Deep Ray
Orazio Pinti
Assad A. Oberai

Deep Learning and Computational Physics

Springer

Deep Learning and Computational Physics

Deep Ray · Orazio Pinti · Assad A. Oberai

Deep Learning and Computational Physics

Springer

Deep Ray
Department of Mathematics
University of Maryland
College Park, MD, USA

Orazio Pinti
Pasteur Labs
Brooklyn, NY, USA

Assad A. Oberai
Department of Aerospace and Mechanical Engineering
USC Viterbi School of Engineering
University of Southern California
Los Angeles, CA, USA

ISBN 978-3-031-59344-4 ISBN 978-3-031-59345-1 (eBook)
https://doi.org/10.1007/978-3-031-59345-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Assad dedicates this book to his parents, Sudha and Har Krishan, and his wife, Vidya, for their unwavering love and support.
Deep dedicates this book to his grandfather Tarun Basu for the unwavering belief in Deep's journey.
Orazio dedicates this book to his parents, Paola ed Egidio, in recognition of their unconditional love and silent sacrifices.

Preface

These notes were compiled as lecture notes for a course developed and taught at the University of the Southern California. They should be accessible to a typical engineering graduate student with a strong background in Applied Mathematics.

The main objective of these notes is to introduce a student who is familiar with concepts in linear algebra and partial differential equations to select topics in deep learning. These lecture notes exploit the strong connections between deep learning algorithms and the more conventional techniques of computational physics to achieve two goals. First, they use concepts from computational physics to develop an understanding of deep learning algorithms. Not surprisingly, many concepts in deep learning can be connected to similar concepts in computational physics, and one can utilize this connection to better understand these algorithms. Second, several novel deep learning algorithms can be used to solve challenging problems in computational physics. Thus, they offer someone who is interested in modeling a physical phenomena with a complementary set of tools.

College Park, USA — Deep Ray
Brooklyn, USA — Orazio Pinti
Los Angeles, USA — Assad A. Oberai

Acknowledgements

The authors would like to acknowledge Prof. Alvaro Cotinho and Dr. Dhruv Patel for their insightful feedback on an earlier version of this book, and Drs. Agnimitra Dasgupta, Saeed Moazami and Dr. Anirban Chandra for proof reading portions of this book.

Contents

1 Introduction 1
1.1 Computational Physics 1
1.2 Machine Learning 2
1.2.1 Examples of ML 2
1.2.2 Types of ML Algorithms Based Tasks 3
1.3 Artificial Intelligence, Machine Learning and Deep Learning 4
1.4 Machine Learning and Computational Physics 5
1.5 Computational Exercise 6

2 Introduction to Deep Neural Networks 9
2.1 MLP Architecture 9
2.2 Activation Functions 11
2.2.1 Linear Activation 12
2.2.2 Rectified Linear Unit (ReLU) 12
2.2.3 Leaky ReLU 12
2.2.4 Logistic Function 13
2.2.5 Tanh 13
2.2.6 Sine 13
2.3 Expressivity of a Network 14
2.3.1 Universal Approximation Results 16
2.4 Training, Validation and Testing of Neural Networks 17
2.5 Overfitting and How to Avoid It 19
2.5.1 Regularization 20
2.6 Gradient Descent 22
2.6.1 Convergence 23
2.7 Some Advanced Optimization Algorithms 24
2.7.1 Momentum Methods 25
2.7.2 Adam 26
2.7.3 Stochastic Optimization 26
2.8 Calculating Gradients Using Back-Propagation 29
2.9 Regression Versus Classification 31

2.10 Computational Exercise . . . 34
2.10.1 Expressivity of Deep Neural Networks . . . 34
2.10.2 Training an MLP for a Regression Problem . . . 36

3 Residual Neural Networks . . . 41
3.1 Vanishing Gradients in Deep Networks . . . 41
3.2 ResNets . . . 42
3.3 Connections with ODEs . . . 45
3.4 Neural ODEs . . . 46

4 Convolutional Neural Networks . . . 49
4.1 Functions and Images . . . 49
4.2 Convolutions of Functions . . . 50
4.2.1 Example 1 . . . 50
4.2.2 Example 2 . . . 52
4.3 Discrete Convolutions . . . 53
4.4 Connection to Finite Difference Approximations . . . 54
4.5 Convolution Layers . . . 55
4.5.1 Average and Max Pooling . . . 57
4.5.2 Convolution for Inputs with Multiple Channels . . . 57
4.6 Convolution Neural Network (CNN) . . . 58
4.7 Transpose Convolution Layers . . . 61
4.8 UpSampling . . . 63
4.9 Image-to-Image Transformations . . . 64
4.10 Computational Exercise: Convolutional Neural Networks (CNNs) . . . 65

5 Solving PDEs with Neural Networks . . . 71
5.1 Finite Difference Method . . . 72
5.2 Spectral Collocation Method . . . 74
5.3 Physics-Informed Neural Networks (PINNs) . . . 77
5.4 Extending PINNs to a More General PDE . . . 80
5.5 Error Analysis for PINNs . . . 82
5.6 Data Assimilation Using PINNs . . . 83
5.7 Some Existing PINN Formulations . . . 84
5.8 Computational Exercise: Physics Informed Neural Networks (PINNs) . . . 85

6 Operator Networks . . . 87
6.1 Parametrized PDEs . . . 88
6.2 Operators . . . 88
6.3 Deep Operator Network (DeepONet) Architecture . . . 90
6.3.1 Training DeepONets . . . 92
6.3.2 Error Analysis for DeepONets . . . 93
6.4 Physics-Informed DeepONets . . . 94
6.5 DeepONets and Their Applications . . . 95

6.6 Fourier Neural Operator (FNO) 96
6.6.1 Discretization of the Fourier Neural Operator 99
6.6.2 The Use of Fourier Transforms 100
6.7 Variationally Mimetic Operator Network (VarMiON) 102
6.7.1 Background 102
6.7.2 VarMiON Architecture 105
6.7.3 Training the VarMiON 106
6.7.4 Error Estimates of VarMiON Approximation 107
6.8 Mesh Graph Networks 108
6.8.1 Background 109
6.8.2 Architecture of MGNs 111
6.8.3 Training MGNs 114
6.9 Computational Exercise: Deep Operator Networks (DeepONets) 117

7 Generative Deep Learning 121
7.1 Generative Algorithms 121
7.2 Introductory Concepts in Probability 122
7.2.1 Random Variables 123
7.2.2 Cumulative Distribution Function 123
7.2.3 Probability Density Function 125
7.2.4 Examples of Important Random Variables 126
7.2.5 Expectation and Variance of RVs 128
7.2.6 Random Vectors 129
7.2.7 Joint Probability Density Function 130
7.2.8 Examples of Important Random Vectors 130
7.2.9 Expectation and Covariance of Random Vectors 131
7.2.10 Marginal and Conditional Probability Density Functions 132
7.3 Pure Generative Problem 134
7.3.1 GANs 134
7.3.2 Score-Based Diffusion Models 138
7.4 Conditional Generative Algorithms 143
7.4.1 Conditional GANs 143
7.4.2 Conditional Diffusion Models 145

References 147

About the Authors

Deep Ray is an Assistant Professor of Mathematics at the University of Maryland. He earned his Bachelor of Mathematics from University of Delhi, followed by a Masters and PhD in Mathematics from Tata Institute of Fundamental Research - Center for Applicable Mathematics. He has held research positions at ETH Zurich, EPFL, Rice University and University of Southern California. Deep's research lies at the interface of conventional numerical analysis and machine learning. He focuses on identifying computational bottlenecks in existing numerical algorithms and resolving them by the careful integration of machine learning tools. He has used such techniques to design efficient shock-capturing methods, build deep learning-based surrogate models to solve partial differential equations, develop differentiable models for constrained optimization, and solve Bayesian inference problems arising in real-world applications.

Orazio Pinti is a Research Scientist at Pasteur Labs, working in the field of scientific machine learning and computational physics. He holds a BSc and MSc from the Polytechnic University of Turin, and a PhD from the University of Southern California, all in Aerospace Engineering. His interests include applied mathematics, machine learning, and computational science, with a focus on reduced-order and multi-fidelity modeling.

Assad A. Oberai is the Hughes Professor of Aerospace and Mechanical Engineering in the Viterbi School of Engineering. He earned a Bachelor of Engineering degree from Osmania University, an MS from the University of Colorado, and a PhD from Stanford University all in Mechanical Engineering. He has held academic appointments at Boston University, Rensselaer Polytechnic Institute, and the University of Southern California. Assad leads a group that designs, implements, and applies data- and physics-based models and algorithms to solve problems in engineering and science. Problems such as better detection, diagnosis, and care of diseases like cancer, understanding the role of mechanics and physics in medicine and biology, modeling the evolution of multi-physics and multiscale systems, and reduced-order models for aerospace and mechanical systems. Assad is a Fellow of the American

Academy of Mechanics, American Society of Mechanical Engineers, the American Institute of Medical and Biological Engineering, and the United States Association of Computational Mechanics.

Chapter 1
Introduction

The theory and algorithms for statistical learning and data-driven algorithms have been around since the early 19th century. But the first hints of modern *machine learning* can be traced back to the 1943 work of McCulloch and Pitts [63] who proposed the first model of an *artificial neuron* loosely based on the functioning of a biological neuron in vertebrates. Arthur Samuel is popularly credited to have coined the term "machine learning" in 1959, when he was at IBM performing research on teaching a computer to play checkers [93]. Machine learning has been very successful in applications such as computer vision, speech recognition and natural language processing. But the last few years have also witnessed the emergence of machine learning (in particular deep learning) algorithms to solve physics-driven problems, such as approximating solutions to partial differential equations and inverse problems.

This course deals with topics that lie at the interface of *computational physics* and *machine learning*. Before we can appreciate the need to combine both these important concepts, we need to understand what each of them mean on their own.

1.1 Computational Physics

Computational physics plays a fundamental role in solving many problems in fields of science and engineering. To gain an understanding of this concept, we briefly outline the key steps involved in solving a physical problem:

1. Consider a physical phenomena and collect measurements of some observable of interest. For example, the measurements of the water height and wave direction obtain from ocean buoys, when studying oceanic waves.
2. Based on the observations, postulate a *physical law*. For instance, you observe that the total mass of fluid in a closed-system is conserved for all time.

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_1

3. Write down a mathematical description of the law. This could make use of ordinary differential equations (ODEs), partial differential equations (PDEs), integral equations, etc.
4. Once the mathematical model is framed, solve for the solution of the system. There are two ways to obtain this:
   (a) In certain situations an exact analytical form of the solution can be obtained. For instance one could solve ODEs/PDEs using separation of variables, Laplace transforms, Fourier transforms or integrating factors.
   (b) In most scenarios, exact expressions of the solution cannot be obtained and must be suitable approximated using a numerical algorithm. For instance, one could use forward or backward Euler, mid-point rule, or Runge-Kutta schemes for solving systems of ODEs [14]; or one could use finite difference/volume/element methods for solving PDEs [3].
5. Once the algorithm to evaluate the solution (exactly or approximately) is designed, use it to validate the mathematical model, i.e., see if the predictions agree with the data collected.

   All these steps broadly describe what computational physics entails.

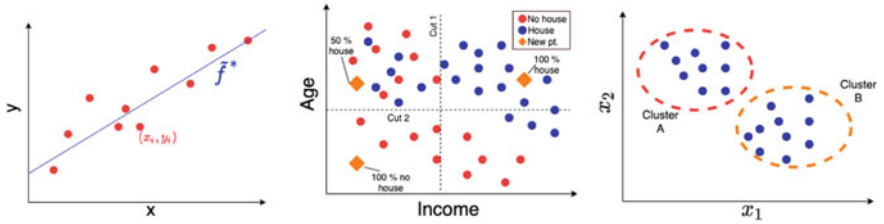
1.2 Machine Learning

Unlike computational physics, machine learning (ML) does not require the postulation of a physical law. The general steps involved are:

1. Collect data by observing the physical phenomena, by real-time measurements of some observable or by using a numerical solver approximating the phenomenon.
2. Train a suitable algorithm using the collected data, with the aim of discovering a pattern or relation between the various samples. See Sect. 1.2.1 for some concrete examples.
3. Once trained, use the ML algorithm to make future predictions, and validate it with additional collected data.

1.2.1 Examples of ML

1. **Regression algorithms**: Given the set of pairwise data $\{(x_i, y_i) : 1 \leq i \leq N\}$ which corresponds to some unknown function $y = f(x)$, fit a polynomial (or any other basis) to this data set in order to approximate f. For instance, find the coefficients a, b of the linear fit $\tilde{f}(x; a, b) = ax + b$ to minimize the error



(a) Linear regression (b) Decision tree (c) Clustering

Fig. 1.1 Examples of ML

$$\Pi(a, b) = \sum_{i=1}^{N} |y_i - \tilde{f}(x_i)|^2.$$

If $(a^*, b^*) = \arg\min_{a,b} \Pi(a, b)$, then we can consider $\tilde{f}^*(x) := \tilde{f}(x; a^*, b^*)$ to be the approximation of $f(x)$ (see Fig. 1.1a).

2. **Decision trees**: We are given a dataset from a sample population, containing the features: age and income. Furthermore, the data is divided into two groups; an individual in Group A owns a house while an individual in Group B does not. Then, given the features of a new data point, we would like to predict the probability of this new individual owning a house. Decision trees can be used to solve this classification problem. The way a typical decision tree works is by making cuts that maximize the group-based separation for the samples in the dataset (see Fig. 1.1b). Then, based on these cuts, the algorithm determines the probability of belonging to a particular class/group for a new point.
3. **Clustering algorithms**: Given a set of data with a number of features per sample, find cluster/patterns in the data (see Fig. 1.1c).

1.2.2 Types of ML Algorithms Based Tasks

Broadly speaking, there are four types of ML algorithms:

1. **Supervised learning**: Given the data $\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : 1 \leq i \leq N\}$, predict $\hat{\boldsymbol{y}}$ for some new $\hat{\boldsymbol{x}}$, such that $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) \notin \mathcal{S}$. For instance, given a set of images and image labels (e.g. dog, cat, cow, etc.), train a classification ML algorithm to learn the relation between images and labels, and use it to predict the label of a new image.
2. **Unsupervised learning**: Given the data $\mathcal{S} = \{\boldsymbol{x}_i \in \Omega_x : 1 \leq i \leq N\}$, find a relation among different regions of Ω_x. For instance, find clusters in the dataset, or find an expression for the probability distribution $p_x(\boldsymbol{x})$ governing the spread of this data and generate new samples from it.
3. **Semi-supervised learning**: This family of methods falls between the supervised and unsupervised learning families. They typically make use of a combination

of labelled and unlabelled data for training. For example, we are given 10,000 images that are unlabeled and only 50 images that are labeled. Can we use this dataset to develop an image classification algorithm?

4. **Re-inforcement learning**: The methods belonging to this family learn based on rewards or penalties for decisions taken. Thus, a suitable path/policy is learned to maximize the reward. This strategy can be used to train algorithms to play chess or Go.

In this course, we will primarily focus on the first two types of ML algorithms.

1.3 Artificial Intelligence, Machine Learning and Deep Learning

At times, the terms Artificial Intelligence (AI), ML and Deep Learning (DL) are used interchangeably. In reality, these are three related but different concepts. This can be understood by looking at the Venn diagram in Fig. 1.2.

AI refers to a system with human-like intelligence. While ML is a key component of an AI system, there are other ingredients involved. A self-driving car is a prototypical example of AI. Let's take a closer look at the design of such a system (see Fig. 1.3). A car is mounted with a camera which takes live images/video of the road ahead. These frames are then passed to an ML algorithm which performs a semantic segmentation, i.e., segments out different regions of the frame and classifies the type of object (car, tree, road, sky, etc.) in each segment. Once this segmentation is done, it is passed to a *decision system* that decides what the next action of the car should be based on this segmented image. This information then passes through a control module that actually controls the mechanical actions of the car. This entire process mimics what a real driver would do, and is thus artificial intelligence.

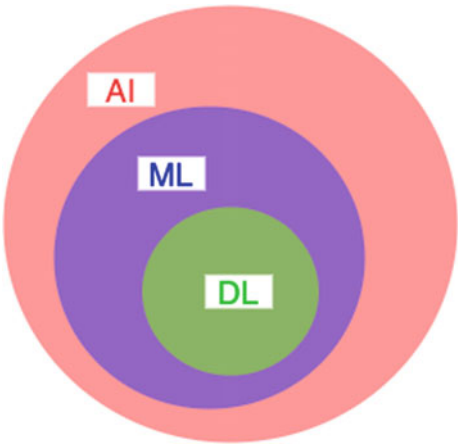


Fig. 1.2 The relation between AI, ML and DL

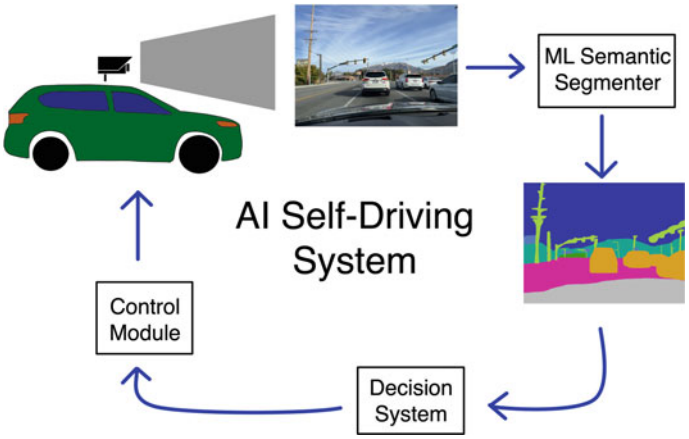


Fig. 1.3 Schematic of AI system for a self-driving car

On the other hand, machine learning (ML) are the components of this system that are trained using data. That is they learn through data. In the example above, the Semantic Segmenter is one such system. There are many ML algorithms that can perform this task using data, and we will learn about some of these in this course. The Decision System could also be an ML component—where the appropriate decision to be made is learnt from prior data. However, it could also be a non-ML rule-based expert system.

DL is a subset of ML algorithms. The simplest form of a DL architecture, known as a feed-forward network, comprises a number of layers of non-linear transformations. This architecture is loosely motivated by how signals are transmitted by the central nervous in living organisms. We will study the DL architecture in greater detail in Chap. 2.

1.4 Machine Learning and Computational Physics

Now that we have a better understanding of computational physics and ML, the next obvious question would be "why do we need to look at a combination of the two?" We list down a few motivations below:

- For complex patterns of "physical" data, ML provides an alternate route to representing mathematical laws. Consider a physical process that contains two important components. Of these, one is well understood and has a trusted mathematical model, and the other is poorly understood and does not have a mathematical description. In this scenario, one may use computational physics for the first component and ML for the second. A concrete example of this would be a system governed by conservation of energy and a complex constitutive model. For the

former we may have a well understood mathematical model, while for the latter we may have to rely on ML to develope a model.
- ML in general is very data hungry. But the knowledge of physics can help restrict the manifold on which the input and solution/predictions lie. With such constraints, we can reduce the amount of data required to train the ML algorithm.
- Tools for analyzing computational physics (functional analysis, numerical analysis, notions of convergence to exact solutions, probabilistic frameworks) carry over to ML. Applying these tools to ML helps us better understand and design better ML algorithms.

We briefly summarize the various topics that will be covered in this course:

- Deep Neural Networks (MLPs) and their convergence.
- Stochastic gradient descent and how it is related to ODEs.
- Resnets and their connections with non-linear ODEs (Neural ODEs).
- Convolutional neural networks and their connection to PDEs.
- Deep Learning algorithms for solving PDEs.
- Deep Learning algorithms for approximating operators.
- Generative algorithms and their connection to computational physics.
- Generative algorithms for solving probabilistic inverse problems.

1.5 Computational Exercise

In order to effectively use the various ML algorithms discussed in this course, and solve the computational exercises, a good grasp on Python programming and PyTorch is required. Listed below are various resources and tutorials about the necessary programming concepts.

1. **Python basics, NumPy and plotting**: A good resource for those who have never used Python before, or even for those who are familiar with Python and want to freshen up their programming knowledge, is https://www.w3schools.com/python/. This tutorial covers important Python modules such as `NumPy`, `Pandas` and `SciPy`, describe how to generate plots using `Matplotlib`, and how to read/write files.
2. **PyTorch**: The codes used in this course will be written using PyTorch, which is a machine learning framework originally developed by Meta AI. You can install PyTorch locally on your machines by following the instructions given here https://pytorch.org/get-started/locally/. Various tutorials on using PyTorch can be found here https://pytorch.org/tutorials/.
3. **Google Colab**: If you do not wish do install PyTorch locally, or do not have the appropriate hardware to train deep learning models, you could also use Google Colab https://colab.research.google.com. Colab is essentially a combination of Jupyter notebook and Google Drive, and only requires you to have a Google

account. The attractive thing about Colab is that it comes preinstalled with many useful packages (like NumPy, PyTorch etc.), so everyone can use it without worrying about installing the correct versions of libraries and dependencies. Furthermore, it runs entirely on the cloud and can be launched and used directly through the web browser. It also gives free access to powerful GPUs and TPUs.

Chapter 2
Introduction to Deep Neural Networks

In this chapter, we introduce the simplest deep learning architecture used in practice, which is known as the *multilayer perceptron* (MLP). We will discuss its various components, its ability to approximate functions with different regularity (*universal approximation results*), and the various training paradigms to learn the various parameters (and hyperparameters) without overfitting the training dataset.

2.1 MLP Architecture

Let us define our goal as the approximation of a function $\boldsymbol{f} : \boldsymbol{x} \in \mathbb{R}^d \mapsto \boldsymbol{y} \in \mathbb{R}^D$ using an MLP, which we denote as $\boldsymbol{\mathcal{F}}$.The computing units of an MLP, called *artificial neurons*, are stacked in a number of consecutive layers. The zeroth layer of $\boldsymbol{\mathcal{F}}$ is called the *source layer*, which is not a computing layer but is only responsible for providing an input (of dimension d) to the network. The last layer of $\boldsymbol{\mathcal{F}}$ is known as the *output layer*, which outputs the network's prediction (of dimension D). Every other layer in between is known as a *hidden layer*. The number of neurons in a layer defines the *width* of that layer. A schematic of an MLP with 2 hidden layers is shown in Fig. 2.1.

To understand the operations occurring inside an MLP, let us define some notation. We consider a network with L hidden layers (thus $L + 2$ layers in total), with the width of layer l denoted as H_l for $l = 0, 1, \ldots, L + 1$. Note that for consistency with the target function $\boldsymbol{f}$ that we are trying to approximate, we must have $H_0 = d$ and $H_{L+1} = D$. Let us denote the output vector for l-th layer by $\boldsymbol{x}^{(l)} \in \mathbb{R}^{H_l}$, which will serve as the input to the next layer. We set $\boldsymbol{x}^{(0)} = \boldsymbol{x} \in \mathbb{R}^d$ which will be the input signal provided by the source layer. In each layer l, $1 \leq l \leq L + 1$, the i-th neuron performs an affine transformation on that layer input $\boldsymbol{x}^{(l-1)}$ followed by a non-linear transformation

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_2

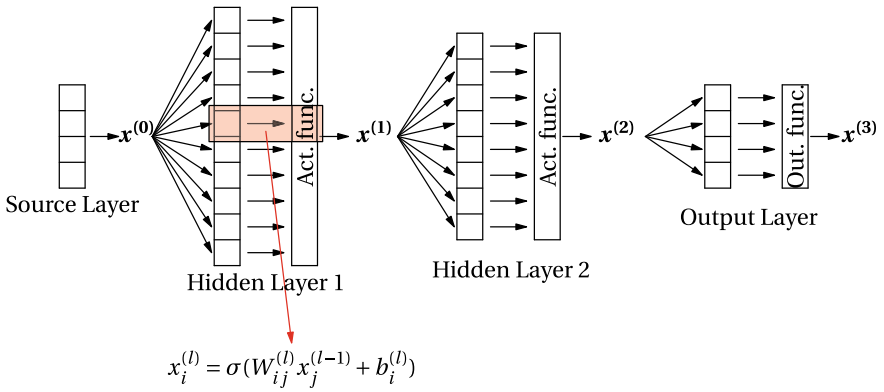


Fig. 2.1 MLP with 2 hidden layers, with a depiction of the transformation occurring inside each neuron

$$x_i^{(l)} = \sigma\Big(\underbrace{W_{ij}^{(l)} x_j^{(l-1)}}_{\text{Einstein sum}} + b_i^{(l)}\Big), \quad 1 \le i \le H_l, \quad 1 \le j \le H_{l-1} \tag{2.1}$$

where $W_{ij}^{(l)}$ and $b_i^{(l)}$ are respectively known as the weights and bias associated with i-th neuron of layer l. The function $\sigma(.)$ is known as the *activation function*, and plays a pivotal role in helping the network to represent non-linear complex functions. If we set $\boldsymbol{W}^{(l)} \in \mathbb{R}^{H_{l-1} \times H_l}$ to be the weight matrix for layer l and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{H_l}$ to be the bias vector for layer l, then we can re-write the action of the whole layer as

$$\boldsymbol{x}^{(l)} = \sigma\left(\mathcal{A}^{(l)}(\boldsymbol{x}^{(l-1)})\right), \quad \mathcal{A}^{(l)}(\boldsymbol{x}^{(l-1)}) = \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{2.2}$$

where the activation function is applied component-wise. Thus, the action of the whole network $\boldsymbol{\mathcal{F}} : \mathbb{R}^d \mapsto \mathbb{R}^D$ can be mathematically seen as a composition of alternating affine transformations and component-wise activations

$$\boldsymbol{\mathcal{F}}(\boldsymbol{x}) = \mathcal{A}^{(L+1)} \circ \sigma \circ \mathcal{A}^{(L)} \circ \sigma \circ \mathcal{A}^{(L-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)}(\boldsymbol{x}). \tag{2.3}$$

We make a few remarks here:

1. For simplicity of the representation, we assume that the same activation function is used across all layers of the network. However, this is not a strict rule. In fact, there is recent evidence that suggests that alternating the activation function from layer to layer leads to better neural networks [119].
2. In the network formulation shown in (2.3), the output layer is purely affine. At times, there might be an output function O at the end of the output layer, which is typically used to reformulate the output into a suitable form. We will see examples of such functions later in this chapter.

3. We use the term *depth* of the network to denote the number of computing layers in the MLP, i.e. the number of hidden layers and the output layer, which would be $L + 1$ as per the notations used above.

The learnable parameters of the network are all the weights and biases, which we represent as

$$\boldsymbol{\theta} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}_{l=1}^{L+1} \in \mathbb{R}^{N_\theta}$$

where N_θ denotes the total number of parameters of the network. The network $\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta})$ represents a family of parameterized functions, where $\boldsymbol{\theta}$ needs to suitably chosen such that the network approximates the target function $f(\boldsymbol{x})$ at the input $\boldsymbol{x}$.

Question 2.1.1 *Prove that* $N_\theta = \sum_{l=1}^{L+1}(H_{l-1} + 1)H_l$.

2.2 Activation Functions

The activation function is perhaps the most important component of an MLP. A large number of activations are available in literature, each with its own advantages and disadvantages. Let us take a look at a few of these options (also see Fig. 2.2).

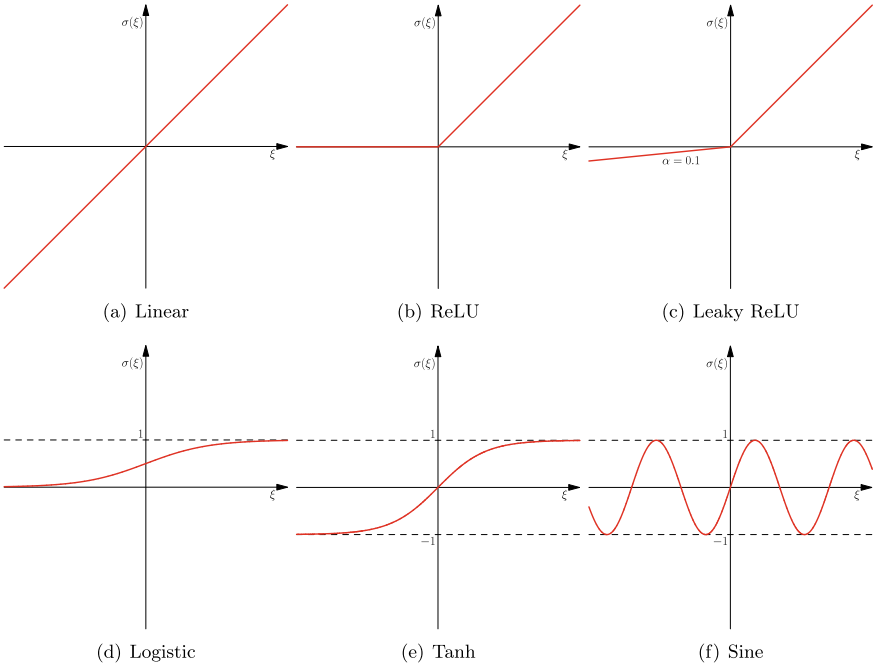


Fig. 2.2 Examples of activation functions

2.2.1 Linear Activation

The simplest activation corresponds to $\sigma(\xi) = \xi$. Some features of this function are

- The function is infinitely smooth, but all derivatives beyond the second derivative are zero.
- The range of the function is $(-\infty, \infty)$.
- Using the linear activation function (in all layers) will reduce the entire network to a single affine transformation of the input $\boldsymbol{x}$. In other words, the network will be nothing more that a linear approximation of the target function $\boldsymbol{f}$, which is not useful if $\boldsymbol{f}$ is non-linear.

2.2.2 Rectified Linear Unit (ReLU)

This function is piecewise linear and defined as

$$\sigma(\xi) = \max\{0, \xi\} = \begin{cases} \xi, & \text{if } \xi \geq 0 \\ 0, & \text{if } \xi < 0 \end{cases}. \tag{2.4}$$

This is one of the most popular activation functions used in practice. Some features of this function are:

- The function is continuous, while its derivative will be piecewise constant with a jump at $\xi = 0$. The second derivative will be a dirac function concentrated at $\xi = 0$. In other words, the higher-order derivatives (greater than 1) are not well-defined.
- The range of the function is $[0, \infty)$.

2.2.3 Leaky ReLU

The ReLU activation leads to a null output from a neuron if the affine transformation of the neuron is negative. This can lead to the phenomena of *dying neurons* [62] while training a neural network, where neurons drops out completely from the network and no longer contribute to the final prediction. To overcome this challenge, a leaky version ReLU was designed

$$\sigma(\xi; \alpha) = \begin{cases} \xi, & \text{if } \xi \geq 0 \\ \alpha\xi, & \text{if } \xi < 0 \end{cases} \tag{2.5}$$

where α becomes a network *hyper-parameter*. Some features of this function are:

- The derivatives of Leaky ReLU behave in the same way as those for ReLU. However, the first derivative (except at $\xi = 0$) is non-zero.
- The range of the function is $(-\infty, \infty)$.

2.2.4 Logistic Function

The Logistic or Sigmoid activation function is given by

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \tag{2.6}$$

and has the following properties

- The function is infinitely smooth and monotonic.
- The range of the function is (0, 1), i.e., the function is bounded. Such a function is useful in representing probabilities. It is also useful in representing output functions that are bounded.
- Since the derivative quickly decays to zero away from $\xi = 0$, this activation function can lead to slow convergence of the network training algorithm.

2.2.5 Tanh

The tanh function can be seen as a symmetric extension of the logistic function

$$\sigma(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}} \tag{2.7}$$

and has the following properties

- The function is infinitely smooth and monotonic.
- The range of the function is $(-1, 1)$, i.e., the function is bounded. Note that it maps zeros input to zero, while pushing positive (negative) inputs to $+1$ (-1).
- Similar to the logistic function, the derivative of tanh quickly decays to zero away from $\xi = 0$ and can thus lead to slow training.

2.2.6 Sine

Recently, the sine function, i.e., $\sigma(\xi) = \sin(\xi)$ has been proposed as an efficient activation function [100]. It has the best features of all the activation function discussed above:

- The function is infinitely smooth.
- The range of the function is $[-1, 1]$, i.e., the function is bounded.
- None of the derivatives of this function decay to zero.
- A network that uses this activation function can be viewed as a nonlinear generalization of the Fourier series.
- The use this activation function has lead to impressive results in implicit representation of functions [100].

Question 2.2.1 *Can you think of an MLP architecture with the sine activation function, which leads to an approximation very similar to a Fourier series expansion?*

Remark 2.2.1 There are many more activation functions that are used in a comprehensive list of activation functions can be found in [25].

2.3 Expressivity of a Network

Let us try to understand the effect of increasing N_θ in an MLP. In popular parlance this would be referred to as examining the effect of increasing *expressivity* of a network. To see this, let us consider a simple example using the ReLU activation function, i.e., $\sigma(\xi) = \max\{\xi, 0\}$. We set $d = D = 1$, $L = 1$ and the parameters

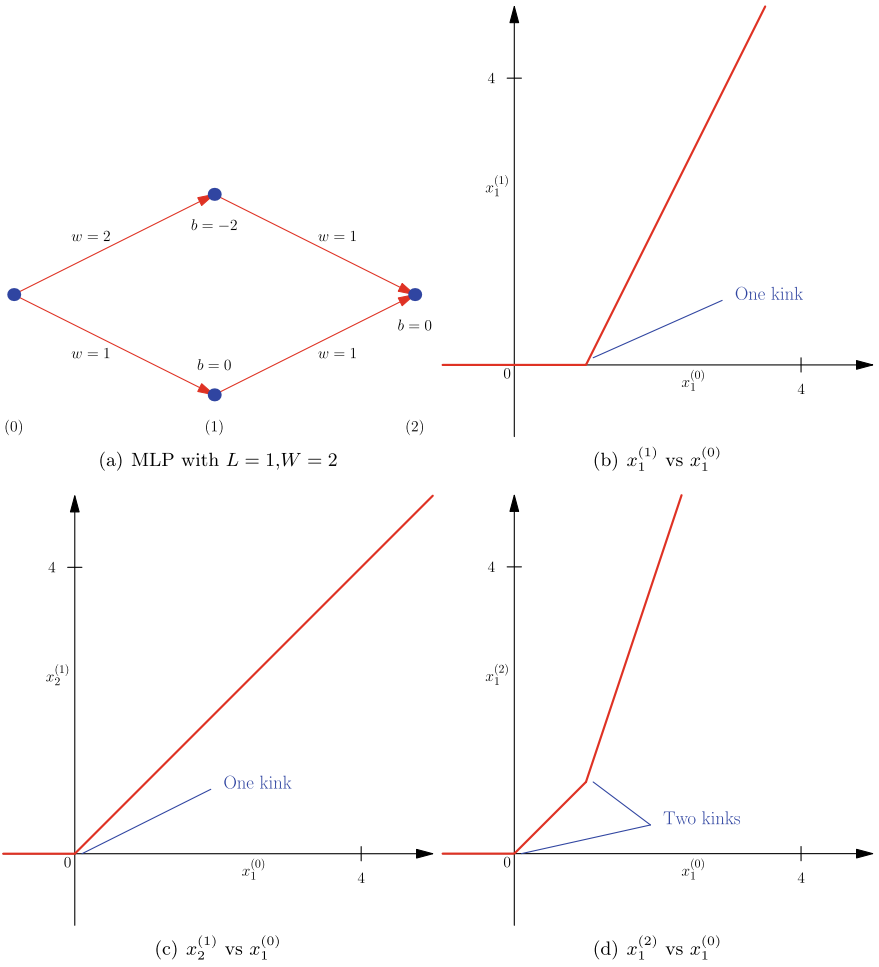
$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \boldsymbol{b}^{(1)} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}, \quad \boldsymbol{W}^{(2)} = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad b^{(2)} = 0.$$

as shown in Fig. 2.3a. Then the various layer outputs are

$$x_1^{(1)} = \max\{2x_1^{(0)} - 2, 0\}, \quad x_2^{(1)} = \max\{x_1^{(0)}, 0\}, \quad x_1^{(2)} = \max\{2x_1^{(0)} - 2, 0\} + \max\{x_1^{(0)}, 0\}.$$

Notice that while the the output $\boldsymbol{x}^{(1)}$ of the hidden layer (see Fig. 2.3b, c) have only one corner/kink, the final output ends up having two kinks (see Fig. 2.3d).

We generalize this formulation to a bigger network with L hidden layers each of width H. Then one can expect that $x_i^{(1)}$, $1 \leq i \leq H$ will have a single kink, with the location and angle of the kink depending on the weights and bias associated with each neuron of the hidden layer. The vector $\boldsymbol{x}^{(1)}$ is passed to the next hidden layer, where each neuron will combine the single kinks and give an output with possibly H kinks. Once again, the location and angles of the H kinks in the output from each neuron of the second hidden layer will be different. The location of the kinks will be different because each neuron is allowed a different bias, and therefore can induce a different shift. Continuing this argument, one can expect the number of kinks to increase as H, H^2, H^3 as it passes through the various hidden layers with width H. In general the total number of kinks can grow as H^L. Further, by appropriately selecting the weights and the biases in the network one could select the location of the



(a) MLP with $L = 1, W = 2$

(b) $x_1^{(1)}$ vs $x_1^{(0)}$

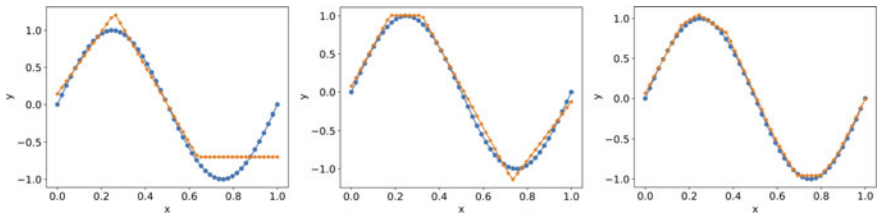
(c) $x_2^{(1)}$ vs $x_1^{(0)}$

(d) $x_1^{(2)}$ vs $x_1^{(0)}$

Fig. 2.3 Examples to understand the expressivity of neural networks

kinks, and the slope in the segment between the kinks. Thus one could approximate any continuous function by generating a piece-wise continuous approximation. In other words, the network has the ability to become more expressive as the depth (and width) of the network is increased.

As an illustration, in Fig. 2.4 we show the approximation of $f(x) = \sin(2\pi x)$ for $x \in [0, 1]$ using neural networks with varying depths (in terms of L) and a fixed hidden layer width $H = 3$, trained on 50 equi-spaced (in x) training samples. Note that as the depth increases, the network develops more kinks, allowing it to bend enough to better approximate the target function.



(a) $L = 2$ (b) $L = 3$ (c) $L = 4$

Fig. 2.4 Approximating $\sin(2\pi x)$ using an MLP with $H = 3$ for each hidden layer and varying depth. The blue ($-\bullet$) curves denote the true function/data while the orange ($-\star$) curves denote the predictions by the trained networks

2.3.1 Universal Approximation Results

To quantify the expressivity of networks in a mathematically rigorous manner, we look at some results about the approximation properties of MLPs. For these results, we assume $K \subset \mathbb{R}^d$ is a closed and bounded set. Note that this is a reasonable requirement since in most cases the input data is scaled to lie in some closed and bounded set. For example, $K = [0, 1]^d$, that is, a cube in d-dimensions.

Theorem 2.3.1 (Pinkus [83]) *Let $f : K \to \mathbb{R}$, i.e., $D = 1$, be a continuous function. Then given an $\epsilon > 0$, there exists an MLP with a single hidden layer ($L = 1$), arbitrary width H and a non-polynomial continuous activation σ such that*

$$\max_{\boldsymbol{x} \in K} |\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}) - f(\boldsymbol{x})| \leq \epsilon.$$

This theorem states that a network with a single hidden layer can approximate any continuous function to within any specific point-wise error if one is allowed to select an arbitrarily large value for its width.

Theorem 2.3.2 (Kidger [49]) *Let $\boldsymbol{f} : K \to \mathbb{R}^D$ be a continuous vector-valued function. Then given an $\epsilon > 0$, there exists an MLP with arbitrary number of hidden layers L, each having width $H \geq d + D + 2$, a continuous activation σ (with some additional mild conditions), such that*

$$\max_{\boldsymbol{x} \in K} \|\boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta}) - \boldsymbol{f}(\boldsymbol{x})\| \leq \epsilon.$$

Here $\|\cdot\|$ denotes the Euclidean 2-norm of a vector.

This theorem provides an analogous result for a network with a fixed and finite width, and an arbitrary large depth.

Theorem 2.3.3 (Yarotsky [119]) *Let $f : K \to \mathbb{R}$ be a function with two continuous derivates, i.e., $f \in C^2(K)$. Consider an MLP with ReLU activations and $H \geq 2d + 10$. Then there exists a network with this configuration such that the error converges as*

$$\max_{\boldsymbol{x} \in K} |\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}) - \boldsymbol{f}(\boldsymbol{x})| \leq C(N_\theta)^{-4}$$

where C is a constant depending on the number of network parameters.

The theorem above specifies how the approximation error for a network can vary as the number of parameters in the network, denoted by N_θ, changes.

Theoretical results like those mentioned above help demystify the "black-box" nature of neural network, and serve as useful practical guidelines when designing network architectures.

2.4 Training, Validation and Testing of Neural Networks

Now that we have a better understanding of the architecture of MLPs, we would like to discuss how the parameters of these networks are set to approximate some target function. We restrict our discussions to the framework of supervised learning.

Let us assume that we are given a dataset of pairwise samples $\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : 1 \leq i \leq N\}$ corresponding to a target function $\boldsymbol{f} : \boldsymbol{x} \mapsto \boldsymbol{y}$. We wish to approximate this function using the neural network

$$\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})$$

where $\boldsymbol{\theta}$ are the network parameters defined before, while $\boldsymbol{\Theta}$ corresponds to the *hyper-parameters* of the network . These include quantities such as the depth $L + 1$, width H, type of activation function σ, etc. The strategy to design a robust network involves three steps:

1. Find the optimal values of $\boldsymbol{\theta}$ (for a fixed $\boldsymbol{\Theta}$) in the *training* phase.
2. Find the optimal values of $\boldsymbol{\Theta}$ in the *validation* phase.
3. Test the performance of the network on unseen data in the *testing* phase.

To accomplish these three tasks, it is customary to split the dataset $\mathcal{S}$ into three distinct parts: a *training set* with N_{train} samples, a *validation set* with N_{val} samples and *test set* with N_{test} samples, with $N = N_{\text{train}} + N_{\text{val}} + N_{\text{test}}$. Typically, one uses around 60% of the samples as training samples, 20% as validation samples and the remaining 20% for testing.

Splitting the dataset is necessary as neural networks are heavily over-parameterized functions. The large number of degrees of freedom available to model the data can lead to over-fitting the data. This happens when the error or noise present in the data drives the behavior of the network more than the underlying input-output relation

itself. Thus, a part of the data is used to determine $\boldsymbol{\theta}$, and another part to determine the hyper-parameters $\boldsymbol{\Theta}$. The remainder of the data is kept aside for testing the performance of the trained network on unseen data, i.e., the network's ability to *generalize* well.

Now let us discuss how this split is used during the three phases in further details:
Training: Training the network makes use of the training set $\mathcal{S}_{\text{train}}$ to solve the following optimization problem: Find

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi_{\text{train}}(\boldsymbol{\theta}), \quad \text{where} \quad \Pi_{\text{train}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{\substack{i=1 \\ (\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{S}_{\text{train}}}}^{N_{\text{train}}} \|\boldsymbol{y}_i - \boldsymbol{\mathcal{F}}(\boldsymbol{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2$$

for some fixed $\boldsymbol{\Theta}$. The optimal $\boldsymbol{\theta}^*$ is obtained using a suitable gradient based algorithm (will be discussed later). The function Π_{train} is referred to as the *training loss function*. In the example above we have used the mean-squared loss function. Later we will consider other types of loss functions.

Validation: Validation of the network involves using the validation set $\mathcal{S}_{\text{val}}$ to solve the following optimization problem: Find

$$\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} \Pi_{\text{val}}(\boldsymbol{\Theta}), \quad \text{where} \quad \Pi_{\text{val}}(\boldsymbol{\Theta}) = \frac{1}{N_{\text{val}}} \sum_{\substack{i=1 \\ (\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{S}_{\text{val}}}}^{N_{\text{val}}} \|\boldsymbol{y}_i - \boldsymbol{\mathcal{F}}(\boldsymbol{x}_i; \boldsymbol{\theta}^*, \boldsymbol{\Theta})\|^2.$$

The optimal $\boldsymbol{\Theta}^*$ is obtained using techniques such as (random or tensor) grid search. Note that the optimal $\boldsymbol{\theta}^*$ depends on the choice of $\boldsymbol{\Theta}$, i.e., $\boldsymbol{\theta}^* = \boldsymbol{\theta}^*(\boldsymbol{\Theta})$. For ease of notation, we have suppressed this dependency here.

Testing: Once the "best" network is obtained, characterized by $\boldsymbol{\theta}^*$ and $\boldsymbol{\Theta}^*$, it is evaluated on the test set $\mathcal{S}_{\text{test}}$ to estimate the networks performance on data not used during the first two phases.

$$\Pi_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{\substack{i=1 \\ (\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{S}_{\text{test}}}}^{N_{\text{test}}} \|\boldsymbol{y}_i - \boldsymbol{\mathcal{F}}(\boldsymbol{x}_i; \boldsymbol{\theta}^*, \boldsymbol{\Theta}^*)\|^2.$$

This testing error is also known as the (approximate) *generalizing error* of the network.

Let's see an example to better understand how such a network is obtained

Example 2.4.1 Let us consider an MLP where all hyper-parameters are fixed except for the following flexible choices

$$\sigma \in \{\text{ReLU}, \tanh\}, \quad L \in \{10, 20\}.$$

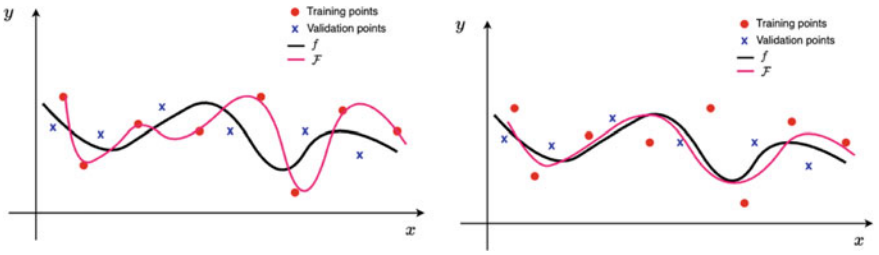
We use the following algorithm

1. For each possible σ, L pair:
   (a) Find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi_{\text{train}}(\boldsymbol{\theta})$
   (b) With this $\boldsymbol{\theta}^*$, evaluate $\Pi_{\text{val}}(\boldsymbol{\Theta})$
2. Select $\boldsymbol{\Theta}^*$ to be the one that gave the smallest value of $\Pi_{\text{val}}(\boldsymbol{\Theta})$.
3. Finally, report Π_{test} for this $\boldsymbol{\Theta}^*$ and the corresponding $\boldsymbol{\theta}^*$.

Remark 2.4.1 The notion of the *test* dataset is important to understand, as it is often misused in practice. In the "correct" approach, the test data should never be used to improve the performance of the network. Once the network (with optimized $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$) is evaluated on the test set, further changes in $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ to improve the test performance can be seen as "data-snooping", with the test set becoming a glorified validation set.

2.5 Overfitting and How to Avoid It

Neural networks, especially MLPs, are almost always *over-parametrized*, i.e., $N_\theta \gg N_{\text{train}}$ where N_{train} is the number of training samples. This would lead to a highly nonlinear network model with such a situation depicted in Fig. 2.5a. In this figure, we show the approximation of scalar-valued function f (black curve) taking a scalar input x. The magenta curve denotes the network prediction, the red dots are the noisy training points, while the blue crossed are the noisy validation points. An over-parameterized MLP has the tendency to fit the noise in the training set, thus leading to a poor performance on the validation (or any unseen) data. We would like to avoid such an overfitting.



(a) Overfitting network (b) Regularized network

Fig. 2.5 The behavior of an un-regularized/overfit network (**a**) and a regularized network **b** approximating a scalar valued function taking a scalar input x. The black curve denotes the true function f, the magenta curve denotes the network prediction, the red dots are the training points, while the blue crosses are the validation points

The first thing we would like to do is to be able to find out whether we are in this situation. How do we know this? A clue to this answer can be gleaned from Fig. 2.5a by noting the distance between the magenta curve (the MLP prediction) and the validation data points, indicated by the blue crosses. This distance is much larger than the distance between the magenta curve and the training data points. This will translate to $\Pi_{\text{val}} \gg \Pi_{\text{train}}$. Thus whenever one observes a large gap between the validation and training losses, one should think about whether one is overfitting the data.

Now that we have a way of identifying overfitting, the next question to answer is how to avoid it? There are several approaches; however the most popular approach is referred to as regularization and is described next.

2.5.1 Regularization

From Fig. 2.5a, we observe that the one significant difference between the black curve (what we desire) and the magenta curve (what we predict) is that the former is much smoother than the latter. That is the derivative of the black curve with respect to its argument is much smaller than the derivative of the red curve. This tells that we would like to train MLPs such that $|\frac{\partial \mathcal{F}(\boldsymbol{x})}{\partial \boldsymbol{x}}|$ is small. Regularization is a way to achieve this goal.

Consider the output the first hidden layer of a network whose input is a scalar x,

$$x_1^{(1)} = \sigma(W_{11}^{(1)} x + b_1^{(1)}),$$

which gives

$$\frac{\partial x_1^{(1)}}{\partial x} = \sigma'(W_{11}^{(1)} x + b_1^{(1)}) W_{11}^{(1)} \propto W_{11}^{(1)}.$$

Since this derivative scales with $W_{11}^{(1)}$, this tells us that if we limit the value of $W_{11}^{(1)}$ we would also limit the value of this derivative. Of course, this is just the derivative of the output of the first hidden layer of the network. We will show later that the derivative of the final output of the MLP is a product of such derivatives. Therefore if we can control this derivative, and others like this, we can control the overall derivative. The most obvious way to do this is by penalizing large values of the weights in the network. This is precisely what is accomplished by adding a regularization term to the loss function.

The simplest method of regularization involves adding a penalty term to the loss function:

$$\Pi(\boldsymbol{\theta}) \longrightarrow \Pi(\boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|, \quad \alpha \geq 0$$

where α is a regularization hyper-parameter, and $\|\boldsymbol{\theta}\|$ is a suitable norm of the network parameters $\boldsymbol{\theta}$. When the individual components of the vector $\boldsymbol{\theta}$ are large, this term

is also large. Now, in addition to finding the value of parameters that best match the training data, we are also looking for those parameters that are small, and therefore will lead to a smoother output from the MLP.

Let us consider some common types of regularization:

- l_2 **regularization**: Here we use the l_2 norm in the regularization term

$$\|\boldsymbol{\theta}\| = \|\boldsymbol{\theta}\|_2 = \left(\sum_{i=1}^{N_\theta} \theta_i^2\right)^{1/2}.$$

- l_1 **regularization**: Here we use the l_1 norm in the regularization term

$$\|\boldsymbol{\theta}\| = \|\boldsymbol{\theta}\|_1 = \sum_{i=1}^{N_\theta} |\theta_i|,$$

which promotes the sparsity of $\boldsymbol{\theta}$.

In Fig. 2.5b, we depict the predictions with a regularized network. Using the penalty term, we obtain a network with lower complexity, whose predictions are smoother. Further, we note that the mismatch between the prediction and validation points is lowered, but at the cost of a slightly higher error in predicting on the training data. Therefore, we now have $\Pi_{\text{val}} \approx \Pi_{\text{train}}$. In the terminology of statistical learning, this notion is also known as *bias-variance tradeoff* [45], which says that as the model complexity of the prediction model decreases, the *bias* (error introduced by approximating a complicated f by a much simpler $\mathcal{F}$) increases, while the *variance* (sensitivity of $\mathcal{F}$ to change in the training data) decreases. In practice, this is better monitored in terms of the Π_{val} and Π_{train}. As shown in Fig. 2.6, Π_{val} is typically high

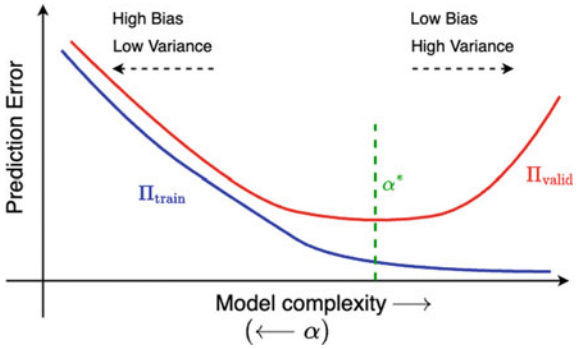


Fig. 2.6 Bias-variance tradeoff. As the model complexity increases (for eg. α decreases to 0), the model bias will decrease, the model variance will increase, the training loss/error will decrease, and the validation loss/error will first decrease but then increase. The optimal α^* has been marked in the figure, which is where the validation loss takes the lowest value

for a complex $\mathcal{F}$ while Π_{train} is low. As the model complexity decreases (say by means of increasing the regularization parameter α), Π_{val} decreases while Π_{train} increases. However, there is typically a sweet spot beyond which both Π_{val} and Π_{train} increase with further simplification of the model. This situation is known as *underfitting* and can occur if α is chosen to be too large. Thus, a careful choice of α (marked as α^* in the figure) becomes important to ensure the network performs well on the training set, while generalizing well to unseen data.

2.6 Gradient Descent

Recall that we wish to solve the minimization problem $\boldsymbol{\theta}^* = \arg\min \Pi(\boldsymbol{\theta})$ in the training phase. This minimization problem can be solved using an iterative optimization algorithm called *gradient descent* (GD), also known as steepest descent. Assuming sufficient smoothness of the loss function with respect to $\boldsymbol{\theta}$, consider the truncated Taylor expansion about $\boldsymbol{\theta}_0$

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) = \Pi(\boldsymbol{\theta}_0) + \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \cdot \Delta\boldsymbol{\theta} + \frac{\partial^2 \Pi}{\partial \theta_i \theta_j}(\hat{\boldsymbol{\theta}}) \Delta\theta_i \Delta\theta_j$$

for some $\hat{\boldsymbol{\theta}}$ in a small neighbourhood of $\boldsymbol{\theta}_0$. When $\|\Delta\boldsymbol{\theta}\|$ is small and assuming $\frac{\partial^2 \Pi}{\partial \theta_i \theta_j}$ is bounded, we can neglect the second order term and just consider the approximation

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) \approx \Pi(\boldsymbol{\theta}_0) + \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \cdot \Delta\boldsymbol{\theta}. \tag{2.8}$$

In order to lower the value of the loss function as much as possible compared to its evaluation at $\boldsymbol{\theta}_0$, i.e. minimize $\Delta\Pi = \Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) - \Pi(\boldsymbol{\theta}_0)$, we need to choose the step $\Delta\boldsymbol{\theta}$ in the opposite direction of the gradient, i.e.:

$$\Delta\boldsymbol{\theta} = -\eta \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \tag{2.9}$$

with the step-size $\eta \geq 0$, also known as the *learning-rate*. This is yet another hyperparameter that we need to tune during the validation phase. Note that by using (2.9) in (2.8), we have

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) \approx \Pi(\boldsymbol{\theta}_0) - \eta \left\| \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \right\|_2^2 \leq \Pi(\boldsymbol{\theta}_0).$$

This is the crux of the GD algorithm, which can be summarized as follows:

1. Initialize $k = 0$ and $\boldsymbol{\theta}_0$
2. While $|\Pi(\boldsymbol{\theta}_k)| > \epsilon_1$, do

(a) Evaluate $\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k)$
(b) Update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k)$
(c) Increment $k = k + 1$.

2.6.1 Convergence

Assume that $\Pi(\boldsymbol{\theta})$ is convex and differentiable, and its gradient is Lipschitz continuous with Lipschitz constant $\mathcal{K}$. Then for a $\eta \leq 1/\mathcal{K}$, the GD updates converges with the rate

$$\|\boldsymbol{\theta}^* - \boldsymbol{\theta}_k\|_2 \leq \frac{C}{k}.$$

However, in most scenarios $\Pi(\boldsymbol{\theta})$ is not convex. If there is more than one minima, then what kind of minima does GD like to pick? To answer this, consider the loss function for a scalar θ as shown in Fig. 2.7, which has two valleys. Let's assume that the profile of $\Pi(\theta)$ in each valley can be locally approximated by a parabola

$$\Pi(\theta) \approx \frac{1}{2} a(\theta - \theta^*)^2$$

where $a > 0$ is the curvature while θ^* is the local minima. Clearly each valley has a different value for a and θ^*. Note that the curvature of the left valley is much smaller than the curvature of the right valley. Let's pick a constant learning rate η and a starting value θ_0 in either of the valleys. Then,

$$\frac{\partial \Pi}{\partial \theta}(\theta_0) = a(\theta_0 - \theta^*)$$

and the new point after a GD update will be

$$\theta_1 = \theta_0 - \eta a(\theta_0 - \theta^*) \iff (\theta_1 - \theta^*) = (\theta_0 - \theta^*)(1 - \eta a)$$

Similarly, it is easy to see that all subsequent iterates can be written as

$$(\theta_{k+1} - \theta^*) = (\theta_k - \theta^*)(1 - \eta a) = (\theta_0 - \theta^*)(1 - \eta a)^k$$

The iterates θ_k would converge to θ^* if $|1 - \eta a| < 1$. Since $a > 0$ in the valleys, we will need the following condition on the learning rate

$$-1 < 1 - \eta a \implies \eta a < 2.$$

If we fix η, then for convergence we need the local curvature to satisfy $a < 2/\eta$. In other words, GD will prefer to converge to a minima with a flat/small curvature, i.e., it will prefer the minima in the left valley. If the starting point is in the right valley,

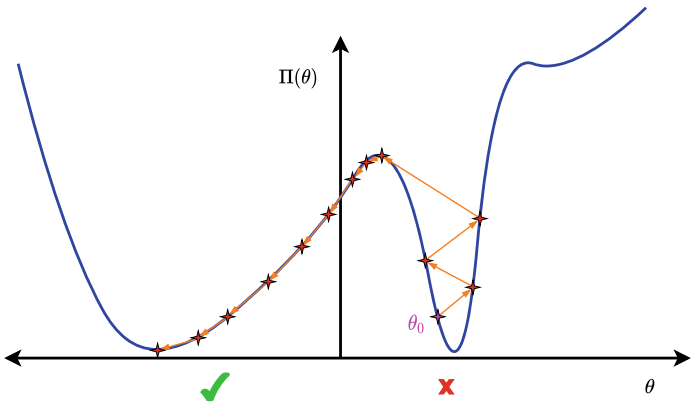


Fig. 2.7 GD prefers flatter minimas

there is a chance that we will keep overshooting the right minima and bounce off the opposite wall till the GD algorithm slingshots θ_k outside the valley (depicted in Fig. 2.7). After this it will enter the left valley with a smaller curvature and gradually move towards its minima.

While it is clear that GD prefers flat minima, what is not clear is why are flat minima better. There is empirical evidence that the parameter values obtained at flat minima tend to generalize better, and therefore are to be preferred [47].

2.7 Some Advanced Optimization Algorithms

We discussed how GD can be used to solve the optimization problem involved in training neural networks. Let us look at a few advanced and popular optimization techniques motivated by GD.

In general, the update formula for most optimization algorithms make use of the following formula

$$[\boldsymbol{\theta}_{k+1}]_i = [\boldsymbol{\theta}_k]_i - [\boldsymbol{\eta}_k]_i [\boldsymbol{g}_k]_i, \quad 1 \leq i \leq N_\theta. \tag{2.10}$$

where we have made use of the notation that for any vector, $\boldsymbol{a}$, $[\boldsymbol{a}]_i$ denotes the i-th component. In the equation above $[\boldsymbol{\eta}_k]_i$ is the component-wise learning rate and the vector-valued function $\boldsymbol{g}$ depends on/approximates the gradient. Note that the learning rate is allowed to depend on the iteration number k.

The GD method can be represented using (2.10) by recognizing

$$[\boldsymbol{\eta}_k]_i = \eta, \quad \boldsymbol{g}_k = \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k).$$

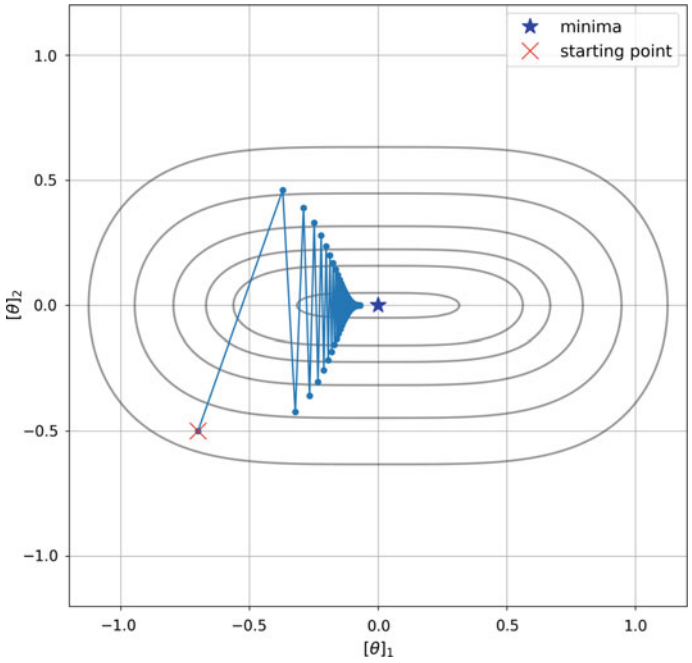


Fig. 2.8 Zig-zagging updates with GD

An issue with the GD method is that the convergence to the minima can be quite slow if η is not suitably chosen. For instance, consider the objective function landscape shown in Fig. 2.8, which has sharper gradients along the $[\boldsymbol{\theta}]_2$ direction compared to the $[\boldsymbol{\theta}]_1$ direction. If we start from a point, such as the one shown in the figure with a red cross-mark, then if η is too large (but still within the stable bounds) the updates will keep zig-zagging their way towards the minima. Ideally, for the particular situation shown in Fig. 2.8, we would like the steps to take longer strides along the $[\boldsymbol{\theta}]_1$ compared to the $[\boldsymbol{\theta}]_2$ direction, thus reaching the minima faster.

Let us look at two popular methods that are able to overcome some of the issues faced by GD.

2.7.1 Momentum Methods

Momentum methods make use of the history of the gradient, instead of just the gradient at the previous step. The formula for the update is given by (2.10) where

$$[\boldsymbol{\eta}_k]_i = \eta, \quad \boldsymbol{g}_k = \beta_1 \boldsymbol{g}_{k-1} + (1-\beta_1)\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k), \quad \boldsymbol{g}_{-1} = 0$$

where $\boldsymbol{g}_k$ is a weighted moving average of the gradient. This weighting is expected to smooth out the zig-zagging seen in Fig. 2.8 by cancelling out the components of gradient along the $[\boldsymbol{\theta}]_2$ direction and causing the updates to move more smoothly towards the minima. A commonly used value for β_1 is 0.9.

2.7.2 Adam

The Adam optimization algorithm (short for "adaptive moment estimation") was introduced by Kingma and Ba [50], and makes use of the history of the gradient as well the second moment (which is a measure of the magnitude) of the gradient. For an initial learning rate η, the updates are once again given by (2.10), where

$$\begin{aligned} \boldsymbol{g}_k &= \beta_1 \boldsymbol{g}_{k-1} + (1-\beta_1)\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k) \\ [\boldsymbol{G}_k]_i &= \beta_2 [\boldsymbol{G}_{k-1}]_i + (1-\beta_2)\left(\frac{\partial \Pi}{\partial \theta_i}(\boldsymbol{\theta}_k)\right)^2 \\ [\boldsymbol{\eta}_k]_i &= \frac{\eta}{\sqrt{[\boldsymbol{G}_k]_i + \epsilon}} \end{aligned} \tag{2.11}$$

In the equations above, $\boldsymbol{g}_k$ and $\boldsymbol{G}_k$ are the weighted running averages of the gradients and the square of the gradients, respectively. The recommended values for the hyper-parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Note that the learning rate for each component is different. In particular, the larger the magnitude of the gradient for a component the smaller is its learning rate. Referring back to the example in Fig. 2.8, this would mean a smaller learning rate for θ_2 in comparison to θ_1, and therefore will help alleviate the zig-zag path of the optimization algorithm.

Remark 2.7.1 The Adam algorithm also has additional correction steps for $\boldsymbol{g}_k$ and $\boldsymbol{G}_k$ to improve the efficiency of the algorithm. See [50] for details.

2.7.3 Stochastic Optimization

We note that the training loss can be rewritten as

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \Pi_i(\boldsymbol{\theta}), \quad \Pi_i(\boldsymbol{\theta}) = \|\boldsymbol{y}_i - \boldsymbol{\mathcal{F}}(\boldsymbol{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2$$

Thus, the gradient of the loss function is

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta})$$

However, taking the summation of gradients can be very expensive since N_{train} is typically very large, $N_{\text{train}} \sim 10^6$. One easy way to circumvent this problem is to use the following update formula (shown here for the GD method)

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k), \tag{2.12}$$

where i is randomly chosen for each update step k. This is known as *stochastic gradient descent*. Remarkably, this modified algorithm does converge assuming that $\Pi_i(\boldsymbol{\theta})$ is convex and differentiable, and $\eta_k \sim 1/\sqrt{k}$ [72].

To illustrate why η_k needs to decay, consider the toy function(s) for $\boldsymbol{\theta} \in \mathbb{R}^2$

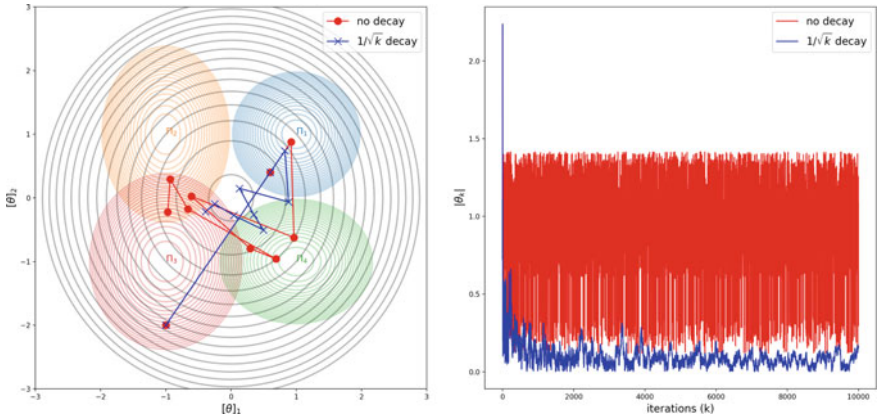
$$\begin{aligned}
&\Pi_1(\boldsymbol{\theta}) = ([\boldsymbol{\theta}]_1 - 1)^2 + ([\boldsymbol{\theta}]_2 - 1)^2, \quad \Pi_2(\boldsymbol{\theta}) = ([\boldsymbol{\theta}]_1 + 1)^2 + 0.5([\boldsymbol{\theta}]_2 - 1)^2, \\
&\Pi_3(\boldsymbol{\theta}) = 0.7([\boldsymbol{\theta}]_1 + 1)^2 + 0.5([\boldsymbol{\theta}]_2 + 1)^2, \quad \Pi_4(\boldsymbol{\theta}) = 0.7([\boldsymbol{\theta}]_1 - 1)^2 + \frac{1}{2}([\boldsymbol{\theta}]_2 + 1)^2, \\
&\Pi(\boldsymbol{\theta}) = \frac{1}{4}\left(\Pi_1(\boldsymbol{\theta}) + \Pi_2(\boldsymbol{\theta}) + \Pi_3(\boldsymbol{\theta}) + \Pi_4(\boldsymbol{\theta})\right).
\end{aligned} \tag{2.13}$$

The contour plots of these functions in shown in Fig. 2.9a, where the black contour plots corresponds to $\Pi(\boldsymbol{\theta})$. Note that the $\boldsymbol{\theta}^* = (0, 0)$ is the unique minima for $\Pi(\boldsymbol{\theta})$. We consider solving with the SGD algorithm with a constant learning rate $\eta_k = 0.4$ and a decaying learning rate $\eta_k = 0.4/\sqrt{k}$. Starting with $\boldsymbol{\theta}_0 = (-1.0, 2.0)$ and randomly selecting $i \in 1, 2, 3, 4$ for each step k, we run the algorithm for 10,000 iterations. The first 10 steps with each learning rate is plotted in Fig. 2.9a. We can clearly see that without any decay in the learning rate, the SGD algorithm keeps overshooting the minima. In fact, this behaviour continues for all future iterations as can be seen in Fig. 2.9b where the norm of the updates does not decay (we expect it to decay to $|\boldsymbol{\theta}^*| = 0$). On the other hand, we quickly move closer to $\boldsymbol{\theta}^*$ if the learning rate decays as $1/\sqrt{k}$.

The reason for reducing the step size as we approach closer to the minima is that far away from the minima for Π, the gradient vector for Π and all the individual Π_i's align quite well. However, as we approach closer to the minima for Π this is not the case and therefore one is required to take smaller steps so as not be thrown off to a region far away from the minima.

In practice, stochastic optimization algorithms are not used for the following reasons:

1. Although the loss function decays with the number of iterations, it fluctuates in a chaotic manner close the the minima and never manages to reach the minima.
2. While handling all samples at once can be computationally expensive, handling a single sample at a time severely under-utilizes the computational and memory resources.



(a) Function contours and paths

(b) Norm of updates

Fig. 2.9 SGD algorithm with and without a decay in the learning rate

However, a compromise can be made by using *mini-batch optimization*. In this strategy, the dataset of N_{train} samples is split into N_{batch} disjoint subsets known as mini-batches. Each mini-batch contains $\overline{N}_{\text{train}} = N_{\text{train}}/N_{\text{batch}}$ samples, which is also referred to as the batch-size. Thus, the gradient of the loss function can be approximated by

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) \approx \frac{1}{\overline{N}_{\text{train}}} \sum_{i \in \text{batch}(j)} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}). \tag{2.14}$$

Note that taking $N_{\text{batch}} = 1$ leads to the original optimization algorithms, while take $N_{\text{batch}} = N_{\text{train}}$ gives the stochastic gradient descent algorithm. One typically chooses a batch-size to maximize the amount of data that can be loaded into the RAM at one time. We define an *epoch* as one full pass through all samples (or mini-batches) of the full training set. The following describes the mini-batch stochastic optimization algorithm:

1. For epoch = 1, …, J
   (a) Randomly shuffle the full training set
   (b) Create N_{batch} mini-batches
   (c) For $i = 1, \ldots, N_{\text{batch}}$
      i. Evaluate the batch gradient using (2.14).
      ii. Update $\boldsymbol{\theta}$ using this gradient and your favorite optimization algorithm (gradient descent, momentum, or Adam).

Remark 2.7.2 There is an interesting study [113] that suggests that stochastic gradient descent might actually help in selecting minima that generalize better. In that study the authors prove that SGD prefers minima whose curvature is more homogeneous. That is, the distribution of the curvature of each of the components of the

loss function is sharp and centered about a small value. This is in contrast to minima where the overall curvature might be small; however the distribution of the curvature of each component of the loss function is more spread out. Then they go on to show (empirically) that the more homogeneous minima tend to generalize better than their heterogeneous counterparts.

2.8 Calculating Gradients Using Back-Propagation

The final piece of the training algorithm that we need to understand is how the gradients are actually evaluated while training the network. Recall the output $\boldsymbol{x}^{(l+1)}$ of layer $l+1$ is given by

$$\text{Affine transform:} \quad \xi_i^{(l+1)} = W_{ij}^{(l+1)} x_j^{(l)} + b_i^{(l+1)}, \quad 1 \le i \le H_{l+1} \tag{2.15}$$

$$\text{Non-linear transform:} \quad x_i^{(l+1)} = \sigma\left(\xi_i^{(l+1)}\right), \quad 1 \le i \le H_{l+1}. \tag{2.16}$$

Given a training sample $(\boldsymbol{x}, \boldsymbol{y})$, set $\boldsymbol{x}^{(0)} = \boldsymbol{x}$. The value of the loss/objective function (for this particular sample) can be evaluated using the forward pass:

1. For $l = 1, \ldots, L+1$
   (a) Evaluate $\xi^{(l)}$ using (2.15).
   (b) Evaluate $\boldsymbol{x}^{(l)}$ using (2.16).
2. Evaluate the loss function for the given sample

$$\Pi(\boldsymbol{\theta}) = \|\boldsymbol{y} - \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2.$$

This operation can be written succinctly in the form of a computational graph as shown in Fig. 2.10. In this figure, the lower portion of the graph represents the evaluation of the loss function Π.

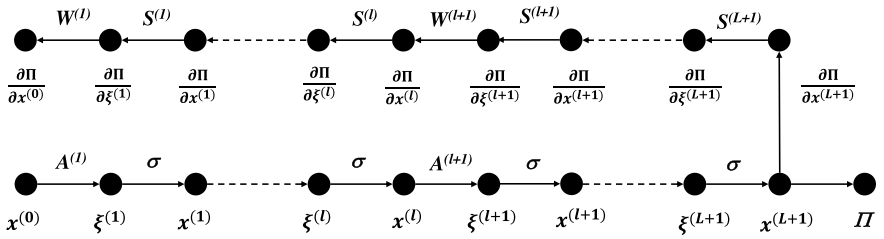


Fig. 2.10 Computational graph for computing the loss function and its derivatives with respect to hidden/latent vectors

We would of course need to repeat this step for all samples in the training set (or a mini-batch for stochastic optimization). For simplicity, we restrict the discussion to the evaluation of the loss and its gradient for a single sample.

In order to update the network parameters, we need $\frac{\partial \Pi}{\partial \boldsymbol{\theta}}$, or more precisely $\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}$, $\frac{\partial \Pi}{\partial \boldsymbol{b}^{(l)}}$ for $1 \leq l \leq L+1$. We will derive expressions for these derivatives by first deriving expressions for $\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}$ and $\frac{\partial \Pi}{\partial \boldsymbol{x}^{(l)}}$.

From the computational graph it is easy to see how each hidden variable in the network is transformed to the next. Recognizing this, and applying the chain rule repeatedly yields

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \cdot \frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{\xi}^{(L+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(L+1)}}{\partial \boldsymbol{x}^{(L)}} \cdots \frac{\partial \boldsymbol{x}^{(l+1)}}{\partial \boldsymbol{\xi}^{(l+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} \cdot \frac{\partial \boldsymbol{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}}. \tag{2.17}$$

In order to evaluate this expression we need to evaluate the following terms:

$$\frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} = -2(\boldsymbol{y} - \boldsymbol{x}^{(L+1)}) \tag{2.18}$$

$$\frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} = \boldsymbol{W}^{(l+1)} \tag{2.19}$$

$$\frac{\partial \boldsymbol{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \equiv \text{diag}[\sigma'(\xi_1^{(l)}), \ldots, \sigma'(\xi_{H_l}^{(l)})], \tag{2.20}$$

where the last two relations hold for any network layer l, H_l is the width of that particular layer, and σ' denotes the derivative of the activation with respect to its argument. Using these relations in (2.17), we arrive at,

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \cdot \boldsymbol{S}^{(L+1)} \cdot \boldsymbol{W}^{(L+1)} \cdots \boldsymbol{S}^{(l+1)} \cdot \boldsymbol{W}^{(l+1)} \cdot \boldsymbol{S}^{(l)}. \tag{2.21}$$

Now consider a vector $\boldsymbol{v} = \boldsymbol{a} \cdot \boldsymbol{M}$, where $\boldsymbol{a}$ is another vector and $\boldsymbol{M}$ is a matrix. Then we may write $\boldsymbol{v} = \boldsymbol{a}^T \boldsymbol{M} = \boldsymbol{M}^T \boldsymbol{a}$. Using this in the equation above, recognizing that the transpose of the product of a series of matrices is equal to the product (in reverse order) of the transpose of the matrices, and recognizing that $\boldsymbol{\Sigma}^{(l)}$ is diagonal and therefore symmetric, we finally arrive at

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \boldsymbol{W}^{(l+1)T} \boldsymbol{S}^{(l+1)} \cdots \boldsymbol{W}^{(L+1)T} \boldsymbol{S}^{(L+1)} [-2(\boldsymbol{y} - \boldsymbol{x}^{(L+1)})]. \tag{2.22}$$

This evaluation can also be represented as a computational graph. In fact, as shown in Fig. 2.10, it can be appended to the original graph, where this part of the computation appears in the upper row of the graph. Note that we are now traversing in the backward direction. Hence the name back propagation.

The final step is to evaluate an explicit expression for $\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}$. This can be done by recognizing,

$$\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \otimes \boldsymbol{x}^{(l-1)}, \tag{2.23}$$

where $[\boldsymbol{x} \otimes \boldsymbol{y}]_{ij} = x_i y_j$ is the outer product. Thus, in order to evaluate $\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}$ we need $\boldsymbol{x}^{(l-1)}$ which is evaluated during the forward phase and $\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}$ which is evaluated during back propagation.

Remark 2.8.1 It is instructive to estimate the computational cost of computing the loss function and its derivatives with respect to the learnable parameters of the MLP. Let us compute this estimate for an MLP with L hidden layers each with width H. The computational cost of the computing the loss function is the cost associated with the lower part of the graph shown in Fig. 2.10. In this graph the cost of each affine transform is $O(H^2)$ flops, while that of each activation layer is $O(H)$. Since we have L such layers, the dominant cost of computing the loss function scales as $O(H^2 L)$. The cost of computing the derivative of the loss function is estimated by estimating the cost of the upper branch of the computational graph, plus the cost of computing the outer product. The former is dominated by the matrix-vector product at each layer with the $\boldsymbol{W}^{(l)^T}$, and scales as $O(H^2 L)$. The cost of computing the outer product scales with the number of entries in each matrix times the number of matrices, and is therefore also given by $O(H^2 L)$. Therefore, the cost of computing the derivative of the loss function scales as $O(2H^2 L)$, which is the same order as the cost of computing the loss function itself. The fact that these costs scale in the same way is critical, making the training of an MLP feasible with reasonable computational resources.

Question 2.8.1 *Can you derive a similar set of expressions and the corresponding algorithm to evaluate* $\frac{\partial \Pi}{\partial \boldsymbol{b}^{(l)}}$*?*

Question 2.8.2 *Can you derive an explicit expression for* $\frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{x}^{(0)}}$*. That is the an expression for the derivative of the output of the network with respect to its input? This is a very useful quantity that finds use in algorithms like physics informed neural networks (see Chap. 5) and Wasserstein generative adversarial networks (see Chap. 7).*

2.9 Regression Versus Classification

Till now, given the labelled dataset $\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : 1 \leq i \leq N\}$, we have considered two types of losses

- The mean square error (MSE)

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \|\boldsymbol{y}_i - \boldsymbol{\mathcal{F}}(\boldsymbol{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2.$$

- The mean absolute error (MAE)

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \|\boldsymbol{y}_i - \boldsymbol{\mathcal{F}}(\boldsymbol{x}_i; \boldsymbol{\theta}, \boldsymbol{\Theta})\|.$$

Neural networks with the above losses can be used to solve various regression problems where the underlying function is highly nonlinear and the inputs/outputs are multi-dimensional.

Example 2.9.1 Given the house/apartment features such as the zip code, the number of bedrooms/bathrooms, carpet area, age of construction, etc, predict the outcomes such as the market selling price, or the number of days on the market.

Now let us consider some examples of classification problems, where the output of the network typically lies in a discrete finite set.

Example 2.9.2 Given the symptoms and blood markers of patients with COVID-19, predict whether they will need to be admitted to ICU. So the input and output for this problem would be

$$\begin{aligned} \boldsymbol{x} &= [\text{pulse rate, temperature, } SPO_2\text{, procalcitonin, } \ldots] \\ \boldsymbol{y} &= [p_1, p_2] \end{aligned}$$

where p_1 is the probability of being admitted to the ICU, while p_2 is the probability of not being admitted. Note that $0 \leq p_1, p_2 \leq 1$ and $p_1 + p_2 = 1$.

Example 2.9.3 Given a set of images of animals, predict whether the animal is a dog, cat or bird. In this case, the input and output should be

$$\begin{aligned} \boldsymbol{x} &= \text{the image} \\ \boldsymbol{y} &= [p_1, p_2, p_2] \end{aligned}$$

where p_1, p_2, p_3 is the probability of being a dog, cat or bird, respectively. Again, $0 \leq p_1, p_2, p_3 \leq 1$ and $p_1 + p_2 + p_3 = 1$.

Since the output for the classification problem corresponds to probabilities, we need to make a few changes to the network:

1. Make use of an output function at the end of the output layer that suitably transforms the output vector into the desired form, i.e., a vector of probabilities. This is typically done using the *softmax function*

$$x_i^{(L+1)} = \frac{\exp(\xi_i^{(L+1)})}{\sum_{j=1}^{C} \exp(\xi_j^{(L+1)})}$$

where C is the number of classes (and also the output dimension). It is easily verified that with this transformation, the components of the $\boldsymbol{x}^{(L+1)}$ form a convex combination, i.e., $x_i^{(L+1)} \in [0, 1]$ and $\sum_{i=1}^{C} x_i^{(L+1)} = 1$.

2. The output labels for the various samples need to be one-hot encoded. In other words, for the sample $(\boldsymbol{x}, \boldsymbol{y})$, the output label $\boldsymbol{y}$ should have dimension $D = C$, and whose component is 1 only for the component signifying the class $\boldsymbol{x}$ belongs to, otherwise 0. For instance, in Example 2.9.3

$$\boldsymbol{y} = \begin{cases} [1, 0, 0]^\top & \text{if } \boldsymbol{x} \text{ is a dog,} \\ [0, 1, 0]^\top & \text{if } \boldsymbol{x} \text{ is a cat,} \\ [0, 0, 1]^\top & \text{if } \boldsymbol{x} \text{ is a bird.} \end{cases}$$

3. Although the MSE or MSA can still be used as the loss function, it is preferable to use the cross-entropy loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \sum_{c=1}^{C} -y_{ci} \log(\mathcal{F}_c(\boldsymbol{x}_i; \boldsymbol{\theta})), \tag{2.24}$$

where y_{ci} is the c-th component of the true label for the i-th sample. The loss function in (2.24) treats y_c and $\mathcal{F}_c$ as probability distributions and measures the discrepancy between the two. It can be shown to be related to the Kullback-Liebler divergence between the two distributions. Compared to MSE, this loss function severely penalizes strongly confident incorrect predictions. This is demonstrated in Example 2.9.4.

Example 2.9.4 Let us consider a binary classification problem, i.e., $C = 2$. For a given $\boldsymbol{x}$, let $\boldsymbol{y} = [0, 1]$ and let the prediction be $\boldsymbol{\mathcal{F}} = [p, 1 - p]$. Clearly, a small value of p is preferred for this sample. Therefore any reasonable cost function should penalize large values of p. Now let us evaluate the error using various loss functions

- MSE Loss $= (0 - p)^2 + (1 - 1 + p)^2 = 2p^2$.
- Cross-entropy Loss $= -(0 \log(p) + 1 \log(1 - p) = -\log(1 - p)$.

Note that both losses penalize large values of p. Also when $p = 0$, both losses are zero. However, as $p \to 1$ (which would lead the wrong prediction), the MSE loss $\to 2$, while the cross-entropy loss $\to \infty$. That is, it strongly penalizes incorrect confident predictions. The two losses are plotted in Fig. 2.11.

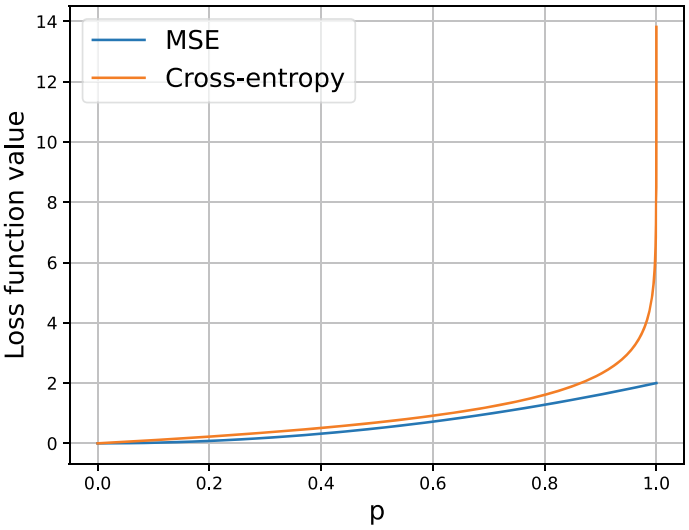


Fig. 2.11 Comparing MSE and cross-entropy losses as a function of the class probability p for a binary classification problem

2.10 Computational Exercise

2.10.1 Expressivity of Deep Neural Networks

The scope of this numerical exercise is to understand how the expressive power of the network varies with depth and width. We begin by loading some necessary Python libraries:

```
# Numpy library
import numpy as np

# Plotting library
import matplotlib.pyplot as plt

# PyTorch libraries
import torch
from torch import nn
```

Next, we define the structure of the network. The class called `MLP` below is inherited from `torch.nn.Module`. The `__init__` method takes the following arguments:

- `input_dim`: dimension of the input vector
- `output_dim`: dimension of the output vector/prediction
- `width`: width of each hidden layer (number of neurons per layer)
- `depth`: depth of the neural network (number of hidden layers + output layer)
- `activation`: type of activation function used in each neuron

The following PyTorch commands are used to define the various layers:

- `torch.nn.Linear(N,M)`: Use to define a dense layer with `M` neurons and receiving an input `N`-dimensional input.
- `torch.nn.ModuleList()`: Holds PyTorch submodules in a list.
- We have shown a few ways to define activation functions.
- To initialize the weights with a uniform distribution you can use `torch.nn.init.uniform_()` and the class function `torch.nn.Module.apply()`.

The second crucial method required in the MLP class is `forward` that takes as argument the network input, and defines the forward pass of the network. Note that the activation function is applied only to the output of the hidden layers.

```
class MLP(nn.Module):
    def __init__(self, input_dim=1, out_dim=1, width=10,
        depth=5, activation='tanh'):
        super(MLP, self).__init__()

        # Want to ensure there is at least one hidden layer
        assert depth > 1

        self.depth=depth

        # Selecting the activation for hidden layers
        if activation == 'tanh':
          self.activation = nn.Tanh()
        elif activation == 'sin':
          self.activation = torch.sin
        elif activation == 'relu':
          self.activation = nn.ReLU()

        # Creat list of layers and include the first hidden layer
        MLP_list = [nn.Linear(input_dim, width)]

        # Remaining hidden layers
        for _ in range(depth - 2):
            MLP_list.append(nn.Linear(width, width))

        # Output layer
        MLP_list.append(nn.Linear(width, out_dim))

        # Adding list of layers as modules
        self.model = nn.ModuleList(MLP_list)

        # Weights initialization
        def init_weights(layer):
          if isinstance(layer, nn.Linear):
              nn.init.uniform_(layer.weight, -1, 1)
              if layer.bias is not None:
                nn.init.uniform_(layer.bias, -1, 1)

        self.model.apply(init_weights)
```

```
    # Defining forward mode of MLP model
    def forward(self, x):

        for i, layer in enumerate(self.model):
            # Apply activation only to hidden layers
            if i < self.depth-1:
                x = self.activation(layer(x))
            else:
                x = layer(x)
        return x
```

With this in hand, perform the following tasks:

1. Using the above class, construct different instances of the network by fixing `input_dim = 2`, `output_dim = 1`, `depth = 15` and varying `width = 5,10,20`. Repeat this with both `tanh` and `sin` activation functions.
   (a) For each configuration, calculate and print the total number of network parameters (weights and biases).
   (b) Create an 201×201 array of equi-spaced points in the domain $[-1, 1] \times [-1, 1]$. Using this as the input to the network, evaluate the network prediction. This will be a 201×201 array of real numbers. You may need to use `.detach().numpy()` on the network prediction to detach the graph and return the output as a `numpy` array. Do this for each network configuration (6 in total). Note that there is **no training involved here**. Plot the output from each network in a two-dimensional contour plot. You will generate 6 such plots (one for each configuration).
   (c) For a given choice of the activation function, what trends do you observe as you vary width?
   (d) What effect does the choice of activation function appear to have?
   (e) Do these trends vary if you re-run the script? What is the cause of this variation, if any?
   (f) What is your conclusion about the expressivity of neural networks as a function of width and/or activation function?

2.10.2 Training an MLP for a Regression Problem

In what follows, a deep neural network is trained to approximate a scalar function. The primary objective is to learn how to train an MLP, tune its hyper-parameters (in this case the regularization parameter), and understand the effect of regularization on the prediction of the network. You can make use of the following Pytorch tutorials for reference:

1. Build the Neural Network: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html
2. Automatic Differentiation with torch.autograd: https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

3. Optimizing Model Parameters: https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html.

To start, the following Python libraries will be necessary:

```
import numpy as np
import torch
from torch import nn
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
```

To create a dataset we will use the function

$$f(x) = \tanh(20(x - 0.5)) + \sin(10\pi x) + 0.3\omega(x)$$

where $\omega \sim N(0, 1)$ is an additive Gaussian noise at each evaluation point. We evaluate this function at 150 points uniformly spaced in [0, 1]. Out of these, 100 randomly selected points are used for training and the remainder for validation. We fix a random generator seed (say to 1), to ensure the same split into training and validation data occurs each time the code is run. In a single plot, we will show the points of the training and validation sets.

```
def target_func(x):
    val = np.tanh(20 * (x-0.5)) + np.sin(10 * np.pi * x)
    # Adding noise
    val += 0.3 * np.random.normal(size=val.shape)
    return val.to(torch.float32)


np.random.seed(1)
Ntrain = 100
Nval   = 50
N      = Ntrain + Nval

x         = torch.linspace(0, 1, N).reshape(N,1)
ind_all   = np.arange(0,N)
np.random.shuffle(ind_all)
ind_train = np.sort(ind_all[0:Ntrain])
ind_val   = np.sort(ind_all[Ntrain:])

x_train = x[ind_train]
x_val   = x[ind_val]
y_train = target_func(x_train)
y_val   = target_func(x_val)

fig, ax = plt.subplots(figsize=(15,5))
ax.plot(x_train,y_train,'-o', label='Training points')
ax.plot(x_val,y_val,'-x', label='Validation points')
plt.legend()
```

Next, we create a custom dataset class (inheritied from `torch.utils.data.Dataset`) consisting of three key class objects:

1. `__init__` which receives paired data samples and stores them as class objects.
2. `__len__` which returns the number of samples in the dataset.
3. `__getitem__` which receives a list of indices and returns the sample pairs corresponding to those indices.

We then use this dataset class to create a dataset object for training data only, and use `torch.utils.data.DataLoader` to create a wrapper that allows to iterate through random batches of the dataset. We use a batch size of 50 with shuffling. How many mini-batches would this lead to?

```
# Create custom dataset class
class CustomDataset(Dataset):
    def __init__(self, samples):
        """
        Initialize the CustomDataset with paired samples.

        Args:
            samples (list of tuples): A list of (x, y) pairs
                representing the dataset samples.
        """
        self.samples = torch.Tensor(samples).to(torch.float32)

    def __len__(self):
        """
        Returns the length of the dataset, i.e., the number of
            samples.
        """
        return len(self.samples)

    def __getitem__(self, idx):
        """
        Returns the sample pairs corresponding to the given list
            of indices.

        Args:
            indices (list): A list of indices to retrieve samples
                for.

        Returns:
            list: A list of (x, y) pairs corresponding to the
                specified indices.
        """
        selected_samples = self.samples[idx]
        return selected_samples

training_samples =
    np.concatenate((x_train.reshape(-1,1),y_train.reshape(-1,1)),
    axis=1)
train_dataset = CustomDataset(samples=training_samples)
train_loader = DataLoader(train_dataset, batch_size=50,
    shuffle=True)
```

The tasks for this exercise are:

1. Use the class defined in previous exercise to define an MLP without an activation function in the output layer, and create a network with `width = 45`, `depth = 8`, `input_dim = output_dim = 1` and $\sin(\cdot)$ activation. Consider four values of the regularization parameter, $\{0.0, 1e-3, 1e-2, 1e-1\}$. Your goal is to find the "optimal" value of this parameter. For each parameter value, train the network 4 times with different weight initializations. Set the maximum epochs to 10,000. To train each instance of the network, you will need to
   (a) Define the optimizer (Adam) with a learning rate of $1e-3$ and the `weight_decay` set as the regularization parameter.
   (b) Create two loops, one over the number of epochs and another over the number of mini-batches.
   (c) In the innermost loop, set the gradient buffers to zero `.zero_grad()`, evaluate the model and training loss use `nn.MSELoss()`, find the gradients using back-propagation (`.backward()`) and update the weights (`.step()`).
2. For each value of the regularization parameter:
   (a) Save the history of the metric evaluated for the training set (`mse`) and the validation set (`val_mse`) for each of the four runs. Compute the average of this history across the four runs. Generate two side-by-side plots for each regularization regularization parameter. In the first plot, show the training metrics as a function of epoch number for each run, as well as the average; in the second, do the same for the validation metrics.
   (b) Report the average value of training and validation metrics at the final epoch. Based on this value, select the optimal value for the regularization parameter. Justify your selection.
   (c) At the completion of each run, evaluate the prediction of the trained network on a set of uniformly spaced 1,000 points in the interval [0, 1]. For efficiency, remember to run the network in evaluation mode inside the `torch.no_grad()` environment. In a single plot (one per regularization regularization parameter) plot this prediction as a function of the input. There should be four curves in each plot. Explain the effect of regularization on these predictions. Which regularization parameter yields prediction curves that are closest to the noise-free target function?

Chapter 3
Residual Neural Networks

Residual networks (or ResNets) were introduced by He et al. [40] in 2015. In this chapter, we will discuss what these networks are, why they were introduced and their relation to ODEs.

3.1 Vanishing Gradients in Deep Networks

While training neural networks, the gradients $\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}$, $\frac{\partial \Pi}{\partial \boldsymbol{b}^{(l)}}$ might become very small. For instance, consider a very deep network, say $L \geq 20$. If $\left|\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}\right| \ll 1$ for $l \leq \bar{l}$, then the contribution of first $\bar{l}$ layers of the network will be negligible, as the influence of their weights on the loss function is small. Because of this depth cut-off, the benefit in terms of expressivity of deep networks is lost.

So why does this happen? Recall from Sect. 2.8 that

$$\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \otimes \boldsymbol{x}^{(l-1)}$$

and

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \prod_{m=l+1}^{L+1} (\boldsymbol{W}^{(m)T} \boldsymbol{S}^{(m)}) \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}}. \tag{3.1}$$

For any matrix, $\boldsymbol{A}$, let $\tau(\boldsymbol{A})$ denote the largest singular value. Then we can bound $\left\|\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}\right\|$ by

$$\left\|\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}\right\| \leq \tau(\boldsymbol{S}^{(l)}) \prod_{m=l+1}^{L+1} (\tau(\boldsymbol{W}^{(m)})\tau(\boldsymbol{S}^{(m)})) \left\|\frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}}\right\|. \tag{3.2}$$

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_3

Recall that $\boldsymbol{S}^{(m)} \equiv \text{diag}[\sigma'(\xi_1^{(m)}), \ldots, \sigma'(\xi_{H_l}^{(m)})]$, where σ' denotes the derivative of σ with respect to its argument. For ReLU its value is either 0 or 1. Therefore $\tau(\boldsymbol{S}^{(m)}) = 1$.

Also, for stability we would want $\tau(\boldsymbol{W}^{(m)}) < 1$. Otherwise the output of the network can become unbounded. In practice this can be enforced by augmenting the loss with a regularization term. Thus, Eq. 3.2 reduces to

$$\left\| \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \right\| \leq \prod_{m=l+1}^{L+1} (\tau(\boldsymbol{W}^{(m)})) \left\| \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \right\|, \tag{3.3}$$

where each term in the product is a scalar less than 1. As the number of terms increases, that is $L - l \gg 1$, this product can, and does, become very small. This typically happens when $L - l \approx 20$, in which case $\left\| \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \right\|$, and therefore $\left\| \frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}} \right\|$, become very small. This issue is called the problem of vanishing gradients. It manifests itself in deep networks where the weights in the inner layers (say $L - l > 20$) do not contribute to the network.

In [40], the authors demonstrate that taking a deeper network can actually lead to an increase in training and validation error. To demonstrate this, we train MLPs with varying depth to approximate the function

$$u(x) = \sin\left(2\pi(x+1)^3\right)\cos(2\pi x), \quad x \in [0, 1]. \tag{3.4}$$

As can be seen in the first column of Figs. 3.1 and 3.2, the training and test (MSE) losses curves shift upwards as the depth of the MLP is increased. In terms of predictions, only depth = 10 leads to a good approximation of the function. With depth = 20, the approximation towards the right of the domain (where high-frequency modes dominate) is poor. If we increase the depth to 40, the MLP seems to learn a constant function. Thus, beyond a certain point, increasing the depth of a network can be counterproductive. Based on our previous discussion on vanishing gradients we know why this is the case. Given this, we would like to come up with a network architecture that addresses the problem of vanishing gradients by ensuring $\left\| \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \right\| \approx \left\| \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \right\|$. This means requiring that when the weights of the network approach small values, the network should approach the identity mapping, and not the null mapping. This is the core idea behind a ResNet architecture.

3.2 ResNets

Consider an MLP with depth 6 (as shown in Fig. 3.3) with a fixed width H for each hidden layer. We add skip connections between the hidden layers in the following manner

$$x_i^{(l)} = \sigma(W_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)}) + x_i^{(l-1)}, \quad 2 \leq l \leq L. \tag{3.5}$$

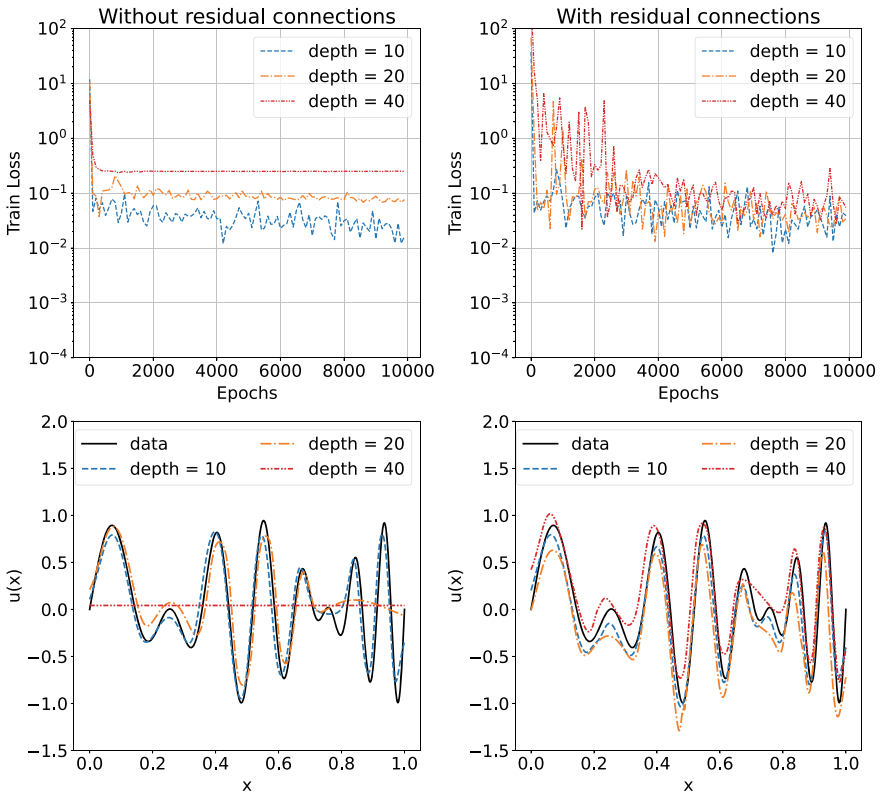


Fig. 3.1 Performance of MLP approximating (3.4) on the training set, without residual connections (left) and with residual connections (right), as the depth is increased

We can make the following observations:

1. If all weights (and biases) were null, then $\boldsymbol{x}^{(5)} = \boldsymbol{x}^{(1)}$, which in turn would imply

$$\frac{\partial \Pi}{\partial \boldsymbol{x}^{(1)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(5)}},$$

   i.e., we will not have the issue of vanishing gradients.
2. The computational graph for forward and back-propagation of a ResNet is shown in Fig. 3.4. Looking at this graph, it is clear that the expression for $\frac{\partial \boldsymbol{x}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}}$ now involves traversing two branches and adding their sum. Therefore, we have

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \prod_{m=l+1}^{L+1} (\boldsymbol{I} + \boldsymbol{W}^{(m)T} \boldsymbol{S}^{(m)}) \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}}. \tag{3.6}$$

Now, if we assume that $\|\boldsymbol{W}^{(m)}\| \ll 1$ via regularization, we have

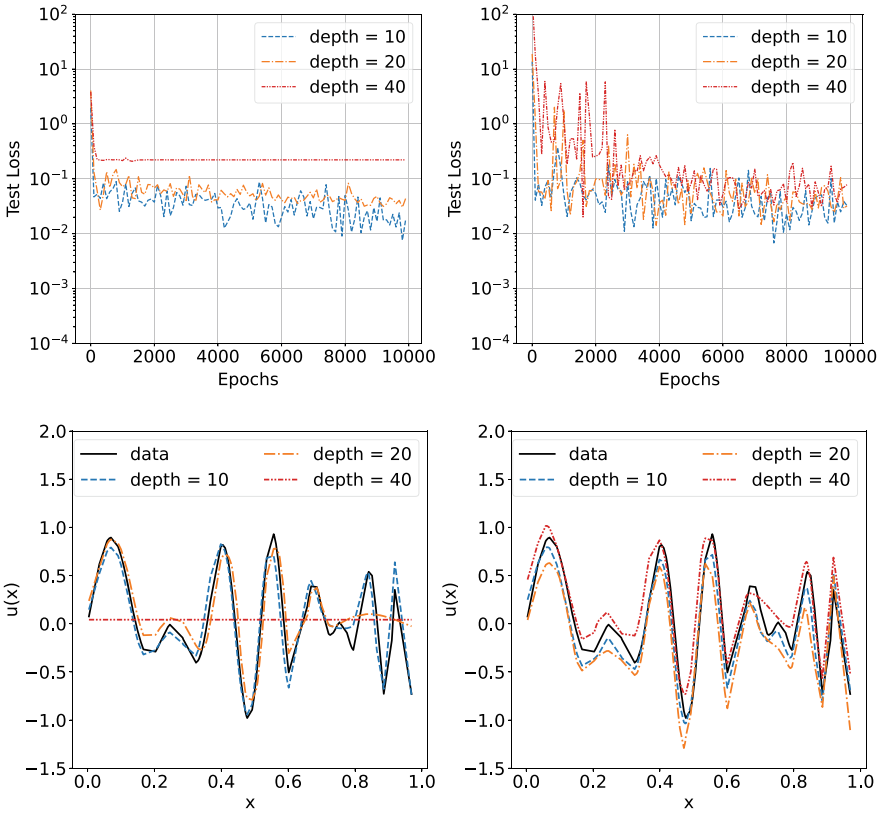


Fig. 3.2 Performance of MLP approximating (3.4) on the test set, without residual connections (left) and with residual connections (right), as the depth is increased

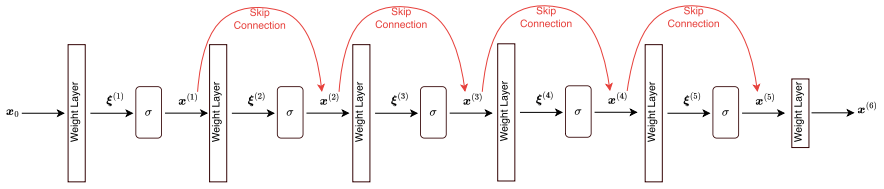


Fig. 3.3 ResNet of depth 6 with skip connections

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \left(\boldsymbol{I} + \sum_{m=l+1}^{L+1} \boldsymbol{W}^{(m)\,T} \boldsymbol{S}^{(m)} + \text{ higher order terms} \right) \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}}. \quad (3.7)$$

In the expression above, even if the individual matrices have small entries, their sum need not approach a zero matrix. This implies that we can create a finite (and

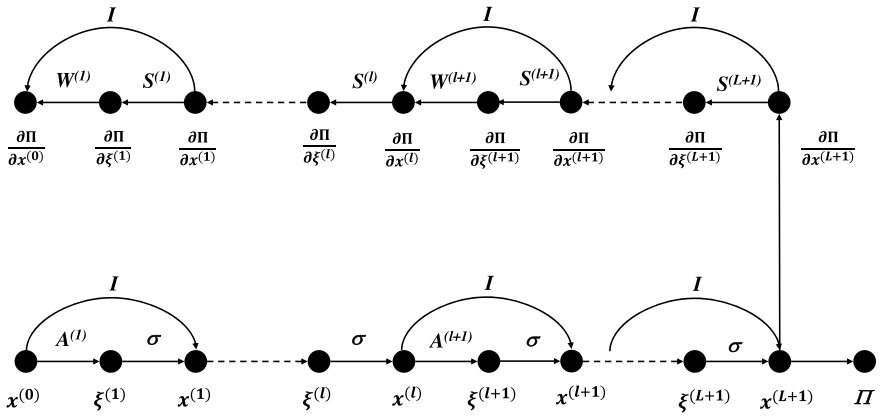


Fig. 3.4 Computational graph for forward and backpropagation in a Resnet

significant) change between the gradients near the input and output layers, while still requiring the weights to be small (via regularization).

We empirically demonstrate the benefit of adding residual skip connections by training MLPs to approximate (3.4) with varying depth, but now including skip connections between hidden layers. The results shown in the second column of Figs. 3.1 and 3.2 clearly show an improvement in the training/test losses, as well as the predictions, for all depths considered. The improvement is more significant for the deeper network, with the depth = 40 MLP clearly capturing the target function $u(x)$, as opposed to the constant function learned by the MLP in the absence of skip connections.

Remark 3.2.1 The above analysis can be extended to cases when the hidden layer width H is not fixed, but the analysis is not as clean. See [40] on how we can do this.

3.3 Connections with ODEs

Let us first consider the special case of a ResNet with $d = D = H$. Recall the relation (3.5), which we can rewrite as

$$\frac{\boldsymbol{x}^{(l)} - \boldsymbol{x}^{(l-1)}}{\Delta t} = \frac{1}{\Delta t}\sigma(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)}) = \frac{1}{\Delta t}\sigma(\boldsymbol{\xi}^{(l)}) \tag{3.8}$$

for some scalar Δt, where we note that $\boldsymbol{\xi}^{(l)}$ is a function of $\boldsymbol{x}^{(l-1)}$ parameterized by $\boldsymbol{\theta}^{(l)} = [\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}]$. Thus, we can further rewrite (3.8) as

$$\frac{\boldsymbol{x}^{(l)} - \boldsymbol{x}^{(l-1)}}{\Delta t} = \boldsymbol{V}(\boldsymbol{x}^{(l-1)}; \boldsymbol{\theta}^{(l)}). \tag{3.9}$$

Now consider a first-order system of (possibly non-linear) ODEs

$$\dot{\boldsymbol{x}} \equiv \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{V}(\boldsymbol{x}, t) \tag{3.10}$$

where we want to find $\boldsymbol{x}(T)$ given some initial state $\boldsymbol{x}(0)$. In order to solve this numerically, we can uniformly divide the temporal domain with a time-step Δt and temporal nodes $t^{(l)} = l\Delta t$, $0 \leq l \leq L+1$, where $(L+1)\Delta t = T$. Define the discrete solution as $\boldsymbol{x}^{(l)} = \boldsymbol{x}(l\Delta t)$. Then, given $\boldsymbol{x}^{(l-1)}$, we can use a time-integrator to approximate the solution $\boldsymbol{x}^{(l)}$. We can consider a method motivated by the forward Euler integrator, where the the LHS of (3.10) is approximated by

$$LHS \approx \frac{\boldsymbol{x}^{(l)} - \boldsymbol{x}^{(l-1)}}{\Delta t}.$$

while the RHS is approximated using a parameter $\boldsymbol{\theta}^{(l)}$ as

$$RHS \approx \boldsymbol{V}(\boldsymbol{x}^{(l-1)}; t^{(l)}) = \boldsymbol{V}(\boldsymbol{x}^{(l-1)}; \boldsymbol{\theta}^{(l)}).$$

where we are allowing the parameters to be different at each time-step. Putting these two together, we get exactly the relation of the ResNet given in (3.9). In other words, a ResNet is nothing but a descritization of a non-linear system of ODEs. We make some comments to further strengthen this connection.

- In a fully trained ResNet we are given $\boldsymbol{x}^{(0)}$ and the weights of a network, and we predict $\boldsymbol{x}^{(L+1)}$.
- In a system of ODEs, we are given $\boldsymbol{x}(0)$ and $\boldsymbol{V}(\boldsymbol{x}, t)$, and we predict $\boldsymbol{x}(T)$.
- Training the ResNet means determining the parameters $\boldsymbol{\theta}$ of the network so that $\boldsymbol{x}^{(L+1)}$ is as close as possible to $\boldsymbol{y}_i$ when $\boldsymbol{x}^{(0)} = \boldsymbol{x}_i$, for $i = 1, \ldots, N_{\text{train}}$.
- When viewed from the analogous ODE point of view, training means determining the right hand side $\boldsymbol{V}(\boldsymbol{x}, t)$ by requiring $\boldsymbol{x}(T)$ to be as close as possible to $\boldsymbol{y}_i$ when $\boldsymbol{x}(0) = \boldsymbol{x}_i$, for $i = 1, \ldots, N_{\text{train}}$.
- In a ResNet we are looking for "one" $\boldsymbol{V}(\boldsymbol{x}, t)$ that will map $\boldsymbol{x}_i$ to $\boldsymbol{y}_i$, for all $1 \leq i \leq N_{\text{train}}$.

3.4 Neural ODEs

Motivated by the connection between ResNets and ODEs, neural ODEs were proposed in [15] which received the Best Paper Award in NeurIPS 2018. Consider a system of ODEs given by

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{V}(\boldsymbol{x}, t) \tag{3.11}$$

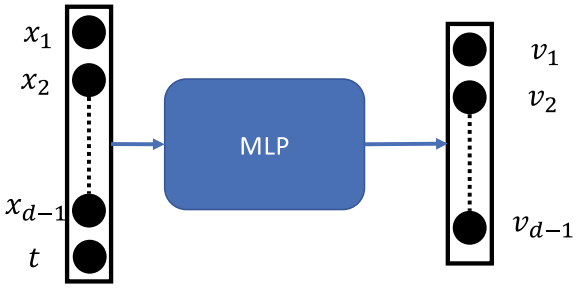


Fig. 3.5 Feed-forward neural network used to model the right hand side in a Neural ODE. The number of dependent variables $= d - 1$

Given $\boldsymbol{x}(0)$, we wish to find $\boldsymbol{x}(T)$. In [15], the RHS, i.e., $\boldsymbol{V}(\boldsymbol{x}, t)$, is defined using a feed-forward neural network with parameters $\boldsymbol{\theta}$ (see Fig. 3.5). The input to the network is $(\boldsymbol{x}, t)$ while the output is $\boldsymbol{V}(\boldsymbol{x}, t)$ (having the same dimension as $\boldsymbol{x}$). With this description, the system (3.11) is solved using a suitable time-marching scheme, such as forward Euler, Runge-Kutta, etc.

So how do we use this network to solve a regression problem? Assume that you are given the labelled training data $\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : 1 \leq i \leq N_{\text{train}}\}$. Here both $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ are assumed to have the same dimension $d - 1$. The key idea is to think of $\boldsymbol{x}_i$ as points in the $d - 1$-dimensional space that represent the initial state of the system, and to think of $\boldsymbol{y}_i$ as points that represent the final state. Then the regression problem becomes finding the RHS of (3.11) that will map the initial points to the final points with minimal amount of error. In other words, find the parameters $\boldsymbol{\theta}$ such that

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} |\boldsymbol{x}_i(T; \boldsymbol{\theta}) - \boldsymbol{y}_i|^2$$

is minimized. Here, $\boldsymbol{x}_i(T; \boldsymbol{\theta})$ denotes the solution (at time $t = T$) to (3.11) with $\boldsymbol{x}(0) = \boldsymbol{x}_i$ and the RHS represented by a feed-forward neural network $\boldsymbol{V}(\boldsymbol{x}, t; \boldsymbol{\theta})$. Note that $\boldsymbol{y}_i$ is the output value that is measured. There is a relatively straightforward way of extending this approach to the case when $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ have different dimensions. In summary, in Neural ODEs one transforms a regression problem to one of finding the nonlinear, time-dependent RHS of a system of ODEs.

This is shown pictorially in Fig. 3.6. In the plot on the left, we have drawn a curve that we wish to approximate using a Neural ODE. We are given $N_{\text{train}} = 7$ data points (x_i, y_i) to train the network. When viewed from a Neural ODE perspective, this means that we are looking for a right-hand side for a scalar ODE whose solution $x(t)$ will be such that when the initial state $x(0)$ is set equal to x_i, the solution at $t = T$ will be equal to y_i. This will be required for all training data points. The solution from the trained Neural ODE is shown in the plot on the right of Fig. 3.6. In this plot each red curve represents a trajectory $x(t)$ of the solution. There are seven such trajectories, each connecting the initial state $x(0) = x_i$ to the final state $x(T) = y_i$ for $i = 1, \ldots, 7$.

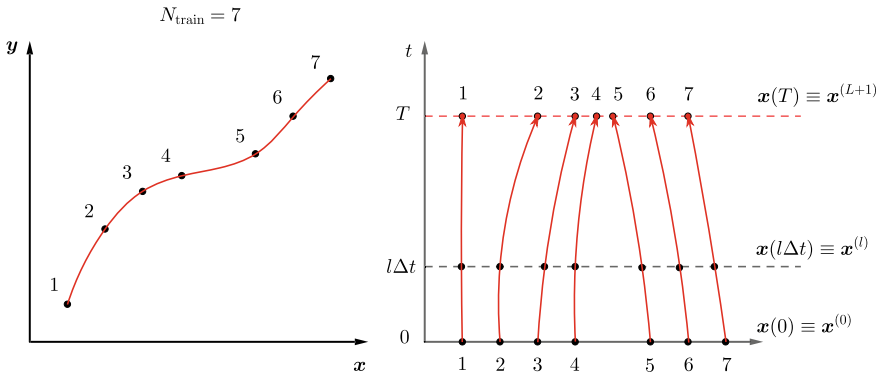


Fig. 3.6 Analogy between regression problems and Neural ODEs

Let us list the advantages and differences when comparing Neural ODEs to ResNets:

- If we interpret the number of time-steps in the Neural ODE as the number of hidden layers L in a ResNet, then the computational cost for both methods is $\mathcal{O}(L)$. This is the cost associated with performing one forward propagation and one backward propagation. However the memory cost (the cost associated with storing the weights of each layer), is different. For the neural ODE all the weights are associated with the feed-forward network used to represent the function $\boldsymbol{V}(\boldsymbol{x}, t; \boldsymbol{\theta})$. Thus the number of weights are independent of the number of time-steps used to solve the ODE. On the other hand, for a ResNet the number of weights increases linearly with the number of layers, therefore the cost of storing them scales as $\mathcal{O}(L)$.
- In Neural ODEs, we can take the limit $\Delta t \to 0$ and study the convergence, since this will not change the size of the network used to represent the RHS. However, this is not computationally feasible to do for ResNets, where $\Delta t \to 0$ corresponds to the network depth $L \to \infty$!
- ResNet uses a forward Euler type method, but in a Neural ODE one can use any time-integrator. Especially, other higher-order explicit time-integrator like the Runge-Kutta methods that converge to the "exact" solution at a faster rate.

Chapter 4
Convolutional Neural Networks

In the previous chapters, we have seen how to construct neural networks using fully-connected layers. We will now look at a different class of layers, called convolution layers [36, 59], which are very useful when handling inputs which are images. These are routinely used in network architectures designed for tasks such as classification of images into different categories [87, 98], performing semantic segmentation on images [66, 81, 108], and transforming images from one type to another [91].

4.1 Functions and Images

Consider a function $u(\boldsymbol{x})$ defined on a two-dimensional domain with $\boldsymbol{x} \in [0, a] \times [0, b] \subset \mathbb{R}^2$. We can visualize the discretized version of this function as an image $\boldsymbol{U} \in \mathbb{R}^{N_1 \times N_2}$, where

$$U[i, j] = u(ih, jh), \quad 1 \leq i \leq N_1, \ 1 \leq j \leq N_2, \ h = \text{ pixel size}. \tag{4.1}$$

Note that the image in (4.1) typically defines a grayscale image where the value of u at each pixel is just the intensity. If we work with color images, then it would be a three-dimensional tensor, with the third dimension corresponding to the red, blue and green channels. In other words, $\boldsymbol{U} \in \mathbb{R}^{N_1 \times N_2 \times 3}$.

If we want to use a fully-connected neural network (MLP) which takes as input a colored 2D image of size 100×100, then the input dimension after unravelling the entire image as a single vector would be 3×10^4, which is very large. This would in turn lead to very large connected layers which is not computationally feasible. Secondly, when unravelling the image, we lose all spatial context of the initial image. Finally, one would expect that local operations, such as edge detection, to be the same in any region of the image. Consider the weights for a fully connected layer. These would be represented by the matrix W_{ij}, where the i index represents the output of the linear transform and the j index represents the input. If the operation was the

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_4

same for every output index, we would apply the same operation for every i and therefore not need the matrix. To address all these issues, we can use the convolution operator on functions. Or, more appropriately, the discrete version of the convolution operator on the discrete version of images.

4.2 Convolutions of Functions

The convolution operator maps functions to functions. Consider the function $u(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^d$, and a sufficiently smooth kernel function $g(\boldsymbol{x})$ which decays as $|\boldsymbol{x}| \to \infty$. Then the convolution operator is given by

$$\overline{u}(\boldsymbol{x}) = \int_{\mathbb{R}^d} g(\boldsymbol{y} - \boldsymbol{x}) u(\boldsymbol{y}) \mathrm{d}\boldsymbol{y}. \tag{4.2}$$

We can interpret the convolution operator as sampling different portions of u by varying $\boldsymbol{x}$. For example, in 1D, let $u(x)$ and $g(x)$ be as shown in Fig. 4.1, and

$$\overline{u}(x) = \int_{\mathbb{R}} g(y - x) u(y) \mathrm{d}y.$$

Consider a point x_0. Then $g(y - x_0)$ shifts the kernel to the location x_0 which will sample the function u in the orange shaded region. Similarly, for another point x_1, $g(y - x_1)$ shifts the kernel to the location x_1 which will sample the function u in the green shaded region. Thus as the kernel moves, it samples u in different windows. Note that the same operation is applied regardless of the value of x. Let us now consider a few typical kernel functions.

4.2.1 Example 1

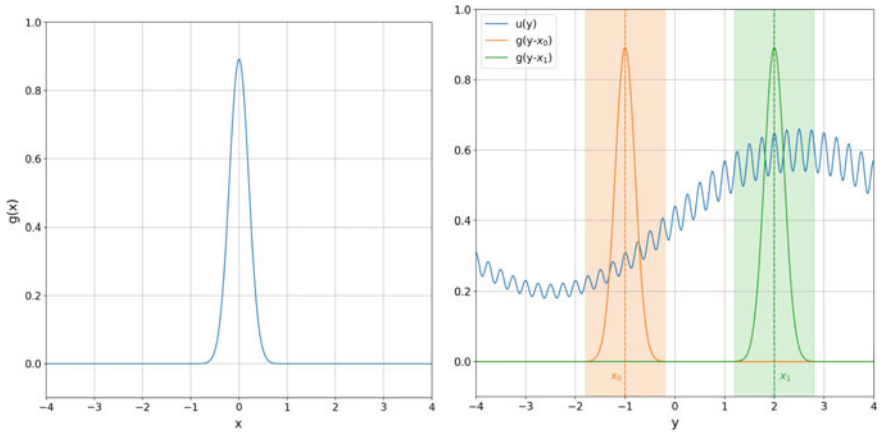
A popular choice is the Gaussian kernel

$$g(\boldsymbol{\xi}) = \rho(|\boldsymbol{\xi}|),$$

where for any scalar r,

$$\rho(r) = \frac{\exp(-r^2/2\sigma^2)}{\sqrt{(2\pi\sigma^2)^d}}.$$

Which is used as a smoothing/blurring filter. Here d is the dimension while σ denotes the spread. This kernel in 1D is shown in Fig. 4.1a for $\sigma = 0.2$. Some important properties of this kernel are:



(a) Kernel $g(x)$ (b) Sampling

Fig. 4.1 Sampling with shifted kernel in 1D

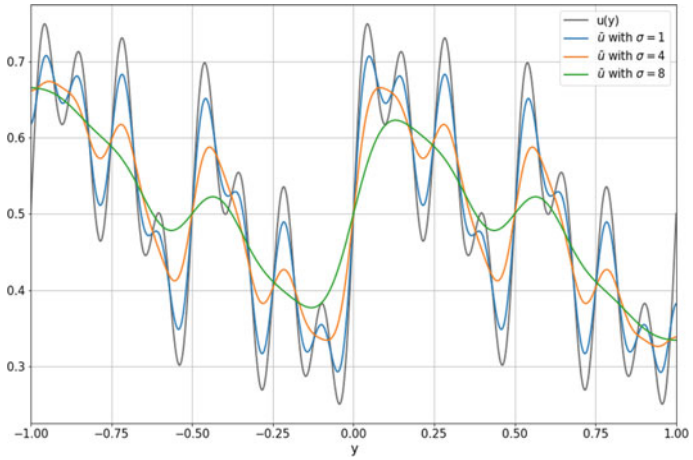


Fig. 4.2 1D convolution with Gaussian kernel

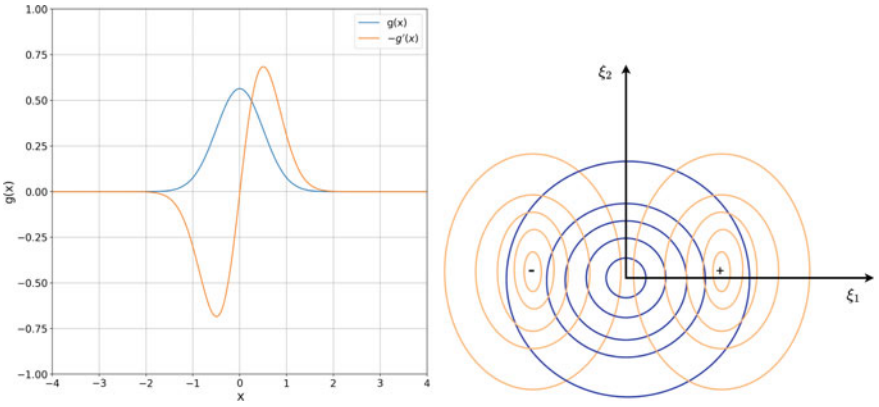
- It is isotropic, as it only depends on the magnitude of $\boldsymbol{\xi}$.
- The integral over the whole domain is unity.
- It is parameterized by σ, which represents a length-scale cut-off, as scales finer than σ are filtered out by the convolution. This smoothing effect is demonstrated in Fig. 4.2.

4.2.2 Example 2

Let us consider another example of a kernel that would produce the derivative of a smooth version of u. In 2D, we want this to look like (note: in the equation below ρ is the Gaussian kernel),

$$\begin{aligned}\overline{u}(\boldsymbol{x}) &= \frac{\partial}{\partial x_1}\left(\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\rho(|\boldsymbol{y}-\boldsymbol{x}|)u(\boldsymbol{y})dy_1dy_2\right)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{\partial\rho(|\boldsymbol{y}-\boldsymbol{x}|)}{\partial x_1}u(\boldsymbol{y})dy_1dy_2\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\underbrace{\left(-\frac{\partial\rho(|\boldsymbol{y}-\boldsymbol{x}|)}{\partial y_1}\right)}_{\text{required kernel}}u(\boldsymbol{y})dy_1dy_2\end{aligned} \tag{4.3}$$

This kernel is shown in both 1D and 2D in Fig. 4.3. Note that the action of this kernel looks like a smoothed finite difference operation. That is the region to the left of the center of the kernel is weighted by a negative value and the region to the right is weighted by a positive value and the integral involves summing the contributions from these regions to compute the finite difference.



(a) 1D (b) 2D, with a derivative along the 1-direction

Fig. 4.3 Derivative kernel. The blue curve denotes the Gaussian kernel and the orange curve denotes the derivative

4.3 Discrete Convolutions

In this section, we derive an expression for the discrete version of the convolution. To get started we represent u, $\bar{u}$ and the kernel g with their discrete counterparts. That is,

$$U[i,j] = u(ih, jh), \quad -\infty \leq i \leq \infty, \ -\infty \leq j \leq \infty, \ h = \text{ pixel size} \tag{4.4}$$

$$\bar{U}[i,j] = \bar{u}(ih, jh), \quad -\infty \leq i \leq \infty, \ -\infty \leq j \leq \infty, \tag{4.5}$$

$$G[i,j] = g(ih, jh), \quad -\infty \leq i \leq \infty, \ -\infty \leq j \leq \infty. \tag{4.6}$$

As in the continuous case, we will assume that G vanishes after a certain distance

$$G[i,j] = 0, \quad |i|, |j| > \bar{N}.$$

To evaluate the discrete convolution in 2D, consider (4.2) and discretize it using a suitable quadrature rule. Then, using the expressions above we will have

$$\overline{U}[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} G[m-i, n-j]U[m,n] \tag{4.7}$$

where we have absorbed the measure h^2 into the definition of the kernel. Let $m' = m - i$ and $n' = n - j$. Then

$$\overline{U}[i,j] = \sum_{m'=-\infty}^{\infty} \sum_{n'=-\infty}^{\infty} G[m', n']U[i+m', j+n']. \tag{4.8}$$

Since G vanishes outside the limit of its width, the limits of the sum are reduced by excluding all the pixels over which the convolution will be zero,

$$\overline{U}[i,j] = \sum_{m'=-\bar{N}}^{\bar{N}} \sum_{n'=-\bar{N}}^{\bar{N}} G[m', n']U[i+m', j+n']. \tag{4.9}$$

This is the final expression for a discrete convolution.

Let's consider some examples:

- A smoothing kernel would be an approximation of the Gaussian kernel shown in Fig. 4.3b. Hence its values would be

$$\frac{1}{8}\begin{bmatrix} \frac{1}{4} & 1 & \frac{1}{4} \\ 1 & 3 & 1 \\ \frac{1}{4} & 1 & \frac{1}{4} \end{bmatrix} \approx \text{Gaussian kernel with some } \sigma$$

- Kernels that lead to the derivative along the x_1-direction and x_2-direction are given by (see Fig. 4.3b for a derivative along the x_1-direction)

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

- Similarly, the second derivatives along the x_1 and x_2-directions are given by kernels of the form

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & -1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

- The Laplacian is given by the kernel of the type

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Remark 4.3.1 Equation 4.9 is precisely how a convolution is applied in deep learning. Thus, the convolution is entirely determined by

$$G[m, n], \quad |m|, |n| \leq \bar{N},$$

which become the learnable parameters of the convolution layer, with the number of parameters being $(2\bar{N} + 1)^2$. Further, the kernel of the convolution has a **width** $k = (2\bar{N} + 1)$ in each direction.

Remark 4.3.2 We can have different $\bar{N}$ in different directions. That is, we can have kernels with different widths along each direction. Moreover, the widths are allowed to be even as opposed to odd numbers considered above.

4.4 Connection to Finite Difference Approximations

There is a very strong connection between the concept of convolution and the stencil of a finite difference scheme. This is made clear in the discussion below.

Say some function $u(x_1, x_2)$ is represented on a finite grid, where the grid points are indexed by (i, j) with a grid size h. Using the notation $U[i, j] = u(x_1^i, x_2^j)$ and applying a Taylor series expansion about (i, j), we have

$$\frac{U[i+1, j] - U[i-1, j]}{2h} = \frac{u(x_1^{i+1}, x_2^j) - u(x_1^{i-1}, x_2^j)}{2h} = \frac{\partial}{\partial x_1} u(x_1^i, x_2^j) + \mathcal{O}(h^2).$$

The operation above is *identical to* the operation of a discrete convolution with weights given by

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \frac{1}{2h} \approx \frac{\partial u}{\partial x_1}.$$

Thus we may say that this convolution operation approximates a derivative along the 1-direction.

Similarly, we can show that

$$\frac{U[i+1, j] - 2U[i, j] + U[i-1, j]}{h^2} = \frac{\partial}{\partial x_1^2} u(x_1^i, x_2^j) + \mathcal{O}(h^2).$$

and thus the convolution with the kernel given by

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} \frac{1}{h^2} \approx \frac{\partial u}{\partial x_1^2},$$

approximates the computation of the second derivative along the 1-direction.

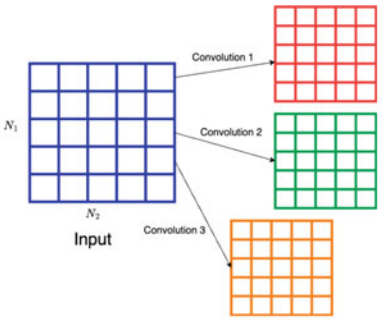
4.5 Convolution Layers

We are now ready to discuss the explicit structure and action of convolution layers. As we proceed, we should keep a few key points in mind:

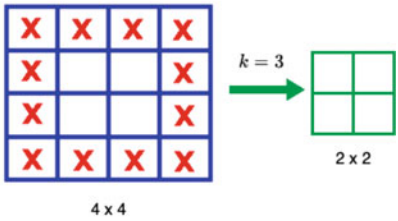
- Each convolution layer consists of several discrete convolutions, each with its own kernel.
- The weights of the kernel, which determine its action (smoothing, first derivative, second derivative etc.), are learnable parameters and are determined when training the network. Thus the way to think about the learning process is that the network learns the operations (convolutions) that are appropriate for its task. The task can be a classification problem, for example.

Let us assume we have an $N_1 \times N_2$ (grayscale) image as an input to a convolution layer comprising multiple convolutions. Each convolution will generate a different image, as shown in Fig. 4.4a. The trainable parameters of this layer are the weights of each convolution kernel. If we assume that the width of the kernels is $k = (2\bar{N} + 1)$ in each direction, and there are P kernels, then the number of trainable weights will be $P \times k^2$.

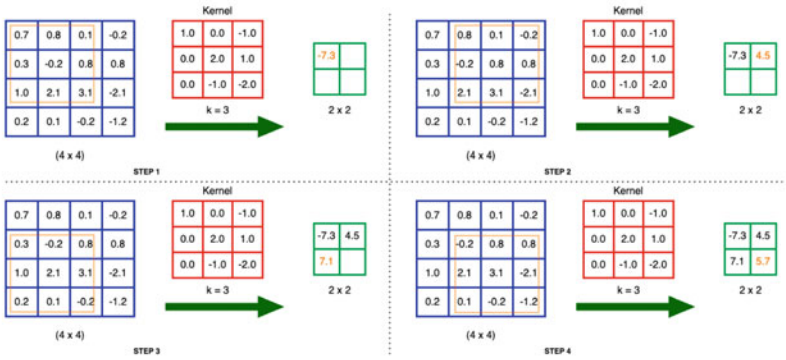
Next let us consider the size of the output image after applying a single kernel operation. Note that we will not be able to apply the kernel on the boundary pixels since there are no pixel-values available beyond the image boundary (see Fig. 4.4b). Thus, we will have to skip $\bar{N}$ pixels at each boundary when applying the kernel,



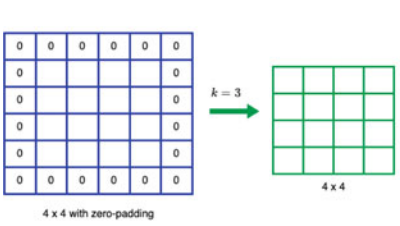
(a) Action of a convolution layer



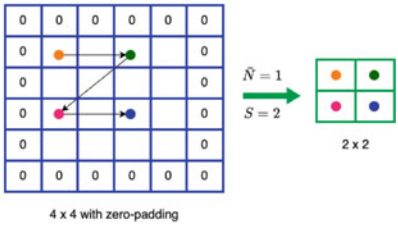
(b) Convolution without padding



(c) Example of a convolution action (step-by-step)



(d) Convolution with zero-padding



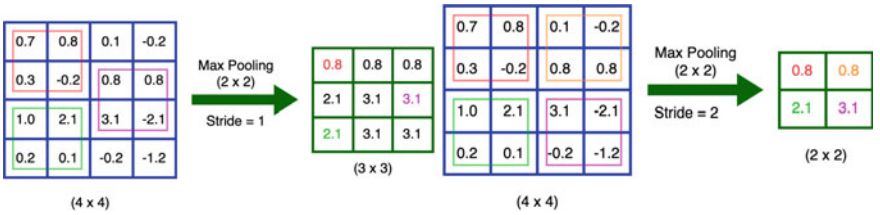
(e) Convolution with zero-padding and stride 2

Fig. 4.4 Action of a convolution layer/kernel

leading to an output image of shape $(N_1 - \bar{N}) \times (N_2 - \bar{N})$. A concrete example of a kernel of width 3 acting on an image of shape 4×4 is shown step-by-step in Fig. 4.4c.

One way to overcome this is by *padding* the image with $\bar{N}$ pixels with zero value on each edge. Now we can apply the kernel on the boundary pixels and the output image will be the same size as the input image, as can be seen in Fig. 4.4d.

Another feature of convolutions is known as the *stride* which determines the number of pixels by which the kernel is shifted as we move over the image. In the examples above, the stride was 1 in both directions. In practice, we can choose a



(a) Stride 1 (b) Stride 2

Fig. 4.5 Max pooling applied to an image over patches of size (2×2)

stride > 1 which will further shrink the size of the output image. For instance, if stride was taken as S in each direction (with zero-padding applied), then the output image size would reduce by a factor of S in each direction (see Fig. 4.4e).

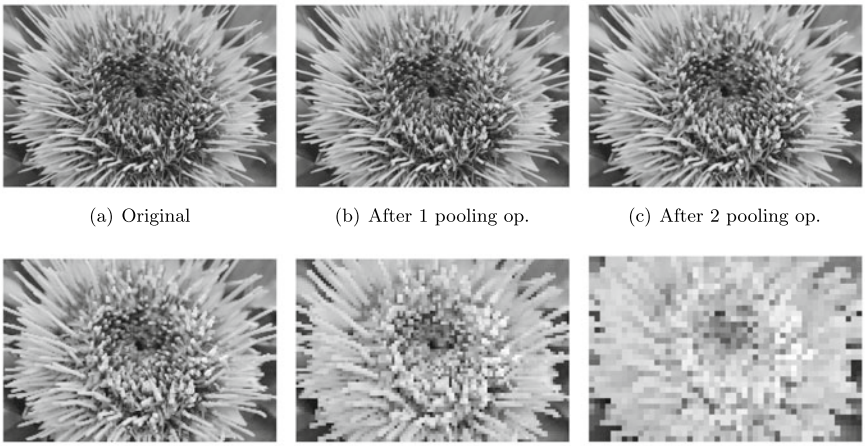
4.5.1 Average and Max Pooling

Pooling operations are generally used to reduce the size of an image, and allowing one to step through different scales of the image. If applied on an image of size $N \times N$ over patches of size $S \times S$, the new image will have dimensions $\frac{N}{S} \times \frac{N}{S}$, where S is the stride of the pooling operation. This is shown in Fig. 4.5 for $S = 2$. Note it is typical to select the patch of pixels over which the max or average is computed to be $(S \times S)$, where S is the stride. This is true for Fig. 4.5b but not for Fig. 4.5a.

Also, we show in Fig. 4.6 how pooling allows us to move through various scales of the image, where the image gets coarser as more pooling operations are applied. Note that pooling operations do not have any trainable parameters. The pooling operation has strong analog in similar operators that are used when designing multigrid preconditioners for solving linear systems of algebraic equations.

4.5.2 Convolution for Inputs with Multiple Channels

Assume that the input to a convolution layer is of size $N_1 \times N_2 \times Q$, where Q is the number of channels in the input image. Then a convolution layer applies P convolutions on this input and give an output of size $M_1 \times M_2 \times P$. Note that both the spatial resolution as well as the number of channels of the output image might be different from the input image. Furthermore, if a single convolution in the layer uses kernels of width $k = 2\bar{N} + 1$, then each kernel will be of the shape $k \times k \times Q$, i.e., the kernel will have $k \times k$ weights for each of the Q input channels of the input image. Each convolution will act on the input to give an output image of shape



(a) Original (b) After 1 pooling op. (c) After 2 pooling op.

(d) After 3 pooling op. (e) After 4 pooling op. (f) After 5 pooling op.

Fig. 4.6 Max pooling applied repeatedly to an image over patches of size (2×2) with stride 2

$M_1 \times M_2 \times 1$. The output of each convolution are stacked together to give the final output of the convolution layer. This can be written as,

$$\bar{U}[i, j, p] = \sum_{m=-\bar{N}}^{\bar{N}} \sum_{n=-\bar{N}}^{\bar{N}} \sum_{q=1}^{Q} g_p[m, n, q] U[i+m, j+n, q], \qquad 1 \le i \le M_1,\ 1 \le j \le M_2,\ 1 \le p \le P,$$

where g_p is the kernel of the p-th convolution in the layer. Note that the total number of trainable parameters will be $P \times k^2 \times Q$. This is the type of convolutional layer most frequently encountered in a convolutional neural network, which is described in the following section.

4.6 Convolution Neural Network (CNN)

Now let's put all the elements together to form the full network. Consider an image classification problem. Then the functional form of a typical CNN used for this task is given by $\hat{\boldsymbol{y}} = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta})$ where $\boldsymbol{x} \in \mathbb{R}^{N_1 \times N_2 \times Q}$ is the input image with Q channels, while $\hat{\boldsymbol{y}} \in \mathbb{R}^C$ is the predicted probability vector whose i-th component of denotes the probability that the input image belongs the i-th class among a total of C classes. The true $\boldsymbol{y}$ are typically one-hot encoded.

A possible architecture of this network is shown in Fig. 4.7. This consists of a number of convolution layers followed by pooling layers, which will reduce the spatial resolution of the input image while increasing the number of channels. The output of the final pooling layer is flattened to form a vector, which is then passed though a number of fully connected layers with some activation function (say ReLU).

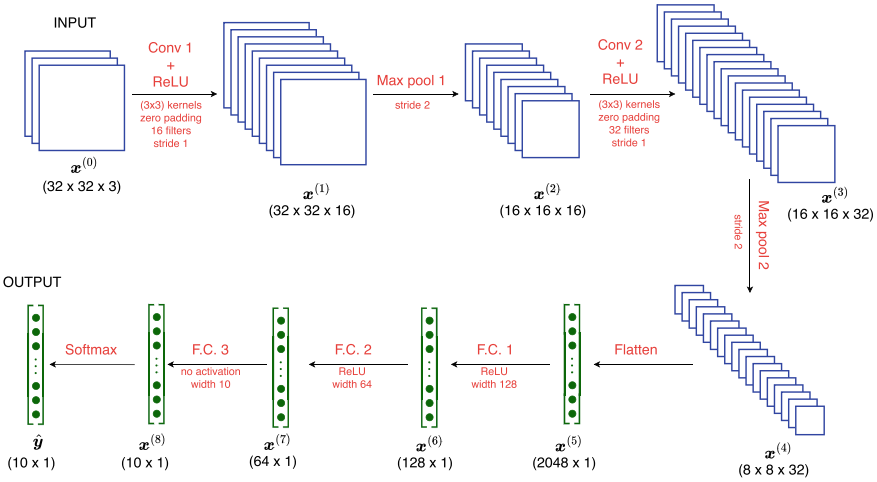


Fig. 4.7 Example of a CNN architecture for an image classification problem, for 10 classes

The final fully connected layer reduces the size of the vector to C (which is taken to be 10 in the figure), which is then passed through a softmax function to generate the predicted probability vector $\hat{\boldsymbol{y}}$. Since we are solving a classification problem, the loss function is taken to be the cross-entropy function

$$\Pi(\boldsymbol{\theta}) = -\sum_{i=1}^{N_{\text{train}}}\sum_{c=1}^{C}\left[y_c^{(i)}\log(\hat{y}_c^{(i)})\right] = -\sum_{i=1}^{N_{\text{train}}}\sum_{c=1}^{C}\left[y_c^{(i)}\log\left(\mathcal{F}_c(\boldsymbol{x}^{(i)};\boldsymbol{\theta})\right)\right].$$

We train the CNN by trying to find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}}\Pi(\boldsymbol{\theta})$ with the final network being $\boldsymbol{\mathcal{F}}(\boldsymbol{x};\boldsymbol{\theta}^*)$.

We make some important remarks:

1. The convolution operation is also a linear operation on the input, as is the case for a fully connected layer. The only difference is that in a fully-connected layer, the weights connect every pixel in the output to every pixel of the input, while in a convolution layer the weights connect one pixel of the output only to a patch of pixels in the input. Furthermore, the same weights are applied on each patch of the input.
2. In the CNN shown in Fig. 4.7, the convolution layers can be interpreted as encoding information about the input image, while the fully connected layers can be interpreted as using this information to solve the classification problem. This is why in the literature, convolution layers are said to perform *feature selection*. Further, the part of the network leading up to the "flattened" vector is sometimes referred to as the *encoder*.

3. It is common practice to apply activation to the output of the convolution layer along with a bias

$$x^{(l+1)}[i, j, p] = \sigma(\sum_{m,n,c} g_p[m, n, c]x^{(l)}[i + m, j + n, c] + b_p)$$

where a single bias b_p is used for a given output channel p.
4. In the example above we considered an image classification problem. That is, the network was a transform from an image to a class label. We can think of other similar cases. For example, when the network is a transform from an image to a real number. This might have several useful applications in computational physics. Consider the case where you want to create an enstrophy calculator. That is a network that will take as input images of the velocity components of a fluid defined on a grid, and produce as output the integral of the square of the vorticity (called the enstrophy) over the entire domain. Another example would be a network that takes as input the components of the displacement of an elastic solid and produces as output the total strain energy stored within the solid.
5. The architecture we have considered allows us to transform images to vectors, which is useful in problems involving image classification, image analysis, etc. However, there is another architecture that does the opposite, i.e., maps vectors into images. This is useful in applications involving image synthesis. Finally, by putting these two architecture together, we can transform an image to a vector and back to another image. Such image-to-image transformations are useful in applications such as image semantic segmentation. These ideas are described in the following Sections.
6. It is worth taking a moment to analyze how convolution layers act on images and why they are so useful. When dealing with images input in the context of deep learning, a first naive approach could be to flatten the image and feed it to a regular fully connected MLP. However, this would lead to different problems. In fact, for regular images, the size of the flattened input would be extremely large. In that case, we would have two possibilities when defining the architecture of the network:

   (a) We can size the first layers of the network to have a width comparable to the (large) dimension of the input.
   (b) We can have a sharp decrease in the width of the second hidden layer.

   Either of the strategies would lead to issues. In the first case, the size of the network would be too large. Hence, there would be too many trainable parameters which would require an unrealistic amount of data to train the network. In the second case, the compression of information happening between the first two layers would be too aggressive, making it very hard for the network to learn how to extract the right features across different images. Moreover, important spatial relationship among pixels (like edges, shapes, etc.) are lost by flattening an image. Ideally we would like to leverage these relations as much as possible, since they carry important spatial information. Convolution layers can solve both issues.

They allow the input to be a 2D image, while drastically decreasing the number of learnable parameters needed for the feature extraction task. In fact, kernels introduce a limited number of parameters compared to a classic fully connected layer. Since the same kernel is applied at different pixel locations in an image, i.e. *parameter sharing*, they utilize the computational resources in an efficient and smart manner.

4.7 Transpose Convolution Layers

We have seen how convolution and pooling layers can be used to scale down images. We now consider layers that do the opposite, i.e., scale up images. To understand what this operation would look like, let us look at a few examples.

1. Consider a 1D image of size 4

$$\text{Input} = \begin{bmatrix} u_1, & u_2, & u_3, & u_4 \end{bmatrix},$$

and a kernel of size 3×1

$$\boldsymbol{k} = \begin{bmatrix} x, & y, & z \end{bmatrix}.$$

Consider a convolution layer with the kernel $\boldsymbol{k}$, stride 1 and zero-padding layer of size 1. Then, the output of the layer acting on the input is

$$\text{Output} = \begin{bmatrix} yu_1 + zu_2, & xu_1 + yu_2 + zu_3, & xu_2 + yu_3 + zu_4, & xu_3 + yu_4 \end{bmatrix}$$

The steps involved in the convolution operator are: pad, dot-product, stride. Note that using padding and stride 1 have ensured the output has the same size as the input.

2. Consider another convolution with the same kernel $\boldsymbol{k}$, zero-padding layer 1 but stride 2. Then, the output of the layer acting on the same input as earlier is

$$\text{Output} = \begin{bmatrix} yu_1 + zu_2, & xu_2 + yu_3 + zu_4 \end{bmatrix}$$

Note that the size of the output has reduced by a factor of 2. In other words, increasing the stride has allowed us to downsample the input. The question we want to ask is whether we can perform an upsampling in a similar way? This can indeed be done by transposing every operation in a convolution layer.

- Instead of using a dot-product (inner-product), we will use an outer-product.
- Instead of skipping pixels in the input (stride) we will skip pixels in the output.
- Instead of padding, we will need to crop the output.

3. Let us now see an example of a transpose convolution layer. Consider a 1D input image of size 2×1

$$\text{Input} = \begin{bmatrix} u_1, & u_2 \end{bmatrix}$$

and a kernel of size 3×1

$$\boldsymbol{k} = \begin{bmatrix} x, & y, & z \end{bmatrix}.$$

if we perform the outer-product of the input with $\boldsymbol{k}$, we will get

$$\text{Outer product} = \begin{bmatrix} u_1 x & u_1 y & u_1 z \\ u_2 x & u_2 y & u_2 z \end{bmatrix}.$$

If we use a stride of 2 (or in general s), we need to shift the entries in the i-th row of the outer-product to the right by $2(i-1)$ (or in general $s(i-1)$), and fill the empty spaces in the final matrix with zeros

$$\text{Striding} = \begin{bmatrix} u_1 x & u_1 y & u_1 z & 0 & 0 \\ 0 & 0 & u_2 x & u_2 y & u_2 z \end{bmatrix}.$$

After striding is performed we need to add the entries in each column and crop the vector to get the output

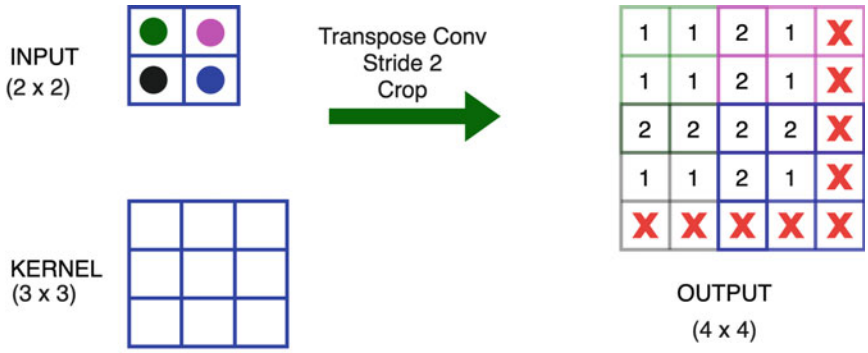
$$\text{Output} = \text{Crop}(\begin{bmatrix} u_1 x, & u_1 y, & u_1 z + u_2 x, & u_2 y, & u_2 z \end{bmatrix}) = \begin{bmatrix} u_1 x, & u_1 y, & u_1 z + u_2 x, & u_2 y \end{bmatrix}$$

where we have cropped out the last few elements (by convention) to get an output which has 2 times the size of the input.

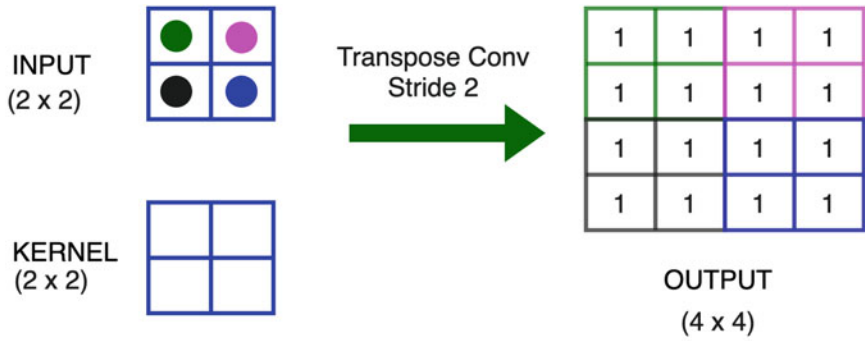
4. We consider transpose convolution in 2D applied on a 2D image of size (2×2). The kernel is taken to be of shape (3×3) with stride 2 and padding (cropping). The action of this transpose convolution is shown in Fig. 4.8a, where we first obtain an image of size (5×5) which is then cropped to give an output of size (4×4). Note that the output pixels get an unequal contribution from the various patches (indicated by numbers inside each cell/pixel of the output), which leads to an undesirable phenomena called *checker-boarding*. Checker-boarding refers to pixel-to-pixel variations in the values of the output image. A nice discussion on this, with a useful visual toolbox can be found here [74].
5. One way to avoid checker-boarding, is by ensuring that the filter size is an integer multiple of the stride. Let us repeat the previous example but with a (2×2) kernel. The operation is illustrated in Fig. 4.8b. In this case, we do not need to pad (crop) and each output pixel has an equal contribution.

We make some remarks:

1. Transpose convolution layers are also called fractionally-strided layers, because for every one step in the input, we take greater than one step in the output. This is the opposite of what happens in a convolution layer. In a convolution layer, we take step greater than one step in the input image, for each step in the output image.



(a) kernel size (3×3), checker-boarding



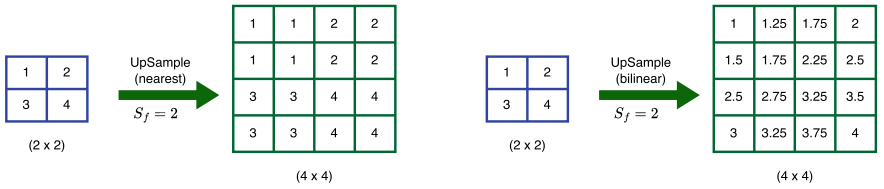
(b) kernel size (2×2), no checker-boarding

Fig. 4.8 Example of a transpose convolution operation. The cells marked with red X's in the output are cropped out. The numbers denote the number of patches that contributed to the value in the output pixel

2. Transpose convolutions are a tool of choice for upscaling *through learnable parameters*.
3. Upscaling is typically done with a reduction in the number of channels, which is once again the opposite of what is done in convolution layers.

4.8 UpSampling

An alternate and simpler way to upscale an image is by using an UpSampling layer defined by the hyperparameter S_f known as the *scale factor*. UpSampling layers can be seen as inverses of pooling layers, and also do not have any trainable parameters. When an image of shape $N_1 \times N_2 \times Q$ is fed into an UpSampling layer, the output image will have the shape $(S_f N_1) \times (S_f N_2) \times Q$, i.e., the image resolution is scaled by a factor of S_f in each spatial direction. Another way to interpret this action is that



(a) Nearest neighbour (b) Bilinear

Fig. 4.9 UpSampling applied to an image with a scale factor $S_f = 2$

each (i, j)-th pixel in the input image generates an independent patch of $S_f \times S_f$ of pixels in the output image, the values of which are determined by a suitable interpolation algorithm. The two popular UpSampling interpolation algorithms are:

- **Nearest neighbour interpolation**: This strategy involves copying the value in the (i, j)-th pixel in the input image to corresponding patch of $S_f \times S_f$ of pixels in the output image. An example of this is shown in Fig. 4.9a.
- **Bilinear interpolation:** This strategy involves using a weighted combination of the values in the (i, j)-th pixel and its neighbouring pixels in the input image to evaluate the values in the corresponding patch of $S_f \times S_f$ of pixels in the output image. An example of this is shown in Fig. 4.9b.

The UpSampling layer is typically used in combination with Convolution layers for up scaling an image, in place of a Transpose Convolution layers.

4.9 Image-to-Image Transformations

Image-to-image transformations can be seen analogous to function-to-function transformations. These types of networks are typically used in computer vision, super-resolution, style transfer, and also in computational physics where we (say) map the source field (right hand side of the PDE) to the solution of the PDE.

We will discuss a particular type of network for such transformations, which is known as U-Net [91]. In a U-Net (see Fig. 4.10) there is a downward branch which takes an input image and downscales the images using a number of convolution layers and pooling operations. As we go down this branch, the number of channels typically increases. After we reach the coarsest level, we have an upward branch that scales up the image and reduced the number of channels using upsampling or transpose convolution-type operations, to finally give the output image. In addition to these branches, the U-Net also makes use of skip connections that combines information at a particular scale in the downward branch to the information in the upward branch at the same scale (typically by concatenating along the channel-direction). In the upscaling branch of the U-Net, if you consider the activation at one point, you will see they come from two different sources. One of these is the from the

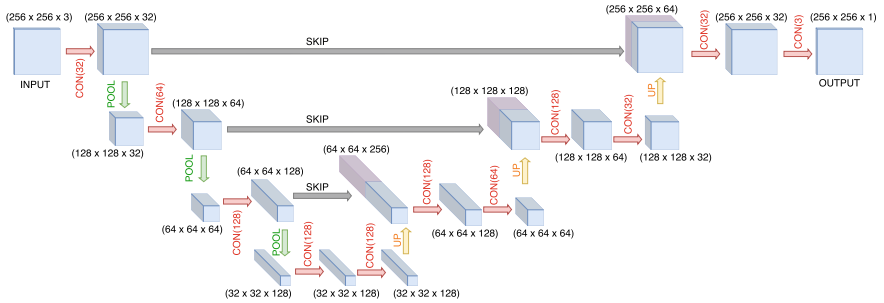


Fig. 4.10 Example of a U-Net mapping an input of shape $256 \times 256 \times 3$ to an output image of shape $256 \times 256 \times 1$. CON(P) denotes a convolution layer with P kernels and appropriate zero padding to preserve the spatial resolution; POOL denotes a max (or average) pooling layer halving the spatial resolution; UP denotes an upsampling layer that doubles the spatial resolution; SKIP denotes a skip connection that concatenates along the channel-direction

same spatial scale in the down-scaling branch of the U-Net, and the other is from the coarser scales of the upscaling branch of the U-net. The U-Net architecture shown in Fig. 4.10 maps an image of shape $256 \times 256 \times 3$ to an output image with the same spatial resolution but with a single channel. Such U-Net models are typically used for image segmentation tasks.

Remark 4.9.1 The U-net architecture shares many common features with the V-cycle that is typically used in multigrid preconditioners.

Remark 4.9.2 We can also think of a the U-net as an encoder-decoder network with the additional feature of including skip connections.

4.10 Computational Exercise: Convolutional Neural Networks (CNNs)

The objectives of this exercise are:

1. Implement, train and test a simple CNN-based model to classify the handwritten MNIST digits.
2. Recognize that the same classifier can be used to solve a classification problem in mechanics. In this problem we imagine than the MNIST digits represent a two-dimensional plate with a soft background material. Embedded within this soft background is a stiff inclusion in the form of a handwritten digit. This plate is then sheared and the two-dimensional displacement field within it is measured. We refer to this as the Mechanical MNIST data [57]. Given this displacement field within the plate, we wish to classify the inclusion as one of the ten hand-written digits.

3. Recognize that while a human expert can look at the MNIST images and easily classify them correctly, this is not the case for the Mechanical MNIST data. It is very hard for a human expert to look at the displacement fields and infer what the shape of the inclusion will look like. The question is whether a CNN-based classifier can do both these tasks well.

   Import required libraries:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
```

Gathering and Processing Data

The sources of data for the MNIST and the Mechanical MNIST problems are different. Therefore, you have to spend some time carefully gathering and processing data. This is often the case when working on a problem in deep learning. So, it is good to get used to it.

1. For the MNIST data, use the command: `torchvision.datasets.MNIST()`. You should use it twice: once for defining the training set with the train argument set to `train=True`, and another time for the testing set with `train=False`. This data set contains $N_{\text{train}} = 60{,}000$ images and labels for training, and $N_{\text{test}} = 10{,}000$ images and labels for testing. Each image is a 28×28 array of pixels and the each label takes on integer values in the interval $[0, 9]$.

```
transform = transforms.Compose([transforms.ToTensor()])
# Load MNIST training dataset
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
    transform=transform, download=True)
# Load MNIST test dataset
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
    transform=transform, download=True)
```

2. You can access the images and the labels by using `train_dataset.data` and `train_dataset.targets`, respectively.
3. The Mechanical MNIST dataset is stored at https://open.bu.edu/handle/2144/39429. Download the displacement files corresponding to loading Step 5. You will see that this dataset contains 28×28 arrays of vertical and horizontal displacements for N_{train} training and N_{test} test samples. It does not contain the corresponding labels (that is which digit does the displacement fields correspond to); however, the ordering for the samples is the same as that for the MNIST data. Therefore, we will use the labels from the MNIST data. You can download and store this data in your Google drive so that you can upload it in Colab.

```
from google.colab import drive
import os
drive.mount('/content/drive')
# Specify file paths "file_path" for train and test data
    in your Google Drive folder
train_disp_x_path = '/content/drive/My Drive/Colab
    Notebooks/file_path/summary_dispx_train_step5.txt'
train_disp_y_path = '/content/drive/My Drive/Colab
    Notebooks/file_path/summary_dispy_train_step5.txt'
test_disp_x_path = '/content/drive/My Drive/Colab
    Notebooks/file_path/summary_dispx_test_step5.txt'
test_disp_y_path = '/content/drive/My Drive/Colab
    Notebooks/file_path/summary_dispy_test_step5.txt'
```

4. Process both the MNIST and the Mechanical MNIST data so that you have:

   - For the MNIST data, numpy arrays: (a) `mnist_train` of size $N_{\text{train}} \times 1 \times 28 \times 28$ of training images, (b) `mnist_test` of size $N_{\text{test}} \times 1 \times 28 \times 28$ of test images, (c) `y_train` of size $N_{\text{train}} \times 10$ of one-hot-encoded (categorical) training labels, and (d) `y_test` of size $N_{\text{test}} \times 10$ of one-hot-encoded (categorical) training labels.
   - For the Mechanical MNIST data, numpy arrays: (a) `mnist_mech_train` of size $N_{\text{train}} \times 2 \times 28 \times 28$ of training images, and (b) `mnist_mech_test` of size $N_{\text{test}} \times 2 \times 28 \times 28$ of test images. The label arrays for the Mechanical MNIST are the same as the ones for the MNIST dataset.

   Note that this task will require things like converting text and torch tensors to numpy arrays, and reshaping and concatenating arrays to get them to be of desired shape.
5. For the first five training and test samples create a figure each that contains (a) a plot of the MNIST image, (b) plots of the two Mechancial MNIST displacements, and (c) the corresponding label. On the basis of the physical experiment, explain why the displacement images look the way they look.

Constructing CNN-based Classifiers

Construct a classifier each for the MNIST and the Mechanical MNIST data with the following architecture:

- Convolution layer with 32 filters (kernel size $= 3 \times 3$), zero padding, stride 1 and `ReLU` activation.
- A 2×2 Max. pooling layer with stride 2.
- Convolution layer with 128 filters (kernel size $= 3 \times 3$), zero padding, stride 1 and `ReLU` activation.
- A 2×2 Max. pooling layer with stride 2.

- Convolution layer with 256 filters (kernel size $= 3 \times 3$), zero padding, stride 1 and `ReLU` activation.
- A 2×2 Max. pooling layer with stride 2.
- A flattening layer.
- A fully connected layer with 256 neurons and `ReLU` activation.
- A final fully connected layer with width = number of classes = 10 and no activation. Note, we don't need to specify a soft-max activation becuase that is built into the cross entropy loss.

Given that the input image is of size 28×28, what will be the size of the intermediate tensors after each layer of the above CNN?

Training and Testing

Train the MNIST and Mechanical MNIST networks with the following parameters:

- l_2 regularization of 10^{-5}.
- optimizer = Adam.
- learning rate set to the default value.
- number of epochs = 30.
- loss function = `CrossEntropyLoss()`.
- batch size = 100.

1. While training the networks, save the training loss and accuracy after each mini-batch iteration. (a) Plot the loss versus iteration number for the MNIST and Mechanical MNIST networks on the same plot. Use a log scale for the vertical axis. (b) Plot the accuracy on the training set versus iteration number for both networks. Do not use the log scale in this plot. What do you observe?
2. For the trained networks, what is the accuracy on the test set? Which network does better? Why do you think this is the case.
3. For the MNIST network, find three cases when the prediction is incorrect. For each case plot the MNIST image, the correct label and the incorrect label. What do you observe?
4. For the Mechanical MNIST network, find three cases when the prediction is incorrect. For each case plot the Mechanical MNIST images (the displacement fields), the correct label and the incorrect label. What do you observe?

Remarks

- You can use https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html for reference.
- The command below checks whether a GPU is available. You will know that it is available if the command prints `cuda`:
  device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
  print(device)

Chapter 5
Solving PDEs with Neural Networks

Partial differential equations (PDEs) are used to model a multitude of phenomena encountered in science and engineering. However, a closed-form expression of the solution is rarely available for most practical problems. Thus, numerical algorithms are employed to approximately solve PDEs. Some commonly used methods include finite difference/volume methods, finite element methods, spectral Galerkin methods, and also deep neural networks! To better appreciate some of these methods, especially deep neural networks, let us consider a simple model problem describing the scalar advection-diffusion problem in one-dimension:

Find $u(x)$ in the interval $x \in (0, \ell)$ such that

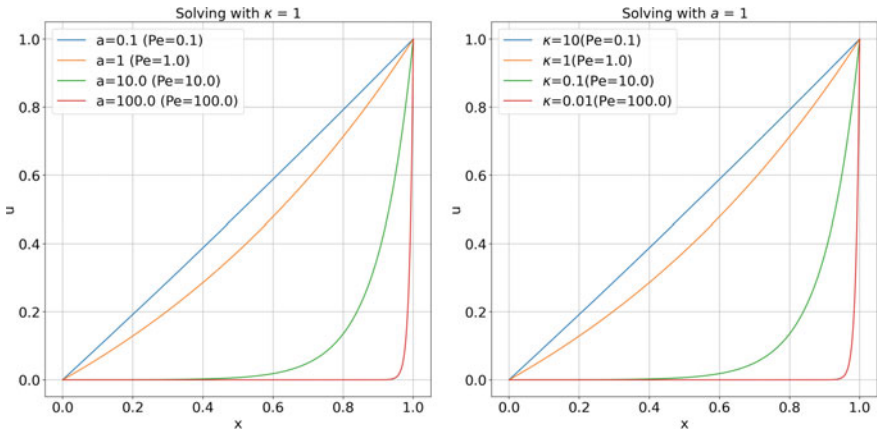
$$\begin{aligned} a\frac{\mathrm{d}u}{\mathrm{d}x} - \kappa\frac{\mathrm{d}^2u}{\mathrm{d}x^2} &= f(x), \quad x \in (0, \ell) \\ u(0) &= 0 \\ u(\ell) &= 1 \end{aligned} \tag{5.1}$$

where a denotes the advective velocity, κ is the diffusion coefficient while $f(x)$ is the source. Such equations are used to model many physical phenomena, such as the transport of pollutant by fluids, or modelling the flow of electrons through semiconductors. The multi-dimensional version of this problem will take the form

$$\begin{aligned} \boldsymbol{a} \cdot \nabla u(\boldsymbol{x}) - \kappa \Delta u(\boldsymbol{x}) &= f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{5.2}$$

Note that the model problem is a linear PDE (ODE in the one-dimensional case). When the velocity or the diffusivity depend on the concentration we are led to a nonlinear version of this equation. For example, replacing the velocity a by u leads to the viscous Burgers equation.

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_5



(a) Fixed κ (b) Fixed a

Fig. 5.1 Exact solution of (5.1) with $\ell = 1$

The solution to (5.1) for $f \equiv 0$ can be analytically written as

$$u(x) = \frac{1 - \exp(ax/\kappa)}{1 - \exp(a\ell/\kappa)}$$

where $a\ell/\kappa$ is known as the Peclet number (Pe) and measures the ratio of the strength of advection to the strength of diffusion. We plot the solution for varying values of a and κ in Fig. 5.1. Note that for small Pe, the solution is essentially a straight line. But as Pe increases, the solution starts to bend and forming a steeper boundary layer near the right boundary. The thickness of this boundary layer is given by $\delta \approx \ell/\text{Pe}$.

We will now consider a few methods to numerically solve this toy problem.

5.1 Finite Difference Method

The finite difference method [101, 105] is one of the most popular methods for solving PDEs numerically. The core idea is to approximate the derivatives in the differential equation at hand with finite difference quotients. More specifically, the key steps of a finite difference scheme are as follows:

1. Discretize the domain into a grid of points, with the goal being to find the solution at these points.
2. Approximate the derivates with finite difference approximations at these points. This leads to a system of (linear or non-linear) algebraic equations.
3. Solve this system using a suitable algorithm to find the solution.

Applying these steps to (5.1) leads to:

1. Discretize the domain into $N+1$ points, with $x_i = ih$, $0 \le i \le N$ where $h = \ell/N$. We wish to solve for $u(x_i) = u_i$. We also know from the boundary conditions that $u_0 = 0$ and $u_N = 1$.
2. Use the approximations

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) = \frac{u_{i+1} - u_{i-1}}{2h} + \mathcal{O}(h^2)$$
$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_i) = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}(h^2)$$

Note that both approximations used above are second-order accurate. They are "central difference" approximations, as they weigh points on either side of the i-th point with the same magnitude. It is worth mentioning that in the limit of large Peclet number, a central difference approximation of the advective term is not ideal since it leads to numerical instability. In such a case, an "upwind" approximation is preferred. Applying the approximations to the PDE at x_i, $1 \le i \le N-1$

$$a\frac{u_{i+1} - u_{i-1}}{2h} - \kappa\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = f_i$$
$$\iff u_{i+1}\underbrace{\left(\frac{a}{2h} - \frac{\kappa}{h^2}\right)}_{\gamma} + u_i\underbrace{\left(\frac{2\kappa}{h^2}\right)}_{\beta} + u_{i-1}\underbrace{\left(-\frac{a}{2h} - \frac{\kappa}{h^2}\right)}_{\alpha} = f_i$$

Looking at each node where the solution is unknown (recall that $u_0 = 0$ and $u_N = 1$ are known),

$$\begin{aligned} \beta u_1 + \gamma u_2 &= -\alpha u_0 + f_1 \\ \alpha u_{i-1} + \beta u_i + \gamma u_{i+1} &= f_i, \quad \forall\, 2 \le i \le N-2 \\ \alpha u_{N-2} + \beta u_{N-1} &= -\gamma u_N + f_{N-1} \end{aligned} \tag{5.3}$$

Combining all the $N-1$ equations in (5.3), we get the following linear system

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{5.4}$$

where the tridiagonal matrix $\boldsymbol{K}$ and the other vectors in (5.4) are defined as

$$\boldsymbol{K} = \begin{bmatrix} \beta & \gamma & & 0 \\ \alpha & \ddots & \ddots & \\ & \ddots & \ddots & \gamma \\ 0 & & \alpha & \beta \end{bmatrix} \in \mathbb{R}^{(N-1)\times(N-1)},$$

$$\boldsymbol{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_{N-2} & u_{N-1} \end{bmatrix}^\top \in \mathbb{R}^{N-1},$$

$$\boldsymbol{f} = \begin{bmatrix} -\alpha u_0 + f_1 & f_2 & f_3 \cdots & f_{N-2} & f_{N-1} & -\gamma u_N + f_{N-1} \end{bmatrix}^\top \in \mathbb{R}^{N-1}$$

3. Solve $\boldsymbol{u} = \boldsymbol{K}^{-1}\boldsymbol{f}$.

Note that:

- In practice, we never actually invert $\boldsymbol{K}$ as it is computationally expensive. We instead use smart numerical algorithms to solve the system (5.4). For instance, one can use the Thomas tridiagonal algorithm for this particular system, which is a simplified version of Gaussian elimination.
- We only obtain an approximation $u_i \approx u(x_i)$. To reduce the approximation error, we could reduce the mesh size h. Alternatively, we could use higher-order finite difference approximations which would lead to a "wider stencil" to approximate the derivates at each point.
- We can think of each point where we "apply" the PDE as a collocation point. This idea of applying the PDE at collocation points is shared by the next method we consider. It is also shared by the method using neural networks to solve PDEs.

5.2 Spectral Collocation Method

The spectral collocation method [63, 123] seeks a solution written as an expansion in terms of a set of smooth and global basis functions. The basis functions are chosen a priori, whereas the coefficients of the expansion are unknowns, and are computed by requiring that the numerical solution of the PDE is exact at a set of so-called collocation points. More specifically, this approach involves the following steps.

1. Select a set of global basis functions with the following properties:

   (a) It forms complete basis in the space of functions being considered.
   (b) The functions are smooth enough so that derivatives can be evaluated.
   (c) The functions and their derivatives are easy to evaluate.

   For instance, one can use the Chebyshev polynomials defined on $\xi \in (-1, 1)$, given by the following recurrence relation

$$T_0(\xi) = 1, \quad T_1(\xi) = \xi, \quad T_{n+1}(\xi) = 2\xi T_n(\xi) - T_{n-1}(\xi)$$

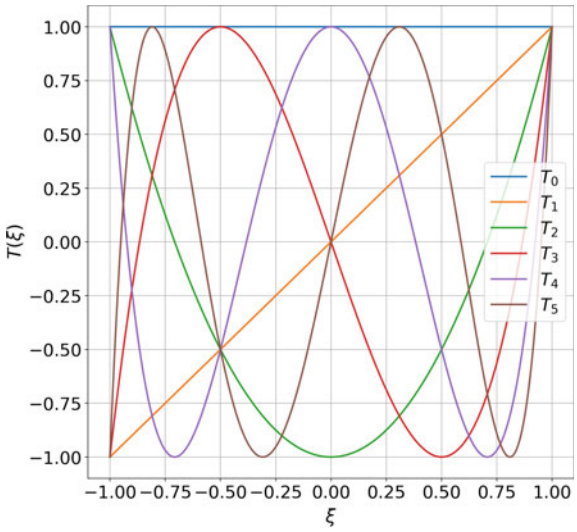


Fig. 5.2 First few Chebyshev polynomials

The first few Chebyshev polynomials are shown in Fig. 5.2. Note that this basis satisfies all the required properties listed above. It is easy to evaluate at any point because one can use the recurrence relation above and the values of the two lower-order polynomials to evaluate the Chebyshev polynomial of the subsequent order. One can also take derivatives of the recurrence relation above to obtain a recurrence relation for derivatives of all orders.

2. Write the solution as a linear combination of the basis functions $\{\phi_n(x)\}_{n=0}^{N}$

$$u(x) = \sum_{n=0}^{N} u_n \phi_n(x) \tag{5.5}$$

where u_n are the basis coefficients. For our toy problem (5.1) (assuming $\ell = 1$), we will use the Chebyshev polynomials $\phi_n(x) = T_n(2x - 1)$, where the argument is transformed to use these functions on the interval $(0, 1)$.

3. Evaluate the derivates for the PDE, which for our toy problem will be

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}x}(x) &= \sum_{n=0}^{N} u_n \phi_n'(x) = \sum_{n=0}^{N} u_n 2\, T_n'(2x-1) \\ \frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x) &= \sum_{n=0}^{N} u_n \phi_n''(x) = \sum_{n=0}^{N} u_n 4\, T_n''(2x-1) \end{aligned} \tag{5.6}$$

4. Use the boundary conditions of the PDE. For the specific case of (5.1),

$$\begin{aligned} u(0) = 0 &\implies \sum_{n=0}^{N} u_n \phi_n(0) = \sum_{n=0}^{N} u_n T_n(-1) = 0, \\ u(1) = 1 &\implies \sum_{n=0}^{N} u_n \phi_n(1) = \sum_{n=0}^{N} u_n T_n(1) = 1 \end{aligned} \tag{5.7}$$

which leads to 2 linear equations for $N+1$ coefficients. We then consider a set of (suitably chosen) nodes x_i, $1 \leq i \leq N-1$ in the interior of the domain, i.e. the collocation points, and use the derivatives expressions from (5.6) in the PDE evaluated at these $N-1$ nodes

$$\begin{aligned} & a \sum_{n=0}^{N} u_n \phi_n'(x_i) - \kappa \sum_{n=0}^{N} u_n \phi_n''(x_i) = f(x_i) \\ \implies & \sum_{n=0}^{N} u_n \left(2a T_n'(2x_i - 1) - 4\kappa T_n''(2x_i - 1) \right) = f(x_i) \end{aligned} \tag{5.8}$$

This leads to an additional $N-1$ equations for the $N+1$ coefficients. Combining (5.7) and (5.8) leads to the following linear system

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{5.9}$$

where

$$\begin{aligned} \boldsymbol{K} &\in \mathbb{R}^{(N+1)\times(N+1)}, \\ \boldsymbol{u} &= \begin{bmatrix} u_0 & u_2 & \cdots & u_{N-1} & u_N \end{bmatrix}^\top \in \mathbb{R}^{N+1}, \\ \boldsymbol{f} &= \begin{bmatrix} 0 & f(x_1) & f(x_1) & \cdots & f(x_{N-2}) & f(x_{N-1}) & 1 \end{bmatrix}^\top \in \mathbb{R}^{N+1} \end{aligned}$$

5. Solve $\boldsymbol{u} = \boldsymbol{K}^{-1}\boldsymbol{f}$.

We need to choose the collocation/quadrature points x_i properly, so that $\boldsymbol{K}$ has desirable properties that make the linear system (5.9) easier to solve. These include invertibility, positive-definiteness, sparseness, etc.

Remark 5.2.1 The method is called a collocation method as the PDE is evaluated at the specific collocation/quadrature points x_i.

Remark 5.2.2 When working with a non-linear PDE, we will end up with a non-linear systems of algebraic equations for the coefficients $u_0, \ldots, u_N$, which is typically solved by Newton's method.

Let us look at a least-square variant for finding the coefficients of the expansion of the spectral methods. As done earlier, we still represent the solution using (5.5) and compute its derivates. Then, the coefficients $\boldsymbol{u}$ are found by minimizing the following loss function

$$\begin{aligned}\Pi(\boldsymbol{u}) &= \Pi_{\text{int}}(\boldsymbol{u}) + \lambda \Pi_{\text{bc}}(\boldsymbol{u}) \\ \Pi_{\text{bc}}(\boldsymbol{u}) &= \left| \sum_{n=0}^{N} u_n \phi_n(0) - 0 \right|^2 + \left| \sum_{n=0}^{N} u_n \phi_n(1) - 1 \right|^2 \\ \Pi_{\text{int}}(\boldsymbol{u}) &= \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left| a \sum_{n=0}^{N} u_n \phi_n'(x_i) - \kappa \sum_{n=0}^{N} u_n \phi_n''(x_i) - f(x_i) \right|^2\end{aligned} \tag{5.10}$$

This can be solved using any of the gradient-based methods we have seen in Chap. 2. This approach is especially useful when treating non-linear PDEs. In fact, in those cases it is not be possible to write a linear system in terms of the coefficients such as (5.9). A few things to note here

- λ is a parameter used to scale the interior loss and boundary loss differently.
- The number of interior points x_i can be chosen independently of the number of basis functions. In other words, N_{train} does not have to be the same as N.
- We will see in the next section how this variant of the spectral method is very similar to how deep neural networks are used to solve PDEs.

5.3 Physics-Informed Neural Networks (PINNs)

The idea of learning the solution of a PDE using a neural network constrained by structure of the PDE operator was first considered in the early 90 s by Dassanayake and Phan-Thien [23], where they solved simple PDEs on fixed meshes. This was closely followed by the work by Lagaris et al. [54, 55] where the networks were constrained to strongly satisfy the boundary conditions. With the renewed interest in using machine learning tools in solving PDEs, this idea was rediscovered in 2019 by Raissi et al. [85], and was given the term PINNs (physics-informed neural networks). The basic idea of PINNs is similar to regression, except that the loss function $\Pi(\boldsymbol{\theta})$ contains derivate operators arising in the PDE being considered. We outline the main steps below for a one-dimensional scalar PDE, which can easily be extended to multi-dimensional systems of PDEs. We recommend that the reader thinks about the similarities and differences between this method and the spectral collocation method described in the previous section.

1. Select a neural network as a function representation of the PDE solution:

$$u(x; \boldsymbol{\theta}) = \mathcal{F}(x; \boldsymbol{\theta}). \tag{5.11}$$

   Some crucial properties required by this representation are:

   (a) Do we have completeness with the representation, i.e., can we accurately approximate the necessary class of function using the representation? The

answer is **yes**, because of the universal approximation theorems of neural networks (see Sect. 2.3.1).

(b) Is the representation smooth? The answer is **yes** provided the activation function is smooth, such as tanh, sin, etc. Note that we cannot use ReLU since it does not possess smooth derivatives.

(c) Is it easy to evaluate? The answer is **yes**, due to a quick forward propagation pass using neural networks (with a reasonable size).

(d) Is it easy to evaluate derivatives? The answer is **yes**, due to back-propagation. This will be discussed in detail below.

2. Given the representation (5.11), we need to find $\boldsymbol{\theta}$ such that the PDE is satisfied in some suitable form. Compare this with spectral collocation approximation given by (5.5), where we need to determine the coefficients u_n. Note that while the dependence on the coefficients u_n in (5.5) is linear, the dependence on $\boldsymbol{\theta}$ in (5.11) can be highly non-linear.
3. Next we want to find the derivatives of the representation. Consider the computational graph of the network as shown in Fig. 5.3. It comprises alternate steps of affine transformations and component-wise nonlinear transformation. The derivative of the output with respect to the input can be evaluated by back-propagation. The graph in Fig. 5.3 is obtained by simply setting $\Pi = \boldsymbol{x}^{(L+1)}$ in the graph shown in Fig. 2.10. Further, once we recognize that $\frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{x}^{(L+1)}} = \boldsymbol{I}$, the identity matrix, we can easily read from this graph that

$$\frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{x}^{(0)}} = \boldsymbol{W}^{(1)T} \boldsymbol{S}^{(1)} \boldsymbol{W}^{(2)T} \boldsymbol{S}^{(2)} \cdots \boldsymbol{W}^{(L)T} \boldsymbol{S}^{(L)} \boldsymbol{W}^{(L+1)T} \boldsymbol{S}^{(L+1)}$$

Hence, the evaluation of $\frac{\mathrm{d}u}{\mathrm{d}x}$ requires the extention of the original graph with a backward branch used to evaluate the derivative of the activation function for each component of the vectors $\boldsymbol{\xi}^{(l)}$ (see Fig. 5.3). The second derivative $\frac{\mathrm{d}^2u}{\mathrm{d}x^2}$ is evaluated by performing back-propagation of the extended graph. To evaluate higher order derivatives, the graph will need to be extended further in a similar manner. This is what happens behind the scenes in Pytorch when a call to `autograd`[1] is made.
4. Insert the functional representation of the solution (5.11) into the PDE to find the parameters $\boldsymbol{\theta}$. To do this, we first define a set of points $\mathcal{S} = \{x_i : 1 \leq i \leq N_{\text{train}}\}$ used to train the network, analogous to the set of collocation points in the spectral collocation methods. Thereafter, we need to define the loss function (specialized to our toy problem (5.1))

[1] For details see https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html.

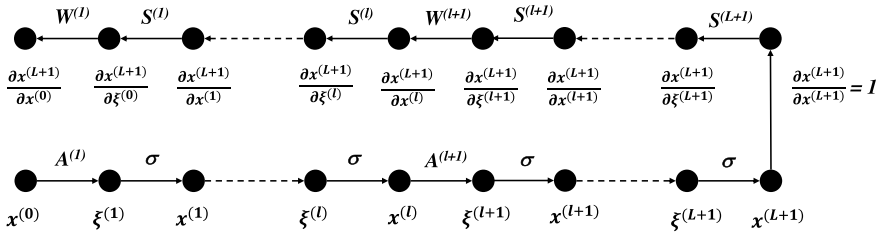


Fig. 5.3 Extended graph to evaluate derivatives with respect to network input

$$
\begin{aligned}
\Pi(\boldsymbol{\theta}) &= \Pi_{\text{int}}(\boldsymbol{\theta}) + \lambda_b \Pi_{\text{b}}(\boldsymbol{\theta}), \\
\Pi_{\text{b}}(\boldsymbol{\theta}) &= (\mathcal{F}(0;\boldsymbol{\theta}) - 0)^2 + (\mathcal{F}(1;\boldsymbol{\theta}) - 1)^2 , \\
\Pi_{\text{int}}(\boldsymbol{\theta}) &= \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left(a\mathcal{F}'(x_i;\boldsymbol{\theta}) - \kappa \mathcal{F}''(x_i;\boldsymbol{\theta})_n - f(x_i) \right)^2 .
\end{aligned}
\tag{5.12}
$$

After training the network, i.e. solving the minimization problem $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$, the solution writes $u^*(x) = \mathcal{F}(x;\boldsymbol{\theta}^*)$. Note that this is exactly what is done for the least squares variant of the spectral collocation method, where the coefficients u_n are solved by minimizing a similar loss.

We make a few remarks:

- If we find a $\boldsymbol{\theta}^*$ for which $\Pi(\boldsymbol{\theta}^*) = 0$, this implies $\Pi_{\text{int}}(\boldsymbol{\theta}^*) = 0$ and $\Pi_{\text{b}}(\boldsymbol{\theta}^*) = 0$. In other words, the PDE residuals are zero at the collocation points. This will lead to a good solution as long as the collocation points cover the domain well. At any other non-collocation points in the domain, the PINN solution can be treated as an interpolated value. This is similar to what happens with the spectral collocation approximation (5.5).
- There are various ways to improve the accuracy of PINNs, such as
  - Increasing the number of collocation points.
  - Changing the hyper-parameter λ_b weighting the boundary loss.
  - Increasing the size of the network i.e., increasing N_θ.
- The boundary conditions (BCs) of a differential equation carry fundamental physical properties of the phenomena we are trying to describe, and it is paramount that those are satisfied by our numerical solution. In the framework of PINNs, BCs are enforced as a soft constraint via the penalization term $\Pi_{\text{b}}(\boldsymbol{\theta})$. Hence, the hyper-parameter λ_b plays a crucial role in the training of the network, as it balances the interplay between the two loss terms during the minimization process. If the gradients of the different loss terms are not adequately scaled, the convergence to a solution that satisfies both the BCs and the PDE itself can be extremely slow. This is particularly exacerbated for stiff PDEs. To address this issue, different self-

adaptive techniques to tune the value of λ_b during training have been proposed [11, 110], including self- and residual-based attention [4, 64].

5.4 Extending PINNs to a More General PDE

Consider a general PDE problem: Find the solution $\boldsymbol{u} : \Omega \subset \mathbb{R}^d \to \mathbb{R}^D$ such that

$$\begin{aligned} L(\boldsymbol{u}(\boldsymbol{x})) &= \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\ B(\boldsymbol{u}(\boldsymbol{x})) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{5.13}$$

where L is the differential operator, $\boldsymbol{f}$ is the known forcing term, B is the boundary operator, and g is the non-homogeneous part of boundary condition (also prescribed).

As an example, we can consider the three-dimensional incompressible Navier-Stokes equation solving for the velocity field $\boldsymbol{v} = [v_1, v_2, v_3]$ and pressure p on $\Omega = \Omega_S \times [0, T]$. Here Ω_S is the three dimensions spatial domain and $[0, T]$ is the time interval of interest. The equation is given by

$$\begin{aligned} \frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} + \nabla p - \nu \Delta \boldsymbol{v} &= \boldsymbol{f}, \quad \forall\, (\boldsymbol{s}, t) \in \Omega \\ \nabla \cdot \boldsymbol{u} &= 0, \quad \forall\, (\boldsymbol{s}, t) \in \Omega \\ \boldsymbol{v} &= \boldsymbol{0}, \quad \forall\, (\boldsymbol{s}, t) \in \partial\Omega_S \times [0, T] \\ \boldsymbol{v}(\boldsymbol{s}, 0) &= \boldsymbol{v}_0(\boldsymbol{s}), \quad \forall\, \boldsymbol{s} \in \Omega_S. \end{aligned} \tag{5.14}$$

The first equation above is the balance of linear moment. The second equation enforces the conservation of mass. The third equation is the no-slip boundary condition which is used when the boundary is rigid and fixed. The fourth equation is the prescription of the initial velocity field. The prescribed data for (5.14) comprises the kinematic viscosity ν, the forcing function $\boldsymbol{f}(\boldsymbol{s}, t)$, and the initial velocity of the fluid $\boldsymbol{v}_0(\boldsymbol{s})$. Given the prescribed data we wish to solve for the velocity field $\boldsymbol{v}$ and the pressure field p on Ω.

To design a PINN for (5.13), the input to the network should be the independent variables $\boldsymbol{x}$ and the output should be the solution vector $\boldsymbol{u}$. For the specific case of the Navier-Stokes system (5.14), the input to the network would be $\boldsymbol{x} = [s_1, s_2, s_3, t] \in \mathbb{R}^4$, while the output vector would be $\boldsymbol{u} = [v_1, v_2, v_3, p] \in \mathbb{R}^4$. The simplest schematic of a PINN that achieves this is shown in Fig. 5.4. The key design steps would be the following:

1. Construct the loss functions
   - Define the interior residual $R(\boldsymbol{u}) = L(\boldsymbol{u}) - \boldsymbol{f}$.
   - Define the boundary residual $R_b(\boldsymbol{u}) = B(\boldsymbol{u}) - g$.

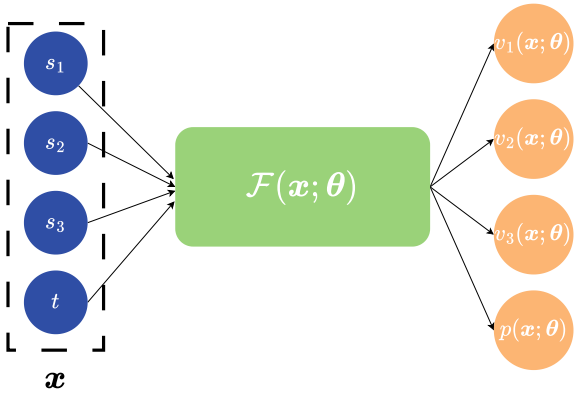


Fig. 5.4 Schematic of a PINN to solve (5.14)

- Select suitable N_v collocation points in the interior of the domain and N_b points on the domain boundary to evaluate the residuals. These could be chosen as based on quadrature rules, such as Gaussian, Lobatto, Uniform, Random, etc.

Then the loss function is

$$\Pi(\boldsymbol{\theta}) = \Pi_{\text{int}}(\boldsymbol{\theta}) + \lambda_b \Pi_{\text{b}}(\boldsymbol{\theta})$$

$$\Pi_{\text{int}}(\boldsymbol{\theta}) = \frac{1}{N_v} \sum_{i=1}^{N_v} |R(\boldsymbol{\mathcal{F}}(\boldsymbol{x}_i;\boldsymbol{\theta})|^2$$

$$\Pi_{\text{b}}(\boldsymbol{\theta}) = \frac{1}{N_b} \sum_{i=1}^{N_b} |R_b(\boldsymbol{\mathcal{F}}(\boldsymbol{x}_i;\boldsymbol{\theta})|^2$$

2. Train the network: find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$, and set the solution as $\boldsymbol{u}^*(\boldsymbol{x}) = \boldsymbol{\mathcal{F}}(\boldsymbol{x};\boldsymbol{\theta}^*)$

We make some remarks here:

- It is implicitly assumed that a weight regularization term is also added to the loss $\Pi(\boldsymbol{\theta})$.
- Is $\boldsymbol{u}^*(\boldsymbol{x})$ the exact solution to the PDE? The answer is **No**!
  - Firstly, $\Pi(\boldsymbol{\theta}^*)$ may not be zero.
  - Even if $\Pi(\boldsymbol{\theta}^*)$ is identically zero, it only means that the residuals vanishes at the collocation points. However, that does not guarantee that the residuals will vanish everywhere in the domain. For that to happen, we need to choose every point in the domain as a collocation point, i.e., N_v, $N_b \to \infty$, and then require the corresponding loss function to be zero!
  - Also, with a fixed network (N_θ fixed) we cannot represent all functions. For that, we will need $N_\theta \to \infty$.

- In practice, we only compute $\Pi(\boldsymbol{\theta}^*)$. Is the solution error $\|\boldsymbol{e}\| = \|\boldsymbol{u}^* - \boldsymbol{u}\|$ related to this loss value? And if it is, can we say that this error will be small as long as the loss is small? This is what we try to answer in next section.

5.5 Error Analysis for PINNs

In order to evaluate the error $\boldsymbol{e} = \boldsymbol{u}^* - \boldsymbol{u}$, we need to know the exact solution $\boldsymbol{u}$ which is not available in general. We consider a way of overcoming this issue, by restricting our discussion to linear PDEs i.e., L and B are assumed to be linear operators. Note that if $\boldsymbol{u}$ is the exact solution, then

$$L(\boldsymbol{e}) = L(\boldsymbol{u}^* - \boldsymbol{u}) = L(\boldsymbol{u}^*) - L(\boldsymbol{u}) = L(\boldsymbol{u}^*) - \boldsymbol{f} = R(\boldsymbol{u}^*) \tag{5.15}$$

and

$$B(\boldsymbol{e}) = B(\boldsymbol{u}^* - \boldsymbol{u}) = B(\boldsymbol{u}^*) - B(\boldsymbol{u}) = B(\boldsymbol{u}^*) - g = R_b(\boldsymbol{u}^*). \tag{5.16}$$

Thus, (5.15), (5.16) lead to a PDE for $\boldsymbol{e}$ driven by the residuals of the MLP solution,

$$\begin{aligned} L(\boldsymbol{e}) &= R(\boldsymbol{u}^*), \quad \text{in } \Omega \\ B(\boldsymbol{e}) &= R_b(\boldsymbol{u}^*), \quad \text{on } \partial\Omega. \end{aligned} \tag{5.17}$$

If the residuals of $\boldsymbol{u}^*$ were zero, then $\boldsymbol{e} = 0$. Unfortunately, these residuals are not zero. The most that we can say is that they are small at the collocation points. However, from the theory of stability of well-posed PDEs, we have

$$\|\boldsymbol{e}\|_{L^2(\Omega)} \leq C_1 \left(\|R(\boldsymbol{u}^*)\|_{L^2(\Omega)} + \|R_b(\boldsymbol{u}^*)\|_{L^2(\partial\Omega)} \right) \tag{5.18}$$

where C_1 is a stability constant that depends on the PDE, the domain Ω, etc. Such a condition can typically be obtained for all well-posed PDEs. It says that if the terms driving the PDE are small, then the solution to the PDE will also be small. This equation tells us that we can control the error if we can control the residuals for the MLP solution. However, in practice we know and control Π_{int}, Π_{b} and not $\|R(\boldsymbol{u}^*)\|^2_{L^2(\Omega)}$, $\|R_b(\boldsymbol{u}^*)\|^2_{L^2(\partial\Omega)}$. The question then becomes, are these quantities related. This is answered in the analysis below. Firstly, we assume the N_v interior collocation points to be suitable quadrature nodes, which leads to the following quadrature approximation for any function ℓ (with sufficient regularity)

$$\int_\Omega \ell(\boldsymbol{x}) \mathrm{d}x = \frac{m_\Omega}{N_v} \sum_{i=1}^{N_v} w_i \ell(\boldsymbol{x}_i) + C_2 N_v^{-\alpha} \tag{5.19}$$

where m_Ω is the measure of the domain Ω, $\{w_i\}$ are the quadrature weights, and the constants $C_2 > 0$ and rate $\alpha > 0$ depend on the type of quadrature used. Next, we can write

$$\begin{aligned} \|R(\boldsymbol{u}^*)\|^2_{L^2(\Omega)} &= \left| m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*) + \|R(\boldsymbol{u}^*)\|^2_{L^2(\Omega)} - m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*) \right| \\ &\leq m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*) + \left| \|R(\boldsymbol{u}^*)\|_{L^2(\Omega)} - m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*) \right| \\ &\leq m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*) + C_2 N_v^{-\alpha} \end{aligned} \tag{5.20}$$

In the equation above, the first line is obtained by adding and subtracting the term $m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)$, the second line is obtained by using the triangle inequality, while the third line is obtained using (5.19) for the integrand $R(\boldsymbol{u}^*)^2$ with the quadrature weights chosen as $w_i = 1$ for simplicity. Then by using the relation $(a+b)^{1/2} \leq a^{1/2} + b^{1/2}$ for $a, b \geq 0$, we get from (5.20)

$$\|R(\boldsymbol{u}^*)\| \leq m_\Omega^{1/2} \Pi_{\text{int}}(\boldsymbol{\theta}^*) + C_2^{1/2} N_v^{-\alpha/2} \tag{5.21}$$

Similarly for the boundary residual

$$\|R_b(\boldsymbol{u}^*)\|_{L^2(\partial\Omega)} \leq m_{\partial\Omega}^{1/2} \Pi_{\text{b}}(\boldsymbol{\theta}^*)^{1/2} + C_3^{1/2} (N_b)^{-\beta/2} \tag{5.22}$$

where $m_{\partial\Omega}$ is the measure of $\partial\Omega$, while C_3 and $\beta > 0$ will depend on the type of boundary quadrature rule considered.

Combining (5.18), (5.21) and (5.22), we get

$$\|\boldsymbol{e}\|_{L^2(\Omega)} \leq C_1 \left(\underbrace{m_\Omega^{1/2} \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2} + m_{\partial\Omega}^{1/2} \Pi_{\text{b}}(\boldsymbol{\theta}^*)^{1/2}}_{\text{reduced by } N_\theta \uparrow} + \underbrace{C_2^{1/2} (N_v)^{-\alpha/2} + C_3^{1/2} (N_b)^{-\beta/2}}_{\text{reduced by } N_v, N_b \uparrow} \right) \tag{5.23}$$

This equation tells us that it is possible to control the error in the PINNs solution by reducing the loss functions (by increasing N_θ) and by increasing the number of interior and boundary collocation points. For further details about this analysis, the reader is referred to [69].

5.6 Data Assimilation Using PINNs

The problem of data assimilation is often encountered in science and engineering. In this problem, we are able to make a few sparse measurements of a quantity, and using these we wish to evaluate it everywhere on a fine grid. We are also given a physical principle (in the form of a PDE) that the variable of interest must adhere to.

Let us assume that we are given a set of sparse measurements of some quantity u on the domain Ω

$$u_i = u(\boldsymbol{x}_i), \quad \boldsymbol{x}_i \in \Omega, \quad 1 \leq i \leq M$$

Furthermore, we are given that u satisfies some constraint $R(u) = 0$ on Ω. Here R could correspond to the PDE residuals (5.17) or any other physical constraint. Then, data assimilation corresponds to using this information to find the value of u at any $\boldsymbol{x} \in \Omega$.

We can solve this problem using PINNs. First, we represent u using a neural network $\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta})$. Next, we define a loss function

$$\Pi(\boldsymbol{\theta}) = \underbrace{\frac{\lambda_I}{M} \sum_{i=1}^{M} (u_i - \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta}))^2}_{\text{data matching}} + \underbrace{\frac{1}{N_v} \sum_{i=M+1}^{M+N_v} |R(\mathcal{F}(\boldsymbol{x}_i, \boldsymbol{\theta}))|^2}_{\text{physical constraint}} + \underbrace{\lambda \|\boldsymbol{\theta}\|^2}_{\text{smoothness}}$$

where $\boldsymbol{x}_i$, $M + 1 \leq i \leq M + N_v$ are suitable collocation points chosen to evaluate the residual, while λ_I, λ are hyper-parameters. Then we train the network by finding $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$, and set the PINNs solution as $\boldsymbol{u}^*(\boldsymbol{x}) = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta}^*)$. Note that the only difference between this formulation and the PINN formulation in the previous section is that fact that the boundary condition term in the former is replaced by the data matching term in the latter.

5.7 Some Existing PINN Formulations

Over the last few years, PINNs have been used to solve a number of PDE problems, as well as integro-differential models [75, 76, 122] and stochastic differential equations [115]. Several gradient pathologies and instabilities of PINNs were identified in [110, 112] along with a discussion of potential remedies [109]. The generalization error analysis of PINNs is available for linear second-order elliptic and parabolic PDEs [99]. In addition to the error estimates for forward PDE problems [69] discussed in Sect. 5.5, estimates for PDE-based inverse problems have also been addressed [68]. In [22], the authors analyze how the generalization error for PINNs trained to solve the Navier-Stokes equations can be controlled by reducing the training error.

The PINN formulations discussed in the present chapter makes use of the strong form of the PDE to construct the residual losses. However, it is more meaningful to consider the variational (weak) form of the PDE, especially when the solution is expected to have lower regularity. Thus, *variational PINNs* were designed in [48], which made use of the Petrov-Galerkin formulation for PDEs. Another variant of PINNs, called *weak PINNS*, have been designed to accurately approximate the entropy solutions for conservation laws. To solve PDEs on very complicated domains, the *XPINN* framework was proposed in [19] which deploys multiple neural networks in parallel to approximate the local solution in smaller sub-domains. We direct interested readers to [18] for a more extensive review on currently available PINN-type approaches.

5.8 Computational Exercise: Physics Informed Neural Networks (PINNs)

In this exercise, you will use feed-forward networks to solve the nonlinear diffusion-advection-reaction equation

$$\mathcal{L}u \equiv u''(x) - \mathrm{Pe}\, u'(x) + \mathrm{Da}\, u(1-u) = 0 \qquad x \in (0,1) \tag{5.24}$$

$$u(0) = 0 \tag{5.25}$$

$$u(1) = 1. \tag{5.26}$$

In this equation, the non-dimensional number Pe is the Peclet number which measures the strength of advection relative to diffusion, and Da is the Damkohler number, which measures the strength of reaction to diffusion.

1. Use the MLP class developed previously to create a feed-forward network of `width=18` and `depth=6`, and with input and output dimensions set to 1. This will be used to represent the function $u = u(x;\boldsymbol{\theta})$ which will be solution to the PDE above. The weights and biases ($\boldsymbol{\theta}$) should be initialized using a standard normal distribution (zero mean and unitary standard deviation). All hidden layers should make use of a `tanh` activation function, while no activation should be used in the output layer. Use l_2 regularization in all layers with a parameter 1e-7.
2. Create an array of $N = 100$ uniformly spaced points in $[0, 1]$. Train a neural network with the following loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} (\mathcal{L}u(x_i;\boldsymbol{\theta}))^2 + \lambda_b\left(u(0;\boldsymbol{\theta})^2 + (u(1;\boldsymbol{\theta}) - 1)^2\right)$$

which is the sum of the interior residual and a scaled boundary residual.

```
def loss_fun(x_train, model, Pe, Da):
    u = model(x_train)
    u_x = torch.autograd.grad(
            u, x_train,
            grad_outputs=torch.ones_like(x_train),
            create_graph=True)[0]
    u_xx = torch.autograd.grad(
            u_x, x_train,
            grad_outputs=torch.ones_like(x_train),
            create_graph=True)[0]
    R_int = torch.mean(torch.square(u_xx - Pe * u_x + Da * u *
        (1.0 - u)))
    R_bc  = torch.square(u[0]) + torch.square(u[-1] - 1.0)

    return R_int,R_bc
```

3. Use a learning rate of 1e-4 and $\lambda_b = 10$ for the training. Consider four different sets of values for the non-dimensional parameters:

   - $P_e = 0.01$, $Da = 0.01$ (diffusion dominates).
   - $P_e = 20$, $Da = 0.01$ (advection dominates).
   - $P_e = 0.01$, $Da = 60$ (reaction dominates).
   - $P_e = 15$, $Da = 40$ (advection and reaction dominate).

4. For each set of parameter values with $\lambda_b = 10$

   - Train the network for 40,000 epochs **without mini-batches**. Save the history of the interior loss and boundary loss.
   - Generate a plot of the interior and boundary as a function of epoch number. One plot for each set of parameter values.
   - Using the fully trained network (at the end of 40,000 epochs) evaluate predicted solution at the test points contained in the first column of the reference solution files provided to you.
   - Generate a plot of the network solution and the reference solution as a function of the spatial coordinate, x. One plot for each set of parameter values.

5. Repeat the previous step for $\lambda_b = 300$ and $\lambda_b = 10{,}000$.
6. Based on the results obtained answer the following questions:

   - Which set of parameter values is the hardest to solve for?
   - Does the same value of λ_b work for all sets of parameter values? Provide a justification for your observations.
   - Extra credit: try different options (activation functions, network size, learning rate, regularization parameter, and the hyperparameters) to see if you can arrive at a good solution for the most challening set of parameters.

Chapter 6
Operator Networks

Recall that a typical MLP $\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta})$ is a function that takes as input $\boldsymbol{x} \in \mathbb{R}^d$ and gives an output $\boldsymbol{y} \in \mathbb{R}^d$ with trainable weights $\boldsymbol{\theta}$. Also, as we discussed in Chap. 5, a PINN is a network of the form $\boldsymbol{u}(\boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta})$ taking as input the independent variable $\boldsymbol{x}$ of the underlying PDE and giving the approximate solution $\boldsymbol{u}(\boldsymbol{x}; \boldsymbol{\theta})$ (of the PDE) as output. The network is trained by minimizing the weighted sum of the PDE and boundary residual. However, this is just one instance of the solution of the PDE for some given boundary condition and source term. For instance, if we consider the PDE

$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega = [0, 1] \times [0, 1] \\ u(\boldsymbol{x}) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{6.1}$$

and train a PINN $\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta})$ to minimize the loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_v} \sum_{i=1}^{N_v} |\nabla \cdot (\kappa \nabla \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta})) - f(\boldsymbol{x}_i)|^2 + \frac{\lambda_b}{N_b} \sum_{i=1}^{N_b} |\mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta}) - g(\boldsymbol{x}_i)|^2$$

Then, if $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$, the PINN solving (6.1) will be $u^*(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}^*)$. However, if we change f or g in (6.1), we have no reason to believe that the same trained network would work. In fact, we would need to retrain the network (with perhaps the same architecture) for the new f and g. This can be quite cumbersome to do, and we would ideally like to avoid it. In this chapter, we will see ways by which we can overcome this issue.

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_6

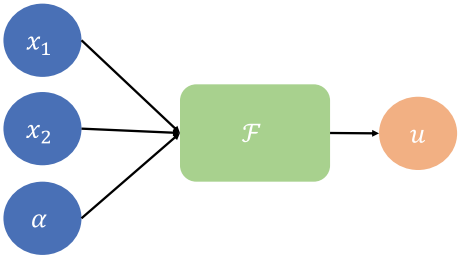


Fig. 6.1 Schematic of a PINN with a parameterized input

6.1 Parametrized PDEs

Assume the the source term f in (6.1) is given as a parametric function $f(\boldsymbol{x}; \alpha)$. For instance, we could have

$$f(x_1, x_2; \alpha) = 4x_1(1 - x_1)\, sin(\alpha x_2)$$

Then we could train a PINN that accommodates for the parametrization by considering a network that takes as input both $\boldsymbol{x}$ and α, i.e., $\mathcal{F}(\boldsymbol{x}, \alpha; \boldsymbol{\theta})$. This is shown in Fig. 6.1 This network can be trained by minimizing the loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_a}\sum_{j=1}^{N_a}\left[\frac{1}{N_v}\sum_{i=1}^{N_v}|\nabla\cdot(\kappa\nabla\mathcal{F}(\boldsymbol{x}_i, \alpha_j; \boldsymbol{\theta})) - f(\boldsymbol{x}_i, \alpha_j)|^2 + \frac{\lambda_b}{N_b}\sum_{i=1}^{N_b}|\mathcal{F}(\boldsymbol{x}_i, \alpha_j; \boldsymbol{\theta}) - g(\boldsymbol{x}_i)|^2\right]$$

Note that we have to also consider collocation points $\{\alpha_j\}_{j=1}^{N_a}$ for the parameter α while constructing the loss function. If $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}}\Pi(\boldsymbol{\theta})$, then the solution to the parameterized PDE would be $u^*(\boldsymbol{x}, \alpha) = \mathcal{F}(\boldsymbol{x}, \alpha; \boldsymbol{\theta}^*)$. Further, for any new value of $\alpha = \hat{\alpha}$ we could find the solution by evaluating $\mathcal{F}(\boldsymbol{x}, \hat{\alpha}; \boldsymbol{\theta}^*)$. We could use the same approach if there was a way of parameterizing the functions $\kappa(\boldsymbol{x})$ and $g(\boldsymbol{x})$.

However, what if we wanted the solution for an arbitrary, non-parametric f? In order to do this, we need to find a way to approximate **operators that map functions to functions.**

6.2 Operators

Consider a class of functions $\boldsymbol{a}(\boldsymbol{y}) \in A$ such that $\boldsymbol{a} : \Omega_Y \rightarrow \mathbb{R}^D$. The functions in this class might have certain properties, such as $\boldsymbol{a} \in C(\Omega_Y)$ or $\boldsymbol{a} \in L^2(\Omega_Y)$. Also consider the operator $\mathcal{N} : A \mapsto C(\Omega_X)$, with $\boldsymbol{u}(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{a})(\boldsymbol{x})$ for $\boldsymbol{x} \in \Omega_X$. Let us see some examples of operators $\mathcal{N}$.

1. Consider the PDE (6.1) for which we assume that the conductivity κ and boundary condition g are given and fixed functions. For this PDE, $\Omega_X = \Omega_Y = \Omega$ and the operator $\mathcal{N}$ maps the RHS f to the solution (temperature) u of the PDE. That is, $u(\boldsymbol{x}) = \mathcal{N}(f)(\boldsymbol{x})$. The input and the output to the operator are related by the PDE.
2. Consider the PDE (6.1) but we assume that the conductivity field κ might change for the model, instead of the RHS. Then, $\Omega_X = \Omega_Y = \Omega$ and the operator $\mathcal{N}$ maps the conductivity κ to the solution u of the PDE. That is, $u = \mathcal{N}(\kappa)(\boldsymbol{x})$. The input and the output to the operator are related by the PDE where it is assumed that f and g are given and fixed.
3. Once again, consider the same PDE (6.1) but with conductivity and the boundary condition being allowed to change. Then, the operator $\mathcal{N}$ maps the boundary condition g and the conductivity κ to the solution u of the PDE. That is, $u = \mathcal{N}(\kappa, g)(\boldsymbol{x})$. In this case the input to the operator are two functions (g, κ) and the output is a single function. Therefore $\Omega_X = \Omega$, while $\Omega_Y = \Omega \times \partial\Omega$. The input and the output are related through the solution to the PDE where it is assumed that f is given and fixed.
4. Consider the equations of linear isotropic elasticity posed on a three-dimensional domain $\Omega \subset \mathbb{R}^3$,

$$\begin{aligned} \nabla \cdot \left(\lambda(\nabla \cdot \boldsymbol{u})\boldsymbol{I} + 2\mu\nabla^S(\boldsymbol{u})\right) &= \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\ \boldsymbol{u}(\boldsymbol{x}) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{6.2}$$

   where we are interested in how the solution of the PDE changes as the source function f is varied. Thus, we are interested in the operator defined by $\boldsymbol{u}(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{f})(\boldsymbol{x})$ whose input function is $\boldsymbol{f} : \Omega \to \mathbb{R}^3$, and the output function is $\boldsymbol{u} : \Omega \to \mathbb{R}^3$. The two are related by the equations above where λ, μ, g are given and fixed.
5. Now, consider a different PDE. In particular, the advection-diffusion-reaction equation,

$$\begin{aligned} \frac{\partial u}{\partial t} + \boldsymbol{a} \cdot \nabla u - \kappa\nabla^2 u + u(1-u) &= f, \quad (\boldsymbol{x}, t) \in \Omega \times (0, T] \\ u(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t), \quad (\boldsymbol{x}, t) \in \partial\Omega \times (0, T] \\ u(\boldsymbol{x}, 0) &= u_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega. \end{aligned} \tag{6.3}$$

   We want to find the operator $\mathcal{N}$ that maps the initial condition u_0 to the solution u at the final time T, i.e., $u(\boldsymbol{x}, T) = \mathcal{N}(u_0)(\boldsymbol{x})$. In this case $\Omega_X = \Omega_Y = \Omega$. Further, the input and the output functions are related to each other via the solution of the PDE above with $\boldsymbol{a}$, κ, f, g given and fixed.

Remark 6.2.1 It is often useful to determine whether an operator is linear or nonlinear. This is because if it is linear it can be well approximated by another linear operator. In the cases considered above the operators in examples 1 and 4 were linear whereas those in examples 2, 3, and 5 were nonlinear.

We are interested in networks that approximate the operator $\mathcal{N}$. We will see how we can do this in the following sections. These types of networks are often referred to as Operator Networks. There are two popular versions of these networks. One is referred to as a Deep Operator Network, or a DeepONet, and the other is referred to as a Fourier Neural Operator. We describe the DeepONet and its extensions in the following three sections. Thereafter, we describe the Fourier Neural Operator, the VarMiON, which is an operator network that draws from variational principles, and Mesh Graph Networks that are adept at working with data defined on unstructured meshes.

6.3 Deep Operator Network (DeepONet) Architecture

Operator networks were first proposed by Chen and Chen [17], where they considered only shallow networks with a single hidden layer. This idea was rediscovered and extended to deep architectures more recently in [61] and were called DeepONets. A standard DeepONet comprises two neural networks. We describe below its construction to approximate an operator $\mathcal{N} : A \to U$, where A is a set of functions of the form $a : \Omega_Y \subset \mathbb{R}^d \to \mathbb{R}$ while U consists of functions of the form $u : \Omega_X \subset \mathbb{R}^D \to \mathbb{R}$. Furthermore, we assume that point-wise evaluations of both class of functions is possible. The architecture for the DeepONet for this operator is illustrated in Fig. 6.2. It is explained below:

- Fix M distinct sensor points $\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(M)}$ in Ω_Y.
- Sample a function $a \in A$ at these sensor points to get the vector $\boldsymbol{a} = [a(\boldsymbol{y}^{(1)}), ..., a(\boldsymbol{y}^{(M)})]^\top \in \mathbb{R}^M$.
- Supply $\boldsymbol{a}$ as the input to a sub-network, called the *branch net* $\mathcal{B}(.; \boldsymbol{\theta}_B) : \mathbb{R}^M \to \mathbb{R}^p$, whose output would be the vector $\boldsymbol{\beta} = [\beta_1(\boldsymbol{a}), ..., \beta_p(\boldsymbol{a})]^\top \in \mathbb{R}^p$. Here $\boldsymbol{\theta}_B$ are the trainable parameters of the branch net. The dimension of the output of the branch is relatively small, say $p \approx 100$.
- Supply $\boldsymbol{x} \in \Omega_X$ as an input to a second sub-network, called the trunk net $\mathcal{T}(.; \boldsymbol{\theta}_T) : \mathbb{R}^D \to \mathbb{R}^p$, whose output would be the vector $\boldsymbol{\tau} = [\tau_1(\boldsymbol{x}), ..., \tau_p(\boldsymbol{x})]^\top \in \mathbb{R}^p$. Here $\boldsymbol{\theta}_T$ are the trainable parameters of the trunk net.
- Take a dot product of the outputs of the branch and trunk nets to get the final output of the DeepONet $\widetilde{\mathcal{N}}(., .; \boldsymbol{\theta}) : \mathbb{R}^D \times \mathbb{R}^M \to \mathbb{R}$ which will approximate the value of $u(\boldsymbol{x})$

$$u(\boldsymbol{x}) \approx u(\boldsymbol{x}; \boldsymbol{\theta}) = \widetilde{\mathcal{N}}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta}) = \sum_{k=1}^{p} \beta_k(\boldsymbol{a}) \tau_k(\boldsymbol{x}). \tag{6.4}$$

  where the trainable parameters of the DeepONet will be the combined parameters of the branch and trunk nets, i.e., $\boldsymbol{\theta} = [\boldsymbol{\theta}_T, \boldsymbol{\theta}_M]$.

In the above construction, once the DeepONet is trained (we will discuss the training in the following section), it will approximate the underlying operator $\mathcal{N}$, and

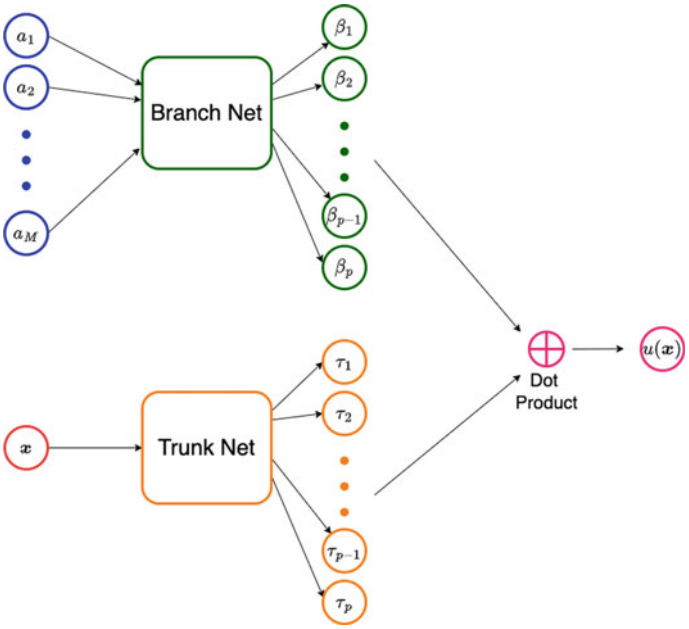


Fig. 6.2 Schematic of a DeepONet

allow us to approximate the value of any $\mathcal{N}(a)(\boldsymbol{x})$ for any $a \in A$ and any $\boldsymbol{x} \in \Omega_X$. Note that in the construction of the DeepONet, the M sensor points need to be pre-defined and cannot change during the training and evaluation phases.

We can make the following observations regarding the DeepONet architecture:

1. The expression in (6.4) has the form of representing the solution as the sum of a series of coefficients and functions. The coefficients are determined by the branch network, while the functions are determined by the trunk network. In that sense the DeepONet construction is similar to the formulation in the spectral method or the finite element method. There is a critical difference though. In these methods, the basis functions are pre-determined and selected by the user. However, in the DeepONet these functions are determined by the trunk network and their final form depends on the data used to train the DeepONet.
2. Architecture of the branch sub-network: When the nodes for sampling the input function are chosen randomly, the appropriate architecture for the branch network comprises fully connected layers. Further recognizing that the dimension of the input to this network can be rather large $M \approx 10^4$, while the output is typically small $p \approx 10^2$, this network can be thought of as an encoder.
   When the nodes for sampling the input function are chosen as a tensorized grid, the appropriate architecture for the branch network comprises convolutional layers. In that case, this network maps an image of large dimension ($M \approx 10^4$) to a latent vector of small dimension, $p \approx 10^2$. Thus it is best represented by a convolutional neural network.

3. Broadly speaking, there are two ways of improving the expressivity of a DeepONet. These involve increase the number of network parameters in the branch and trunk sub-networks, and increasing the dimension p of the latent vectors formed by these sub-networks.

6.3.1 Training DeepONets

Training a DeepONet is typically supervised, and requires pairwise data. The following are the main steps involved:

1. Select N_1 representative function $a^{(i)}$, $1 \leq i \leq N_1$ from the set A. Evaluate the values of these functions at the M sensor points, i.e., $a_j^{(i)} = a^{(i)}(\boldsymbol{y}^{(j)})$ for $1 \leq j \leq M$. This gives us the vectors $\boldsymbol{a}^{(i)} = [a^{(i)}(\boldsymbol{y}^{(1)}), ..., a^{(i)}(\boldsymbol{y}^{(M)})]^\top \in \mathbb{R}^M$ for each $1 \leq i \leq N_1$.
2. For each $a^{(i)}$, determine (numerically or analytically) the corresponding functions $u^{(i)}$ given by the operator $\mathcal{N}$.
3. Sample the function $u^{(i)}$ at N_2 points in Ω_X, i.e., $u^{(i)}(\boldsymbol{x}^{(k)})$ for $1 \leq k \leq N_2$.
4. Construct the training set

$$\mathcal{S} = \left\{ \left(\boldsymbol{a}^{(i)}, \boldsymbol{x}^{(k)}, u^{(i)}(\boldsymbol{x}^{(k)}) \right) : 1 \leq i \leq N_1,\ 1 \leq k \leq N_2 \right\}$$

   which will have $N_1 \times N_2$ samples.
5. Define the loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_1 N_2} \sum_{i=1}^{N_1} \sum_{k=1}^{N_2} |\widetilde{N}(\boldsymbol{x}^{(k)}, \boldsymbol{a}^{(i)}; \boldsymbol{\theta}) - u^{(i)}(\boldsymbol{x}^{(k)})|^2. \tag{6.5}$$

6. Training the DeepONet corresponds to finding $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$.
7. Once trained, then given any new $a \in A$ sampled at the M sensor points (which gives the vector $\boldsymbol{a} \in \mathbb{R}^M$), and a new point $\boldsymbol{x} \in \Omega_X$, we can evaluate the corresponding prediction $u^*(\boldsymbol{x}) = \widetilde{N}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta}^*)$.

Remark 6.3.1 We need not choose the same N_2 points across all i in the training set. In fact, these can be chosen randomly leading to a more diverse dataset.

Remark 6.3.2 The DeepONet can be easily extended to the case where the input comprises multiple functions. In this case, the trunk network remains the same, however the branch network now has multiple vectors as input. The case corresponding to two input functions, $a(\boldsymbol{y})$ and $b(\boldsymbol{y})$, which when sampled yield the vectors, $\boldsymbol{a}$ and $\boldsymbol{b}$, is shown in Fig. 6.3.

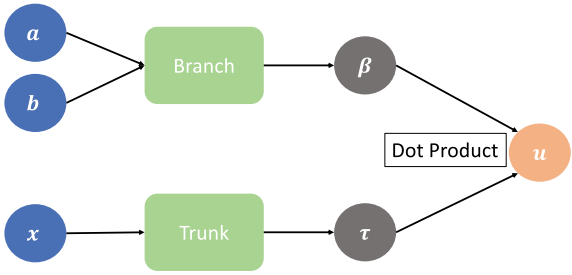


Fig. 6.3 Schematic of a DeepONet with two input functions

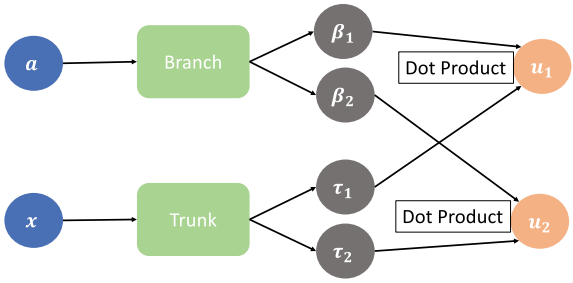


Fig. 6.4 Schematic of a DeepONet with two output functions

Remark 6.3.3 The DeepONet can be easily extended to the case where the output comprises multiple functions (say D such functions). In this case, the output of the branch and trunk network leads to D vectors each with dimension p. The solution is then obtained by taking the dot product of each one of these vectors. The case corresponding to two output functions $u_1(\boldsymbol{x})$ and $u_2(\boldsymbol{x})$ is shown in Fig. 6.4.

6.3.2 *Error Analysis for DeepONets*

A natural question one can ask is "how well do DeepONets approximate operators?" One of the first universal approximation theorem for a shallow version of DeepONets was provided by Chen and Chen [17]:

Theorem 6.3.1 *Suppose Ω_X and Ω_Y are compact sets in $\mathbb{R}^D$ and $\mathbb{R}^d$ (or more generally a Banach space), respectively. Let V be a compact subset of $C(\Omega_Y)$ and $\mathcal{N}$ be a nonlinear, continuous operator mapping V into $C(\Omega_X)$. Then given $\epsilon > 0$, there exists a DeepONet $\widetilde{\mathcal{N}}$ with*

- *A branch sub-network taking inputs sampled at M sensor nodes, with a single hidden layer of width H and activation function $\sigma \in (TW)$, and output layer with (latent) dimension p without activation,*
- *A trunk sub-network with no hidden layers, and an output layer with dimension p and activation σ (as also used in the branch hidden layer)*

such that

$$\max_{\substack{\boldsymbol{x}\in\Omega_X \\ a\in V}} |\widetilde{\mathcal{N}}(\boldsymbol{x}, a; \boldsymbol{\theta}) - \mathcal{N}(a)(\boldsymbol{x})| < \epsilon$$

for a large enough M, H and p.

In the above theorem, the space (TW) denotes the space of *Tauber-Wiener* functions from $\mathbb{R}$ to $\mathbb{R}$, where $\sigma \in (TW)$ if all linear combinations of the form $\sum_{i=1}^{r} c_i \sigma(w_i x + d_i)$ are dense in every $C[a, b]$. In a follow-up paper [16], the same authors extended Theorem 6.3.1 by allowing σ to be chosen as radial basis functions (RBFs).

Motivated by the results by Chen and Chen, a universal approximation result for a deeper version of the network was obtained in [61]. A more general version of this result by removing the compactness assumptions on the spaces, was proposed in [56]. In fact, an in-depth analysis of the DeepONet generalization error was carried out in [56], where it was shown that the size of the DeepONet (the number of trainable parameters) in general scales exponentially as $\mathcal{O}(\epsilon^{-\kappa_\epsilon})$ for $\kappa_\epsilon \geq 0$ with $\kappa_\epsilon \xrightarrow{\epsilon\to 0} \infty$, which the authors call the *curse of dimensionality*. They further provide some concrete examples of operators, some of which correspond to solution operators for PDEs, where the curse of dimensionality can be broken, i.e., the size of the DeepONet grows algebraically in $1/\epsilon$ instead of exponentially.

6.4 Physics-Informed DeepONets

Recall that DeepONets approximate $u(\boldsymbol{x}) = \mathcal{N}(a)(\boldsymbol{x}) \approx \widetilde{\mathcal{N}}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta})$. Assume that the pair a and u satisfy a PDE. For example,

$$\begin{aligned} \nabla \cdot (\kappa u) &= f \text{ in } \Omega \\ u &= g \text{ on } \partial\Omega \end{aligned} \tag{6.6}$$

where κ and g are prescribed. To construct the operator $\widetilde{\mathcal{N}}$ that maps f to u, we need to solve the PDE externally using a traditional numerical solver to define the target labels in the loss function (6.5). However, in addition to this, we can also use a PINN-type loss function and add that to the total loss. This is the idea of Physics-Informed DeepONets proposed in [111]. So for the above model PDE, the physics-based loss would would be,

$$\Pi_p(\boldsymbol{\theta}) = \frac{1}{\bar{N}_1}\frac{1}{\bar{N}_2}\sum_{i=1}^{\bar{N}_1}\sum_{k=1}^{\bar{N}_2}\left|\nabla_{\boldsymbol{x}} \cdot \left(\kappa \nabla_{\boldsymbol{x}} \widetilde{\mathcal{N}}(\boldsymbol{x}^{(k)}, \boldsymbol{f}^{(i)}; \boldsymbol{\theta})\right) - f^{(i)}(\boldsymbol{x}^{(k)})\right|^2. \tag{6.7}$$

This is in addition to the standard data-driven loss function which, for this example is given by

$$\Pi_d(\boldsymbol{\theta}) = \frac{1}{N_1 N_2} \sum_{i=1}^{N_1} \sum_{k=1}^{N_2} |\widetilde{N}(\boldsymbol{x}^{(k)}, \boldsymbol{f}^{(i)}; \boldsymbol{\theta}) - u^{(i)}(\boldsymbol{x}^{(k)})|^2. \tag{6.8}$$

The total loss function is a weighted sum of these two terms:

$$\Pi(\boldsymbol{\theta}) = \Pi_d(\boldsymbol{\theta}) + \lambda \Pi_p(\boldsymbol{\theta}), \tag{6.9}$$

where λ is a hyper-parameter. A few comments are in order:

1. The output sensor points used in the physics-based loss function are usually distinct from the output sensor points used in the data-driven loss term. The former represent the locations at which we wish to minimize the residual of the PDE, while the latter represent the points at which the solution is available to us through external means. The total number of the output sensor points in the physics-based portion of the loss function is denoted by $\bar{N}_2$, whereas in the data-driven loss function it is denoted by N_2.
2. The set of input functions used to construct the physics-based loss function is usually distinct from the set of input functions used to construct the data-driven loss function. The former set represents the functions for which we wish to minimize the residual of the PDE, while the latter set represents the collection of input functions for which the solution is available to us through external means. The total number of functions in the set used to construct the the physics-based portion of the loss function is denoted by $\bar{N}_1$, whereas the total number of functions in the set used to construct the data-driven portion of the loss function is denoted by N_1.

As earlier, we train the network by finding $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$ and approximate the solution for a new a by $u^*(\boldsymbol{x}) = \widetilde{N}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta}^*)$. The advantages of adding the extra physics-based loss are:

1. It reduces the demand on the amount of data in the data-driven loss term. What this means is that we don't have to generate as many solutions of the PDE for training the DeepONet.
2. It makes the network more robust in that it becomes more likely to produce accurate solutions for the type of input functions not included in the training set for the data-driven loss term.

6.5 DeepONets and Their Applications

DeepONet has been successfully applied in various fields, spanning heat conduction [53], biomedical applications [120], fracture mechanics [35], multi-scale modeling using elastic and hyper-elastic materials [121], and the response of power-grids response to part failures or disturbances [70]. Considerable efforts have been invested

in extending the DeepONet framework for uncertainty quantification applications, in order to improve generalization and provide an uncertainty measure associated with its predictions. For example, the Variational Bayes DeepONet (VB-DeepONet) uses variational inference to estimate a high dimensional posterior distributions within a Bayesian framework [27]. In [124] the authors propose a multi-resolution autoencoder DeepONet model, referred to as MultiAuto-DeepONet, to tackle high-dimensional stochastic problems. Here, an encoder is used to reduce the dimensionality of the stochastic inputs, and a pair of DeepONets with a common branch network are used in combination as a decoder. Finally, an uncertainty quantification approach for operator learning based on randomized prior ensembles, called UQDeepONet, with an associated efficient implementation as a JAX library, is described in [117]. DeepONets have been studied also in conjunction with transfer learning [34, 114] and preconditioning strategies [52]. In [38], a variation of DeepONet with a residual U-Net as trunk network is used to predict nonlinear elastic- plastic stress response for complex geometries obtained from topology optimization under variable loads. For time- and path-dependent phenomena, such as plasticity, Sequential Deep Operator Networks (S-DeepONet) [39] were proposed to take into account the causal and time relationships in the data by using a sequence of appropriate models like the long short-term memory (LSTM) units in the branch network of the standard DeepONet.

6.6 Fourier Neural Operator (FNO)

Fourier Neural Operators (FNOs) [58] is an alternative framework of approximating operators, based on the principle of first formulating the algorithm in the infinite dimensional setting and discretizing afterwards. Our approach in developing the architecture for a FNO will be to begin with the architecture of a typical feed-forward MLP that maps a scalar to another scalar, and systematically extend it to a version that maps a scalar valued function to another scalar valued function.

In Fig. 6.5 we have plotted the computational graph of an MLP. We are focused only on the forward part (not the back-propagation) part of this network. For simplcity, we assume that the input $\boldsymbol{x}^{(0)} = x$ is a scalar and the output $\boldsymbol{x}^{(L+1)} = y$ is also a scalar. Further all the other hidden variables (with the exception of $\boldsymbol{\xi}^{(L+1)}$) are vectors with H components. That is, the width of each hidden layer is H.

The first step in this process will be to replace the input and the output with functions. The input will now be the function $a(\boldsymbol{x}) : \Omega \mapsto \mathbb{R}$. Similarly the output

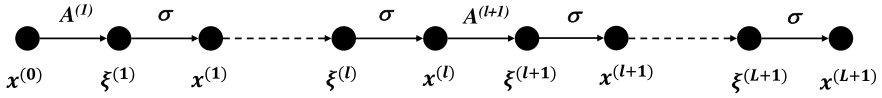


Fig. 6.5 Computational graph for a feed-forward MLP

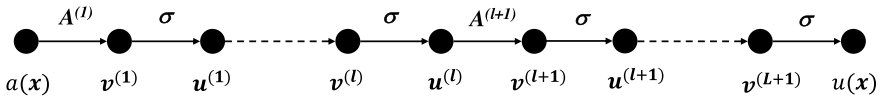


Fig. 6.6 Computational graph for a feed-forward Fourier Neural Operator (FNO) network

is the function $u(\boldsymbol{x}) : \Omega \mapsto \mathbb{R}$. This leads us to the computational graph shown in Fig. 6.6.

The next step is consider the variables in the hidden layers. In the MLP, these were all vectors with H components. In the FNO, these will be vector-valued functions with H components. That is,

$$\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(L)}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(L)} : \Omega \mapsto \mathbb{R}^H. \tag{6.10}$$

As shown in Fig. 6.6, $\boldsymbol{v}^{(n)}$ and $\boldsymbol{u}^{(n)}$ are the counterparts of $\boldsymbol{\xi}^{(n)}$ and $\boldsymbol{x}^{(n)}$, respectively. Further since $\boldsymbol{\xi}^{(L+1)}$ was a scalar, we will set $\boldsymbol{v}^{(L+1)}$ to be a scalar-valued function.

We are now done with extending the input, the output and the variables in the hidden layers from vectors to functions. Next, we need to extend the operators that transform vectors to vectors within an MLP to those that transform functions to functions within an FNO.

We begin with the operator $\boldsymbol{A}^{(1)}$, which in an MLP is an affine map from a vector with one component to a vector with H components. Its straightforward extension to functions is,

$$\boldsymbol{v}^{(1)}(\boldsymbol{x}) = \boldsymbol{A}^{(1)}(a)(\boldsymbol{x}), \tag{6.11}$$

where

$$v_i^{(1)}(\boldsymbol{x}) = W_i^{(1)} a(\boldsymbol{x}) + b_i^{(1)}, \qquad i = 1, \ldots, H. \tag{6.12}$$

Here $W_i^{(1)}$ and $b_i^{(1)}$ are the weights and biases associated with this layer.

Similarly, in an MLP the operator $\boldsymbol{A}^{(L+1)}$ is an affine map from a vector with H components to a vector with 1 component. Its straightforward extension to functions is,

$$v^{(L+1)}(\boldsymbol{x}) = \boldsymbol{A}^{(L+1)}(\boldsymbol{u}^{(L)})(\boldsymbol{x}), \tag{6.13}$$

where

$$v^{(L+1)}(\boldsymbol{x}) = W_i^{(L+1)} u_i^{(L)}(\boldsymbol{x}) + b^{(L+1)}, \qquad i = 1, \ldots, H. \tag{6.14}$$

In the equation above the summation over the index i (from 1 to H) is implied, and $W_i^{(L+1)}$ and $b^{(L+1)}$ are the weights and the bias associated with this layer.

Next we describe the action of the activation on input functions in each layer. It is a simple extension of the activation function applied to the point-wise values of the input function. That is,

$$\boldsymbol{u}^{(n)}(\boldsymbol{x}) = \sigma(\boldsymbol{v}^{(n)})(\boldsymbol{x}), \tag{6.15}$$

where

$$u_i^{(n)}(\boldsymbol{x}) = \sigma(v_i^{(n)}(\boldsymbol{x})), \qquad i = 1, \ldots, H. \tag{6.16}$$

Finally it remains to extend the operators $\boldsymbol{A}^{(n)}$, $n = 2, \ldots, L$ to functions. These are defined as,

$$\boldsymbol{v}^{(n+1)}(\boldsymbol{x}) = \boldsymbol{A}^{(n+1)}(\boldsymbol{u}^{(n)})(\boldsymbol{x}), \tag{6.17}$$

where

$$v_i^{(n+1)}(\boldsymbol{x}) = W_{ij}^{(n+1)} u_j^{(n)}(\boldsymbol{x}) + b_i^{(n+1)} \tag{6.18}$$

$$+ \int_\Omega \kappa_{ij}^{(n+1)}(\boldsymbol{y} - \boldsymbol{x}) u_j^{(n)}(\boldsymbol{y}) d\boldsymbol{y}, \qquad i = 1, \ldots, H. \tag{6.19}$$

In the equation above the summation over the dummy index j (from 1 to H) is implied. The new term that appears in this equation is a convolution. It is motivated by the observation that a large class of linear operators can be represented as convolutions. An example is the so-called Green's operator which maps the right hand side (also called the forcing function) of a linear PDE to its solution. The functions $\kappa_{ij}^{(n+1)}(\boldsymbol{z})$ are called the kernels of the convolution. We note that there are H^2 of these functions in each layer.

It is instructive to examine a specific case of a convolution. Let us consider $\Omega = [0, L_1] \times [0, L_2]$, where we denote the two coordinates by either x_1 and x_2, or y_1 or y_2. In this case we may write the convolution as,

$$v_i(x_1, x_2) = \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(y_1 - x_1, y_2 - x_2) u_j(y_1, y_2)\, dy_2 dy_1, \qquad i = 1, \ldots, H. \tag{6.20}$$

In the equation above, we have dropped the superscripts since they are not relevant to the discussion.

Remark 6.6.1 We may interpret the FNO as a sequence of an affine transform and convolution followed by a point-wise nonlinear activation. This combination of linear and nonlinear (activation) operations allows us to approximate nonlinear operator using this architecture.

Remark 6.6.2 It is instructive to list all the trainable entities in a FNO. First we list all the trainable parameters:

$$W_i^{(1)}, W_{ij}^{(2)}, \ldots, W_{ij}^{(L)}, W_i^{(L+1)}; b_i^{(1)}, b_i^{(2)}, \ldots, b_i^{(L)}, b^{(L+1)} \qquad \forall\, i, j = 1, \ldots, H. \tag{6.21}$$

Thereafter, all the trainable kernel functions

$$\kappa_{ij}^{(n)}(\boldsymbol{z}), \qquad \forall\, i, j = 1, \ldots, H, \;\; n = 2, \ldots, L. \tag{6.22}$$

The neural operator introduced in this section acts directly on functions and transforms them into functions. However, when implementing this operator on a computer the functions have to be represented discretely. This is described in the following section.

6.6.1 Discretization of the Fourier Neural Operator

The functions that appear in the neural operator described in the previous section are:

$$a, \boldsymbol{v}^{(1)}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{v}^{(L)}, \boldsymbol{u}^{(L)}, v^{(L+1)}, u. \tag{6.23}$$

Each of these functions is defined on the domain Ω. We discretize this domain with N uniformly distributed points, and represent each function using its values at these points.

As an example, in two dimensions, with $\Omega = [0, L_1] \times [0, L_2]$, we represent the function $a(x_1, x_2)$ as,

$$a[m, n] = a(x_{1m}, x_{2n}), \quad m = 1 \ldots, N_1, \; n = 1 \ldots, N_2. \tag{6.24}$$

where

$$x_{1m} = (m-1) \times \frac{L_1}{N_1 - 1} \tag{6.25}$$

$$x_{2n} = (n-1) \times \frac{L_2}{N_2 - 1}. \tag{6.26}$$

The same representation will be used for all other functions.

We now have to consider the discrete version of all operations acting on these functions as well. This is described below for the special case of $\Omega = [0, L_1] \times [0, L_2]$.

We begin with the operator $\boldsymbol{A}^{(1)}$. The discretized version is

$$\boldsymbol{v}^{(1)}[m, n] = \boldsymbol{A}^{(1)}(a)[m, n], \tag{6.27}$$

where

$$v_i^{(1)}[m, n] = W_i^{(1)} a[m, n] + b_i^{(1)}, \qquad i = 1, \ldots, H. \tag{6.28}$$

Similarly, the discretized version of the operator $\boldsymbol{A}^{(L+1)}$ is,

$$v^{(L+1)}[m, n] = \boldsymbol{A}^{(L+1)}(\boldsymbol{u}^{(L)})[m, n], \tag{6.29}$$

where

$$v^{(L+1)}[m,n] = W_i^{(L+1)} u_i^{(L)}[m,n] + b^{(L+1)}, \qquad i = 1, \ldots, H. \tag{6.30}$$

Next we describe the action of the activation function on discretized input functions. It is given by

$$\boldsymbol{u}^{(n)}[m,n] = \sigma(\boldsymbol{v}^{(n)})[m,n], \tag{6.31}$$

where

$$u_i^{(n)}[m,n] = \sigma(v_i^{(n)}[m,n]), \qquad i = 1, \ldots, H. \tag{6.32}$$

Finally it remains to develop the discrete version of the operators $\boldsymbol{A}^{(n)}$, $n = 2, \ldots, L$. These are defined as,

$$\boldsymbol{v}^{(p+1)}[m,n] = \boldsymbol{A}^{(p+1)}(\boldsymbol{u}^{(p)})[m,n], \tag{6.33}$$

where

$$v_i^{(p+1)}[m,n] = W_{ij}^{(p+1)} u_j^{(p)}[m,n] + b_i^{(p+1)} \tag{6.34}$$

$$+ \sum_{r=1}^{N_1} \sum_{s=1}^{N_2} \kappa_{ij}^{(p+1)}[r-m, s-n] u_j^{(p)}[r,s] h_1 h_2, \qquad i = 1, \ldots, H, \tag{6.35}$$

where $h_1 = \frac{L_1}{N_1 - 1}$ and $h_2 = \frac{L_2}{N_2 - 1}$. Note that the integral in the convolution is now replaced by a sum over all the grid points. Computing this integral (approximation) for each value of i and m, n involves $O(N_1 N_2 H)$ flops. And since this needs to be done for H different values of i, N_1 values of M, and N_2 values of j, the total cost of discretizing the convolution operation is $O(N_1^2 N_2^2 H^2) = O(N^2 H^2)$, where $N = N_1 \times N_2$. The factor of N^2 in this cost is not acceptable and makes the implementation of this algorithm impractical. In the following section we describe how the use of Fourier Transforms (forward and inverse) overcomes this bottleneck and leads to a practical algorithm. This is also the reason that this algorithm is referred to as a "Fourier Neural Operator".

6.6.2 The Use of Fourier Transforms

Consider a periodic function $u(x_2, x_2)$ defined on $\Omega \equiv [0, L_1] \times [0, L_2]$. If this function is sufficiently smooth it may be approximated by a truncated Fourier series,

$$u(x_1, x_2) \approx \sum_{m=-N_1/2}^{N_1/2} \sum_{n=-N_2/2}^{N_2/2} \hat{u}[m,n] e^{2\pi \mathrm{i}(\frac{m x_1}{L_1} + \frac{n x_2}{L_2})}. \tag{6.36}$$

Here N_1 and N_2 are even integers, the coefficients $\hat{u}[m,n]$ are the Fourier coefficients and $\mathrm{i} = \sqrt{-1}$. We note that while the function u is real-valued the coefficients are complex-valued. However, since u is real-valued, they obey the rule $\hat{u}[-m,-n] = \hat{u}^*[m,n]$, where $(.)^*$ denotes the complex-conjugate of a complex number. The approximation can be made more accurate by increasing N_1 and N_2, and as these numbers tend to infinity, we recover the equality. The relation above is often referred to as the inverse Fourier transform, since it maps the Fourier coefficients to the function in the physical space.

The forward Fourier transform (which maps the function in the physical space to the Fourier coefficients) can be obtained from the relation above by

1. Multiplying both sides by $e^{-2\pi\mathrm{i}(\frac{rx_1}{L_1}+\frac{sx_2}{L_2})}$, where r and s are integers.
2. Integrating both sides over Ω.
3. Recognizing that the integral $\int_\Omega e^{2\pi\mathrm{i}(\frac{(m-r)x_1}{L_1}+\frac{(n-s)x_2}{L_2})} dx_1 dx_2$ is non-zero only when $m = r$ and $n = s$, and in that case it evaluates to $L_1 L_2$.

These steps yield the final relation:

$$\hat{u}[r,s] = \frac{1}{L_1 L_2} \int_0^{L_1} \int_0^{L_2} u(x_1,x_2) e^{-2\pi\mathrm{i}(\frac{rx_1}{L_1}+\frac{sx_2}{L_2})} dx_1 dx_2. \tag{6.37}$$

We now describe how Fourier transforms can be used to evaluate the convolution efficiently. To do this we consider the special case of 2D convolution in (6.20). We begin with substituting $u_j(y_1,y_2) = \sum_{m=-N_1/2}^{N_1/2} \sum_{n=-N_2/2}^{N_2/2} \hat{u}_j[m,n] e^{2\pi\mathrm{i}(\frac{my_1}{L_1}+\frac{ny_2}{L_2})}$ in this equation to get,

$$\begin{aligned}
v_i(x_1,x_2) &= \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(y_1-x_1, y_2-x_2) \sum_{m,n} \hat{u}_j[m,n] e^{2\pi\mathrm{i}(\frac{my_1}{L_1}+\frac{ny_2}{L_2})} \, dy_2 dy_1 \\
&= \sum_{m,n} \hat{u}_j[m,n] \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(y_1-x_1, y_2-x_2) e^{2\pi\mathrm{i}(\frac{my_1}{L_1}+\frac{ny_2}{L_2})} \, dy_2 dy_1 \\
&= \sum_{m,n} \hat{u}_j[m,n] \int_{-x_1}^{L_1-x_1} \int_{-x_2}^{L_2-x_2} \kappa_{ij}(z_1,z_2) e^{2\pi\mathrm{i}(\frac{m(z_1+x_1)}{L_1}+\frac{n(z_2+x_2)}{L_2})} \, dz_2 dz_1 \\
&= \sum_{m,n} \hat{u}_j[m,n] e^{2\pi\mathrm{i}(\frac{mx_1}{L_1}+\frac{nx_2}{L_2})} \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(z_1,z_2) e^{2\pi\mathrm{i}(\frac{mz_1}{L_1}+\frac{nz_2}{L_2})} \, dz_2 dz_1 \\
&= L_1 L_2 \sum_{m,n} \hat{u}_j[m,n] \hat{\kappa}_{ij}[-m,-n] e^{2\pi\mathrm{i}(\frac{mx_1}{L_1}+\frac{nx_2}{L_2})}.
\end{aligned} \tag{6.38}$$

In the development above, in going from the first to the second line we have taken the summation outside the integral and recognized that the coefficients $\hat{u}_j[m,n]$ do not depend on y_1 and y_2. In going from the second to the third line we have introduced the variables $z_1 = y_1 - x_1$ and $z_2 = y_2 - x_2$. In going from the third to the fourth line we have made use of the fact that the functions $\kappa_{ij}(z_1,z_2)$ are periodic. Finally in going

from the fourth to the fifth line we have made use of the definition of the Fourier Transform (6.37). This final relation tells us that the convolution can be computed by:

1. Computing the Fourier Transform of u_j.
2. Computing the Fourier Transform of κ_{ij}.
3. Computing the product of the coefficients of these two transforms.
4. Computing the inverse Fourier Transform of the product.

Next, we account for the fact that we will only work with the discrete forms of the functions u_j and κ_{ij}. This means that we evaluate the inverse Fourier transform (6.36) at a finite set of grid points. Further, it means that we have to approximate the integral in the Fourier transform (6.37). This alternate form is given by

$$\hat{u}[r,s] = \frac{h_1 h_2}{L_1 L_2} \sum_{m=1}^{N_1} \sum_{n=1}^{N_2} u[m,n] e^{-2\pi \mathrm{i}(\frac{r x_{1m}}{L_1} + \frac{s x_{2n}}{L_2})}. \tag{6.39}$$

Here $h_1 = \frac{L_1}{N_1}$ and $h_2 = \frac{L_2}{N_2}$, $x_{1m} = (m-1)h_1$ and $x_{2n} = (n-1)h_2$.

The final observation is that the evaluating the sums in (6.36) and (6.39) require $O(N^2)$ operations. This would make the evaluation of the convolution via the Fourier method impractical except for when N is very small. However, the use of Fast Fourier Transform (FFT) reduces this cost to $O(N \log N)$. Thus the cost of implementing the convolution reduces to $O(N \log N H^2)$. This makes the implementation of Fourier Neural Operators practical.

6.7 Variationally Mimetic Operator Network (VarMiON)

The form of the PDEs considered so far, for instance (6.1), are written in the *strong form* and assume sufficient regularity of the PDE solution and data. However, it is more practical to consider the *weak (variational) form* when this regularity is not guaranteed (see, for example, [44]). In [78], an operator network was constructed that mimics the variational form of the PDE, where the architecture of the various sub-components of the network were motivated by the variational structure. Such operator networks are called Variationally Mimetic Operator Networks (VarMiONs). To understand the VarMiON architecture, we first provide some background of the variational form for linear elliptic PDEs and it's discrete counterpart.

6.7.1 Background

Variational form Let $\Omega \in \mathbb{R}^D$ be an open, bounded domain with boundary Γ. The boundary is further partitioned into the Dirichlet boundary Γ_g and the Neumann

boundary Γ_η. Define the space $H^1_g = \{u \in H^1(\Omega) \,:\, u\big|_{\Gamma_g} = 0\}$, where the Sobolev function space $H^1(\Omega) = \{u \in L^2(\Omega) : \partial_i u \in L^2(\Omega)\, \forall\, 1 \leq i \leq D\}$. Consider the following model PDE model as a canonical example for second-order linear elliptic PDEs

$$\begin{aligned} \nabla \cdot (\kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x})) &= f(\boldsymbol{x}), \qquad && \forall\, \boldsymbol{x} \in \Omega, \\ \kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) &= q(\boldsymbol{x}), && \forall\, \boldsymbol{x} \in \Gamma_q, \\ u(\boldsymbol{x}) &= 0, && \forall\, \boldsymbol{x} \in \Gamma_g, \end{aligned} \tag{6.40}$$

where $f \in \mathcal{F} \subset L^2(\Omega)$ is the source term, $q \in \mathcal{Q} \subset L^2(\Gamma_q)$ is the boundary flux data, and $\kappa \in \mathcal{K} \subset L^\infty(\Omega)$ is the spatially varying permeability of the media, and $\boldsymbol{n}(\boldsymbol{x})$ is the outward normal to the boundary Γ_q. Note that (6.40) is essentially the extension of (6.1) where we now include a Neumann boundary.

The variational formulation of (6.40) is obtained by integrating the PDE against a weighting function $w \in H^1_g$ and applying an integration-by-parts. In doing so, the RHS of the PDE becomes

$$(w, f) := \int_\Omega w(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \tag{6.41}$$

where $(.,.)$ denotes the $L^2(\Omega)$ inner-product, while the LHS of the PDE becomes

$$\begin{aligned} \int_\Omega w(\boldsymbol{x})\, (\nabla \cdot (\kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x})))\, \mathrm{d}\boldsymbol{x} &= -\int_\Omega \nabla w(\boldsymbol{x}) \cdot (\kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x})) \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} w(\boldsymbol{x}) \kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \\ &= -\int_\Omega \nabla w(\boldsymbol{x}) \cdot (\kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x})) \mathrm{d}\boldsymbol{x} + \int_{\Gamma_g} w(\boldsymbol{x}) \kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \\ &\quad + \int_{\Gamma_q} w(\boldsymbol{x}) \kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \\ &= -\int_\Omega \nabla w(\boldsymbol{x}) \cdot (\kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x})) \mathrm{d}\boldsymbol{x} + \int_{\Gamma_q} w(\boldsymbol{x}) q(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \end{aligned} \tag{6.42}$$

where the final expression above is obtained by using the fact that w vanishes on Γ_g, and the given Neumann boundary condition in (6.40). We denote the inner-product on $L^2(\Gamma_q)$ as $(.,.)_{\Gamma_q}$ and define the bilinear form

$$a(w, u; \kappa) := -\int_\Omega \nabla w(\boldsymbol{x}) \cdot (\kappa(\boldsymbol{x}) \nabla u(\boldsymbol{x})) \mathrm{d}\boldsymbol{x}. \tag{6.43}$$

Note that a is linear in w and u, but does not need be linear in κ which parametrizes this bilinear form. Combining (6.41), (6.42) and (6.43), the variational problem is defined by: Find $u \in H^1_g$ such that $\forall\, w \in H^1_g$

$$a(w, u; \kappa) = (w, f) + (w, q)_{\Gamma_q}, \tag{6.44}$$

If the solution and PDE data have sufficient regularity, we can recover the strong form (6.40) from the weak form (6.44). We refer interested readers to [13, 28] for additional details.

Consider the solution operator

$$\mathcal{N} : \mathcal{F} \times \mathcal{K} \times \mathcal{Q} \longrightarrow \mathcal{V} \subset H_g^1, \quad \mathcal{N}(f, \kappa, q)(\boldsymbol{x}) = u(\boldsymbol{x}) \tag{6.45}$$

which maps the data (f, κ, q) to the unique solution u of (6.44). We are interested in approximating $\mathcal{N}$ using a VarMiON. To do so, we need to first describe a discrete framework for (6.44)

Discrete variational form Let us consider the class of numerical solvers for (6.44) that approximate the solution function space $\mathcal{V}$ by the space $\mathcal{V}^h$ spanned by a finite set of continuous basis functions $\{\phi_i(\boldsymbol{x})\}_{i=1}^n$. Typical examples include the Finite Element Method (FEM) or Proper Orthogonal Decomposition (POD) basis approximating the solutions of (6.44) [8, 44]. Then any function $v^h \in \mathcal{V}^h$ can be be expressed as a linear combination of the finite basis, i.e.,

$$v^h(\boldsymbol{x}) = v_i \phi_i(\boldsymbol{x}) = \boldsymbol{V}^\top \boldsymbol{\Phi}(\boldsymbol{x}), \quad \boldsymbol{V} = (v_1, \ldots, v_n)^\top, \quad \boldsymbol{\Phi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), \ldots, \phi_n(\boldsymbol{x}))^\top.$$

We also define the space $\mathcal{V}_q^h = \{v\big|_{\Gamma_q} : v \in \mathcal{V}^h\}$ for the function on the Neumann boundary.

We project the PDE data (f, κ, q) onto these finite dimensional spaces, with the projections given by

$$f^h(\boldsymbol{x}) = \boldsymbol{F}^\top \boldsymbol{\Phi}(\boldsymbol{x}) \in \mathcal{V}^h, \quad \kappa^h(\boldsymbol{x}) = \boldsymbol{K}^\top \boldsymbol{\Phi}(\boldsymbol{x}) \in \mathcal{V}^h, \quad q^h(\boldsymbol{x}) = \boldsymbol{Q}^\top \boldsymbol{\Phi}(\boldsymbol{x})\big|_{\Gamma_q} \in \mathcal{V}_q^h. \tag{6.46}$$

where coefficients $\boldsymbol{F}, \boldsymbol{K}, \boldsymbol{Q}$ will depend on the choice of the basis functions. If the approximate solution is represented as $u^h(\boldsymbol{x}) = \boldsymbol{U}^\top \boldsymbol{\Phi}(\boldsymbol{x})$, then using (6.46) in (6.44) gives us the following system of equations describing the discrete variational form

$$\boldsymbol{S}(\kappa^h)\boldsymbol{U} = \boldsymbol{M}_1 \boldsymbol{F} + \boldsymbol{M}_2 \boldsymbol{Q} \tag{6.47}$$

where the matrices are given by

$$S_{ij}(\kappa^h) = a(\phi_i, \phi_j; \kappa^h), \quad [M_1]_{ij} = (\phi_i, \phi_j), \quad [M_2]_{ij} = (\phi_i, \phi_j)_{\Gamma_q} \quad 1 \le i, j \le n. \tag{6.48}$$

In order to recover the solution coefficients $\boldsymbol{U}$ from (6.47), we make assume that $\boldsymbol{S}(\kappa^h)$ is invertible for all $\kappa^h \in \mathcal{V}^h$. We can now define the discrete solution operator

$$\mathcal{N}^h : \mathcal{V}^h \times \mathcal{V}^h \times \mathcal{V}_q^h \to \mathcal{V}^h, \quad \mathcal{N}^h(f^h, \kappa^h, q^h)(\boldsymbol{x}) := u^h(\boldsymbol{x}) = (\boldsymbol{B}_1(f^h, \kappa^h) + \boldsymbol{B}_2(q^h, \kappa^h))^\top \boldsymbol{\Phi}(\boldsymbol{x}) \tag{6.49}$$

where

$$\boldsymbol{B}_1(f^h, \kappa^h) = \boldsymbol{S}^{-1}(\kappa^h)\boldsymbol{M}_1\boldsymbol{F}, \quad \boldsymbol{B}_2(q^h, \theta^h) = \boldsymbol{S}^{-1}(\kappa^h)\boldsymbol{M}_2\boldsymbol{Q}. \tag{6.50}$$

The VarMiON is constructed to mimic the structure of $\mathcal{N}^h$. Note that $\mathcal{N}^h$ is linear in $\boldsymbol{F}$ and $\boldsymbol{Q}$, but might be non-linear in κ^h. We will keep these obervations in mind while discussing the VarMiON architecture.

6.7.2 VarMiON Architecture

In order to feed (f, κ, q) into the VarMiON network, we need to sample the function at sensor nodes (as was also done for DeepONets). Let $\{\boldsymbol{y}^{(j)}\}_{j=1}^{M}$ be the sensor nodes in Ω for f and κ, while $\{\boldsymbol{y}_b^{(j)}\}_{j=1}^{M'}$ be the sensor nodes on Γ_q for q. We define the input vectors

$$\widetilde{\boldsymbol{F}} = (f(\boldsymbol{y}^{(1)}), \ldots, f(\boldsymbol{y}^{(M)}))^\top, \quad \widetilde{\boldsymbol{K}} = (\kappa(\boldsymbol{y}^{(1)}), \ldots, \kappa(\boldsymbol{y}^{(M)}))^\top, \quad \widetilde{\boldsymbol{Q}} = (q(\boldsymbol{y}_b^{(1)}), \ldots, q(\boldsymbol{y}_b^{(M')}))^\top \tag{6.51}$$

Then the schematic of the VarMiON architecture to solve the PDE (6.40) is shown in Fig. 6.7, which comprises:

- a **non-linear branch** taking the input $\widetilde{\boldsymbol{K}} \in \mathbb{R}^M$, which is transformed into a matrix output $\widetilde{\boldsymbol{D}}(\widetilde{\boldsymbol{K}}) \in \mathbb{R}^{p\times p}$. Here p is the latent dimension of the VarMiON.
- a **linear branch** taking the input $\widetilde{\boldsymbol{F}} \in \mathbb{R}^M$ and transforming it as $\boldsymbol{A}_1\widetilde{\boldsymbol{F}}$, where $\boldsymbol{A}_1 \in \mathbb{R}^{p\times M}$ is a learnable matrix. The output of this branch is acted upon by the matrix $\boldsymbol{D}$ to give $\boldsymbol{\beta}_1(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}) = \widetilde{\boldsymbol{D}}(\widetilde{\boldsymbol{K}})\boldsymbol{A}_1\widetilde{\boldsymbol{F}} \in \mathbb{R}^p$.

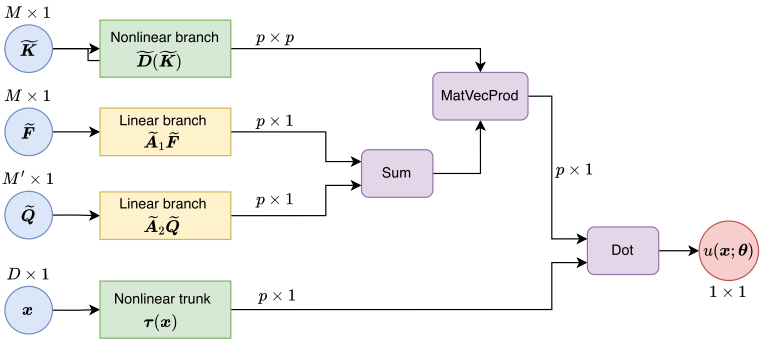


Fig. 6.7 VarMiON architecture

- a **linear branch** taking the input $\widetilde{\boldsymbol{Q}} \in \mathbb{R}^{M'}$ and transforming it as $\boldsymbol{A}_2 \widetilde{\boldsymbol{Q}}$, where $\boldsymbol{A}_2 \in \mathbb{R}^{p \times M'}$ is a learnable matrix. The output of this branch is acted upon by the matrix $\widetilde{\boldsymbol{D}}$ to give $\boldsymbol{\beta}_2(\widetilde{\boldsymbol{Q}}, \widetilde{\boldsymbol{K}}) = \widetilde{\boldsymbol{D}}(\widetilde{\boldsymbol{K}})\boldsymbol{A}_2 \widetilde{\boldsymbol{Q}} \in \mathbb{R}^p$.
- a **non-linear trunk** taking the input $\boldsymbol{x} \in \mathbb{R}^D$, which gives the output $\boldsymbol{\tau}(\boldsymbol{x}) = (\tau_1(\boldsymbol{x}), \ldots, \tau_p(\boldsymbol{x}))^\top$, where each $\tau_i : \mathbb{R}^D \to \mathbb{R}$ is a trainable network.

Let $\mathcal{V}^\tau$ be the space spanned by the trunk functions τ. Then the final VarMiON operator is given by

$$\widetilde{\mathcal{N}}(.;\boldsymbol{\theta}) : \mathbb{R}^M \times \mathbb{R}^M \times \mathbb{R}^{M'} \to \mathcal{V}^\tau, \quad \widetilde{\mathcal{N}}(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}, \widetilde{\boldsymbol{Q}}; \boldsymbol{\theta})(\boldsymbol{x}) = u(\boldsymbol{x};\boldsymbol{\theta}) = \left(\boldsymbol{\beta}_1(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}) + \boldsymbol{\beta}_2(\widetilde{\boldsymbol{Q}}, \widetilde{\boldsymbol{K}})\right)^\top \boldsymbol{\tau}. \tag{6.52}$$

where $\boldsymbol{\theta}$ are all the trainable parameters of the VarMiON. Note the similarity between (6.49) and (6.52). In particular, $\boldsymbol{B}_1$ (respectively $\boldsymbol{B}_2$) have the same structure as $\boldsymbol{\beta}_1$ (respectively $\boldsymbol{\beta}_2$). Further, the note the branch network for $\widetilde{\boldsymbol{F}}$ and $\widetilde{\boldsymbol{Q}}$ are linear, motivated by the linearity of $\mathcal{N}^h$ with respect to $\boldsymbol{F}$ and $\boldsymbol{Q}$. In this sense, the VarMiON mimics the discrete variational form of the PDE.

6.7.3 Training the VarMiON

The VarMiON is trained in a supervised manner similar to the DeepONet, where the training data is generated using a numerical solver (for instance the same one used to describe the discrete variational form).

1. Sample N_1 representative function triplets $(f^{(i)}, \kappa^{(i)}, q^{(i)})$, $1 \le i \le N_1$ from the set function space $\mathcal{F} \times \mathcal{K} \times \mathcal{Q}$.
2. Evaluate the values of these N_1 triples at the sensor points (according to (6.51)) to get the branch input vectors $\widetilde{\boldsymbol{F}}^{(i)}$, $\widetilde{\boldsymbol{K}}^{(i)}$, $\widetilde{\boldsymbol{Q}}^{(i)}$ for all $1 \le u \le N_1$.
3. For each N_1 triplet, determine (numerically) the corresponding solution functions $u^{h,(i)}$ given by the operator $\mathcal{N}^h$.
4. Sample the function $u^{h,(i)}$ at N_2 points in Ω_X, i.e., $u^{h,(i)}(\boldsymbol{x}^{(j)})$ for $1 \le j \le N_2$.
5. Construct the training set

$$\mathcal{S} = \left\{ \left(\widetilde{\boldsymbol{F}}^{(i)}, \widetilde{\boldsymbol{K}}^{(i)}, \widetilde{\boldsymbol{Q}}^{(i)}, \boldsymbol{x}^{(j)}, u^{h,(i)}(\boldsymbol{x}^{(j)}) \right) : 1 \le i \le N_1,\ 1 \le j \le N_2 \right\}$$

   which will have $N_1 \times N_2$ samples.
6. Define the loss function

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_1 N_2} \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} |\widetilde{N}(\widetilde{\boldsymbol{F}}^{(i)}, \widetilde{\boldsymbol{K}}^{(i)}, \widetilde{\boldsymbol{Q}}^{(i)}; \boldsymbol{\theta})(\boldsymbol{x}^{(j)}) - u^{(i)}(\boldsymbol{x}^{(j)})|^2. \tag{6.53}$$

7. Training the VarMiON corresponds to finding $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$.

Once trained, then given any new (f, κ, q) sampled at the sensor points, and a new point $\boldsymbol{x} \in \Omega_X$, we can evaluate the corresponding prediction $u^*(\boldsymbol{x}) = \widetilde{N}(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}, \widetilde{\boldsymbol{Q}}; \boldsymbol{\theta}^*)(\boldsymbol{x})$.

6.7.4 *Error Estimates of VarMiON Approximation*

Let (f, κ, q) be any triplet in $\mathcal{F} \times \mathcal{K} \times \mathcal{Q}$. Then, we can define the VarMiON error in approximating the true solution as

$$\mathcal{E}(f, \kappa, q) := \|\mathcal{N}(f, \kappa, q) - \widetilde{\mathcal{N}}(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}, \widetilde{\boldsymbol{Q}}; \boldsymbol{\theta}^*)\|_{L^2(\Omega)}. \tag{6.54}$$

Let us also define the error between $\mathcal{N}$ and $\mathcal{N}^h$, i.e., the error incurred while approximating the true solution with the numerical solver (used to generate training samples)

$$\mathcal{E}_h(f, \kappa, q) := \|\mathcal{N}(f, \kappa, q) - \mathcal{N}^h(f^h, \kappa^h, q^h; \theta^*)\|_{L^2(\Omega)}, \tag{6.55}$$

and that between $\mathcal{N}^h$ and the VarMiON approximation as

$$\widetilde{\mathcal{E}}(f, \kappa, q) := \|\mathcal{N}^h(f^h, \kappa^h, q^h) - \widetilde{\mathcal{N}}(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}, \widetilde{\boldsymbol{Q}}; \boldsymbol{\theta}^*)\|_{L^2(\Omega)}. \tag{6.56}$$

Further, for any (f, κ, q), (f', κ', q'), we define the corresponding change in the true solution as

$$\mathcal{E}_{\text{stab}}[(f, \kappa, q), (f', \kappa', q')] := \|\mathcal{N}(f, \kappa, q) - \mathcal{N}(f', \kappa', q')\|_{L^2(\Omega)}, \tag{6.57}$$

and the change in the VarMiON solution as

$$\widetilde{\mathcal{E}}_{\text{stab}}[(f, \kappa, q), (f', \kappa', q')] := \|\widetilde{\mathcal{N}}(\widetilde{\boldsymbol{F}}, \widetilde{\boldsymbol{K}}, \widetilde{\boldsymbol{Q}}; \boldsymbol{\theta}^*) - \widetilde{\mathcal{N}}(\widetilde{\boldsymbol{F}}', \widetilde{\boldsymbol{K}}', \widetilde{\boldsymbol{Q}}'; \boldsymbol{\theta}^*)\|_{L^2(\Omega)}. \tag{6.58}$$

Note that (6.57) and (6.58) describe the stability of the true and VarMiON solution operator, respectively.

Now, for any training sample triplet $(f^{(i)}, \kappa^{(i)}, q^{(i)})$, we can use triangle inequality repeatedly to get the bound (see [78] for details)

$$\begin{aligned}\mathcal{E}(f, \kappa, q) \leq\ & \mathcal{E}_{\text{stab}}[(f, \kappa, q), (f^{(i)}, \kappa^{(i)}, q^{(i)})] + \mathcal{E}_h(f^{(i)}, \kappa^{(i)}, q^{(i)}) + \widetilde{\mathcal{E}}(f^{(i)}, \kappa^{(i)}, q^{(i)}) \\ & + \widetilde{\mathcal{E}}_{\text{stab}}[(f, \kappa, q), (f^{(i)}, \kappa^{(i)}, q^{(i)})].\end{aligned} \tag{6.59}$$

Thus, a bound for the generalization error can be obtained by determining bounds for each of the four terms on the right of (6.57).

The authors in [78] obtained a bound of the following form (under certain assumptions)

$$\mathcal{E}(f,\kappa,q) \leq C\left(\epsilon_h + \epsilon_s + \sqrt{\epsilon_t} + \frac{1}{M^{\alpha/2}} + \frac{1}{(M')^{\alpha'/2}} + \frac{1}{N_2^{\gamma/2}}\right) \tag{6.60}$$

where ϵ_h is the bound on the numerical solver error (6.55), ϵ_t is the VarMiON error on the training set (which is tracked while training), ϵ_s is the a measure of the maximum distance between any $(f,\kappa,q) \in \mathcal{F}\times\mathcal{K}\times\mathcal{Q}$ and the set of N_1 samples in the training set, and α, α', γ are associated with the descritization of the input and output of the VarMiON (similar to α and β arising in the PINNs error estimate (5.23)). The constant C depends on the stability constants associated with the stability of $\mathcal{N}$ and $\widetilde{\mathcal{N}}$.

We make few comments about (6.60):

- The error estimate reveals the various sources of error. In general, we cannot expect the VarMiON error to be smaller than the error associated with generating the training samples.
- If the training samples are generated by a Galerkin FEM method, then ϵ_h would correspond to the interpolation error that depends on the element size and the order of the basis elements (see [44] for details).
- We can make ϵ_s smaller by carefully adding more training samples so that the space $\mathcal{F}\times\mathcal{K}\times\mathcal{Q}$ is better "covered" by these samples. However, we might also need to increase the size of the VarMiON in order to maintain the training error ϵ_t at the level.
- Choosing a finer discretization for the inputs and outputs, i.e., a larger M, M' and N_2, would lower the quadrature error (governed by the last three terms on (6.60)).

Remark 6.7.1 It has also been shown in [78] that the inverse of the matrix $\boldsymbol{S}(\kappa^h)$ arising in the discrete variational formulation (6.47) can be approximated in terms of its counterpart $\widetilde{\boldsymbol{D}}(\widetilde{\boldsymbol{K}})$ in the VarMiON.

Remark 6.7.2 A VarMiON-type formulation has also been proposed in [78] for the scalar non-linear advection-diffusion-reaction problem, and been numerically tested on the regularized Eikonal equations.

6.8 Mesh Graph Networks

The type of operator networks we discuss next are called Mesh Graph Networks (MGNs) [82]. MGNs provide a deep learning framework to operate directly on mesh-based data, commonly used in computational physics. This is done by interpreting mesh-based data as graphs, and using a type of neural network architecture called Graph Neural Networks (GNNs) [6, 96, 125]. GNNs generalize the concept of CNNs to graph-like structures, and are useful and efficient tools for approximating functions that map graphs to graphs.

MGNs can be applied in different realms of computational physics, whenever the quantity to be predicted is defined over a (possibly unstructured) computational mesh; this includes problems that are typically solved using methods from computational fluid dynamics (CFD) or finite element analysis (FEA). This approach can be used for time-dependent problems [82] and steady-state parametric problems [30]. The main steps in applying MGNs are:

1. *Mesh to graph representation.* In this step, the mesh-based data are transformed into instances of a graph. The quantities that are to be predicted by the MGN are used to define the features (or attributes) of the nodes of the graph, and the connectivity of the mesh is used to define the edges of the graph. The edges can also be equipped with their own attributes, which usually carry geometrical information from the physical domain.
2. *Feature encoding.* As is common in several applications in machine learning, the features of the nodes and edges are embedded (or encoded) in a latent space. The encoding is usually performed with a shallow MLP or some other trainable model, which is shared among all the nodes of the graph. In an MGN all significant calculations are carried out with these latent features.
3. *Processing.* This involves updating the features, which is performed via message passing blocks [29]. This represents a mechanism to propagate information across the graph, and thereby update a function that is defined on the graph. The rationale for the update is that the update to the features of a node depends on its own features and the information it gathers from its neighbors. A single message passing block can be divided into 3 sub-steps: (*i*) message computation, (*ii*) message aggregation, and (*iii*) feature update. Usually, multiple message passing blocks are applied sequentially, so that a node can be influenced by nodes that are further away than its nearest neighbors, and can therefore observe a larger portion of the domain.
4. *Features decoding.* At the end of the message processing step, the updated latent representation has to be decoded back to the physical function(s) it represents. This is once again carried out by a small MLP that is shared by all nodes of the graph.

In what follows, we will first provide some background on graph theory and mesh-based data, and then delve into the details of how MGNs work, how they are trained, and how they are used to make predictions.

6.8.1 Background

Graphs A graph G is an object defined as the pair $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$ are its vertices or nodes, and $E = \{e_{ij} | \, i, \, j \in V\}$ are its edges. The nodes, denoted with an integer index $i \in \{1, \ldots, n\}$, are the fundamental units of the graph and are usually equipped with a set of attributes or features $\mathbf{v}_i$. The edges $\{e_{ij}\}$ carry information about the relations across nodes – if the edge e_{ij} exists, then

the nodes i and j are connected. This connection can be interpreted in many ways, depending on the problem at hand. If we decide to associate each edge with a weight, representing the strength of the connection, we call it a weighted graph.

A common way to represent a graph is through its adjacency matrix $\boldsymbol{A}$. This is an $n \times n$ matrix whose row and column indices represent the vertices of the graph, and the entries A_{ij} indicate whether or not two nodes are connected. In the case of a simple graph, the entries of $\boldsymbol{A}$ are either 1 or 0, depending on whether two nodes are connected or not. For a weighted graph, the entries are the non-negative real numbers that represent the strength of the connections.

We also distinguish between directed and undirected graphs. For directed graphs, the edges represent one-way, non-symmetric connections. On the other hand, for undirected graphs the connections are symmetric. We can deduce that for an undirected graph the adjacency matrix is symmetric, whereas this is not necessarily the case for a directed graph.

It is important to note that a graph is an abstract object that may not have a physical interpretation. Fundamentally, it is a nonlinear data structure composed of a collection of nodes that are related to each other in some way. When thinking of a graph, it is common to imagine a particular *embedding*, presumably in a Euclidean space, for the sake of visualizing the graph. Embedding the graph in a Euclidean space (say the plane) simply means associating each vertex of the graph with a point in the plane, and connecting with line segments the nodes that are connected to each other. Every proper embedding preserves the inner topology of the graph. There is nothing fundamental about any given proper embedding; however, there are embeddings that are more favorable than others if we want to discover or highlight some properties of the graph.

One way to study the properties of a graph is to compute and analyze the eigenvalues and eigenfunctions of its adjacency matrix, and other matrices that are derived from it. This branch of mathematics is called spectral graph theory and has a rich history of applications in machine learning that include clustering [73, 107], semi-supervised learning [1, 10, 43], and multi-fidelity modeling [84].

As an example, if we were to use a graph to describe a social network of human beings, then each person could be a node, and the their age, weight, interests, and income could be the nodal attributes. The edges could be used to represent whether two persons know and interact with each other, and the elements of the adjacency matrix could denote the strength of this connection.

Alternatively, if we were to us a graph to define the state of an elastic solid, then each material point can be a node whose features include the local loading, the stress state, and the material properties, whereas the edges can incorporate the physical distance among nodes.

Mesh-based simulation in computational physics In a typical problem in computational physics, a mathematical model is usually used to describe the behavior or the evolution of a system in terms of a set of PDEs, defined over a specific domain in space and time. Solving these differential equations analytically is often not possible. In these cases, numerical methods are employed to solve these equations. These

include the finite difference method, the finite volume method and the finite element method. In all these methods, the space-time domain is discretized into smaller sub-domains that are used to solve the PDE approximately. The data structure that represent this discretization is often called the "mesh", and consists of nodes, volumetric elements, faces and edges. Once the nodes are numbered, the information regarding the edges is contained in a connectivity matrix which determines which nodes in the computational mesh are connected to each other.

Computational meshes may be structured or unstructured. In three spatial dimensions, a structured mesh is composed of cuboids that divide the physical domain in a regular, grid-like fashion. This mesh is very efficient to work with since its representation benefits from a one-to-one correspondence with a simple Euclidean grid. This means that the connections among nodes can be easily recovered from their numbers and do not require special data structures. On the other hand, unstructured meshes cannot be mapped back to a Euclidean grid and are generally composed by elements of different shapes that include tetrahedrons, wedges, prisms and hexahedrons. Unlike structured meshes, unstructured meshes require special data structures to store information about how different entities are connected to each other. However, they are can easily represent complex physical domains.

Based on the description of a computational mesh and a graph, it should be clear to the reader that these two concepts are closely related to each other. In fact, it is easy to see that one can represent the nodes and the edges in a computational mesh as a as a graph $M = (V,\ E)$. In this graph, the physical quantities defined at the nodes (for example, pressure and velocity for the incompressible Navier Stokes equations) can be thought of as the attributes of the node. Furthermore the information contained in the connectivity matrix can be used as adjacency matrix. For reasons that will become more clear, a bidirectional graph is preferable for this purpose, implying that the connections between the nodes are not symmetric.

6.8.2 Architecture of MGNs

MGNs comprise an Encode-Process-Decode architecture that takes a graph $M = (V,\ E)$ as input and produces another graph as output. In this context, M is a mesh-based representation of the state of a physical system, hence the number of nodes $|V| = n$ of this graph is equal to the number of computational nodes of the input mesh. All vertices and edges are equipped with a set of features, denoted by $\mathbf{V}_i \in \mathbb{R}^{d_V}$ and $\mathbf{E}_{ij} \in \mathbb{R}^{d_E}$. The vertex features, $\mathbf{V}_i$, include all the physical quantities that characterize the input state. The edge features, $\mathbf{E}_{ij}$, can include geometrical properties of the nodes, and are used to communicate information about the physical domain to the model. For example, these could be the absolute or the relative coordinates of the vertices, that is $\mathbf{E}_{ij} = (\mathbf{x}_i, \mathbf{x}_{ij})$, where $\mathbf{x}_{ij} = \mathbf{x}_i - \mathbf{x}_j$.

The map from the input to the output graph is accomplished via the three steps described below.

Encoding The first involves encoding all the node and edge features into a latent space using the encoders $\boldsymbol{\epsilon}_E$ and $\boldsymbol{\epsilon}_V$. That is,

$$\mathbf{e}_{ij} = \boldsymbol{\epsilon}_E \left(\mathbf{E}_{ij};\, \boldsymbol{\theta}_{\boldsymbol{\epsilon}_E}\right) \tag{6.61}$$

$$\mathbf{v}_i = \boldsymbol{\epsilon}_V \left(\mathbf{V}_i;\, \boldsymbol{\theta}_{\boldsymbol{\epsilon}_V}\right) \tag{6.62}$$

Here $\boldsymbol{\epsilon}_E(\cdot;\, \boldsymbol{\theta}_{\boldsymbol{\epsilon}_E})$ and $\boldsymbol{\epsilon}_V(\cdot;\, \boldsymbol{\theta}_{\boldsymbol{\epsilon}_V})$ are MLPs with trainable parameters $\boldsymbol{\theta}_{\boldsymbol{\epsilon}_E}$ and $\boldsymbol{\theta}_{\boldsymbol{\epsilon}_V}$. We note that the dimension of the latent vector is usually higher than the number of features. Thus the encoders embed the node and edge features into a higher-dimensional space.

Processing This phase is the core of the MGN algorithm. It consists of L identical message passing steps, which iteratively update the node and edge features for an entity based on its neighbor's features. Each step is accomplished by a neural network with trainable parameters, and the steps are applied iteratively. This operation is given by $l \in \{1, \ldots, L\}$,

$$\mathbf{e}_{ij}^{(l)} = \boldsymbol{f}_E^{(l)}(\mathbf{e}_{ij}^{(l-1)},\, \mathbf{v}_i^{(l-1)},\, \mathbf{v}_j^{(l-1)};\, \boldsymbol{\theta}_{\boldsymbol{f}_E}^{(l)}) \tag{6.63}$$

$$\mathbf{e}_i^{(l)} = \sum_{j \in N(i)} \mathbf{e}_{ij}^{(l)} \tag{6.64}$$

$$\mathbf{v}_i^{(l)} = \boldsymbol{f}_V^{(l)} \left(\mathbf{v}_i^{(l-1)},\, \mathbf{e}_i^{(l)};\, \boldsymbol{\theta}_{\boldsymbol{f}_V}^{(l)}\right) \tag{6.65}$$

where $\boldsymbol{f}_E^{(l)}(\cdot;\, \boldsymbol{\theta}_{\boldsymbol{f}_E}^{(l)})$ and $\boldsymbol{f}_V^{(l)}(\cdot;\, \boldsymbol{\theta}_{\boldsymbol{f}_V}^{(l)})$ are MLPs with trainable parameters $\boldsymbol{\theta}_{\boldsymbol{f}_E}^{(l)}$ and $\boldsymbol{\theta}_{\boldsymbol{f}_V}^{(l)}$, respectively, and $N(i)$ is the set of all neighboring nodes for the node i. Note that the MLPs are not the same across the L steps.

To understand the rationale behind this step, consider the node i. During each message passing step, the features for all edges that include this node ($\mathbf{e}_{ij}^{(l)}$) are updated using their values at the previous step ($\mathbf{e}_{ij}^{(l-1)}$), and the values of the features of this node ($\mathbf{v}_i^{(l-1)}$) and its neighbors ($\mathbf{v}_j^{(l-1)}$, $j \in N(i)$) at the previous time step. Thus we may interpret this update as a "message" from the nodes j to the node i. In the next step all messages $\mathbf{e}_{ij}^{(l)}$ are aggregated into $\mathbf{e}_i^{(l)}$, by a sum. This represents the overall message sent to node i from all its neighbors. It is worth nothing that the message aggregation operation must be invariant to a permutation of the individual messages, because there is no intrinsic order in a graph. Given this, we can choose more elaborate aggregation operation than a simple sum as long as they respect this important property. Finally, the features for the i-th node ($\mathbf{v}_i^{(l)}$) are updated using their previous value ($\mathbf{v}_i^{(l-1)}$) and the aggregated message ($\mathbf{e}_i^{(l)}$).

The rationale for multiple massage passing is that after one step, each node gathers information only from its closest neighbors. However, after the second block, the neighbor has gathered information from its neighbors, and thus the original node can now be influenced by nodes that are 2 hops away. In this way after L steps each node is influenced by nodes that are L hops away. Therefore, using multiple

message passing steps presents a way to incorporate increasingly non-local effects on the solution to a system.

The message passing step (6.63) also makes it clear that the message from the node j to the node i is not the same as the message from the node i to the node j. This bi-directionality in the messages is also the reason why bi-directed graph is used for representing the mesh.

Decoding To compute the features of the target output graph, the latent node features at the end of the processing blocks ($\mathbf{v}_i^{(L)}$) need to be "decoded" to the physical space. This is accomplished via the decoder network $\boldsymbol{\delta}_V$, which is yet another MLP with trainable parameters $\boldsymbol{\theta}_{\delta_V}$. This operation is given by

$$\bar{\mathbf{V}}_i = \boldsymbol{\delta}_V \left(\mathbf{v}_i^{(L)};\ \boldsymbol{\theta}_{\delta_V} \right) \tag{6.66}$$

The complete MeshGraphNets algorithm is reported in Algorithm (6.1), and a schematic of the workflow is showed in Fig. 6.8.

Algorithm 6.1: MeshGraphNets

Input: $M = (V,\ E)$

▷ Encoding

for $i \in V,\ j \in N(i)$ **do**
  $\mathbf{v}_i^{(0)} = \boldsymbol{\epsilon}_V \left(\mathbf{V}_i;\ \boldsymbol{\theta}_{\epsilon_V} \right)$
  $\mathbf{e}_{ij}^{(0)} = \boldsymbol{\epsilon}_E \left(\mathbf{E}_{ij};\ \boldsymbol{\theta}_{\epsilon_E} \right)$
end

▷ Processing

for $l = 1,\ \ldots,\ L$ **do**
  for $i \in V$ **do**
    for $j \in N(i)$ **do**
      $\mathbf{e}_{ij}^{(l)} = \boldsymbol{f}_E^{(l)}(\mathbf{e}_{ij}^{(l-1)},\ \mathbf{v}_i^{(l-1)},\ \mathbf{v}_j^{(l-1)};\ \boldsymbol{\theta}_{\boldsymbol{f}_E}^{(l)})$ ▷ Message computation
    end
    $\mathbf{e}_i^{(l)} = \sum_{j \in N(i)} \mathbf{e}_{ij}^{(l)}$ ▷ Aggregation
    $\mathbf{v}_i^{(l)} = \boldsymbol{f}_V^{(l)} \left(\mathbf{v}_i^{(l-1)},\ \mathbf{e}_i^{(l)};\ \boldsymbol{\theta}_{\boldsymbol{f}_V}^{(l)} \right)$ ▷ Update
  end
end

▷ Decoding

for $i \in V$ **do**
  $\bar{\mathbf{V}}_i = \boldsymbol{\delta}_V \left(\mathbf{v}_i^{(L)};\ \boldsymbol{\theta}_{\delta_V} \right)$
end

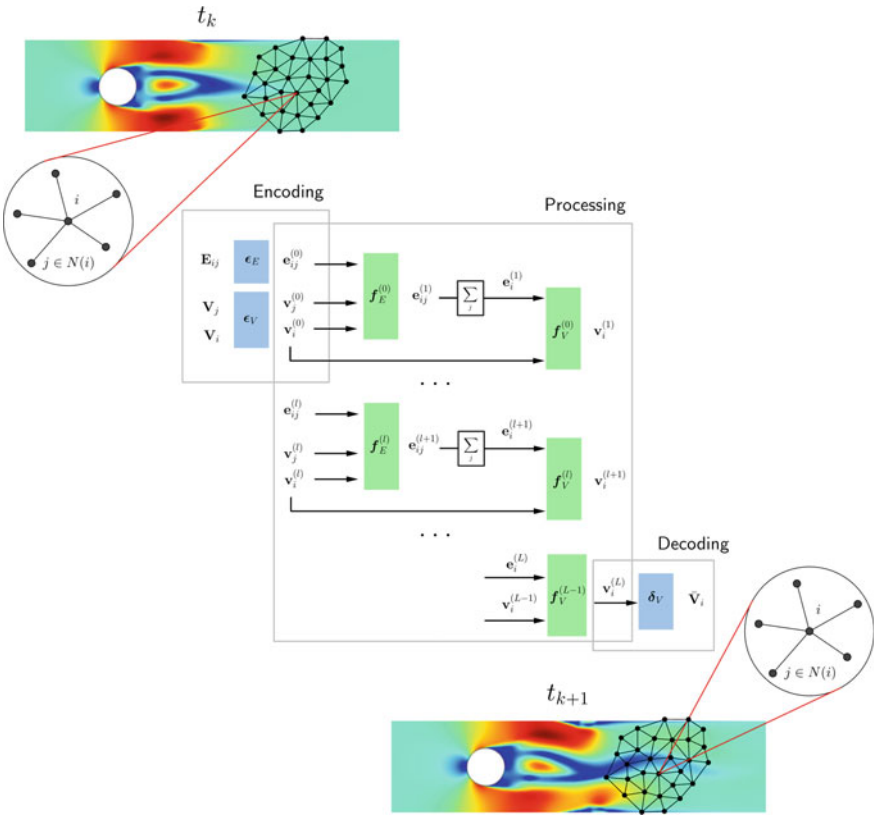


Fig. 6.8 Schematic of the MGN operator workflow

6.8.3 Training MGNs

In this section we describe how the MGN is trained. For this it is worth remembering that an MGN is a graph-to-graph model – that is it takes a graph as input and generates another graph as output.

The neural networks that comprise an MGN are the encoders ϵ_E, ϵ_V, the L pairs of update networks $f_E^{(l)}$, $f_V^{(l)}, l = 1, \cdots, L$, and the decoder δ_V. Let us denote the set of all the weights and biases of these networks as $\theta = [\theta_{\epsilon_V}, \theta_{\epsilon_E}, \theta_{f_E}^{(0)}, \theta_{f_V}^{(0)}, \ldots, \theta_{f_E}^{(L-1)}, \theta_{f_V}^{(L-1)}, \theta_{\delta_V}]$. Then, given a set of observed input and output graphs, training an MGN amounts to finding the optimal set of parameters θ^* that minimize the discrepancy between the predicted and the observed output graphs. The discrepancy between two graphs can be defined by a mean measure of the difference between the attributes of their nodes. The type of observed input and output graph depends upon the problem we wish to solve. In this context, it is useful to consider steady state and time-dependent problems separately.

Steady-state problems As discussed at the beginning of the chapter, for steady-state PDEs, the goal is to approximate the operator $\mathcal{N}$ that maps an input function $a \in A$ to the solution u of the governing equations. A few examples were provided, such as solving the heat equation for the temperature u on a given domain Ω_X with different source terms f, conductivity fields κ, or boundary conditions g. Or, also, solving an elastic problem for the displacement $\boldsymbol{u}$ of a solid subject to different loading conditions $\boldsymbol{f}$, with specified material properties. A surrogate model for these kind of problems can be used for real-time applications, uncertainty quantification, or end-to-end optimization.

Analogously to other operator networks, one can build a training dataset by solving the physical problem for different instances of the input function a.

$$\mathcal{S} = \{(\boldsymbol{a}^{(k)},\ \boldsymbol{u}^{(k)}) : k = 1,\ \ldots,\ N\}$$

where $\boldsymbol{a}^{(k)}$ and $\boldsymbol{u}^{(k)}$ store the values of input function and final solution at every grid point of the mesh, respectively.

For an MGN, a graph representation of these data is needed. To accomplish this we define graphs for the input functions $M_a^{(k)}$ and for the output solutions $M_u^{(k)}$, so the graph-pairs can be assembled as

$$\mathcal{S}_{\mathcal{M}} = \{(M_a^{(k)},\ M_u^{(k)}) : k = 1,\ \ldots,\ N\}$$

We note that the attributes type of the nodes of M_a are different than the attributes for the nodes of M_u. For example, in the case of the heat equation, the attributes for the i-th node of the input graph M_a are the values of the source term, f_i, or the conductivity, κ_i, at that node. Whereas, the attribute at the same node for the output graph M_u is the value of the temperature attained at that node. Similarly, for the elasticity problem, the nodal attributes of the input graph can be the values of the body force or material properties at that node, and the nodal attributes of the output graph can be the values of the displacement and stress fields at that node.

The loss function for the MGN is given by

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N|V|} \sum_{k=1}^{N} \sum_{i \in V^k} |\mathbf{V}_i^{(k)} - \hat{\mathbf{V}}_i^{(k)}|^2. \tag{6.67}$$

Here, given the i-th node, and the k-th input-output graph pair, $\mathbf{V}_i^{(k)}$ are the observed nodal features of output graph, and $\bar{\mathbf{V}}_i^{(k)}$ are nodal features of the output graph predicted by the MGN while using the corresponding input graph as input. The loss described above is the mean-square loss, though other loss terms, like the mean absolute difference are also possible. Some regularization of the type described in Sect. 2.5.1 is also typically added. Finally, training the MGN corresponds to finding $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$. Once the training is complete, the MGN model can be used to

make predictions for a new instance of an input functions a, by using as input the corresponding input graph M_a.

Time dependent problems When dealing with time dependent problems, the goal is usually to predict the state of a system $u(t,\ \boldsymbol{x}),\ \boldsymbol{x} \in \Omega_X$ at any instant in the future $t > t_0$ given the initial condition $u(t_0,\ \boldsymbol{x})$. The application of a MGN to this problem relies on using the MGN auto-regressively. That is, the MGN is trained to take the solution at a given time t as input to predict the solution at $t + \Delta t$. Once trained, this MGN can be applied regressively to march the solution over multiples of Δt. The training of this type of MGN is described next.

The first step involves solving for the evolution of the system of interest for N time steps $t_k = k\Delta t,\ k = 1,\ \ldots,\ N$. This requires the generation of a computational mesh to discretize the domain Ω_X, and the use of a numerical solver to integrate the equations in time. The result is stored in a collection of vectors,

$$\mathcal{S} = \{\boldsymbol{u}^{(k)} : k = 0,\ \ldots,\ N\},$$

where $u_i^{(k)} = u(t^k,\ \boldsymbol{x}_i)$ is the solution at time t^k at the i-th node whose coordinates are denoted by $\boldsymbol{x}_i$.

The next step involves building a graph representation for this data set. This is done by using the nodes of the mesh to define the vertices of the graph, the vectors $\boldsymbol{u}^k$ to define the nodal attributes, and the connectivity of the mesh to define the edges. The result is a finite set of graphs $\{M^{(0)},\ \ldots,\ M^{(N)}\}$, with $M^{(k)} = M(t_k) = (V^k,\ E^k)$. This set can also be interpreted as a sequence of snapshots at different time stamps t_k of an evolving graph, where the attributes of each node are allowed to change with time.

Given this, the MGN can be thought of as the operator that maps $M^{(k-1)} \rightarrow M^{(k)}$ for any value of k. We note that it is the same MGN that is applied for all values of k. To train this MGN we build the input-out pairs of graphs

$$\mathcal{S}_{\mathcal{M}} = \{(M^{(0)},\ M^{(1)}),\ (M^{(1)},\ M^{(2)}),\ \ldots,\ (M^{(N-1)},\ M^{(N)})\}.$$

The loss function for the MGN is given by (6.67) where once again, given the i-th node, and the k-th input-output graph pair, $\mathbf{V}_i^{(k)}$ are the observed nodal features of output graph, and $\bar{\mathbf{V}}_i^{(k)}$ are nodal features of the output graph predicted by the MGN while using the corresponding input graph as input. The MGN is trained by finding the values of the network parameters that minimize this loss.

Once the MGN is trained it is applied to increment the solution from time t to $t + \Delta t$. By consider the output of the MGN at $t + \Delta t$ as the input for the next application of the MGN, we can compute the solution at $t + 2\Delta t$. These recursive steps can be applied several times to advance the solution over a substantial time interval. As an example consider the compressible Navier Stokes equations solved in complex but fixed domain. The solution involves determining the pressure (p), density (ρ), and the velocity field ($\boldsymbol{u}$) at every node on the mesh at every time step.

Thus after N such time steps we have $N+1$ graphs $\{M^{(0)}, \ldots, M^{(N)}\}$ where the attributes of the node i at time step k is $\mathbf{V}_i^{(k)} = (p_i^k, \rho_i^k, \boldsymbol{u}_i^k)$.

Remark 6.8.1 In the original paper [82], for the output graph, the nodal attributes are the time derivatives of the physical quantities, that are denoted by $\dot{\mathbf{V}}_i^{(k)}$. These are integrated in time with an explicit Euler scheme to obtain the desired physical quantities, $\mathbf{V}_i^{(k+1)} = \mathbf{V}_i^{(k)} + \Delta t \dot{\mathbf{V}}_i^{(k)}$.

Remark 6.8.2 It is worth noting that for time-dependent problems, a trained MGN works only for the fixed time step size that was used to generate the training data.

Remark 6.8.3 In this chapter we described an MGN that was applied to a problem with a mesh that did not change with time. However, for some problems, like those based on a Lagrangian [9, 42], or an Arbitrary Lagrangian Eulerian (ALE) description [24, 41], the mesh itself evolves with time. This is particularly useful for modeling moving material interfaces. The MGN framework can also be applied to such problems. In fact, the original MGN reference [82], contains the appropriate approach to solve these types of problems.

Remark 6.8.4 In order to distinguish boundary nodes from interior nodes, one may introduce a simple one-hot-encoded nodal attribute and use it in the MGN formulation (Fig. 6.8).

6.9 Computational Exercise: Deep Operator Networks (DeepONets)

In this exercise you will implement, train and test a DeepONet. The DeepONet will map a function defined on the domain $\Omega = [0, 1]$ to another function also defined on the same domain. The input function is the initial condition to the viscous Burgers equation, and the output function is the solution at $t = 1$.

The architecture of the DeepONet is shown in Fig. 6.9. The branch will map a vector, $\boldsymbol{a} \in \mathbb{R}^k$ to the another vector, $\boldsymbol{b} \in \mathbb{R}^p$. The trunk vector will map $x \in \mathbb{R}^1$ to the another vector, $\boldsymbol{b} \in \mathbb{R}^p$. The output of the network will be the dot product $u = \boldsymbol{b} \cdot \boldsymbol{\tau}$.

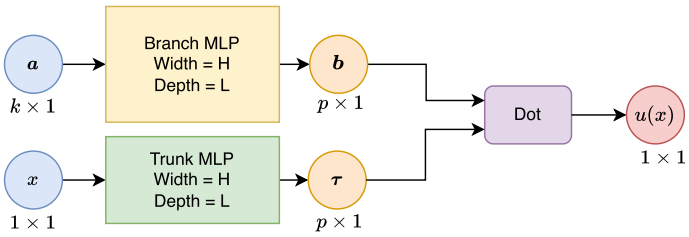


Fig. 6.9 Architecture of the DeepONet

There are three tasks:

1. Download the folder `DeepONet_datasets.zip` where you will find the following `numpy` files:

   ```
   train_branch_input.npy, train_trunk_input.npy, train_output.npy
   ```

   Load them on your notebook, determine the dimensions of the arrays in these files and explain why each dimension makes sense. Using this information, determine the value of k (see Fig. 6.2) for this problem. Create the training dataset for the DeepONet using the class `CustomDataset` by combining all these arrays:

   ```
   class CreateDataset(Dataset):
       def __init__(self,branch_input,trunk_input,output):
           self.branch_input = torch.Tensor(branch_input)
           self.trunk_input = torch.Tensor(trunk_input)
           self.output      = torch.Tensor(output)

       def __len__(self):
           return len(self.branch_input)

       def __getitem__(self, idx):
           if torch.is_tensor(idx):
               idx = idx.tolist()
           return self.branch_input[idx],
               self.trunk_input[idx],self.output[idx]
   ```

2. Train the DeepOnet with the architecture shown in Fig. 6.2. Within the training loop, define the branch and trunk networks using the `MLP` class from Chap. 2. Then generate the final model output by computing the dot product of the output from these networks. Pay close attention to the following:

   (a) Use $L = 6$, $H = 20$, $p = 25$ and `tanh` activation.
   (b) Use

   ```
   all_params = list(branch_net.parameters()) + list(trunk_net.parameters())
   ```

   to combine the parameters from the branch and trunk networks.
   (c) Set batch size to 100, maximum epochs to 4,000, regularization to 10^{-7} and learning rate to 10^{-3}. Use Adam optimizer with `betas=(0.5,0.9)`.

   Generate loss versus epochs curve.
3. Using the test data provided to you, for each input function, generate the predicted solution.

   (a) On a single figure plot the following:
      i. The input function. To plot this you will need `test_u0_at_sensors.npy`, `x_sensor.npy`.
      ii. The predicted output function for which you will need `x_fullfield.npy` and your trained DeepONet model.

iii. The true output function given by `test_u_fullfield.npy`, `x_fullfield.npy`.

(b) On a second figure, plot the the error = predicted - true output solution.

We have provided 5 test cases. So this means that you are required to generate 5 $\times$ 2 figures for the test data.

Chapter 7
Generative Deep Learning

Generative AI has captured the interest of a large section of our society of late. This interest has been created by the impressive results obtained by AI tools like ChatGPT, DallE, and Stability.ai. While the application domains for these tools differ significantly, and the details of the algorithms within them are also different, they share a common feature in that they are generative algorithms. Although we may have some intuitive feel for what the word generative implies, it is worth defining it with some rigor in order to make progress in understanding these algorithms. This is the main focus of this chapter. That is to provide the reader with the background required to understand the broad principles behind generative algorithms, and also to give them some specific example of a few (and by no means all) such algorithms.

7.1 Generative Algorithms

For our purpose, we group generative algorithms into to two broad classes. The first class is of "pure generative" algorithms. These are algorithms that are trained on samples of data drawn from an underlying, unknown, probability distribution, and once trained they generate new samples that are also drawn from the same distribution. For a simple example consider a dataset like CelebA [60] which comprises RGB images of celebrities. Then one can consider developing a generative algorithm which uses all these images for training, and then during execution will produce an image which will look remarkably like a human face, but will not be the same as one of the faces used for training the algorithm. Or perhaps consider the following example which stems from to physics and engineering. Consider an algorithm that is trained on lots of images of the micro-structure of a binary composite material, and then produces new microstructural images that are different and yet look a lot like the training images.

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1_7

The second class of generative algorithms we will address is that of "conditional generative" algorithms. In this case, the training data consists of pairs of entities, where each pair is sampled from the joint probability distribution of these entities. Once the algorithm is trained, the user supplies a fixed instance of one of these entities and the algorithm generates samples from a probability distribution that is conditioned on the input provided by the user. As an examples of this problem, we consider a small modification to one of the pure generative problems describe above. For training data, we consider the image of the face of a celebrity from CelebA and a label that denotes their gender. This pair of entities is used to train the algorithm. Once the algorithm is trained, the user supplies the label to the algorithm, say label = male. Then the algorithm generates new faces that look like (but are not the same as) all the male faces it was trained on.

It is relatively easy to see how conditional generative algorithms can play a very important role in physics and engineering. In particular, they are very useful in solving a large class of problems that are collectively referred to as inverse problems. The best way to understand an inverse problem is to consider an example.

Let's say that we are in possession of a metal plate with an initial temperature distribution. However, we have no way of measuring this distribution. On the other hand, we are able to measure the temperature at a later time but this measurement is noisy and sparse. That is, we are able to measure with a certain degree of accuracy only the temperature at some of locations on the plate. The inverse problem is then: given this sparse and noisy temperature field measured at a later time, generate samples from the probability distribution of the initial temperature field conditioned on the measurement. In order to connect this problem to the one in the previous paragraph, think of the measurement as the label, and the initial temperature field as an image of the face. Then once again, we are given the label and we wish to generate multiple samples of the face. If the reader stretches themselves, they will see that there are many important problems in science and engineering that can be cast into this type of a formulation of an inverse problem.

7.2 Introductory Concepts in Probability

Our qualitative discussion of generative problems in the previous paragraphs has made several references to probability distributions and density functions without really defining what these entities are. Most readers will have heard about them in their professional lives, and some likely have a strong understanding. In either case, one thing is clear, in order to understand generative algorithms at a level that is deeper than cursory, it is necessary to have some understanding of probability and statistics. In this section, we provide a very brief and admittedly incomplete summary of concepts in probability that are key to understanding generative algorithms. For a detailed understanding, we refer to reader to many outstanding texts on these topics [31, 71, 92].

7.2.1 Random Variables

We begin our discussion with the definition of a random variable. A **random variable** X attains values on the real line and comes equipped with a cumulative distribution function. We note that formally a random variable is associated with a sample space, event class and the probability law for a random event, and is defined as a function that maps the sample space of the random events to the real line. This association equips the random variable with a cumulative distribution function and in turn equips the cumulative distribution function with certain properties that are defined in the following section.

In order to make things precise let us define some examples of random variables (RVs):

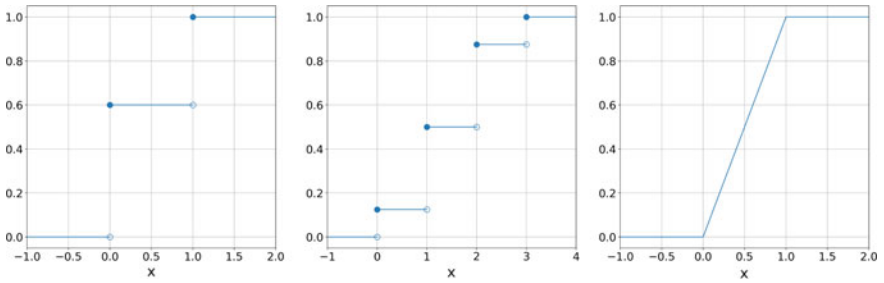
1. Consider a switch that can either be "on" or "off'. Further the probability of "on" $= p$, and probability of "off" $= 1 - p$, with $p \in [0, 1]$. Then we may define the RV
$$X = \begin{cases} 0 & \text{if switch = off} \\ 1 & \text{if switch = on.} \end{cases} \tag{7.1}$$
This is also known as a *Bernoulli Random Variable*.
2. Now consider a fair coin with equal probability of heads (H) or tails (T). Consider three consecutive tosses of this coin and define the random variable,
$$X = \text{Number of heads in three consecutive tosses.} \tag{7.2}$$
3. Now consider an unbiased spinner that stops at any angle in the interval $(0, 2\pi]$ with equal probability. We define a RV to be equal to the value of this angle divided by 2π. That is,
$$X = \text{angle at which the spinner comes to rest}/2\pi. \tag{7.3}$$

We note that the first two examples correspond to a discrete RV whereas the third example is that of a continuous RV. Next we define the cumulative distribution function and also list its properties.

7.2.2 Cumulative Distribution Function

The **cumulative distribution function** (cdf) of a random variable X is given by

$$F_X(x) = P[\xi : X(\xi) \leq x]$$



(a) Bernoulli (b) 3 coin tosses (c) Spinner

Fig. 7.1 Examples of cumulative distribution functions

which defines a probability on $\mathbb{R}$ of X taking values in the interval $(-\infty, x]$. Let us define the cdf for the above examples:

1. For the Bernoulli RV defined by (7.1)

   - if $x < 0$, then $F_X(x) = P[\text{neither on or off}] = 0$
   - if $0 \leq x < 1$, then $F_X(x) = P[\text{off}] = 1 - p$
   - if $x \geq 1$, then $F_X(x) = P[\text{either on or off}] = 1$

   The full cdf is shown in Fig. 7.1a.
2. For the RV defined by (7.2)

   - if $x < 0$, then $F_X(x) = P[\text{negative number of H}] = 0$
   - if $0 \leq x < 1$, then $F_X(x) = P[\#\text{ of H} = 0] = P[\text{TTT}] = 1/8$
   - if $1 \leq x < 2$, then $F_X(x) = P[\#\text{ of H} = 0, 1] = P[\text{TTT}] + P[\text{HTT or THT or } TTH] = 1/8 + 3/8 = 4/8$
   - if $2 \leq x < 3$, then $F_X(x) = P[\#\text{ of H} = 0, 1, 2] = 1/8 + 3/8 + 3/8 = 7/8$
   - if $x \geq 3$, the $F_X(x) = P[\#\text{ of H} = 0, 1, 2, 3] = 1$

   The full cdf is shown in Fig. 7.1b.
3. For the spinner experiment with the RV defined by (7.3)

$$F_X(x) = P[X \leq x] = P[\text{angle at rest} \leq 2\pi x]$$

   - if $x < 0$, then $F_X(x) = P[\text{angle at rest} < 0] = 0$
   - if $0 \leq x < 1$, then $F_X(x) = P[\text{angle at rest } \in (0, 2\pi x]] = \dfrac{\text{length of arc with angle} 2\pi x}{\text{length of arc with angle } 2\pi} = x$
   - if $x \geq 1$, then $F_X(x) = P[\text{angle at rest } \leq 2\pi] = 1$

   The full cdf is shown in Fig. 7.1c.

Let us discuss some properties of F_X. We note that these properties are inherited by the cdf through its association with an underlying random event.

1. $0 \leq F_X(x) \leq 1$.
2. $\lim_{x\to\infty} F_X(x) = 1$.
3. $\lim_{x\to-\infty} F_X(x) = 0$.
4. F_X is monotonically increasing.
5. The cdf is always continuous from the right

$$F_X(x) = \lim_{h\to 0+} F_x(x+h).$$

Note that the F_X for discrete RV (see Fig. 7.1) are discontinuous at finitely many x. In fact, the cdf for discrete RVs can be written as a finite sum of the form

$$F_X(x) = \sum_{k=1}^{K} p_k H(x - x_k), \quad \sum_{k=1}^{K} p_k = 1,$$

where p_k is the probability mass and H is the Heaviside function

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}.$$

Remark 7.2.1 Once we have the F_X we can calculate the probability that X will take values in "any" interval in $\mathbb{R}$, i.e., we can compute $P[a < X \leq b]$. Note that

$$\begin{aligned} F_X(b) = P[X \leq b] &= P[(X \leq a) \cup (a < X \leq b)] \\ &= P[X \leq a] + P[a < X \leq b] \qquad \text{(mutually exclusive events)} \\ &= F_X(a) + P[a < X \leq b]. \end{aligned}$$

Thus,

$$P[a < X \leq b] = F_X(b) - F_X(a).$$

7.2.3 Probability Density Function

The **probability density function** (pdf) f_X for a continuous RV X is defined as the derivative of the cdf F_X with respect to its argument. That is, for a continuous F_X, we have

$$f_X(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_X(x). \tag{7.4}$$

The pdf enjoys the following properties that it inherits from the cdf:

1. $f_X(x) \geq 0,\ \forall x \in \mathbb{R}$, since F_X is monotonically increasing.
2. $\lim_{x\to-\infty} f_X(x) = \lim_{x\to\infty} f_X(x) = 0$.
3. Integrating (7.4) from $(-\infty, x]$ gives us

$$\int_{-\infty}^{x} f_X(y)\mathrm{d}y = F_X(x) - \lim_{x\to-\infty} F_X(x) = F_X(x).$$

4. Also

$$P[a < X \leq b] = F_X(b) - F_X(a) = \int_{-\infty}^{b} f_X(y)\mathrm{d}y - \int_{-\infty}^{a} f_X(y)\mathrm{d}y = \int_{a}^{b} f_X(y)\mathrm{d}y.$$

   Thus, the integral of a pdf in an interval gives the "probability mass" which is the probability that the RV lies in that interval. This is the reason why the pdf is called a "density".
5. A very important property is that a pdf always integrates to 1,

$$\int_{-\infty}^{\infty} f_X(y)\mathrm{d}y = \lim_{x\to\infty} F_X(x) - \lim_{x\to-\infty} F_X(x) = 1.$$

6. For a very small $h > 0$, we have that

$$P[a < X \leq a + h] = \int_{a}^{a+h} f_X(y)\mathrm{d}y \approx h f_X(a).$$

   That is, we can interpret the pdf at a point x as a measure of the likelihood that the random variable will attain a value in a small neighborhood of x. Also, note that as $h \to 0^+$, $P[a < X \leq a + h] \to 0$. That is, for a continuous RV the probability of attaining a single value is zero.

A few remarks are in order. First, for a discrete RV the pdf contains a sum of Dirac-distributions and has to be interpreted in the sense of distributions. Second, in the following development we will work almost exclusively with continuous RVs and therefore we will assume the existence of a well-defined pdf. Finally, when working with generative models, we will find it much more convenient to work with pdfs than cdfs. Thus, it is recommended that the reader familiarizes themselves with the concept of a pdf before proceeding further.

7.2.4 Examples of Important Random Variables

We now look at some important random variables and the associated cdf and pdfs (also see Fig. 7.2). We note that an RV is completely defined by either its cdf or pdf. Thus defining an RV entails defining the functional form of either of these functions.

1. **Uniform RV**: As the name suggests, for a Uniform RV the pdf is uniformly distributed over a finite interval. That is, for some interval $(a, b]$, the pdf is given by

$$f_X(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in (a, b] \\ 0 & \text{other wise} \end{cases},$$

while the cdf is given by

$$F_X(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{b-a} & \text{if } x \in (a, b] \\ 1 & \text{if } x > b \end{cases}.$$

2. **Exponential RV**: This RV is used to model lifetime of devices/humans after a critical event. In this case, the RV X represents the time to failure and the probability that the failure will occur after some time x decreases exponentially. That is, $P[X > x] = e^{-\lambda x}$ where $\lambda > 0$ is a model parameter which denotes the rate of failure. Thus, the cdf is given by

$$F_X(x) = P[X \leq x] = 1 - P[X > x] = 1 - e^{-\lambda x}.$$

Therefore, the pdf is given by

$$f_X(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_X(x) = \lambda e^{-\lambda x}.$$

3. **Gaussian RV**: This RV is used to model the distribution of naturally occurring quantities like like height, weight, etc. within a population. In fact, through the Central Limit Theorem, this is also the distribution given by an aggregate of many RVs. In this case, the pdf is given by

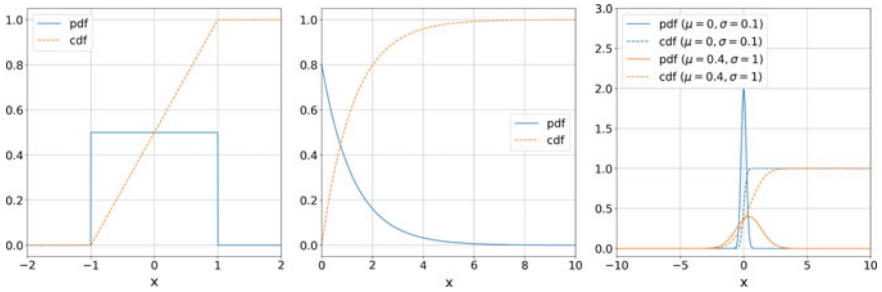
$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{7.5}$$

which is parameterized by the parameter μ which denotes the center of this distribution, and the parameter σ which denotes its spread. We will shortly see that μ turns out to be the mean of the distribution and σ^2 turns out to be the variance. The corresponding cdf is given by

$$F_X(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right], \quad \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t.$$

The Gaussian pdf (7.5) is often concisely represented by the symbol $N(\mu, \sigma)$.

In probabilistic Machine Learning one makes extensive use of uniform and Gaussian random variables.



(a) Uniform RV ($a = -1, b = 1$) (b) Exponential RV ($\lambda = 0.8$) (c) Gaussian RV

Fig. 7.2 Continuous random variables

7.2.5 Expectation and Variance of RVs

Given a RV X with pdf f_X, we can calculate its **expected value** or **expectation** or **mean** as

$$\mu_X := \mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) \mathrm{d}x.$$

The expectation has the following properties:

- Note that if a pdf is symmetric about $x = m$, then $\mathbb{E}[X] = m$. To see this, note that $(m - x) f_X(x)$ will be anti-symmetric about m. Thus

$$0 = \int_{-\infty}^{\infty} (m - x) f_X(x) \mathrm{d}x = m \int_{-\infty}^{\infty} f_X(x) \mathrm{d}x - \int_{-\infty}^{\infty} x f_X(x) \mathrm{d}x \implies \int_{-\infty}^{\infty} x f_X(x) \mathrm{d}x = m.$$

  Using this property, we can easily say the mean for a uniform RV is $(a + b)/2$, while for a Gaussian RV it is μ.
- $\mathbb{E}[c] = c$ for a constant c.
- We can calculate the expected value of functions of RVs as

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x) \mathrm{d}x.$$

- The expectation is linear, i.e.,

$$\mathbb{E}[g(X) + ch(X)] = \mathbb{E}[g(X)] + c\mathbb{E}[h(X)].$$

The **variance** of a RV measures its variation about the mean. It is evaluated as

$$\mathrm{VAR}[X] = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x) \mathrm{d}x.$$

Furthermore, we denote the **standard deviation** as

$$\sigma_X := \text{STD}[X] = \sqrt{\text{VAR}[X]}.$$

For a uniform RV

$$\text{VAR}[X] = \int_a^b \left(x - \frac{b+a}{2}\right)^2 \frac{1}{b-a} \mathrm{d}x = \frac{(b-a)^2}{12}.$$

For a Gaussian RV, we first use the property that the pdf integrates to unity to write

$$\int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \mathrm{d}x = \sqrt{2\pi}\sigma.$$

Taking a derivative with respect to σ on both sides lead to

$$\int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} (x-\mu)^2 \sigma^{-3} \mathrm{d}x = \sqrt{2\pi}$$

which after some algebra gives us

$$\text{VAR}[X] = \int_{-\infty}^{\infty} (x-\mu)^2 \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \mathrm{d}x = \sigma^2.$$

Thus the variance of a Gaussian RV is σ^2 and its standard deviation is σ.

7.2.6 Random Vectors

Recall that we defined a random variable as an entity that attains values on the real line and comes equipped with a cdf (or alternately a pdf). Similarly, we may define a random vector as an entity that attains values in $\mathbb{R}^d$, where d is a positive integer, and comes equipped with a joint cdf, or a joint pdf. We will find it very useful to work with joint pdfs rather than joint cdfs and therefore we will almost exclusively focus on defining random vectors through their joint pdfs. We also note that just like a random variable, a random vector is also associated with the sample space, event class and the probability law for a random event. It is defined as a function that maps the sample space of the random event to $\mathbb{R}^d$. In that sense, a random variable is a special case of a random vector when $d = 1$.

To consider a simple example, let's return to the spinner. We now spin the spinner twice and measure $\psi_1 \in (0, 2\pi]$, $\psi_2 \in (0, 2\pi]$. In this case, we can define a random vector $\boldsymbol{X}$, where

$$X_1 = \frac{\psi_1}{2\pi}, \quad X_2 = \frac{\psi_2}{2\pi}.$$

Note that for this random vector $d = 2$.

7.2.7 Joint Probability Density Function

For a jointly continuous random vector the joint pdf, denoted by $f_{\boldsymbol{X}} : \mathbb{R}^d \mapsto \mathbb{R}$, is such that the probability of $\boldsymbol{X}$ landing in $\Omega \subset \mathbb{R}^d$ is given by the integral of the joint pdf over Ω. That is,

$$P[\boldsymbol{x} \in \Omega] = \int_{\Omega} f_{\boldsymbol{X}}(\boldsymbol{x}) d\boldsymbol{x}.$$

The joint pdf also inherits the following properties:

- $f_{\boldsymbol{X}}(\boldsymbol{x}) = 0$ as $|\boldsymbol{x}| \to \infty$.
- $f_{\boldsymbol{X}}(\boldsymbol{x}) \geq 0, \forall \boldsymbol{x} \in \mathbb{R}^d$.
- $\int_{\mathbb{R}^d} f_{\boldsymbol{X}}(\boldsymbol{x}) d\boldsymbol{x} = 1$.

It is easy to see that for the two-spinner example described above the joint pdf is given by

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } x_1 \in (0, 1] \text{ and } x_2 \in (0, 1] \\ 0 & \text{otherwise} \end{cases}.$$

7.2.8 Examples of Important Random Vectors

We now describe two important classes of random vectors that find common use in generative models. These are joint uniform random variables and joint Gaussian random variables. In each case we "define" the random vector by presenting an analytical expression for the joint pdf and describing the parameters that appear in it.

1. **Joint uniform RV:** Consider the $d-$dimensional box formed by the tensor product of the intervals $(a_i, b_i], i = 1, \cdots, d$, and $b_i = a_i + \delta_i, \delta_i > 0$. Then the joint pdf for a random vector with uniform distribution in this box is given by

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \begin{cases} \frac{1}{\prod_{i=1}^{d} \delta_i} & \text{if } \boldsymbol{x} \in \text{ the box} \\ 0 & \text{otherwise} \end{cases}. \tag{7.6}$$

2. **Joint Gaussian RV:** The joint pdf for joint Gaussian RVs is given by

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2}\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^{\top} \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right] \tag{7.7}$$

where $\boldsymbol{x} = (x_1, \cdots, x_d)$, $\boldsymbol{\mu} \in \mathbb{R}^d$ is the mean, and $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$ is called the covariance matrix and is symmetric and positive definite. In this case, all components of $\boldsymbol{X}$ are normally distributed, and the covariance (see Sect. 7.2.9) between X_i and X_j is given by Σ_{ij}.

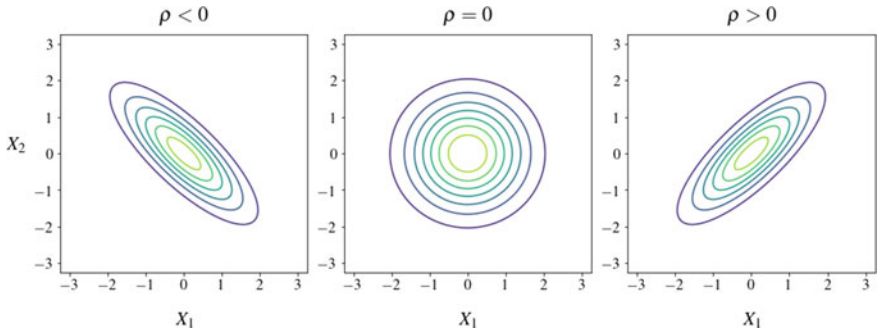


Fig. 7.3 Contours of a bivariate normal distribution for different values of correlation ρ

Often, the joint Gaussian pdf is represented as $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, since $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ completely define the pdf.

It is useful to consider the special case of $d = 2$, then the covariance matrix can be written as

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix}.$$

We can interpret σ_1^2 and σ_2^2 as the variance along the X_1 and X_2 coordinates, and ρ as a parameter that represents the correlation between these variables. In Fig. 7.3, we have plotted contours of this pdf for different values of ρ with $\sigma_1, \sigma_2 = 1$ and $\mu_1, \mu_2 = 0$. We can see that if $\rho \neq 0$, the values of one variable are correlated with the values of the other.

7.2.9 Expectation and Covariance of Random Vectors

Consider the function $g(\boldsymbol{X})$, which can be scalar-, vector-, or tensor-valued, then its expected value is given by

$$\mathbb{E}[g(\boldsymbol{X})] = \int_{\mathbb{R}^d} g(\boldsymbol{x}) f_X(\boldsymbol{x}) \mathrm{d}\boldsymbol{x},$$

as long as the integral is defined.

The simplest instance is $g(\boldsymbol{X}) = \boldsymbol{X}$ which gives us the expectation or the expected value or the mean, $\boldsymbol{\mu}_X \in \mathbb{R}^d$, of the random vector:

$$\boldsymbol{\mu}_X \equiv \mathbb{E}[\boldsymbol{X}] = \int_{\mathbb{R}^d} \boldsymbol{x} f_X(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}.$$

Similarly the covariance, $\text{COV}[\boldsymbol{X}] \in \mathbb{R}^{d\times d}$, of a random vector is defined as,

$$\text{COV}[\boldsymbol{X}] \equiv \mathbb{E}[(\boldsymbol{X}-\mu_{\boldsymbol{X}})\otimes(\boldsymbol{X}-\mu_{\boldsymbol{X}})] = \int_{\mathbb{R}^d}(\boldsymbol{x}-\mu_{\boldsymbol{X}})\otimes(\boldsymbol{x}-\mu_{\boldsymbol{X}})\,f_{\boldsymbol{X}}(\boldsymbol{x})\mathrm{d}\boldsymbol{x}.$$

Here $\otimes$ represents the outer product between two vectors. The i-th diagonal component of the covariance matrix represents the variation in the random variable X_i. The off-diagonal components of the covariance matrix represent the correlation between the two corresponding components of the random vector. The sign of the the off-diagonal component determines the sign of this correlation. If the off-diagonal component is zero, then the two components are said to be uncorrelated.

Using the definitions of the mean and the covariance, and the expression for the joint pdf for a Gaussian random vector (7.7), it easily verified that for a Gaussian random vector the mean, $\boldsymbol{\mu}_{\boldsymbol{X}} = \boldsymbol{\mu}$, and the covariance, $\text{COV}[\boldsymbol{X}] = \boldsymbol{\Sigma}$.

7.2.10 Marginal and Conditional Probability Density Functions

Sometimes, for a random vector it is useful to determine the distribution of some of its components given (a) no knowledge about the remainder of its components, and (b) precise values of the remainder of its components. In this case, the first question is answered by the marginal pdf while the second question is addressed by the conditional pdf. These concepts, which are very important to generative modeling, are described in this section.

Let $\boldsymbol{V} \in \mathbb{R}^{d+D}$ be a random vector. We assume that we can split this vector into its components, $\boldsymbol{V} = (\boldsymbol{X}, \boldsymbol{Y})$, where $\boldsymbol{X} \in \mathbb{R}^d$ and $\boldsymbol{Y} \in \mathbb{R}^D$ are in turn random vectors. Further let the joint pdf for $\boldsymbol{V}$ be denoted by $f_{\boldsymbol{V}}(\boldsymbol{v})$. We may also write this joint pdf as $f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y})$.

First we consider the question, what is the distribution of $\boldsymbol{Y}$ given no knowledge about $\boldsymbol{X}$? The best we can do in this case is assume that $\boldsymbol{X}$ will attain all possible values as determined by the joint pdf. That is, the pdf we are looking for is given by

$$f_{\boldsymbol{Y}}(\boldsymbol{y}) = \int_{\mathbb{R}^d} f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y})\mathrm{d}\boldsymbol{x}.$$

This is the marginal pdf for $\boldsymbol{Y}$. It is easy to show that function defined above satisfies all the properties of a pdf. For example, we note that as required it does integrate to unity,

$$\int_{\mathbb{R}^D} f_{\boldsymbol{Y}}(\boldsymbol{y})\mathrm{d}\boldsymbol{y} = \int_{\mathbb{R}^D}\int_{\mathbb{R}^d} f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y})\mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y} = \int_{\mathbb{R}^{d+D}} f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y})\mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y} = 1$$

where the last equality is obtained from the property of the joint pdf $f_{\boldsymbol{XY}}$.

Note that we can equivalently define a marginal pdf for $\boldsymbol{x}$ given by

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \int_{\mathbb{R}^D} f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y}) \mathrm{d}\boldsymbol{y}.$$

Next we consider the question: what does the distribution of $\boldsymbol{Y}$ look like when we know that $\boldsymbol{X} = \boldsymbol{x}$, a specified value. An intuitive answer might be to consider the joint pdf as a function of $\boldsymbol{y}$ at a fixed value of $\boldsymbol{x}$. This is correct up to a multiplicative constant, which is needed because the joint pdf at a fixed value of $\boldsymbol{x}$ does not integrate to unity. Thus the correct answer is the conditional distribution which is denoted by $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x})$, or in a slightly simpler notation by $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\boldsymbol{x})$. This is given by

$$f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x}) = \frac{f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y})}{f_{\boldsymbol{X}}(\boldsymbol{x})}.$$

We note that in the expression above the term in the numerator is the joint pdf while the term in the denominator is marginal distribution at $\boldsymbol{x}$ defined in (7.2.10). Integrating this expression over $\boldsymbol{y}$ on both sides, the numerator on the right hand sides reduces to the marginal distribution of $\boldsymbol{x}$ at $\boldsymbol{x}$, which is equal to the denominator, and thus the integral evaluates to unity, as it should for a pdf.

Similarly, we can define a distribution for $\boldsymbol{X}$ given $\boldsymbol{Y} = \boldsymbol{y}$, a specified value. This is given by

$$f_{\boldsymbol{X}|\boldsymbol{Y}}(\boldsymbol{x}|\boldsymbol{Y} = \boldsymbol{y}) = \frac{f_{\boldsymbol{XY}}(\boldsymbol{x}, \boldsymbol{y})}{f_{\boldsymbol{Y}}(\boldsymbol{y})}.$$

Using (7.2.10) in the right hand side of (7.2.10), we can write a direct relation between the two conditional distributions,

$$f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x}) = \frac{f_{\boldsymbol{X}|\boldsymbol{Y}}(\boldsymbol{x}|\boldsymbol{Y} = \boldsymbol{y}) f_{\boldsymbol{Y}}(\boldsymbol{y})}{f_{\boldsymbol{X}}(\boldsymbol{x})}.$$

This is the celebrated Bayes rule which finds lots of applications in solving probabilistic inference problems.

We end this brief discussion of introductory concepts in probability by hearkening back to the regression problem we considered in Chap. 2. There, we were given lots of samples of the vectors $(\boldsymbol{x}, \boldsymbol{y})$ and using these, we trained a network that could then be used to determine the value of $\boldsymbol{y}$ for a specified value of $\boldsymbol{x}$. Now we are interested in the probabilistic version of this problem. Once again, we are given lots of samples of random vector $(\boldsymbol{X}, \boldsymbol{Y})$. However, instead of using these to train a network that will yield one value of $\boldsymbol{Y}$ for a specific $\boldsymbol{X} = \boldsymbol{x}$, we want to train a network that would generate samples of $\boldsymbol{Y}$ from the conditional distribution $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x})$. This is the conditional generative problem that we discussed in the Introduction to this chapter. We will return to it in Sect. 7.4. In the next section, we first address the simpler pure generative problem.

7.3 Pure Generative Problem

We now consider the pure generative problem and describe an algorithm to solve it. We are working with a random vector $\boldsymbol{X}$ with d components with a pdf given by $f_{\boldsymbol{X}}$. This pdf is unknown to us. Instead we are given a dataset of realizations $\{\boldsymbol{x}_i\}$ sampled from the density $f_{\boldsymbol{X}}$, which we denote by $\boldsymbol{x}_i \sim f_{\boldsymbol{X}}$. What we would like to do is generate new samples from this underlying pdf.

Some comments regarding the random vector $\boldsymbol{X}$ are in order. $\boldsymbol{X}$ can be used to represent vectors that we may observe in a physical application. So, for example the components of $\boldsymbol{X}$ could represent the pressure and the temperature measured at a specific location within a fluid in a turbulent flow. In this case, the dimension of $\boldsymbol{X}$, $d = 2$.

Now consider the case where the collection of samples is a collection of RGB images of cars with 512×512 pixels. Thus each sample consists of $512 \times 512 \times 3 = 786,432$ real values. In this case $\boldsymbol{X}$ is a random vector with $d = 786,432$, which is a huge number. In the pure generative problem we are attempting to learn this distribution from the samples given to us and then to generate some more samples from this distribution. This is a hard problem! Also, because of the inherent structure of the objects (i.e. the cars) in these images, the various components of the random vector can be expected to be highly correlated, leading to a non-trivial form of $f_{\boldsymbol{X}}$. This correlation also implies that it might be possible to reduce the dimension of $\boldsymbol{X}$ from d to a smaller number and thus make the representation simpler. A number of deep-learning based generative algorithms are available to solve this problem, including variational autoencoders (VAEs) [32, 51, 97], generative adversarial networks (GANs) [5, 33, 37, 79, 80, 116, 118], normalizing flows [77, 90], and diffusion-based models [102, 103]. In this chapter we focus on GANs and diffusion based models.

7.3.1 GANs

GANs were first proposed by Goodfellow et al. [33] in 2014. Since then, many variants of GANs have been proposed which differ based on the network architecture and the objective function used to train the GAN. We begin by describing the abstract problem setup followed by the architecture and training procedure of a GAN.

Consider the dataset $\mathcal{S} = \{\boldsymbol{x}_i \in \Omega_X \subset \mathbb{R}^d : 1 \leq i \leq N_{\text{train}}\}$. We assume that $\mathcal{S}$ consists of realizations of some random vector $\boldsymbol{X}$ with density $f_{\boldsymbol{X}}$, i.e., $\boldsymbol{x}_i \sim f_{\boldsymbol{X}}$. We want to train a GAN to learn $f_{\boldsymbol{X}}$ and generate new samples from it.

A GAN typically comprises two sub-networks, a generator and a discriminator (or critic). The generator is a network of the form

$$g(.;\boldsymbol{\theta}_g) : \Omega_Z \to \Omega_X, \quad g : \boldsymbol{z} \mapsto \boldsymbol{x} \tag{7.8}$$

where $\boldsymbol{\theta}_g$ are the trainable parameters and $\boldsymbol{z} \in \Omega_Z \subset \mathbb{R}^{N_Z}$ is the realization of a random vector $\boldsymbol{Z}$ following a simple distribution, such as an uncorrelated multivariate Gaussian with density (7.7) with $\boldsymbol{\mu} = \mathbf{0}$, $\boldsymbol{\Sigma} = \boldsymbol{I}$. Typically, $N_Z \ll d$ with $\boldsymbol{Z}$ known as the *latent variable* of the GAN.

The architecture of the generator will depend on the size/shape of $\boldsymbol{X}$. If $\boldsymbol{X}$ is a vector, then g can be an MLP with input dimension N_Z and output dimension d. If $\boldsymbol{X}$ is an image, say of shape $H \times W \times 3$, then the generator architecture will have a few fully connected layers, followed by a reshape into a coarse image with many channels, which is pushed through a number of transpose convolution channels that gradually increase the spatial resolution and compress the number of channels to finally scale up to the shape $H \times W \times 3$. This is also known as a *decoder* architecture, similar to the upward branch of a U-Net (see Fig. 4.10). In either case, for a fixed $\boldsymbol{\theta}_g$, the generator g transforms the RV $\boldsymbol{Z}$ to another RV, $\boldsymbol{X}^g = g(\boldsymbol{Z}; \boldsymbol{\theta}_g)$ with density $f_{\boldsymbol{X}}^g$, which corresponds to the latent density $f_{\boldsymbol{Z}}$ pushed-forward by g. We want to choose $\boldsymbol{\theta}_g$ such that $f_{\boldsymbol{X}}^g$ is close to the unknown target distribution $f_{\boldsymbol{X}}$.

The critic network is of the form

$$c(.; \boldsymbol{\theta}_d) : \Omega_X \to \mathbb{R} \tag{7.9}$$

with the trainable parameters $\boldsymbol{\theta}_d$. Once again, the critic architecture will depend on the shape of $\boldsymbol{X}$. If $\boldsymbol{X}$ is a vector then c can be an MLP with input dimension d and output dimension 1. If $\boldsymbol{X}$ is an image, then the critic architecture will have a few convolution layers, followed be a flattening layer and a number of fully connected layers. This is similar to the CNN architecture shown in Fig. 4.7 but with a scalar output and without an output function.

The schematic of the GAN along with the inputs and outputs of the sub-networks is shown in Fig. 7.4. The generator and critic play adversarial roles. The critic is trained to distinguish between true samples coming from $f_{\boldsymbol{X}}$ and fake samples generated by g with the density $f_{\boldsymbol{X}}^g$. The generator, on the other hand, is trained to fool the critic by trying to generate realistic samples, i.e., samples similar to those sampled from $f_{\boldsymbol{X}}$.

We define the objective function describing a *Wasserstein GAN (WGAN)* [5], which has better robustness and convergence properties compared to the original GAN. The objective function is given by

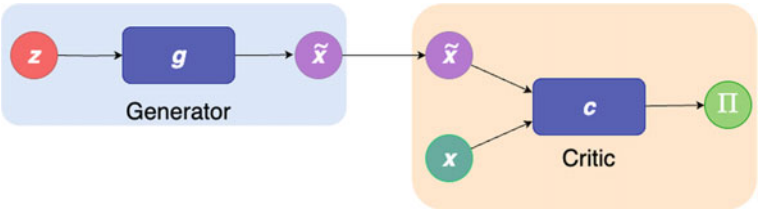


Fig. 7.4 Schematic of a GAN

$$\Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} c(\boldsymbol{x}_i; \boldsymbol{\theta}_d)}_{\text{critic value on real samples}} - \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} c(g(\boldsymbol{z}_i; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d)}_{\text{critic value on fake samples}} \tag{7.10}$$

where $\boldsymbol{x}_i \in \mathcal{S}$ are samples from the true target distribution f_X, while $\boldsymbol{z}_i \sim f_Z$ are passed through g to generate the fake samples. To distinguish between true and fake samples, the critic attains large positive values when evaluated on real samples and large negative values on fake generated samples. Thus, the critic is trained to maximize the objective function. In other words, we want to solve the problem

$$\boldsymbol{\theta}_d^*(\boldsymbol{\theta}_g) = \arg\max_{\boldsymbol{\theta}_d} \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) \text{ for any } \boldsymbol{\theta}_g. \tag{7.11}$$

Note that the optimal parameters of the critic will depend on $\boldsymbol{\theta}_g$. Now to fool the critic, the generator g tries to minimize the objective function,

$$\boldsymbol{\theta}_g^* = \arg\min_{\boldsymbol{\theta}_g} \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d^*). \tag{7.12}$$

Thus, training the WGAN corresponds to solving a minmax optimization problem. We note that the critic and the generator are working in an adversarial manner. That is, while the former is trying to maximize the objective function, the latter is trying to minimize it. Hence the name generative *adversarial* network.

In practice, we need to add a stabilizing term to the critic loss. So the critic is trained to maximize

$$\Pi_c(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) - \frac{\lambda}{\bar{N}} \sum_{i=1}^{\bar{N}} \left(\left\| \frac{\partial c}{\partial \hat{\boldsymbol{x}}}(\hat{\boldsymbol{x}}_i; \boldsymbol{\theta}_d) \right\| - 1 \right)^2 \tag{7.13}$$

where $\hat{\boldsymbol{x}}_i = \alpha \boldsymbol{x}_i + (1 - \alpha) g(\boldsymbol{z}_i; \boldsymbol{\theta}_g)$ and α is sampled from a uniform pdf in (0, 1). The additional term in (7.13) is known as a *gradient penalty* term and is used to constrain the (norm of) gradient of the critic c with respect to its input to be close to 1, and thus be 1-Lipschitz function. For further details on this term, we direct the interested readers to [37].

The iterative Algorithm 1 is used train g and c simultaneously, which is also called *alternating steepest descent*, where η_d and η_g are the learning rates for the critic and the generator, respectively. Note that we take $K > 1$ optimization steps for the critic followed by a single optimization step for the generator. This is because we want to solve the inner maximization problem first so that the critic is able to distinguish between real and fake samples. Although taking a very large K would lead to a more accurate solve of the minmax problem, it would also make the training algorithm computationally intractable for moderately sized networks. Thus, K is typically chosen between 4 and 6 in practice.

Algorithm 1: Algorithm to train a GAN

Input: $\boldsymbol{\theta}_d^0,\ \boldsymbol{\theta}_g^0,\ K,\ N_{\text{epochs}}, \eta_d, \eta_g$
for $n = 1, ..., N_{epochs}$ **do**
  $\hat{\boldsymbol{\theta}}_d \leftarrow \boldsymbol{\theta}_d^{(n-1)}$
  for $k = 1, ..., K$ **do**
    Maximization update:

$$\hat{\boldsymbol{\theta}}_d \leftarrow \hat{\boldsymbol{\theta}}_d + \eta_d \frac{\partial \Pi_c}{\partial \boldsymbol{\theta}_d}(\boldsymbol{\theta}_g^{(n-1)}, \hat{\boldsymbol{\theta}}_d)$$

  end
  $\boldsymbol{\theta}_d^{(n)} \leftarrow \hat{\boldsymbol{\theta}}_d$
  Minimization update:

$$\boldsymbol{\theta}_g^n \leftarrow \boldsymbol{\theta}_g^{(n-1)} - \eta_g \frac{\partial \Pi}{\partial \boldsymbol{\theta}_g}(\boldsymbol{\theta}_g^{(n-1)}, \boldsymbol{\theta}_d^{(n)})$$

end

The minmax problem is a hard optimization problem to solve, and convergence is usually reached after training for many epochs. Alternatively, the critic optimization steps can be done over mini-batches of the training data, with many mini-batches taken per epoch, leading to a similar number of optimization steps for a relatively small number of epochs. As the iterations go on, c becomes better at detecting fake samples and g becomes better at creating samples that can fool the critic.

Under the assumption of infinite capacity ($N_{\boldsymbol{\theta}_g}, N_{\boldsymbol{\theta}_d} \to \infty$), infinite data ($N_{\text{train}} \to \infty$) and a perfect optimizer, we can prove that the generated distribution f_X^g **converges weakly** to the target distribution f_X [5]. This is equivalent to saying

$$\mathbb{E}_Z[\ell(g(\boldsymbol{Z}; \boldsymbol{\theta}_g^*))] \longrightarrow \mathbb{E}_X[\ell(\boldsymbol{X})], \tag{7.14}$$

for every continuous, bounded function ℓ on Ω_X, i.e., $\ell \in C_b(\Omega_X)$.

Once the GAN is trained, we can use the optimized g to generate new samples from $f_X^g \approx f_X$ by first sampling $\boldsymbol{z} \sim f_Z$, and then passing it through the generator to get the sample $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}_g^*)$. Furthermore, due to the weak convergence described above, the statistics (mean, variance, etc.) of the generated samples will converge to the true statistics associated with f_X.

Remark 7.3.1 We make a few important remarks here:

1. Once the GAN is trained, we typically only retain the generator and don't need the critic. The primary role of training the critic is to obtain a suitable g that can generate realistic samples.
2. The reason the term "Wasserstein" appears in the name WGAN is because one can show that solving the minmax problem is equivalent to minimizing the Wasserstein-1 distance between f_X^g and f_X [5, 106]. The Wasserstein-1 distance is a popular metric used to measure discrepancies between two probability distributions.

3. Since the dimension N_Z of the latent variable is typically much smaller than the dimension d of samples in Ω_X, the trained generator also provides a low dimensional representation of high-dimensional data, which can be very useful in several downstream tasks [79, 80].

7.3.2 Score-Based Diffusion Models

If we step back and observe how a trained GAN works we recognize that it generates samples from a "simple" N_z-dimensional Gaussian distribution and maps them such that they are transformed to samples from the desired d-dimensional distribution $f_{\boldsymbol{X}}$. We also note that for a GAN typically $N_Z \ll d$. That is the dimension of the latent space is much smaller than that of the ambient or physical space. This observation leads us to the question whether it is possible to derive other generative algorithms based on this type of transformation, and whether relaxing the requirement $N_Z \ll d$ can lead us to better algorithms. In particular, if we set $N_z = d$ can we somehow address the challenges associated with a GAN. Primarily, can we eliminate the need to solve a challenging, non-linear min-max problem? A class of models, commonly referred to score-based diffusion models [102, 103], accomplishes this goal.

As before we let $\boldsymbol{X}$ be a d-dimensional random vector with density $f_{\boldsymbol{X}}$. However, we now let $\boldsymbol{X}(T)$ be the d-dimensional latent vector. We choose it such that its joint pdf is Gaussian with mean $\boldsymbol{\mu} = \mathbf{0}$, and covariance $\boldsymbol{\Sigma} = \sigma^2(T)\boldsymbol{I}$. That is, the joint pdf of $\boldsymbol{X}(T)$ is $N(\boldsymbol{\mu}, \sigma^2(T)\boldsymbol{I})$. The latent vector is the specific instance of a stochastic process $\boldsymbol{X}(t)$, where t is a time-like variable. The evolution of the joint pdf of this random vector is described in the development below.

Recall, that our goal is to generate samples of $\boldsymbol{X}(T)$ and transform them to samples of $\boldsymbol{X}$. We now derive an algorithm that accomplishes this. The derivation takes the following steps:

1. Derive a convolution that transforms $f_{\boldsymbol{X}}$ to $N(\boldsymbol{\mu}, \sigma^2(T)\boldsymbol{I})$.
2. Derive a time-dependent pde whose solution is equal to this transformation.
3. Derive a backward in time version of this pde. The solution of this pde transforms $N(\boldsymbol{\mu}, \sigma^2(T)\boldsymbol{I})$ to $f_{\boldsymbol{X}}$.
4. Derive and discretize a stochastic ODE corresponding to the pde above. The discretization part will require training a neural network.
5. Generate samples from $N(\boldsymbol{\mu}, \sigma^2(T)\boldsymbol{I})$ and use these as initial states in the discretized ODE above. The final states will then be the desired generated samples from $f_{\boldsymbol{X}}$.

These steps are outlined next.

We first consider the transformation from $f_{\boldsymbol{X}}$ to $f_{\boldsymbol{X}(T)}$. This transformation maps a complex and unknown joint pdf to a Gaussian probability density with zero mean and covariance given by $\sigma^2(T)\boldsymbol{I}$. One way to achieve this is to convolve the original density with a Gaussian kernel as follows,

$$f(\boldsymbol{x}, t) = \int_{\Omega_X} f(\boldsymbol{x}|\boldsymbol{x}', t) f_X(\boldsymbol{x}') d\boldsymbol{x}', \tag{7.15}$$

where

$$f(\boldsymbol{x}|\boldsymbol{x}', t) = \frac{1}{(2\pi\sigma^2(t))^{d/2}} \exp\left[-\frac{|\boldsymbol{x} - \boldsymbol{x}'|^2}{2\sigma^2(t)}\right]. \tag{7.16}$$

Here we think of $t \in (0, T)$ as a time-like scalar variable and assume that $\sigma(t)$ is an increasing function of t with $\sigma(0) = 0$. In this case, if we stipulate that $\sigma(T) \gg 1$, then it is easy to show that $f(\boldsymbol{x}, T) = N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I})$. Also, since $\sigma(0) = 0$, we can show that Gaussian kernel $f(\boldsymbol{x}|\boldsymbol{x}', t)$ tends to the Dirac distribution as $t \to 0$. Therefore, we have $f(\boldsymbol{x}, 0) = f_X$. This concludes the first step in our derivation. Before we move on to the nest step, we make the observation that by running $f(\boldsymbol{x}, t)$ backwards in time, that is from $t = T$ to $t = 0$ we would generate time-dependent density that begins with the Gaussian density $N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I})$ and ends at the data density f_X.

Readers who are familiar with the diffusion equation (or the heat conduction equation) will recognize that $f(\boldsymbol{x}, t)$ given by (7.15) is the solution to the equation

$$\begin{aligned} \frac{\partial f}{\partial t} - \frac{\gamma(t)}{2}\nabla^2 f = 0, \quad (\boldsymbol{x}, t) \in \Omega_X \times (0, T] \\ f(\boldsymbol{x}, 0) = f_X(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega. \end{aligned} \tag{7.17}$$

where $\gamma(t) = \frac{d\sigma^2(t)}{dt}$. The solution to this PDE transforms $f(\boldsymbol{x}, 0) = f_X(\boldsymbol{x})$ to $f(\boldsymbol{x}, T) = N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I})$.

Next, we define a backward time given by $\tau = T - t$. As τ varies from 0 to T, t varies from T to 0. Thus the time-dependent density with which we wish to work is given by

$$\tilde{f}(\boldsymbol{x}, \tau) = f(\boldsymbol{x}, t), \tag{7.18}$$

where $t = T - \tau$. The equation for this density can be derived by substituting relation above into (7.17). It is given by

$$\begin{aligned} \frac{\partial \tilde{f}}{\partial \tau} + \frac{\gamma(T - \tau)}{2}\nabla^2 \tilde{f} = 0, \quad (\boldsymbol{x}, \tau) \in \Omega_X \times (0, T] \\ \tilde{f}(\boldsymbol{x}, 0) = N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I}), \quad \boldsymbol{x} \in \Omega. \end{aligned} \tag{7.19}$$

The solution of this equation will ensure that at $\tau = T$, this density will obtain the desired value, $\tilde{f}(\boldsymbol{x}, T) = f_X(\boldsymbol{x})$.

Equation (7.19) has a problem in that it is the diffusion equation with the opposite sign in the spatial derivative term. It is sometimes referred to as the anti-diffusion equation. More importantly, for our application, this equation does not have a "particle" counterpart that can be used to transform samples drawn from its initial state, $N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I})$, to samples from from its final state, $f_X(\boldsymbol{x})$. As shown below, this

difficulty can be addressed by adding and subtracting another diffusion term with the correct sign to this equation.

We begin with the differential equation in (7.19),

$$\begin{aligned}
\frac{\partial \tilde{f}}{\partial \tau} + \frac{\gamma(T-\tau)}{2}\nabla^2 \tilde{f} &= 0, \\
\Rightarrow \frac{\partial \tilde{f}}{\partial \tau} - \frac{\gamma(T-\tau)}{2}\nabla^2 \tilde{f} + \gamma(T-\tau)\nabla^2 \tilde{f} &= 0, \\
\Rightarrow \frac{\partial \tilde{f}}{\partial \tau} - \frac{\gamma(T-\tau)}{2}\nabla^2 \tilde{f} + \nabla \cdot (\gamma(T-\tau)\nabla \tilde{f}) &= 0, \\
\Rightarrow \frac{\partial \tilde{f}}{\partial \tau} - \frac{\gamma(T-\tau)}{2}\nabla^2 \tilde{f} + \nabla \cdot (\gamma(T-\tau)\nabla \ln(\tilde{f})\tilde{f}) &= 0, \\
\Rightarrow \frac{\partial \tilde{f}}{\partial \tau} - \frac{\gamma(T-\tau)}{2}\nabla^2 \tilde{f} + \nabla \cdot (\gamma(T-\tau)\nabla \ln(f(\boldsymbol{x}, T-\tau))\tilde{f}) &= 0, \\
\Rightarrow \frac{\partial \tilde{f}}{\partial \tau} - \frac{\gamma(T-\tau)}{2}\nabla^2 \tilde{f} + \nabla \cdot (\gamma(T-\tau)\boldsymbol{s}(\boldsymbol{x}, T-\tau))\tilde{f}) &= 0.
\end{aligned} \tag{7.20}$$

In deriving lines 2–4 in the development above, we have performed simple algebraic operations. To arrive at the second from the last line, we have used the relation (7.18), and in the last line we have introduced the definition of the score function for the density $f(\boldsymbol{x}, t)$,

$$\boldsymbol{s}(\boldsymbol{x}, t) = \nabla \ln f(\boldsymbol{x}, t). \tag{7.21}$$

This concludes the third step of our derivation.

The last line of (7.20) represents a drift-diffusion equation for the evolution of $\tilde{f}$ where the diffusion coefficient is given by $\frac{\gamma(T-\tau)}{2}$ and the drift velocity is given by $\gamma(T-\tau)\boldsymbol{s}(\boldsymbol{x}, T-\tau)$. This equation represents the evolution of the probability density for particles whose trajectories are governed by the corresponding stochastic ordinary differential equation. Thus, if samples are drawn from the initial state of this density, that is from $N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I})$ and evolved according to the corresponding stochastic differential equation, they will be transformed to samples from the final state of this density, that is, $f_{\boldsymbol{X}}(\boldsymbol{x})$.

When this stochastic ordinary differential equation is discretized by a specific time-integration scheme (the Euler-Maruyama scheme) we are led to the following algorithm for the evolution of the samples from time $\tau^{(n)}$ to $\tau^{(n)} + \Delta\tau$,

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \gamma(T-\tau^{(n)})\boldsymbol{s}(\boldsymbol{x}^{(n)}, T-\tau^{(n)})\Delta\tau + \sqrt{\Delta\tau\gamma(T-\tau^{(n)})}\boldsymbol{w} \tag{7.22}$$

where $\boldsymbol{x}^{(n)}$ denotes the sample value at time $\tau^{(n)}$ and $\boldsymbol{w} \sim N(\boldsymbol{0}, \boldsymbol{1})$.

We are now ready to describe the generative algorithm that will generate samples from the desired distribution. First, we select a schedule for $\sigma(t)$, which is such that $\sigma(0) = 0$, $\sigma(T) \gg 1$, and for $t \in (0, T)$, $\gamma(t) = \frac{d\sigma^2(t)}{dt} > 0$. Note that selecting a schedule for $\sigma(t)$ means that we also select a schedule for $\gamma(t)$. Next, we divide the

time interval $(0, T)$ into N_T intervals which gives the time increment $\Delta\tau = T/N_T$. Then we generate N_{samples} samples of $x^{(0)}$ from $N(\mathbf{0}, \sigma^2(T)\boldsymbol{I})$. Each of these samples is then evolved through (7.22). In this equation, we known every coefficient, except the score function $\boldsymbol{s}(\boldsymbol{x}, T - \tau_n)$. Therefore, once this function is learnt we can evolve these samples, and by construction, we are guaranteed that at $\tau_n = T$, the samples will have evolved to be samples drawn from $f_X(\boldsymbol{x})$. This algorithm is described in Algorithm 2.

In the next paragraph, we describe how we can learn an approximation to the score function by using a finite number of samples generated from $f_X(\boldsymbol{x})$. These samples form the training data for the generative algorithm, and the approximation for the score function is expressed in terms of a deep neural network.

Learning the Score Function. Let $\boldsymbol{s}(\boldsymbol{x}, t; \boldsymbol{\theta}) : \Omega_X \times (0, T) \mapsto \Omega_X$ be a neural network that maps samples from the domain Ω_X and the time interval $(0, T)$ to samples in Ω_X. Let $\boldsymbol{\theta}$ represent all the trainable parameters in this network. Then from (7.21) we conclude that an appropriate loss function to determine these parameters is given by

$$\Pi(\boldsymbol{\theta}) = \int_0^T \int_{\Omega_X} |\boldsymbol{s}(\boldsymbol{x}, t; \boldsymbol{\theta}) - \nabla \ln f(\boldsymbol{x}, t)|^2 f(\boldsymbol{x}, t) d\boldsymbol{x} dt, \tag{7.23}$$

which is designed to minimize the difference between the true score function and its counterpart approximated by the neural network. By utilizing (7.15) in the above equation and recognizing that the loss function may be shifted by a term that does not depend on $\boldsymbol{\theta}$ (see [103] for details) we arrive at,

$$\Pi(\boldsymbol{\theta}) = \int_0^T \int_{\Omega_X} \int_{\Omega_X} |\boldsymbol{s}(\boldsymbol{x}, t; \boldsymbol{\theta}) - \nabla_{\boldsymbol{x}} \ln f(\boldsymbol{x}|\boldsymbol{x}', t)|^2 f(\boldsymbol{x}|\boldsymbol{x}', t) f_X(\boldsymbol{x}') d\boldsymbol{x}' d\boldsymbol{x} dt. \tag{7.24}$$

Thereafter, using the definition of the Gaussian kernel $f(\boldsymbol{x}|\boldsymbol{x}', t)$ (7.16), we may simplify this as,

$$\Pi(\boldsymbol{\theta}) = \int_0^T \int_{\Omega_X} \int_{\Omega_X} |\boldsymbol{s}(\boldsymbol{x}, t; \boldsymbol{\theta}) + \frac{\boldsymbol{x} - \boldsymbol{x}'}{\sigma^2(t)}|^2 f(\boldsymbol{x}|\boldsymbol{x}', t) f_X(\boldsymbol{x}') d\boldsymbol{x}' d\boldsymbol{x} dt. \tag{7.25}$$

In practise this integral is approximated by its Monte-Carlo approximation,

$$\Pi(\boldsymbol{\theta}) = \frac{1}{K J N_{\text{train}}} \sum_{k=1}^{K} \sum_{j=1}^{J} \sum_{i=1}^{N_{\text{train}}} \left| \boldsymbol{s}(\boldsymbol{x}_i^{(j,k)}, t^{(k)}; \boldsymbol{\theta}) + \frac{\boldsymbol{x}_i^{(j,k)} - \boldsymbol{x}_i}{\sigma^2(t^{(k)})} \right|^2, \tag{7.26}$$

where t_k is sampled from a uniform distribution on the interval $(0, T)$, $\boldsymbol{x}_i$ is the i-th sample from the training dataset $\mathcal{S}$, and $\boldsymbol{x}^{(j,k)} = \boldsymbol{x}_i + \boldsymbol{n}^{(j,k)}$, where $\boldsymbol{n}^{(j,k)}$ is sampled from $N(\mathbf{0}, \sigma^2(t^{(k)})\mathbf{1})$. From the equation above it is clear that given a noisy version of the sample $\boldsymbol{x}_i$, denoted by $\boldsymbol{x}_i^{(j,k)}$, and the time, $t^{(k)}$, the score function learns to compute the noise, $\boldsymbol{n}^{(j,k)}$, scaled by the variance $\sigma^2(t^{(k)})$. This justifies the name

"De-noising diffusion probabilistic modeling" that is often used to describe these models.

Algorithm 2: Algorithm for the score-based diffusion model

Input: $\mathcal{S},\ J,\ K,\ T,\ N_T,\ \sigma^2(t),\ N_{\text{samples}}$

Construct loss function $\Pi(\boldsymbol{\theta}) = \frac{1}{KJN_{\text{train}}} \sum_{k=1}^{K} \sum_{j=1}^{J} \sum_{i=1}^{N_{\text{train}}} |\boldsymbol{s}(\boldsymbol{x}_i^{(j,k)}, t^{(k)}; \boldsymbol{\theta}) + \frac{\boldsymbol{x}_i^{(j,k)} - \boldsymbol{x}_i}{\sigma^2(t^{(k)})}|^2$.

Find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$

Evaluate $\Delta\tau = \frac{T}{N_T}, \gamma(t) = \frac{d\sigma^2(t)}{dt}$

Set $\tau^{(0)} = 0$, $n = 0$, and sample $\boldsymbol{x}_i^{(0)} \sim N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{1}), i = 1, \cdots, N_{\text{samples}}$.

for $n = 1, ..., N_T$ **do**

  for $i = 1, ..., N_{samples}$ **do**

    Stochastic update :

$$\boldsymbol{x}_i^{(n+1)} = \boldsymbol{x}_i^{(n)} + \gamma(T - \tau^{(n)})\boldsymbol{s}(\boldsymbol{x}_i^{(n)}, T - \tau^{(n)}; \boldsymbol{\theta}^*)\Delta\tau + \sqrt{\Delta\tau\gamma(T - \tau^{(n)})}\boldsymbol{w}$$

  end

  $\tau^{(n)} = \tau^{(n)} + \Delta\tau$

  $n = n + 1$

end

Note that the score function takes as input the d-dimensional vector $\boldsymbol{x}$ and the scalar t and generates as output the d-dimensional score function. Thus when $\boldsymbol{x}$ is a vector a logical choice for the architecture for the score function is an MLP that maps a $d + 1$-dimensional vector to a d-dimensional vector. When $\boldsymbol{x}$ represents an image, or a discretized version of a field variable, the architecture is different. In this case the network that approximates the score function is an image-to-image network (U-Net, for example) and the scalar time is used for instance normalization within the network.

The value of $\sigma(T)$ is a hyperparameter for the method. This value should be larger than the largest difference between the samples of $\boldsymbol{x}$. That is it should be larger than $|\boldsymbol{x}_i - \boldsymbol{x}_j|$ for all i, j in the dataset. In practise, this value is computed approximately, and $\sigma(T)$ is set to be slightly greater than this value.

The schedule for $\sigma(t)$ also needs to be selected by the practitioner. There has been considerable work in determining a schedule that works well, and the reader is referred to [46] for prescriptions that perform well.

The number of time steps in the sampling algorithm (N_T) is another hyperparameter. Clearly, the larger this number, the more expensive the sampling stage of the generative algorithm. A reasonable rule of thumb for this number is anywhere between 20 to 1,000.

7.4 Conditional Generative Algorithms

Recall the deterministic problem where given the labelled/pairwise dataset

$$\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : \boldsymbol{x}_i \in \Omega_X \subset \mathbb{R}^d,\ \boldsymbol{y}_i \in \Omega_Y \subset \mathbb{R}^D\}_{i=1}^{N_{\text{train}}} \tag{7.27}$$

we want to find $\boldsymbol{y}$ for a new $\boldsymbol{x}$ not appearing in $\mathcal{S}$. We have seen in the previous chapters how neural networks can be used to solve such a regression (or classification) problem.

Now let us consider the probabilistic version of this problem. We assume that $\boldsymbol{x}$ and $\boldsymbol{y}$ are modelled using the random vectors $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively. Further, let the paired samples in (7.27) be drawn from the unknown joint distribution $f_{\boldsymbol{XY}}$. Then given a realization $\boldsymbol{X} = \hat{\boldsymbol{x}}$, we wish to use $\mathcal{S}$ to determine the conditional distribution $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ and generate samples from it.

There are several popular approaches to solve this probabilistic problem, such as Bayesian neural networks [12], variational inference [21, 86], dropouts [26, 104], deep Boltzman machines [93, 94], or diffusion-based models [103]. In this chapter we will focus on an extension of GANs and diffusion models to solve this problem.

7.4.1 Conditional GANs

Conditional GANs were first proposed in [67] to learn conditional distributions. We will discuss a special variant of these models known as conditional Wasserstein GANS (cWGANs) which were developed in [2], and used to solve a number of physics-based (inverse) problems in [89]. They were slightly modified in [88], and once again used to solve several physics-based inverse problems. In the following section, we describe this modified formulation.

The schematic of a conditional GAN is depicted in Fig. 7.5. The generator is a network of the form

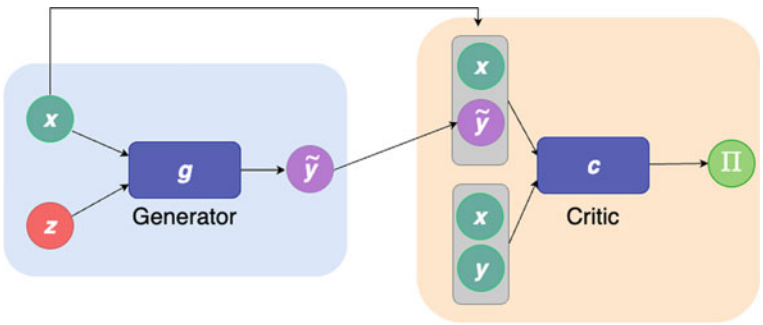


Fig. 7.5 Schematic of a conditional GAN

$$g(.;\boldsymbol{\theta}_g) : \Omega_Z \times \Omega_X \to \Omega_Y, \quad g : (\boldsymbol{z}, \boldsymbol{x}) \mapsto \boldsymbol{y} \tag{7.28}$$

where $\boldsymbol{z} \sim f_{\boldsymbol{Z}}$ is the latent variable. Note that unlike a "simple" GAN, the generator in a conditional GAN also takes as input $\boldsymbol{x}$. For a given value of $\boldsymbol{X} = \hat{\boldsymbol{x}}$, sampling $\boldsymbol{z} \sim f_{\boldsymbol{Z}}$ will generate many samples of $\boldsymbol{y}$ from some induced conditional distribution $f^g_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$. The goal is to prescribe the parameters $\boldsymbol{\theta}_g$ such that $f^g_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ approximates the true conditional $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ for (almost) every value of $\hat{\boldsymbol{x}}$.

The critic is a network of the form

$$c(\cdot, \cdot; \boldsymbol{\theta}_d) : \Omega_X \times \Omega_Y \to \mathbb{R} \tag{7.29}$$

which is trained to distinguish between paired samples $(\boldsymbol{x}, \boldsymbol{y})$ generated from the true joint distribution $f_{\boldsymbol{XY}}$ and the fake pairs $(\boldsymbol{x}, \tilde{\boldsymbol{y}})$ where $\tilde{\boldsymbol{y}}$ is generated by g given (real) $\boldsymbol{x}$.

The objective function for a cWGAN is given by

$$\Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} c(\boldsymbol{x}_i, \boldsymbol{y}_i; \boldsymbol{\theta}_d)}_{\text{critic value on real pairs}} - \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} c(\boldsymbol{x}_i, g(\boldsymbol{z}_i, \boldsymbol{x}_i; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d)}_{\text{critic value on fake pairs}}. \tag{7.30}$$

As earlier, the critic is trained to maximize the objective function (given by (7.11)) while the generator is trained to minimize it (given by (7.12)). Further, a stabilizing gradient penalty term needs to be included when optimizing the critic (see [89]). This term is given by

$$\Pi_c(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) - \frac{\lambda}{\bar{N}} \sum_{i=1}^{\bar{N}} \left(\left\| \nabla c(\hat{\boldsymbol{x}}_i, \boldsymbol{y}_i; \boldsymbol{\theta}_d) \right\| - 1 \right)^2 \tag{7.31}$$

where ∇c denotes the gradient of c with respect to both its arguments, $\hat{\boldsymbol{x}}_i = \alpha \boldsymbol{x}_i + (1 - \alpha) g(\boldsymbol{z}_i; \boldsymbol{\theta}_g)$ and α is sampled from a uniform pdf in (0, 1). The additional term in (7.31) is known as a *gradient penalty* term and is used to constraining the (norm of) gradient of the critic d with respect to its input to be close to 1, and thus be a 1-Lipschitz function.

The optimal weights for the critic are determined by maximizing the objective function (7.11), and the optimal weights for the generator are determined by minimizing the objective function (7.12). Once again, this leads to a min-max problem which is solved using the alternating steepest descent algorithm described earlier.

Under the assumptions of infinite capacity ($N_{\boldsymbol{\theta}_g}, N_{\boldsymbol{\theta}_d} \to \infty$), infinite data ($N_{\text{train}} \to \infty$) and a perfect optimizer, we can prove [88] that the generated conditional distribution $f^g_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ converges in a weak sense to the target condition distribution $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ (on average) for a given $\boldsymbol{X} = \hat{\boldsymbol{x}}$.

Thus when the training for a conditional GAN is over, the generator can be used to generate samples from the conditional distribution for a given $\boldsymbol{X} = \hat{\boldsymbol{x}}$ as follows:

$$\boldsymbol{y}_i = g(\boldsymbol{z}_i, \hat{\boldsymbol{x}}; \boldsymbol{\theta}_g^*), \quad \boldsymbol{z}_i \sim f_{\boldsymbol{Z}}. \tag{7.32}$$

That is, we use $\hat{\boldsymbol{x}}$ as input in one of the channels of the generator and use $\boldsymbol{z}_i$ sampled from $f_{\boldsymbol{Z}}$ as input in the other channel. The output of the generator produces samples of $\boldsymbol{Y}$ that are generated from the desired conditional distribution.

7.4.2 Conditional Diffusion Models

The key differences between the pure generative problem and the conditional generative problem are:

1. In the former we are given samples from the distribution $f_{\boldsymbol{X}}$, whereas in that in the latter we are given data from the joint distribution $f_{\boldsymbol{XY}}$.
2. In the former the goal is to generate some more samples from $f_{\boldsymbol{X}}$, whereas in the latter it is to generate samples from the conditional distribution $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$.

In order to devise a diffusion model that accomplishes this goal, our strategy is to repeat the development for the pure generative problem, while replacing $f_{\boldsymbol{X}}$ with $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$. This derivation is based on the development in [7, 20, 103].

The convolution defined in (7.15) is replaced with

$$f(\boldsymbol{y}, t|\hat{\boldsymbol{x}}) = \int_{\Omega_Y} f(\boldsymbol{y}|\boldsymbol{y}', t) f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}'|\hat{\boldsymbol{x}}) d\boldsymbol{y}', \tag{7.33}$$

where the definition of the Gaussian kernel $f(\boldsymbol{y}|\boldsymbol{y}', t)$ remains the same and is given by (7.16).

Following the steps outlined in Sect. 7.3.2, we can show that the backward in time version of this probability density, defined as $\tilde{f}(\boldsymbol{y}, \tau|\hat{\boldsymbol{x}}) = f(\boldsymbol{y}, t|\hat{\boldsymbol{x}})$, satisfies the partial differential equation,

$$\frac{\partial \tilde{f}}{\partial \tau} - \frac{\gamma(T-\tau)}{2} \nabla^2 \tilde{f} + \nabla \cdot (\gamma(T-\tau) \boldsymbol{s}(\boldsymbol{y}, \hat{\boldsymbol{x}}, T-\tau) \tilde{f}) = 0, \tag{7.34}$$

where the "spatial" derivatives are now along the $\boldsymbol{y}$ coordinates and the appropriate score function is given by

$$\boldsymbol{s}(\boldsymbol{y}, \hat{\boldsymbol{x}}, t) = \nabla \ln f(\boldsymbol{y}, t|\hat{\boldsymbol{x}}). \tag{7.35}$$

When $\tilde{f}$ is initialized to be $N(\boldsymbol{0}, \sigma^2(T)\boldsymbol{I})$ and evolved by solving the above PDE, we are guaranteed that the final state at $\tau = T$ is the desired conditional density, $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$.

This means that if we select samples such that each $\boldsymbol{y}^{(0)}$ is sampled from $N(\mathbf{0}, \sigma^2(T)\boldsymbol{I})$, and then evolve the samples according to,

$$\boldsymbol{y}^{(n+1)} = \boldsymbol{y}^{(n)} + \gamma(T - \tau^{(n)})\boldsymbol{s}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{x}}, T - \tau^{(n)})\Delta\tau + \sqrt{\Delta\tau\gamma(T - \tau^{(n)})}\boldsymbol{w}, \tag{7.36}$$

we are guaranteed that each $\boldsymbol{y}^{(N)}$ will be a sample from the desired conditional density $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}|\hat{\boldsymbol{x}})$.

Now that we have obtained the iterations that will map samples from the standard normal density to the desired conditional density, all that remains is to provide an expression for the score function. Once again we use a neural network to approximate this score function, and train the neural network by defining the loss function to be

$$\Pi(\boldsymbol{\theta}) = \int_0^T \int_{\Omega_Y} \int_{\Omega_X} |\boldsymbol{s}(\boldsymbol{y}, \boldsymbol{x}, t; \boldsymbol{\theta}) - \nabla \ln f(\boldsymbol{y}, t|\boldsymbol{x})|^2 f(\boldsymbol{y}, t|\boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) d\boldsymbol{x} d\boldsymbol{y} dt. \tag{7.37}$$

Following the steps outlined in Sect. 7.3.2, this expression reduces to

$$\Pi(\boldsymbol{\theta}) = \int_0^T \int_{\Omega_Y} \int_{\Omega_Y} \int_{\Omega_X} |\boldsymbol{s}(\boldsymbol{y}, \boldsymbol{x}, t; \boldsymbol{\theta}) + \frac{\boldsymbol{y} - \boldsymbol{y}'}{\sigma^2(t)}|^2 f(\boldsymbol{y}|\boldsymbol{y}', t) f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}'|\boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) d\boldsymbol{x} d\boldsymbol{y} d\boldsymbol{y}' dt. \tag{7.38}$$

Recognizing that $f_{\boldsymbol{Y}|\boldsymbol{X}}(\boldsymbol{y}'|\boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) = f_{\boldsymbol{X}\boldsymbol{Y}}(\boldsymbol{x}, \boldsymbol{y}')$, and replacing the integrals above with their Monte Carlo approximations, we arrive at the final expression for the loss function that is used to train the score function,

$$\Pi(\boldsymbol{\theta}) = \frac{1}{K J N_{\text{train}}} \sum_{k=1}^{K} \sum_{j=1}^{J} \sum_{i=1}^{N_{\text{train}}} \left| \boldsymbol{s}(\boldsymbol{y}_i^{(j,k)}, \boldsymbol{x}_i, t^{(k)}; \boldsymbol{\theta}) + \frac{\boldsymbol{y}_i^{(j,k)} - \boldsymbol{y}_i}{\sigma^2(t^{(k)})} \right|^2, \tag{7.39}$$

where t_k is sampled from a uniform distribution on the interval $(0, T)$, $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is the i-th sample from the training dataset $\mathcal{S}$, and $\boldsymbol{y}^{(j,k)} = \boldsymbol{y}_i + \boldsymbol{n}^{(j,k)}$, where $\boldsymbol{n}^{(j,k)}$ is sampled from $N(\mathbf{0}, \sigma^2(t^{(k)})\mathbf{1})$. The minimization of this loss function will produce an approximate score function which can be used in the iterations given by Eq. (7.36) to generate samples from the desired conditional density for a given input $\hat{\boldsymbol{x}}$.

References

1. Analysis of p-laplacian regularization in semisupervised learning. SIAM J. Math. Anal. **51**, 2085–2120 (2019)
2. J. Adler, O. Öktem, *Deep Bayesian Inversion* (2018). https://arxiv.org/abs/1811.05910
3. W.F. Ames, *Numerical Methods for Partial Differential Equations* (Academic Press, 2014)
4. S.J. Anagnostopoulos, J.D. Toscano, N. Stergiopulos, G.E. Karniadakis, *Residual-based Attention and Connection to Information Bottleneck Theory in Pinns* (2023). arXiv:2307.00379
5. M. Arjovsky, S. Chintala, L. Bottou, Wasserstein generative adversarial networks, in *Proceedings of the 34th International Conference on Machine Learning*, ed. by D. Precup, Y.W. Teh, vol. 70 of Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia, 06–11 Aug. 2017, PMLR, pp. 214–223
6. P.W. Battaglia, J.B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner et al., *Relational Inductive Biases, Deep Learning, and Graph Networks* (2018). arXiv:1806.01261
7. G. Batzolis, J. Stanczuk, C.-B. Schönlieb, C. Etmann, *Conditional Image Generation with Score-based Diffusion Models* (2021). arXiv:2111.13606
8. G. Berkooz, P. Holmes, J.L. Lumley, The proper orthogonal decomposition in the analysis of turbulent flows. Ann. Rev. Fluid Mech. **25**, 539–575 (1993)
9. A. Berlemont, P. Desjonqueres, G. Gouesbet, Particle lagrangian simulation in turbulent flows. Int. J. Multiph. Flow **16**, 19–34 (1990)
10. A.L. Bertozzi, B. Hosseini, H. Li, K. Miller, A.M. Stuart, Posterior consistency of semi-supervised regression on graphs. Inverse Probl. **37**, 105011 (2021)
11. R. Bischof, M. Kraus, *Multi-objective Loss Balancing for Physics-Informed Deep Learning* (2021). http://rgdoi.net/10.13140/RG.2.2.20057.24169
12. C. Blundell, J. Cornebise, K. Kavukcuoglu, D. Wierstra, Weight uncertainty in neural network, in *International Conference on Machine Learning, PMLR*, pp. 1613–1622 (2015)
13. S. Brenner, L. Scott, *The Mathematical Theory of Finite Element Methods Texts in Applied Mathematics* (Springer, New York, 2002)
14. J.C. Butcher, *Numerical Methods for Ordinary Differential Equations* (Wiley, 2016)
15. R.T.Q. Chen, Y. Rubanova, J. Bettencourt, D. Duvenaud, *Neural Ordinary Differential Equations* (2018). https://arxiv.org/abs/1806.07366
16. T. Chen, H. Chen, Approximation capability to functions of several variables, nonlinear functionals, and operators by radial basis function neural networks. IEEE Trans. Neural Netw. **6**, 904–910 (1995)

D. Ray et al., *Deep Learning and Computational Physics*,
https://doi.org/10.1007/978-3-031-59345-1

17. T. Chen, H. Chen, Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. IEEE Trans. Neural Netw. **6**, 911–917 (1995)
18. S. Cuomo, V.S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, F. Piccialli, Scientific machine learning through physics-informed neural networks: where we are and what's next. J. Sci. Comput. **92**, 88 (2022)
19. A.D. Jagtap, G.E. Karniadakis, Extended physics-informed neural networks (xpinns): a generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Commun. Comput. Phys. **28**, 2002–2041 (2020)
20. A. Dasgupta, J. Murgoitio-Esandi, D. Ray, A. Oberai, Conditional score-based generative models for solving physics-based inverse problems, in *NeurIPS 2023 Workshop on Deep Learning and Inverse Problems* (2023)
21. A.K. David, M. Blei, J.D. McAuliffe, Variational inference: a review for statisticians. J. Am. Stat. Assoc. **112**, 859–877 (2017)
22. T. De Ryck, A.D. Jagtap, S. Mishra, Error estimates for physics-informed neural networks approximating the Navier-Stokes equations. IMA J. Numer. Anal. **44**, 83–119 (2023)
23. M.W.M.G. Dissanayake, N. Phan-Thien, Neural-network-based approximations for solving partial differential equations. Commun. Numer. Methods Eng. **10**, 195–201 (1994)
24. J. Donea, S. Giuliani, J.-P. Halleux, An arbitrary lagrangian-eulerian finite element method for transient dynamic fluid-structure interactions. Comput. Methods Appl. Mech. Eng. **33**, 689–723 (1982)
25. S.R. Dubey, S.K. Singh, B.B. Chaudhuri, Activation functions in deep learning: a comprehensive survey and benchmark. Neurocomputing **503**, 92–108 (2022)
26. Y. Gal, Z. Ghahramani, Dropout as a bayesian approximation: Representing model uncertainty in deep learning, in *International Conference on Machine Learning, PMLR*, pp. 1050–1059 (2016)
27. S. Garg, S. Chakraborty, Vb-deeponet: a bayesian operator learning framework for uncertainty quantification. Eng. Appl. Artif. Intell. **118**, 105685 (2023)
28. D. Gilbarg, N. Trudinger, *Elliptic Partial Differential Equations of Second Order Classics in Mathematics* (Springer, Berlin, Heidelberg, 2001)
29. J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, G.E. Dahl, Neural message passing for quantum chemistry, in *International Conference on Machine Learning, PMLR*, pp. 1263–1272 (2017)
30. R.J. Gladstone, H. Rahmani, V. Suryakumar, H. Meidani, M. D'Elia, A. Zareei, Mesh-based GNN surrogates for time-independent PDEs. Sci. Rep. **14**, 3394
31. B.V. Gnedenko, *Theory of Probability* (Routledge, 2018)
32. H. Goh, S. Sheriffdeen, J. Wittmer, T. Bui-Thanh, Solving bayesian inverse problems via variational autoencoders, in *Proceedings of the 2nd Mathematical and Scientific Machine Learning Conference*, ed. by J. Bruna, J. Hesthaven, L. Zdeborova, vol. 145 of Proceedings of Machine Learning Research, PMLR, 16–19 Aug. 2022, pp. 386–425
33. I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, in *Advances in Neural Information Processing Systems*, pp. 2672–2680 (2014)
34. S. Goswami, K. Kontolati, M.D. Shields, G.E. Karniadakis, *Deep Transfer Learning for Partial Differential Equations Under Conditional Shift with Deeponet*, p. 55 (2022). arXiv:2204.09810
35. S. Goswami, M. Yin, Y. Yu, G.E. Karniadakis, A physics-informed variational deeponet for predicting crack path in quasi-brittle materials. Comput. Methods Appl. Mech. Eng. **391**, 114587 (2022)
36. J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai et al., Recent advances in convolutional neural networks. Pattern Recognit. **77**, 354–377 (2018)
37. I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, A.C. Courville, Improved training of wasserstein gans, in *Advances in Neural Information Processing Systems*, pp. 5767–5777 (2017)

38. J. He, S. Koric, S. Kushwaha, J. Park, D. Abueidda, I. Jasiuk, Novel deeponet architecture to predict stresses in elastoplastic structures with variable complex geometries and loads. Comput. Methods Appl. Mech. Eng. **415**, 116277 (2023)
39. J. He, S. Kushwaha, J. Park, S. Koric, D. Abueidda, I. Jasiuk, Sequential deep operator networks (s-deeponet) for predicting full-field solutions under time-dependent loads. Eng. Appl. Artif. Intell. **127**, 107258 (2024)
40. K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition. IEEE Conference on Computer Vision and Pattern Recognition (CVPR) **2016**, 770–778 (2016)
41. C.W. Hirt, A.A. Amsden, J. Cook, An arbitrary lagrangian-eulerian computing method for all flow speeds. J. Comput. Phys. **14**, 227–253 (1974)
42. C.W. Hirt, J. Cook, T.D. Butler, A lagrangian method for calculating the dynamics of an incompressible fluid with free surface. J. Comput. Phys. **5**, 103–124 (1970)
43. F. Hoffmann, B. Hosseini, Z. Ren, A.M. Stuart, Consistency of semi-supervised learning algorithms on graphs: probit and one-hot methods. J. Mach. Learn. Res. **21**, 7549–7603 (2020)
44. T.J. Hughes, *The finite element method: linear static and dynamic finite element analysis* (Cour, Corp, 2012)
45. G. James, D. Witten, T. Hastie, R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R* (Springer Publishing Company, Incorporated, 2014)
46. T. Karras, M. Aittala, T. Aila, S. Laine, Elucidating the design space of diffusion-based generative models. Adv. Neural Inf. Process. Syst. **35**, 26565–26577 (2022)
47. N.S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P.T.P. Tang, On large-batch training for deep learning: Generalization gap and sharp minima, in *International Conference on Learning Representations* (2017). https://openreview.net/forum?id=H1oyRlYgg
48. E. Kharazmi, Z. Zhang, G. Em Karniadakis, *Variational Physics-informed Neural Networks for Solving Partial Differential Equations* (2019)
49. P. Kidger, T. Lyons, Universal approximation with deep narrow networks, in *Proceedings of Thirty Third Conference on Learning Theory*, ed. by J. Abernethy and S. Agarwal, vol. 125 of Proceedings of Machine Learning Research, PMLR, 09–12 July 2020, pp. 2306–2327
50. D.P. Kingma, J. Ba, *Adam: A Method for Stochastic Optimization* (2017). https://arxiv.org/abs/1412.6980v9
51. D.P. Kingma, M. Welling, An introduction to variational autoencoders. Found. Trends ® Mach. Learn. **12**, 307–392 (2019)
52. A. Kopaničáková, G.E. Karniadakis, *Deeponet Based Preconditioning Strategies for Solving Parametric Linear Systems of Equations* (2024). arXiv:2401.02016
53. S. Koric, D.W. Abueidda, Data-driven and physics-informed deep learning operators for solution of heat conduction equation with parametric heat source. Int. J. Heat Mass Transf. **203**, 123809 (2023)
54. I. Lagaris, A. Likas, D. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations. IEEE Trans. Neural Netw. **9**, 987–1000 (1998)
55. I. Lagaris, A. Likas, D. Papageorgiou, Neural-network methods for boundary value problems with irregular boundaries. IEEE Trans. Neural Netw. **11**, 1041–1049 (2000)
56. S. Lanthaler, S. Mishra, G.E. Karniadakis, Error estimates for DeepONets: a deep learning framework in infinite dimensions. Trans. Math. Appl. **6** (2022)
57. E. Lejeune, Mechanical MNIST: A benchmark dataset for mechanical metamodels. Extrem. Mech. Lett. **36**, 100659 (Elsevier, 2020)
58. Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, *Fourier Neural Operator for Parametric Partial Differential Equations* (2020). https://arxiv.org/abs/2010.08895
59. Z. Li, F. Liu, W. Yang, S. Peng, J. Zhou, *A survey of convolutional neural networks: analysis, applications, and prospects* (IEEE Trans. Neural Netw. Learn, Syst, 2021)
60. Z. Liu, P. Luo, X. Wang, X. Tang, Deep learning face attributes in the wild, in *Proceedings of International Conference on Computer Vision (ICCV)*, Dec. 2015

61. L. Lu, P. Jin, G. Pang, Z. Zhang, G.E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. Nat. Mach. Intell. **3**, 218–229 (2021)
62. A.L. Maas, A.Y. Hannun, A.Y. Ng, et al., Rectifier nonlinearities improve neural network acoustic models, in *Proceedings of the ICML*, vol. 30 (2013)
63. M.R. Malik, T.A. Zang, M.Y. Hussaini, A spectral collocation method for the Navier-Stokes equations. J. Comput. Phys. **61**, 64–88 (1985)
64. L. McClenny, U. Braga-Neto, *Self-adaptive Physics-informed Neural Networks Using a Soft Attention Mechanism* (2020). https://arxiv.org/abs/2009.04544
65. W. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**, 115–133 (1943)
66. F. Milletari, N. Navab, S.-A. Ahmadi, V-net: Fully convolutional neural networks for volumetric medical image segmentation, in *Fourth International Conference on 3D Vision (3DV)*, vol. 2016 (IEEE, 2016), pp. 565–571
67. M. Mirza, S. Osindero, *Conditional Generative Adversarial Nets* (2014). https://arxiv.org/abs/1411.1784
68. S. Mishra, R. Molinaro, Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for PDEs. IMA J. Numer. Anal. **42**, 981–1022 (2021)
69. S. Mishra, R. Molinaro, *Estimates on the generalization error of physics-informed neural networks for approximating PDEs* (IMA J. Numer, Anal, 2022)
70. C. Moya, S. Zhang, G. Lin, M. Yue, Deeponet-grid-uq: a trustworthy deep operator framework for predicting the power grid's post-fault trajectories. Neurocomputing **535**, 166–182 (2023)
71. K.P. Murphy, *Machine Learning: a Probabilistic Perspective* (MIT Press, 2012)
72. A. Nemirovski, A. Juditsky, G. Lan, A. Shapiro, Robust stochastic approximation approach to stochastic programming. SIAM J. Optim. **19**, 1574–1609 (2009)
73. A. Ng, M. Jordan, Y. Weiss, On spectral clustering: analysis and an algorithm. Adv. Neural Inf. Process. Syst. **14** (2001)
74. A. Odena, V. Dumoulin, C. Olah, *Deconvolution and Checkerboard Artifacts*, Distill (2016)
75. G. Pang, M. D'Elia, M. Parks, G. Karniadakis, nPINNs: nonlocal physics-informed neural networks for a parametrized nonlocal universal laplacian operator. Algorithms and applications. J. Comput. Phys. **422**, 109760 (2020)
76. G. Pang, L. Lu, G.E. Karniadakis, fPINNs: fractional physics-informed neural networks. SIAM J. Sci. Comput. **41**, A2603–A2626 (2019)
77. G. Papamakarios, E. Nalisnick, D.J. Rezende, S. Mohamed, B. Lakshminarayanan, Normalizing flows for probabilistic modeling and inference. J. Mach. Learn. Res. **22**, 2617–2680 (2021)
78. D. Patel, D. Ray, M.R. Abdelmalik, T.J. Hughes, A.A. Oberai, Variationally mimetic operator networks. Comput. Methods Appl. Mech. Eng. **419**, 116536 (2024)
79. D.V. Patel, A.A. Oberai, Gan-based priors for quantifying uncertainty in supervised learning. SIAM/ASA J. Uncertain. Quantif. **9**, 1314–1343 (2021)
80. D.V. Patel, D. Ray, A.A. Oberai, Solution of physics-based bayesian inverse problems with deep generative priors. Comput. Methods Appl. Mech. Eng. **400**, 115428 (2022)
81. S. Pereira, A. Pinto, V. Alves, C.A. Silva, Brain tumor segmentation using convolutional neural networks in MRI images. IEEE Trans. Med. Imaging **35**, 1240–1251 (2016)
82. T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, P.W. Battaglia, *Learning Mesh-based Simulation with Graph Networks* (2020). arXiv:2010.03409
83. A. Pinkus, Approximation theory of the MLP model in neural networks. Acta Numerica **8**, 143–195 (1999)
84. O. Pinti, A.A. Oberai, Graph laplacian-based spectral multi-fidelity modeling. Sci. Rep. **13**, 16618
85. M. Raissi, P. Perdikaris, G. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. J. Comput. Phys. **378**, 686–707 (2019)

86. R. Ranganath, S. Gerrish, D. Blei, Black box variational inference, in *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, ed. by S. Kaski, J. Corander, vol. 33 of Proceedings of Machine Learning Research, Reykjavik, Iceland, 22–25 Apr. 2014, PMLR, pp. 814–822
87. W. Rawat, Z. Wang, Deep convolutional neural networks for image classification: a comprehensive review. Neural Comput. **29**, 2352–2449 (2017)
88. D. Ray, J. Murgoitio-Esandi, A. Dasgupta, A.A. Oberai, *Solution of Physics-Based Inverse Problems using Conditional Generative Adversarial Networks with full Gradient Penalty* (2023). arXiv:2306.04895
89. D. Ray, H. Ramaswamy, D.V. Patel, A.A. Oberai, *The Efficacy and Generalizability of Conditional GANs for Posterior Inference in Physics-Based Inverse Problems* (2022). https://arxiv.org/abs/2202.07773
90. D. Rezende, S. Mohamed, Variational inference with normalizing flows, in *International Conference on Machine Learning, PMLR*, pp. 1530–1538 (2015)
91. O. Ronneberger, P. Fischer, T. Brox, U-net: convolutional networks for biomedical image segmentation, in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015*, ed. by N. Navab, J. Hornegger, W.M. Wells, A.F. Frangi, Cham (Springer International Publishing, 2015) pp. 234–241
92. S.M. Ross, *Introduction to Probability Models* (Academic Press, 2014)
93. R. Salakhutdinov, G. Hinton, Deep boltzmann machines, in *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, ed. by D. van Dyk, M. Welling, eds., vol. 5 of Proceedings of Machine Learning Research, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr. 2009, PMLR, pp. 448–455
94. R. Salakhutdinov, H. Larochelle, Efficient learning of deep boltzmann machines, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ed. by Y.W. Teh, M. Titterington, vol. 9 of Proceedings of Machine Learning Research, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010, PMLR, pp. 693–700
95. A. Samuel, Some studies in machine learning using the game of checkers. IBM J. Res. Dev. **3**, 210–229 (1959)
96. F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model. IEEE Trans. Neural Netw. **20**, 61–80 (2008)
97. J.K. Seo, K.C. Kim, A. Jargal, K. Lee, B. Harrach, A learning-based method for solving ill-posed nonlinear inverse problems: a simulation study of Lung Eit. SIAM J. Imaging Sci. **12**, 1275–1295 (2019)
98. N. Sharma, V. Jain, A. Mishra, An analysis of convolutional neural networks for image classification. Procedia Comput. Sci. **132**, 377–384 (2018)
99. J. Shin, Y. Darbon, G. Em Karniadakis, On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs. Commun. Comput. Phys. **28**, 2042–2074 (2020)
100. V. Sitzmann, J.N.P. Martel, A.W. Bergman, D.B. Lindell, G. Wetzstein, *Implicit Neural Representations with Periodic Activation Functions* (2020). https://arxiv.org/abs/2006.09661
101. G.D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods* (Oxford University Press, 1985)
102. Y. Song, S. Ermon, Improved techniques for training score-based generative models. Adv. Neural Inf. Process. Syst. **33**, 12438–12448 (2020)
103. Y. Song, J. Sohl-Dickstein, D.P. Kingma, A. Kumar, S. Ermon, B. Poole, *Score-based Generative Modeling Through Stochastic Differential Equations* (2020). arXiv:2011.13456
104. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014)
105. J.W. Thomas, *Numerical Partial Differential Equations: Finite Difference Methods*, vol. 22 (Springer Science & Business Media, 2013)
106. C. Villani, *Optimal Transport: Old and New Grundlehren der mathematischen Wissenschaften* (Springer, Berlin, Heidelberg, 2008)
107. U. Von Luxburg, A tutorial on spectral clustering. Stat. Comput. **17**, 395–416 (2007)

108. P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou, G. Cottrell, Understanding convolution for semantic segmentation, in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, vol. 2018. (IEEE, 2018), pp. 1451–1460
109. S. Wang, S. Sankaran, H. Wang, P. Perdikaris, An expert's guide to training physics-informed neural networks (2023). http://arxiv.org/abs/2308.08468
110. S. Wang, Y. Teng, P. Perdikaris, Understanding and mitigating gradient flow pathologies in physics-informed neural networks. SIAM J. Sci. Comput. **43**, A3055–A3081 (2021)
111. S. Wang, H. Wang, P. Perdikaris, Learning the solution operator of parametric partial differential equations with physics-informed deeponets. Sci. Adv. **7** (2021)
112. S. Wang, X. Yu, P. Perdikaris, When and why PINNs fail to train: a neural tangent kernel perspective. J. Comput. Phys. **449**, 110768 (2022)
113. L. Wu, C. Ma, W. E, How SGD selects the global minima in over-parameterized learning: a dynamical stability perspective, in *Advances in Neural Information Processing Systems*, ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett, vol. 31 (Curran Associates, Inc., 2018)
114. W. Xu, Y. Lu, L. Wang, Transfer learning enhanced deeponet for long-time prediction of evolution equations. Proceedings of the AAAI Conference on Artificial Intelligence **37**, 10629–10636 (2023)
115. L. Yang, D. Zhang, G.E. Karniadakis, Physics-informed generative adversarial networks for stochastic differential equations. SIAM J. Sci. Comput. **42**, A292–A317 (2020)
116. L. Yang, D. Zhang, G.E. Karniadakis, Physics-informed generative adversarial networks for stochastic differential equations. SIAM J. Sci. Comput. **42**, A292–A317 (2020)
117. Y. Yang, G. Kissas, P. Perdikaris, Scalable uncertainty quantification for deep operator networks using randomized priors. Comput. Methods Appl. Mech. Eng. **399**, 115399 (2022)
118. Y. Yang, P. Perdikaris, Adversarial uncertainty quantification in physics-informed neural networks. J. Comput. Phys. **394**, 136–152 (2019)
119. D. Yarotsky, A. Zhevnerchuk, *The Phase Diagram of Approximation Rates for Deep Neural Networks* (2019). https://arxiv.org/abs/1906.09477
120. M. Yin, E. Ban, B.V. Rego, E. Zhang, C. Cavinato, J.D. Humphrey, G. Em Karniadakis, Simulating progressive intramural damage leading to aortic dissection using deeponet: an operator-regression neural network. J. R. Soc. Interface **19**, 20210670 (2022)
121. M. Yin, E. Zhang, Y. Yu, G.E. Karniadakis, Interfacing finite elements with deep neural operators for fast multiscale modeling of mechanics problems. Comput. Methods Appl. Mech. Eng. **402**, 115027 (2022)
122. L. Yuan, Y.-Q. Ni, X.-Y. Deng, S. Hao, A-PINN: auxiliary physics informed neural networks for forward and inverse problems of nonlinear integro-differential equations. J. Comput. Phys. **462**, 111260 (2022)
123. M. Zayernouri, G.E. Karniadakis, Fractional spectral collocation method. SIAM J. Sci. Comput. **36**, A40–A62 (2014)
124. J. Zhang, S. Zhang, G. Lin, *Multiauto-deeponet: a Multi-resolution Autoencoder Deeponet for Nonlinear Dimension Reduction, Uncertainty Quantification and Operator Learning of Forward and Inverse Stochastic Problems* (2022). arXiv:2204.03193
125. J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun, Graph neural networks: a review of methods and applications. AI Open **1**, 57–81 (2020)