DL algorithms due to their similarity to image data, and hence they typically simplify implementations and allow for using stable, established DL components (especially regular convolutional layers). However, for target functions that exhibit an uneven mix of smooth and complex regions, the other two mesh types can have advantages.

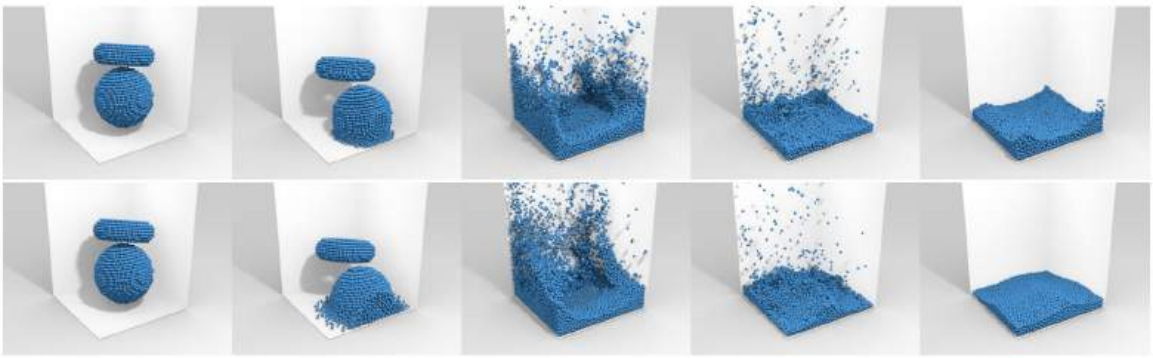


Fig. 45.1: Lagrangian simulations of liquids: the sampling points move with the material, and undergo large changes. In the top row timesteps of a learned simulation, in the bottom row the traditional SPH solver.

45.2 Unstructured meshes and graph neural networks

Within computational sciences the generation of improved mesh structures is a challenging and ongoing effort. The numerous H-,C- and O-type meshes which were proposed with numerous variations over the years for flows around airfoils are a good example.

Unstructured meshes offer the largest flexibility here on the meshing side, but of course need to be supported by the simulator. Interestingly, unstructured meshes share many properties with *graph* neural networks (GNNs), which extend the classic ideas of DL on Cartesian grids to graph structures. Despite growing support, working with GNNs typically causes a fair amount of additional complexity in an implementation, and the arbitrary connectivities call for *message-passing* approaches between the nodes of a graph. This message passing is usually realized using fully-connected layers, instead of convolutions.

Thus, in the following, we will focus on a particle-based method [UPTK19], which offers the same flexibility in terms of spatial adaptivity as GNNs. These were previously employed for a very similar goal [SGGP+20], however, the method below enables a real convolution operator for learning the physical relationships.

45.3 Meshless and particle-based methods

Organizing connectivity explicitly is particularly challenging in dynamic cases, e.g., for Lagrangian representations of moving materials where the connectivity quickly becomes obsolete over time. In such situations, methods that rely on flexible, re-computed connectivities are a good choice. Operations are then defined in terms of a spatial neighborhood around the sampling locations (also called "particles" or just "points"), and due to the lack of an explicit mesh-structure these methods are also known as "meshless" methods. Arguably, different unstructured, graph and meshless variants can typically be translated from one to the other, but nonetheless the rough distinction outlined above gives an indicator for how a method works.

In the following, we will discuss an example targeting splashing liquids as a particularly challenging case. For these simulations, the fluid material moves significantly and is often distributed very non-uniformly.

The general outline of a learned, particle-based simulation is similar to a DL method working on a Cartesian grid: we store data such as the velocity at certain locations, and then repeatedly perform convolutions to create a latent space at each location. Each convolution reads in the latent space content within its support and produces a result, which is activated with a suitable non-linear function such as ReLU. This is done multiple times in parallel to produce a latent space vector, and the resulting latent space vectors at each location serve as inputs for the next stage of convolutions. After expanding the size of the latent space over the course of a few layers, it is contracted again to produce the desired result, e.g., an acceleration.

45.4 Continuous convolutions

A generic, discrete convolution operator to compute the convolution $(f * g)$ between functions f and g has the form

$$(f * g)(\mathbf{x}) = \sum_{\tau \in \Omega} f(\mathbf{x} + \tau) g(\tau),$$

where τ denotes the offset vector, and Ω defines the support of the filter function (typically g).

We transfer this idea to particles and point clouds by evaluating a convolution on a set of i locations $\mathbf{x}_i$ in a radial neighborhood $\mathcal{N}(\mathbf{x}, R)$ around $\mathbf{x}$. Here, R denotes the radius within which the convolution should have support. We define a continuous version of the convolution following [UPTK19]:

$$(f * g)(\mathbf{x}) = \sum_i f(\mathbf{x}_i)\ g(\Lambda(\mathbf{x}_i - \mathbf{x})).$$

Here, the mapping Λ plays a central role: it represents a mapping from the unit ball to the unit cube, which allows us to use a simple grid to represent the unknowns in the convolution kernel. This greatly simplifies the construction and handling of the convolution kernel, and is illustrated in the following figure:

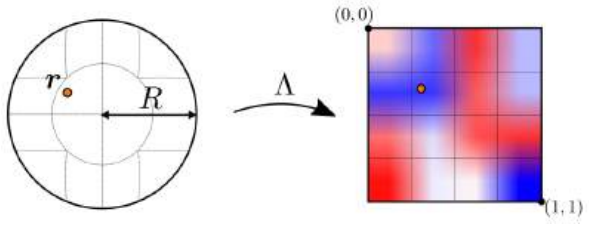


Fig. 45.2: The unit ball to unit cube mapping employed for the kernel function of the continuous convolution.

In a physical setting, e.g., the simulation of fluid dynamics, we can additionally introduce a radial weighting function, denoted as a below to make sure the kernel has a smooth falloff. This yields

$$(f * g)(\mathbf{x}) = \frac{1}{a_{\mathcal{N}}} \sum_i a(\mathbf{x}_i, \mathbf{x})\ f(\mathbf{x}_i)\ g(\Lambda(\mathbf{x}_i - \mathbf{x})),$$

where $a_{\mathcal{N}}$ denotes a normalization factor $a_{\mathcal{N}} = \sum_{i \in \mathcal{N}(\mathbf{x},R)} a(\mathbf{x}_i, \mathbf{x})$. There's is quite some flexibility for a, but below we'll use the following weighting function

$$a(\mathbf{x}_i, \mathbf{x}) = \begin{cases} \left(1 - \frac{\|\mathbf{x}_i - \mathbf{x}\|_2^2}{R^2}\right)^3 & \text{for } \|\mathbf{x}_i - \mathbf{x}\|_2 < R \\ 0 & \text{else.} \end{cases}$$

This ensures that the learned influence smoothly drops to zero for each of the individual convolutions.

For a lean architecture, a small fully-connected layer can be added for each convolution to process the content of the destination particle itself. This makes it possible to use relatively small kernels with even sizes, e.g., sizes of 4^3 [UPTK19].

45.5 Learning the dynamics of liquids

The architecture outlined above can then be trained with a collection of randomized reference data from a particle-based Navier-Stokes solver. The resulting network yields a good accuracy with a very small and efficient model. E.g., compared to GNN-based approaches the continuous convolution requires significantly fewer weights and is faster to evaluate.

Interestingly, a particularly tough case for such a learned solver is a container of liquid that should come to rest. If the training data is not specifically engineered to contain many such cases, the network receives only a relatively small of such cases at training time. Moreover, a simulation typically takes many steps to come to rest (many more than are unrolled for training). Hence the network is not explicitly trained to reproduce such behavior.

Nonetheless, an interesting side-effect of having a trained NN for such a liquid simulation by construction provides a differentiable solver. Based on a pre-trained network, the learned solver then supports optimization via gradient descent, e.g., w.r.t. input parameters such as viscosity. The following image shows an examplary *prediction* task with continuous convolutions from [UPTK19].

Fig. 45.3: An example of a particle-based liquid spreading in a landscape scenario, simulated with learned, continuous convolutions.

45.6 Source code

For a practical implementation of the continuous convolutions, another important step is a fast collection of neighboring particles for $\mathcal{N}$. An efficient example implementation can be found at https://github.com/intel-isl/DeepLagrangianFluids, together with training code for learning the dynamics of liquids, an example of which is shown in the figure above.

CHAPTER

FORTYSIX

GENERATIVE ADVERSARIAL NETWORKS

We've dealt with generative AI techniques and diffusion modeling in detail in *Introduction to Probabilistic Learning*. As outlined there, the fundamental problem to fully represent all possible states of a variable $\mathbf{x}$ under consideration, i.e. to capture its full distribution, is a very old topic. Hence, even before DDPMs&Co. there were techniques to make this possible, and *generative adversarial networks* (GANs) were shown to be powerful tools in this context. While they've been largely replaced by diffsion approaches in research, GANs use a highly interesting approach, and the following sections will give an introduction and show what's possible with GANs.

Traditionally, GANs were employed when the data has ambiguous solutions, and no differentiable physics model is available to disambiguate the data. In such a case a supervised learning would yield an undesirable averaging that can be prevented with a GAN approach.

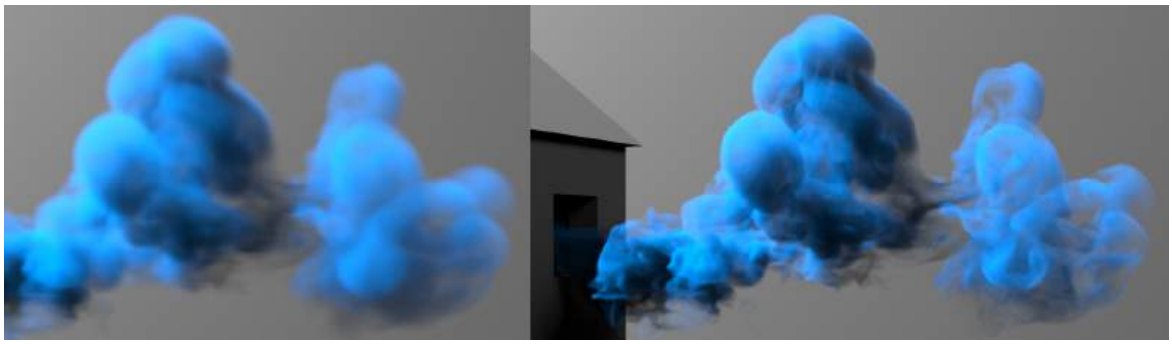


Fig. 46.1: GANs were shown to work well for tasks such as the inference of super-resolution solutions where the range of possible results can be highly ambiguous.

46.1 Maximum likelihood estimation

To train a GAN we have to briefly turn to *classification problems*, which we've managed to ignore up to now. For classification, the learning objective takes a slightly different form than the regression objective in equation (3.2) of *Models and Equations*: We now want to maximize the likelihood of a learned representation f that assigns a probability to an input $\mathbf{x}_i$ given a set of weights θ for a chosen set of i distinct classes. This yields a maximization problem of the form

$$\arg\max_\theta \Pi_i f(\mathbf{x}_i;\theta), \tag{46.1}$$

the classic *maximum likelihood estimation* (MLE). In practice, it is typically turned into a sum of negative log likelihoods to give the learning objective

$$\arg\min_\theta -\sum_i \log f(\mathbf{x}_i;\theta).$$

There are quite a few equivalent viewpoints for this fundamental expression: e.g., it can be seen as minimizing the KL-divergence between the empirical distribution as given by our training data set and the learned one. It likewise represents a maximization of the expectation as defined by the training data, i.e. $\mathbb{E} \log f(\mathbf{x}_i; \theta)$. This in turn is the same as the classical cross-entropy loss for classification problems, i.e., a classifier with a sigmoid as activation function.. The takeaway message here is that the wide-spread training via cross entropy is effectively a maximum likelihood estimation for probabilities over the inputs, as defined in equation (46.1).

46.2 Adversarial training

MLE is a crucial component for GANs: here we have a *generator* that is typically similar to a decoder network, e.g., the second half of an autoencoder from *Model Reduction and Time Series*. For regular GANs, the generator receives a random input vector, denoted with **z**, from which it should produce the desired output.

However, instead of directly training the generator, we employ a second network that serves as loss for the generator. This second network is called *discriminator*, and it has a classification task: to distinguish the generated samples from "real" ones. The real ones are typically provided in the form of a training data set, samples of which will be denoted as **x** below.

For regular GANs training the classification task of the discriminator is typically formulated as

$$\arg\min_{\theta_d} -\frac{1}{2}\mathbb{E} \log D(\mathbf{y}) - \frac{1}{2}\mathbb{E} \log(1 - D(G(\mathbf{z}))$$

which, as outlined above, is a standard binary cross-entropy training for the class of real samples **y**, and the generated ones $G(\mathbf{z})$. With the formulation above, the discriminator is trained to maximize the loss via producing an output of 1 for the real samples, and 0 for the generated ones.

The key for the generator loss is to employ the discriminator and produce samples that are classified as real by the discriminator:

$$\arg\min_{\theta_g} -\frac{1}{2}\mathbb{E} \log D(G(\mathbf{z}))$$

Typically, this training is alternated, performing one step for D and then one for G. Thus the D network is kept constant, and provides a gradient to "steer" G in the right direction to produce samples that are indistinguishable from the real ones. As D is likewise an NN, it is differentiable by construction, and can provide the necessary gradients.

46.3 Regularization

Due to the coupled, alternating training, GAN training has a reputation of being finicky in practice. Instead of a single, non-linear optimization problem, we now have two coupled ones, for which we need to find a fragile balance. (Otherwise we'll get the dreaded *mode-collapse* problem: once one of the two network "collapses" to a trivial solution, the coupled training breaks down.)

To alleviate this problem, regularization is often crucial to achieve a stable training. In the simplest case, we can add an L^1 regularizer w.r.t. reference data with a small coefficient for the generator G. Along those lines, pre-training the generator in a supervised fashion can help to start with a stable state for G. (However, then D usually also needs a certain amount of pre-training to keep the balance.)

46.4 Conditional GANs

For physical problems the regular GANs which generate solutions from the randomized latent-space $\mathbf{z}$ above are not overly useful. Rather, we often have inputs such as parameters, boundary conditions or partial solutions which should be used to infer an output. Such cases represent *conditional* GANs, which means that instead of $G(\mathbf{z})$, we now have $G(\mathbf{x})$, where $\mathbf{x}$ denotes the input data.

A good scenario for conditional GANs are super-resolution networks: These have the task to compute a high-resolution output given a sparse or low-resolution input solution.

46.5 Ambiguous solutions

One of the main advantages of GANs is that they can prevent an undesirable averaging for ambiguous data. E.g., consider the case of super-resolution: a low-resolution observation that serves as input typically has an infinite number of possible high-resolution solutions that would fit the low-res input.

If a data set contains multiple such cases, and we employ supervised training, the network will reliably learn the mean. This averaged solution usually is one that is clearly undesirable, and unlike any of the individual solutions from which it was computed. This is the *multi-modality* problem, i.e. different modes existing as valid equally valid solutions to a problem. For fluids, this can, e.g., happen when we're facing bifurcations, as discussed in *A Teaser Example*.

The following image shows a clear example of how well GANs can circumvent this problem:

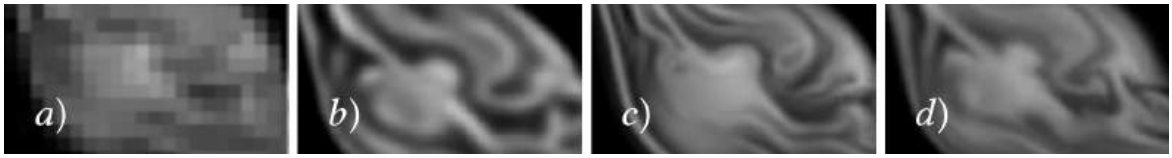


Fig. 46.2: A super-resolution example: a) input, b) supervised result, c) GAN result, d) high-resolution reference.

46.6 Spatio-temporal super-resolution

Naturally, the GAN approach is not limited to spatial resolutions. Previous work has demonstrated that the concept of learned self-supervision extends to space-time solutions, e.g., in the context of super-resolution for fluid simulations [XFCT18].

The following example compares the time derivatives of different solutions:

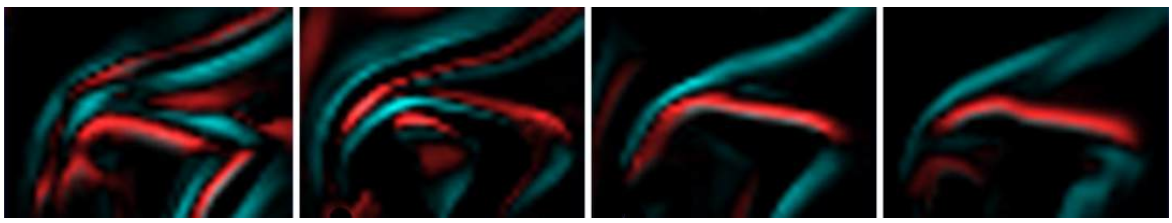


Fig. 46.3: From left to right, time derivatives for: a spatial GAN (i.e. not time aware), a temporally supervised learning, a spatio-temporal GAN, and a reference solution.

As can be seen, the GAN trained with spatio-temporal self-supervision (second from right) closely matches the reference solution on the far right. In this case the discriminator receives reference solutions over time (in the form of triplets), such that it can learn to judge whether the temporal evolution of a generated solution matches that of the reference.

46.7 Physical generative models

As a last example, GANs were also shown to be able to accurately capture solution manifolds of PDEs parametrized by physical parameters [CTS+21]. In this work, Navier-Stokes solutions parametrized by varying buoyancies, vorticity content, boundary conditions, and obstacle geometries were learned by an NN.

This is a highly challenging solution manifold, and requires an extended "cyclic" GAN approach that pushes the discriminator to take all the physical parameters under consideration into account. Interestingly, the generator learns to produce realistic and accurate solutions despite being trained purely on data, i.e. without explicit help in the form of a differentiable physics solver setup. The figure below shows a range of example outputs of a physically-parametrized GAN [CTS+21].

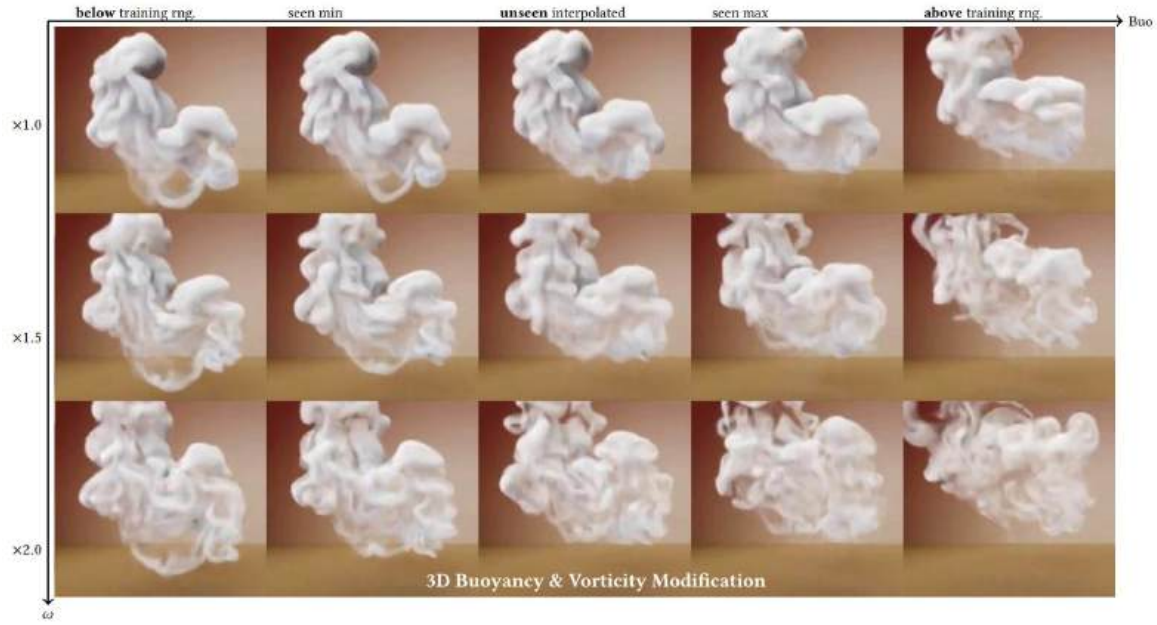


Fig. 46.4: The network can successfully extrapolate to buoyancy settings beyond the range of values seen at training time.

46.8 Discussion

GANs are a powerful learning tool. Note that the discriminator D is really "just" a learned loss function: we can completely discard it at inference time, once the generator is fully trained. Hence it's also not overly crucial how much resources it needs.

However, despite being very powerful tools, it is (given the current state-of-the-art) questionable whether GANs make sense when we have access to a reasonable PDE model. If we can discretize the model equations and include them with a differentiable physics (DP) training (cf. *Introduction to Differentiable Physics*), this will most likely give better results than

trying to approximate the PDE model with a discriminator. The DP training can yield similar benefits to GAN training: it yields a local gradient via the discretized simulator, and in this way prevents undesirable averaging across samples. Hence, combinations of DP training and GANs are also bound to not perform better than either of them in isolation.

That being said, GANs can nonetheless be attractive in situations where DP training is infeasible due to black-box solvers without gradients

46.9 Source code

Due to the complexity of the training setup, we only refer to external open source implementations for practical experiments with physical GANs. E.g., the spatio-temporal GAN from [XFCT18] is available at https://github.com/thunil/tempoGAN.

It also includes several extensions for stabilization, such as L^1 regularization, and generator-discriminator balancing.

Part IX

End Matter

CHAPTER

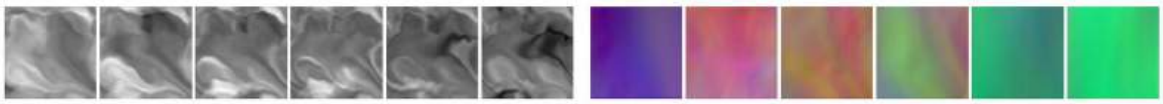
FORTYSEVEN

OUTLOOK

Despite the in-depth discussions and diverse examples we've explored, we've really only begun to tap into the vast potential of physics-based deep learning. The techniques covered in the previous chapters aren't just useful -— they have the power to reshape computational methods for decades to come. As we've seen in the code examples, there's no magic at play; rather, deep learning provides an incredibly powerful new tool to work with complex, non-linear functions.

Crucially, deep learning doesn't replace traditional numerical methods. Instead, it enhances them. Together, they form a groundbreaking synergy, with a huge potential to unlock new frontiers in simulation and modeling. One aspect we haven't yet touched upon is perhaps the most profound: at its core, our ultimate goal is to deepen human understanding of the world. The notion of neural networks as impenetrable "black boxes" is outdated. Instead, they should be seen as just another numerical tool—one that is as interpretable as traditional simulations when used correctly.

Looking ahead, one of the most exciting challenges is to refine our ability to analyze learned networks. By distilling the patterns and structures these networks uncover, we move closer to extracting fundamental, human-readable insights from their solution manifolds. The future of differentiable simulation isn't just about better predictions -— it's about revealing the hidden order of the physical world in ways we've never imagined.



47.1 Some specific directions

Beyond this long-term vision, there are plenty of exciting and immediate next steps. While our deep dives into Burgers' equation and Navier-Stokes solvers have tackled non-trivial challenges, they represent just a fraction of the landscape of PDE models and operators that these techniques can improve. Here are just a few promising directions from other fields:

- Chemical Reaction PDEs often exhibit intricate behaviors due to multi-species interactions. A particularly exciting avenue is training models that can rapidly predict experimental or industrial processes and dynamically adjust control parameters to stabilize them to enable real-time, intelligent control.
- Plasma Simulations share similarities with vorticity-based fluid formulations but introduce additional complexities due to electric and magnetic interactions. This makes them a prime candidate for deep learning methods, especially for plasma fusion experiments and energy generators, where differentiable physics could be a game-changer.
- Weather and Climate Modeling remain among the most critical scientific challenges for humanity. These highly complex, multi-scale systems involve fluid flows intertwined with countless environmental factors. Leveraging deep learning to enhance numerical simulations in this space holds immense potential. Not just for more accurate forecasts, but for unlocking deeper insights into the dynamics of our planet.

47.2 Closing remarks

These are just a few examples, but they illustrate the incredible breadth of opportunities where differentiable physics and deep learning can make an impact. There's lots of exciting research work left to do - the next years and decades definitely won't be boring. □ □

CHAPTER

FORTYEIGHT

NOTATION AND ABBREVIATIONS

48.1 Math notation:

Symbol	Meaning
A	matrix
η	learning rate or step size
Γ	boundary of computational domain Ω
f^*	generic function to be approximated, typically unknown
f	approximate version of f^*
Ω	computational domain
$\mathcal{P}^*$	continuous/ideal physical model
$\mathcal{P}$	discretized physical model, PDE
θ	neural network params
t	time dimension
$\mathbf{u}$	vector-valued velocity
x	neural network input or spatial coordinate
y	neural network output
y^*	learning targets: ground truth, reference or observation data

48.2 Summary of the most important abbreviations:

Abbreviation	Meaning
AI	Mysterious buzzword popping up in all kinds of places these days
BNN	Bayesian neural network
CNN	Convolutional neural network (specific NN architecure)
DDPM	Denoising diffusion probabilistic models (diffusion modeling variant)
DL	Deep Learning
FM	Flow matching (diffusion modeling variant)
FNO	Fourier neural operator (specific NN architecure)
GD	(steepest) Gradient Descent
MLP	Multi-Layer Perceptron, a neural network with fully connected layers
NN	Neural network (a generic one, in contrast to, e.g., a CNN or MLP)
PDE	Partial Differential Equation
PBDL	Physics-Based Deep Learning
SGD	Stochastic Gradient Descent

BIBLIOGRAPHY

[AAC+19] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, and others. Solving rubik's cube with a robot hand. *arXiv:1910.07113*, 2019.

[CCHT21] Li-Wei Chen, Berkay A Cakal, Xiangyu Hu, and Nils Thuerey. Numerical investigation of minimum drag profiles in laminar flow using deep learning surrogates. *Journal of Fluid Mechanics*, 2021. URL: https://ge.in.tum.de/publications/2020-chen-dl-surrogates/.

[CT22] Li-Wei Chen and Nils Thuerey. Towards high-accuracy deep learning inference of compressible turbulent flows over aerofoils. In *Computers and Fluids*. 2022. URL: https://ge.in.tum.de/publications/.

[CRBD19] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *arXiv:1806.07366*, 2019.

[CTS+21] Mengyu Chu, Nils Thuerey, Hans-Peter Seidel, Christian Theobalt, and Rhaleb Zayer. Learning Meaningful Controls for Fluids. *ACM Trans. Graph.*, 2021. URL: https://people.mpi-inf.mpg.de/~mchu/gvv-den2vel/den2vel.html.

[CKM+23] Hyungjin Chung, Jeongsol Kim, Michael Mccann, Marc Klasky, and Jong Chul Ye. Diffusion posterior sampling for general noisy inverse problems. In *International Conference on Learning Representations*. 2023.

[Gol90] H Goldstine. *A history of scientific computing*. ACM, 1990.

[HKT19] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. In *International Conference on Learning Representations*. 2019. URL: https://ge.in.tum.de/publications/2020-iclr-holl/.

[HKT22] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Scale-invariant physics for deep learning. *Advances in Neural Information Processing Systems*, 35:5390–5403, 2022. URL: https://arxiv.org/abs/2109.15048.

[HT23] Benjamin Holzschuh and Nils Thuerey. Solving inverse physics problems with score matching. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[HVT23] Benjamin Holzschuh, Simona Vegetti, and Nils Thuerey. Solving inverse physics problems with score matching. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[KAT+19] Byungsoo Kim, Vinicius C Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep Fluids: A Generative Network for Parameterized Fluid Simulations. *Comp. Grap. Forum*, 38(2):12, 2019. URL: http://www.byungsoo.me/project/deep-fluids/.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: a method for stochastic optimization. *arXiv:1412.6980*, 2014.

[KPB20] Ivan Kobyzev, Simon JD Prince, and Marcus A Brubaker. Normalizing flows: an introduction and review of current methods. *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[KSA+21] Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 2021.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 2012.

[LKA+21] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *ICLR*, 2021.

[LPT25] Mario Lino, Tobias Pfaff, and Nils Thuerey. Learning distributions of complex fluid simulations with diffusion graph networks. In *International Conference on Learning Representations*. 2025.

[LCBH+22] Yaron Lipman, Ricky TQ Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv:2210.02747*, 2022.

[LCT22] Bjoern List, Liwei Chen, and Nils Thuerey. Learned turbulence modelling with differentiable fluid solvers. In *Journal of Fluid Mechanics (929/25)*. 2022. URL: https://ge.in.tum.de/publications/.

[LGL22] Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow straight and fast: learning to generate and transfer data with rectified flow. *arXiv:2209.03003*, 2022.

[MLA+19] Rajesh Maingi, Arnold Lumsdaine, Jean Paul Allain, Luis Chacon, SA Gourlay, and others. Summary of the fesac transformative enabling capabilities panel report. *Fusion Science and Technology*, 75(3):167–177, 2019.

[OMalleyBK+16] Peter JJ O'Malley, Ryan Babbush, Ian D Kivlichan, Jonathan Romero, Jarrod R McClean, Rami Barends, Julian Kelly, Pedram Roushan, Andrew Tranter, Nan Ding, and others. Scalable quantum simulation of molecular energies. *Physical Review X*, 6(3):031007, 2016.

[PUKT22] Lukas Prantl, Benjamin Ummenhofer, Vladlen Koltun, and Nils Thuerey. Guaranteed conservation of momentum for learning particle-based fluid dynamics. *Advances in Neural Information Processing Systems*, 2022.

[Qur19] Mohammed Al Quraishi. Alphafold at casp13. *Bioinformatics*, 35(22):4862–4865, 2019.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[RPK19] Maziar Raissi, Paris Perdikaris, and George Karniadakis. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. 2015.

[SGGP+20] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, 8459–8468. 2020.

[SHT22] Patrick Schnell, Philipp Holl, and Nils Thuerey. Half-inverse gradients for physical deep learning. In *ICLR*. 2022. URL: https://github.com/tum-pbs/half-inverse-gradients.

[SML+15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv:1506.02438*, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, and others. Mastering the game of Go without human knowledge. *Nature*, 2017.

[Sto14] Thomas Stocker. *Climate change 2013: the physical science basis: Working Group I contribution to the Fifth assessment report of the Intergovernmental Panel on Climate Change*. Cambridge university press, 2014.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[TWPH20] Nils Thuerey, Konstantin Weissenow, Lukas Prantl, and Xiangyu Hu. Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows. *AIAA Journal*, 58(1):25–36, 2020. URL: https://ge.in.tum.de/publications/2018-deep-flow-pred/.

[TSSP17] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of Machine Learning Research*, 3424–3433. 2017.

[UBH+20] Kiwon Um, Robert Brand, Philipp Holl, Raymond Fei, and Nils Thuerey. Solver-in-the-loop: learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 2020. URL: https://ge.in.tum.de/publications/2020-um-solver-in-the-loop/.

[UPTK19] Benjamin Ummenhofer, Lukas Prantl, Nils Thuerey, and Vladlen Koltun. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*. 2019. URL: https://ge.in.tum.de/publications/2020-ummenhofer-iclr/.

[Vin11] Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural computation*, 23(7):1661–1674, 2011.

[WBT19] Steffen Wiewel, Moritz Becher, and Nils Thuerey. Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow. *Comp. Grap. Forum*, 38(2):12, 2019. URL: https://ge.in.tum.de/publications/latent-space-physics/.

[WKA+20] Steffen Wiewel, Byungsoo Kim, Vinicius C Azevedo, Barbara Solenthaler, and Nils Thuerey. Latent space subdivision: stable and controllable time predictions for fluid flow. *Symposium on Computer Animation*, 2020. URL: https://ge.in.tum.de/publications/2020-lssubdiv-wiewel/.

[XFCT18] You Xie, Erik Franz, Mengyu Chu, and Nils Thuerey. tempoGAN: A Temporally Coherent, Volumetric GAN for Super-resolution Fluid Flow. *ACM Trans. Graph.*, 2018. URL: https://ge.in.tum.de/publications/tempogan/.

[YK15] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.