Deep Learning and Computational Physics (Lecture Notes)

Deep Ray, Orazio Pinti and Assad A. Oberai[1]

[1]Department of Aerospace and Mechanical Engineering, University of Southern California, Los Angeles, California, USA

Contents

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.

Chapter 1

Introduction

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 bouys, when studying oceanic waves.

2. Based on the observations, postulate a *physical law*. For instance, you observe that the mass of fluid is a closed-system is conserved for all time.

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 obatin 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 integration 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; or one could use finite difference/volume/element methods methods for solving PDEs.

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 are in tune 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 an 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 Section 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

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

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

2. **Decision trees:** We are given a dataset from a sample population, containing the features: age, 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 a 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 the maximize the group-based separation for the samples in the dataset (see Figure 1.1(b)). Then, based on these cuts, the algorithm determines the probability of belonging to a particular class/group for a new point.

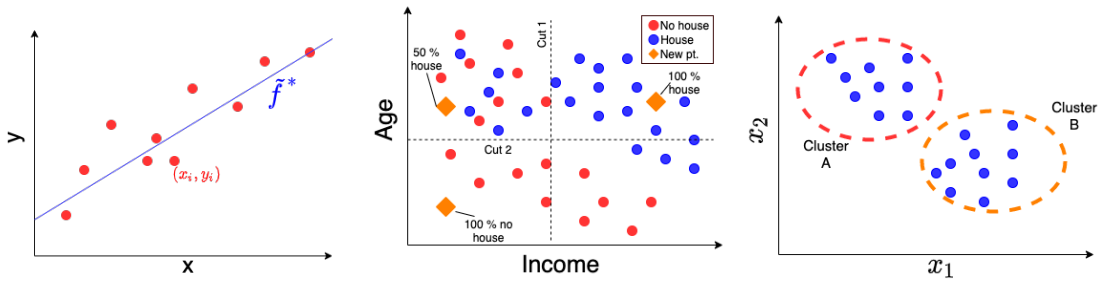
3. **Clustering algorithms:** Given a set of data with a number of features per sample, find cluster/patterns in the data (see Figure 1.1(c)).

1.2.2 Types of ML algorithms based leaning task

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.



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

Figure 1.1: Examples of ML

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 let's say 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 driven by rewards or penalties for decisions taken. Thus, a suitable path/policy is learned to maximize the reward. These kinds of methods are 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 Figure 1.2.

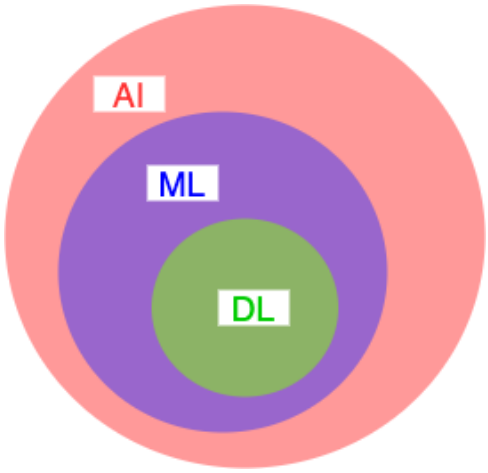


Figure 1.2: The relation between AI, ML and DL

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 Figure 1.3). A car is mounted with a camera which takes lives 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 the next action of the car should be based on this segmented image. This information then passes though a control module that actually controls the mechanical actions of the car. This entire process is mimics what a real driver would do, and is thus artificial intelligence.

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 some 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 be non-ML. Perhaps a rules based expert system.

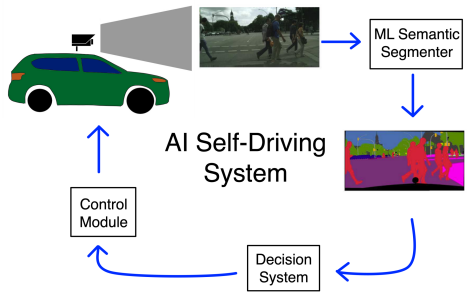


Figure 1.3: Schematic of AI system for a self-driving car. Some illustration taken from [32].

DL is a subset of ML algorithms. The simplest form of a DL architecture, known as a feed-forward network. It comprises a number of layers of non-linear transformations. The non-linear transformations are applied (component-wise) to an affine transformation of an intermediate output. 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 Chapter 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.
- Resnets and their connections with non-linear ODEs (Neural ODEs).
- Recurnets and their connections with nonlinear ODEs.
- Convolutional neural networks and their connection to PDEs.
- Stochastic gradient descent and how it is related to ODEs.
- Deep Learning algorithms for solving PDEs.
- Deep Learning algorithms for approximation operators.
- Generative adversarial algorithms and their connection to computational physics.

Chapter 2

Introduction to deep neural networks

In this chapter, we will take a closer look at the simplest network architecture that is available known as *multilayer perceptron* (MLP).

2.1 MLP architecture

Let us define our objective 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 $\boldsymbol{\mathcal{F}}$. 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 Figure 2.1.

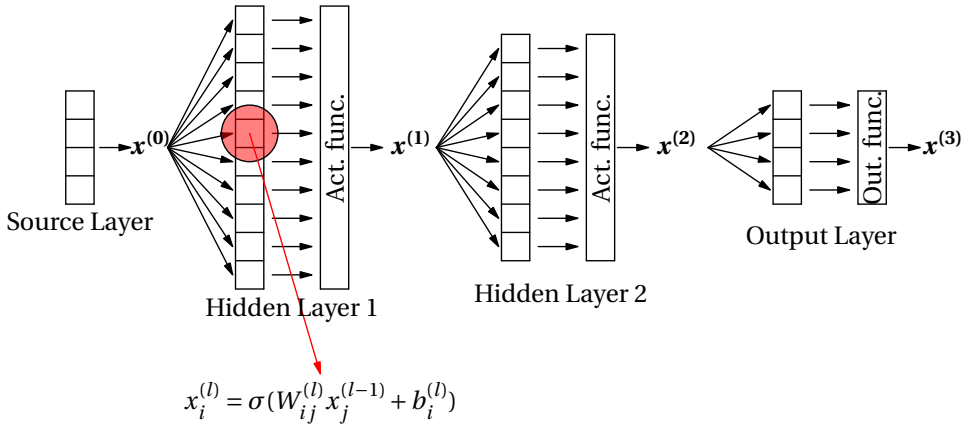


Figure 2.1: MLP with 2 hidden layers

To understand the operations occurring inside an MLP, let us define some notations. We consider a network with L hidden layers, with the width of layer (l) denoted as H_l for $l = 0, 1, ..., L+1$. Note that for consistency with the function 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 input layer. In each layer l, $1 \leq l \leq L+1$, the i-th neuron performs an

affine transformation on that layers input $\boldsymbol{x}^{(l-1)}$ followed by a non-linear transformation

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

where $W_{ij}^{(l)}$ and $b_i^{(l)}$ are respectively known as the weights and bias associated with i-th neuron of layer l, while 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 activation function from layer to layer leads to better neural networks [33].

2. At times, there might be an output function O instead of an activation function 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 the course.

3. We will 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 parameters of the network is all the weights and biases, which we will represent as

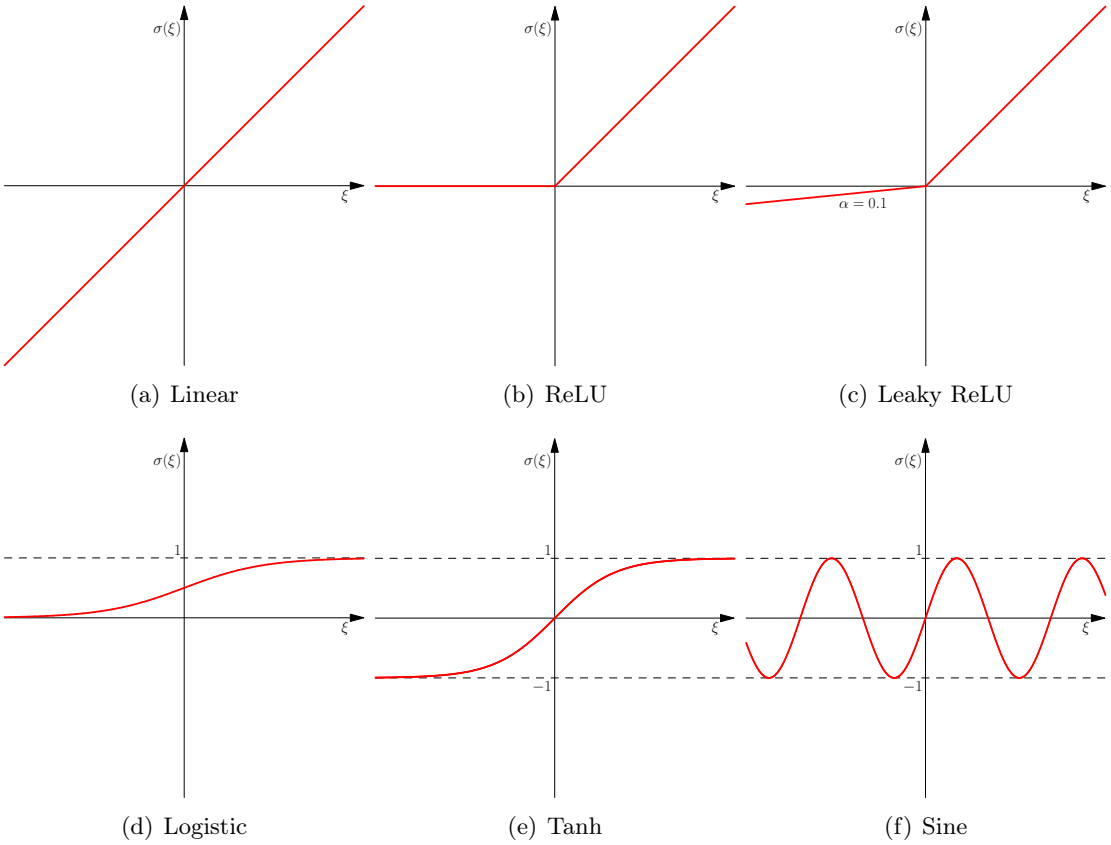
$$\boldsymbol{\theta} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}_{l=1}^{L+1} \in \mathbb{R}^{N_\theta}$$

where N_θ denotes the total number of parameters of the network. The network $\boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta})$ 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 Figure 2.2).



(a) Linear (b) ReLU (c) Leaky ReLU

(d) Logistic (e) Tanh (f) Sine

Figure 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 highly 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 $\xi = 0$. The second derivative will be a dirac function concentrated at $\xi = 0$. In other words, the higher-order derivates (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* [15] 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 derivates of Leaky ReLU behave in the same way as those for ReLU.
- 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.
- Since the derivative quickly decays to zero away from $\xi = 0$, this activation function can lead to slow convergence of the network while training.

2.2.5 Tanh

The tanh function is 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 convergence while training networks.

2.2.6 Sine

Recently, the sine function, i.e., $\sigma(\xi) = \sin(\xi)$ has been proposed as an efficient activation function [27]. 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.

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?*

2.3 Expressivity of a network

Let us try to understand the effects of N_θ increases. 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

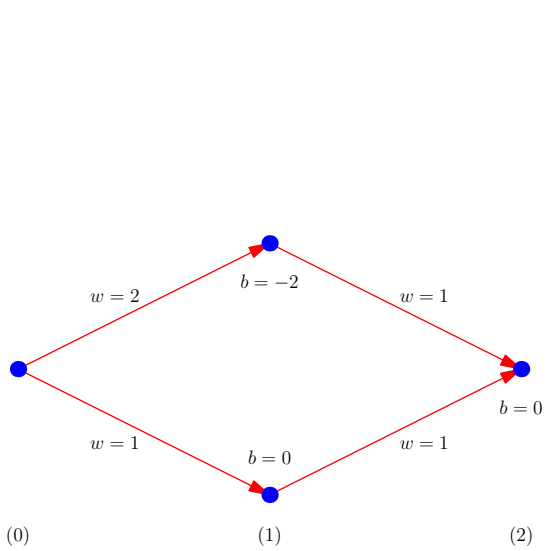
$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \boldsymbol{b}^{(1)} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}, \quad \boldsymbol{W}^{(2)} = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad b^{(2)} = 0.$$

as shown in Figure 2.3(a). Then the various layer outputs are

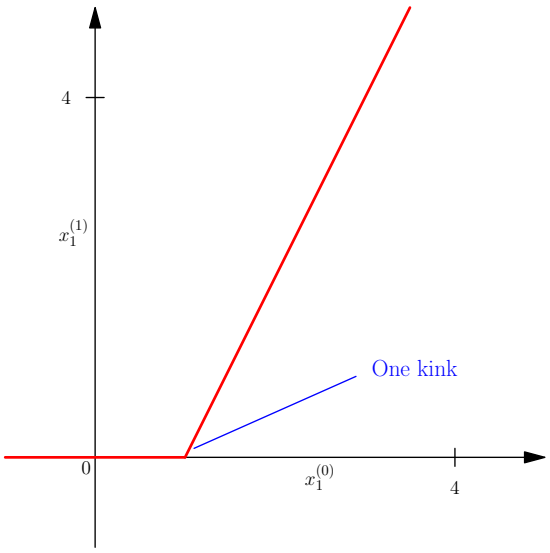
$$x_1^{(1)} = \max\{2x_1^{(0)} - 2, 0\}, \quad x_2^{(1)} = \max\{x_1^{(0)}, 0\}, \quad x_1^{(2)} = \max\{2x_1^{(0)} - 2, 0\} + \max\{x_1^{(0)}, 0\}.$$

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

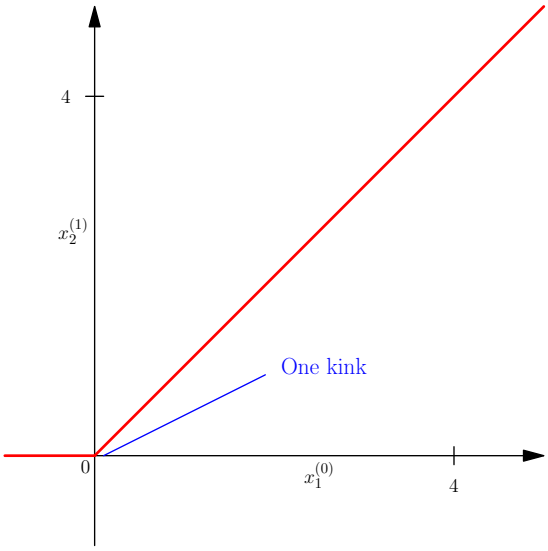
We generalize this formulation to a bigger network with L hidden layers each of width H. Then one can expect that $x_i^{(1)}$, $1 \leq i \leq H$ will have a single kink, with the location and angle of the kink depending on the weights and bias associated with each neuron of the hidden layer. The vector $\boldsymbol{x}^{(1)}$ 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. In other words, the networks have the ability to become more expressive as the depth (and width) of the network is increased.



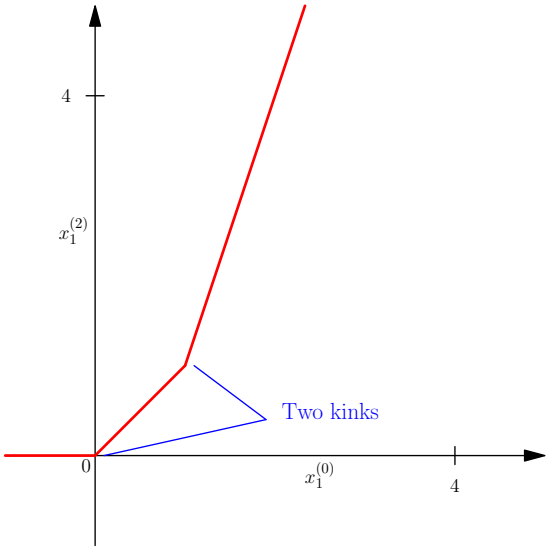
(a) MLP with $L = 1$,$W = 2$



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



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



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

Figure 2.3: Examples to understand the expressivity of neural 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.

Theorem 2.3.1 (Pinkus, 1999 [23]). *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*

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

Theorem 2.3.2 (Kidger, 2020 [9]). *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*

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

Theorem 2.3.3 (Yarotsky, 2021 [33]). *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.

Numerical 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 now 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 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 on the *testing* phase.

To accomplish these three tasks, it is first 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 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 a techniques such as (random or tensor) grid search.

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*

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

We use the following algorithm

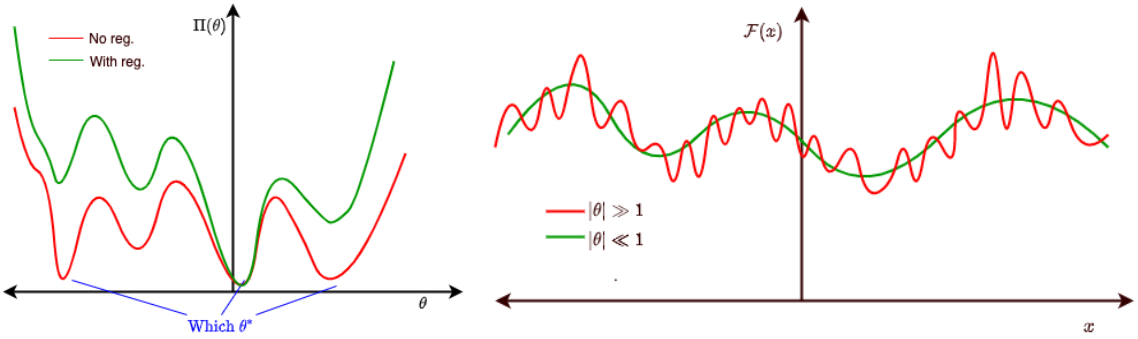
1. *For each possible σ, L pair:*
   (a) *Find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Pi_{train}(\boldsymbol{\theta})$*
   (b) *With this $\boldsymbol{\theta}^*$, evaluate $\Pi_{val}(\boldsymbol{\Theta})$*
2. *Select $\boldsymbol{\Theta}^*$ to be the one that gave the smallest value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

2.5 Generalizability

If we train a network that has a small value of Π_{train} and Π_{val}, does it ensure that Π_{test} will be small? This question is addressed by studying the *generalizability* of the trained network, i.e., it capability to perform well on data not seen while training/validating the network. If the network is trained to overfit the training data, the network will typically lead to poor predictions on test data. Typically, if $\mathcal{S}_{\text{train}}$, $\mathcal{S}_{\text{val}}$ and $\mathcal{S}_{\text{test}}$ are chosen from the same distribution of data, then a small value of $\Pi_{\text{train}}, \Pi_{\text{val}}$ can lead to small values of Π_{test}. Let us look at the commonly used technique to avoid data overfitting, called *regularization*.

2.5.1 Regularization

Neural networks, especially MLPs, are almost always *over-parametrized*, i.e., $N_\theta \gg N$ where N is the number of training samples. This would lead to a highly non-linear network model, for which the loss function $\Pi(\boldsymbol{\theta})$ (where we omit the subscript "train" for brevity) can have a landscape with many local minimas (see Figure 2.4(a)). Then how do we determine which minima leads to a better generalization? To nudge the choice of $\boldsymbol{\theta}^*$ in a more favorable direction, a regularization technique can be employed.



(a) Loss function landscape

(b) Network sensitivity

Figure 2.4: The effect of regularization on the loss function. We have assumed a scalar θ for easier illustration.

The simplest method of regularization involves augmenting 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 $\boldsymbol{\theta}$. This augmentation can change the landscape of $\Pi(\boldsymbol{\theta})$ as illustrated in Figure 2.4(a). In other words, such a regularization encourages the selection of a minima corresponding to smaller values of the parameters $\boldsymbol{\theta}$.

It is not obvious why a smaller value of $\boldsymbol{\theta}$ would be a better choice. To see why this is better, consider the intermediate network output

$$x_1^{(1)} = \sigma(W_{1j}^{(1)} x_j^{(0)} + b_1^{(1)}),$$

which gives

$$\frac{\partial x_1^{(1)}}{\partial x_1^{(0)}} = \sigma'(W_{1j}^{(1)} x_j^{(0)} + b_1^{(1)}) W_{11}^{(1)} \propto W_{11}^{(1)}.$$

Since this derivate scales with $W_{11}^{(1)}$, this implies that $|\frac{\partial \boldsymbol{\mathcal{F}}(\boldsymbol{x})}{\partial x_1^{(0)}}|$ scales with $W_{11}^{(1)}$ as well. If $|W_{11}^{(1)}| \gg 1$, then network would be very sensitive to even small changes in the input $x_1^{(0)}$, i.e., the network would be ill-posed. As illustrated in Figure 2.4(b), using a proper regularization would help avoid over fitting.

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 $\boldsymbol{\theta}$.

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 gradient descent (GD), also known as steepest descent. Consider the 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

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

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.:

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

with the step-size $\eta \geq 0$, also known as the *learning-rate*. This is yet another hyper-parameter that we need to tune during the validation phase. This is the crux of the GD algorithm, and 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 $k = k + 1$

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 as

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

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 Figure 2.5, which has two valleys. Let's assume that the profile of $\Pi(\theta)$ in the each valley can be approximated by a (centered) parabola

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

where $a > 0$ is the curvature of each valley. 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,

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

and the new point after a GD update will be $\theta_1 = \theta_0(1 - a\eta)$. Similarly, it is easy to see that all subsequent iterates write $\theta_{k+1} = \theta_k(1 - a\eta)$. For convergence, we need

$$\left|\frac{\theta_{k+1}}{\theta_k}\right| < 1 \quad \implies |1 - a\eta| < 1.$$

Since $a > 0$ in the valleys, we will need the following condition on the learning rate

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

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, 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. After this it will enter the left valley with a smaller curvature and gradually move towards its minima.

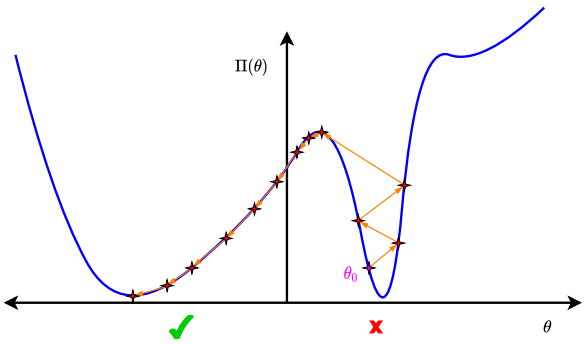


Figure 2.5: GD prefers flatter minimas.

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.

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

$$[\boldsymbol{\theta}_{k+1}]_i = [\boldsymbol{\theta}_k]_i - [\boldsymbol{\eta}_k]_i [\boldsymbol{g}_k]_i, \quad 1 \leq i \leq N_\theta, \tag{2.8}$$

where $[\boldsymbol{\eta}_k]_i$ is the component-wise learning rate and the vector-valued function $\boldsymbol{g}$ depends/approximates the gradient. Note that the notation $[.]_i$ is used to denote the i-th component of the vector. Also note that the learning rate is allowed to depend on the iteration number k. The GD method makes use of

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

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 Figure 2.6, which has sharper gradients along the $[\theta]_2$ direction compared to the $[\theta]_1$ direction. If we start from a point, such as the one shown in the figure, then if η is too large (but still within the stable bounds) the updates will keep zig-zagging its way towards the minima. Ideally, for the particular situation shown in Figure 2.6, we would like the steps to take longer strides along the $[\theta]_1$ compared to the $[\theta]_2$ direction, thus reaching the minima faster.

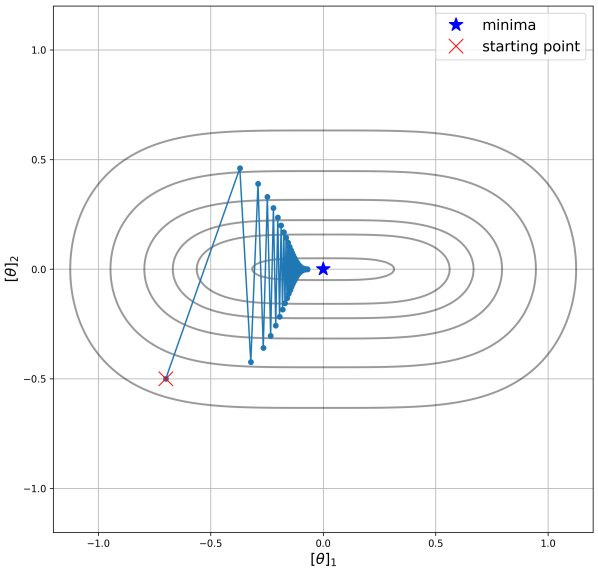


Figure 2.6: Zig-zagging updates with GD.

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

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

where $\boldsymbol{g}_k$ is a weighted moving average of the gradient. This weighting is expected to smoothen out the zig-zagging seen in Figure 2.6 by cancelling out the components of gradient along the $[\theta]_2$ direction and move more smoothly towards the minima. A commonly used value for β_1 is 0.9.

2.7.2 Adam

The Adam optimization was introduced by Kingma and Ba [10], which 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 given by

$$\begin{aligned} \boldsymbol{g}_k &= \beta_1 \boldsymbol{g}_{k-1} + (1-\beta_1)\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k) \\ [\boldsymbol{G}_k]_i &= \beta_2 [\boldsymbol{G}_{k-1}]_i + (1-\beta_2)\left(\frac{\partial \Pi}{\partial \boldsymbol{\theta}_i}(\boldsymbol{\theta}_k)\right)^2 \\ [\boldsymbol{\eta}_k]_i &= \frac{\eta}{\sqrt{[\boldsymbol{G}_k]_i + \epsilon}} \end{aligned} \tag{2.9}$$

where $\boldsymbol{g}_k$ 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 Figure 2.6, 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 [10] 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)

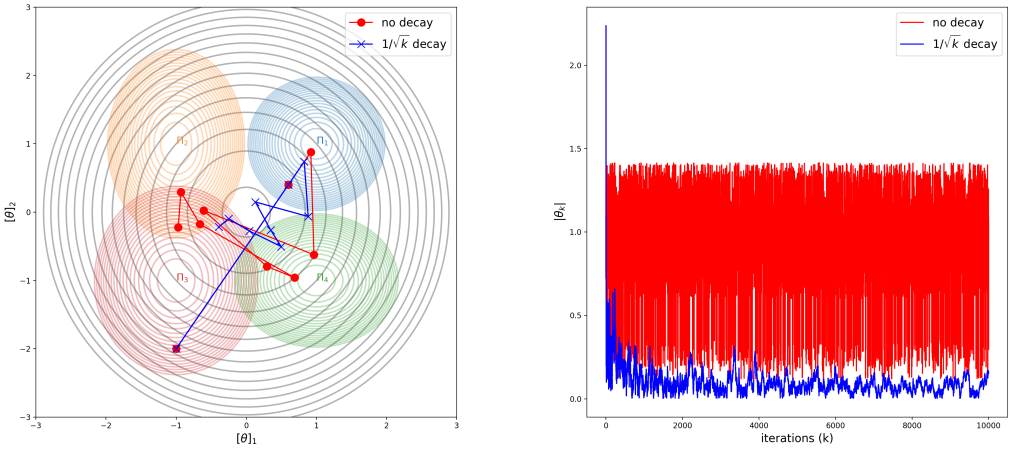
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k), \tag{2.10}$$

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 $\eta_k \sim 1/\sqrt{k}$ [19]. To illustrate why η_k needs to decay, consider the toy function(s) for $\boldsymbol{\theta} \in \mathbb{R}^2$

$$\begin{aligned} &\Pi_1(\boldsymbol{\theta}) = ([\theta]_1 - 1)^2 + ([\theta]_2 - 1)^2, \quad \Pi_2(\boldsymbol{\theta}) = ([\theta]_1 + 1)^2 + 0.5([\theta]_2 - 1)^2, \\ &\Pi_3(\boldsymbol{\theta}) = 0.7([\theta]_1 + 1)^2 + 0.5([\theta]_2 + 1)^2, \quad \Pi_4(\boldsymbol{\theta}) = 0.7([\theta]_1 - 1)^2 + \frac{1}{2}([\theta]_2 + 1)^2, \\ &\Pi(\boldsymbol{\theta}) = \frac{1}{4}\left(\Pi_1(\boldsymbol{\theta}) + \Pi_2(\boldsymbol{\theta}) + \Pi_3(\boldsymbol{\theta}) + \Pi_4(\boldsymbol{\theta})\right). \end{aligned} \tag{2.11}$$

The contour plots of these functions in shown in Figure 2.7(a), 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 Figure 2.7(a). 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 Figure 2.7(b) 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.



(a) Function contours and paths

(b) Norm of updates

Figure 2.7: SGD algorithm with and without a decay in the learning rate.

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 severly under-utilizes the computational and memory resources.

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 also refered to as the batch-size. Thus, the gradient of the loss function can be approximated by

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) \approx \frac{1}{\overline{N}_{\text{train}}} \sum_{i \in \text{batch}(j)} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}). \tag{2.12}$$

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 $i = 1, \cdots, N_{\text{batch}}$
      i. Evaluate the batch gradient using (2.12).
      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 [31] 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 contrast to minima where the overall curvature might be small; however the distribution of the curvature of each component of 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

$$\text{Affine transform:} \quad \xi_i^{(l+1)} = W_{ij}^{(l+1)} x_j^{(l)} + b_i^{(l+1)}, \quad 1 \le i \le H_{l+1} \tag{2.13}$$

$$\text{Non-linear transform:} \quad x_i^{(l+1)} = \sigma\left(\xi_i^{(l+1)}\right), \quad 1 \le i \le H_{l+1}. \tag{2.14}$$

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 $l = 1, ..., L+1$
   (a) Evaluate $\xi^{(l)}$ using (2.13).
   (b) Evaluate $\boldsymbol{x}^{(l)}$ using (2.14).
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 Figure 2.8. In this figure, the lower portion of the graph represents the evaluation of the loss function Π.

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

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \cdot \frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{\xi}^{(L+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(L+1)}}{\partial \boldsymbol{x}^{(L)}} \cdots \frac{\partial \boldsymbol{x}^{(l+1)}}{\partial \boldsymbol{\xi}^{(l+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} \cdot \frac{\partial \boldsymbol{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}}. \tag{2.15}$$

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

$$\frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} = -2(\boldsymbol{y} - \boldsymbol{x}^{(L+1)})^T \tag{2.16}$$

$$\frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} = \boldsymbol{W}^{(l+1)} \tag{2.17}$$

$$\frac{\partial \boldsymbol{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \equiv \text{diag}[\sigma'(\xi_1^{(l)}), \cdots, \sigma'(\xi_{H_l}^{(l)})], \tag{2.18}$$

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 (2.15), we arrive at,

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \cdot \boldsymbol{S}^{(L+1)} \cdot \boldsymbol{W}^{(L+1)} \cdots \boldsymbol{S}^{(l+1)} \cdot \boldsymbol{W}^{(l+1)} \cdot \boldsymbol{S}^{(l)}. \tag{2.19}$$

Taking the transpose, and recognizing that $\boldsymbol{\Sigma}^{(l)}$ is diagonal and therefore symmetric, we finally arrive at

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \boldsymbol{W}^{(l+1)T} \boldsymbol{S}^{(l+1)} \cdots \boldsymbol{W}^{(L+1)T} \boldsymbol{S}^{(L+1)} [-2(\boldsymbol{y} - \boldsymbol{x}^{(L+1)})]. \tag{2.20}$$

This evaluation can also be represented as a computational graph. In fact, as shown in Figure 2.8, it can be appended to the original graph, where this part of the computation appear in the upper row of the graph. Note that we are now traversing in the backward direction. Hence the name back propagation.

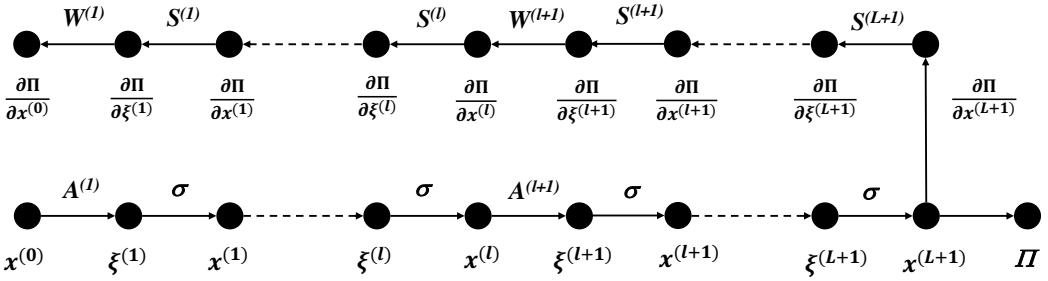


Figure 2.8: Computational graph for computing the loss function and its derivatives with respect to hidden/latent vectors.

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

$$\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \otimes \boldsymbol{x}^{(l-1)}, \tag{2.21}$$

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 evaluted during the forward phase and $\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}$ which is evaluated during back propagation.

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 and Wasserstein generative adversarial networks.*

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*

$$\boldsymbol{x} = [\textit{pulse rate}, \textit{temperature}, SPO_2, \textit{procalcitonin}, ...]$$
$$\boldsymbol{y} = [p_1, p_2]$$

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*

$$\boldsymbol{x} = \textit{the image}$$
$$\boldsymbol{y} = [p_1, p_2, p_2]$$

where p_1, p_2, p_3 *is the probability of being a dog, cat or bird, respectively.*

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*

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

   where C is the number of classes (and also the output dimension). Verify 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

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

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

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \sum_{c=1}^{C} -y_{ci} \log(\mathcal{F}_c(\boldsymbol{x}_i; \boldsymbol{\theta})), \tag{2.22}$$

   where y_{ci} is the c-th component of the true label for the i-th sample. The loss function in (2.22) 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. 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.

Chapter 3

Residual neural networks

Residual networks (or ResNets) were introduced by He et al. [8] 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 Section 2.8 that

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

and

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

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

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

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

Also, for stability we would want $\tau(\boldsymbol{W}^{(m)}) < 1$. Otherwise the output of the network can become unbounded. In practise this is enforced by the regularization term.

Using this in the equation above we have

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

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 $|\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}|$, and therefore $|\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}|$, 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 [8], the authors demonstrate that taking a deeper network can actually lead to an increase in training and validation error (see Figure 3.1). 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 $\left|\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(L+1)}}\right| \approx \left|\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(1)}}\right|$. 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.

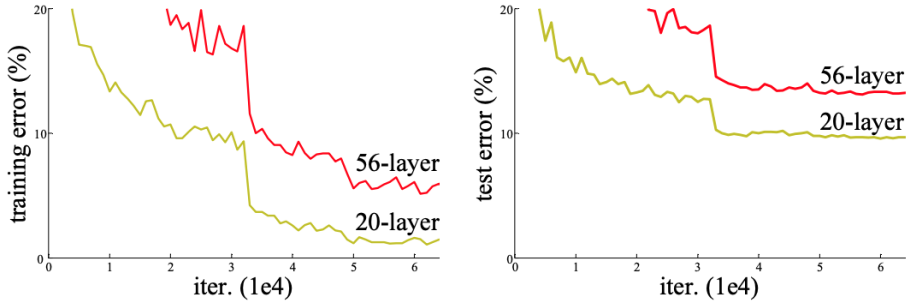


Figure 3.1: Training error (left) and test error (right) on CIFAR-10 data with "plain" deep networks (taken from [8]).

3.2 ResNets

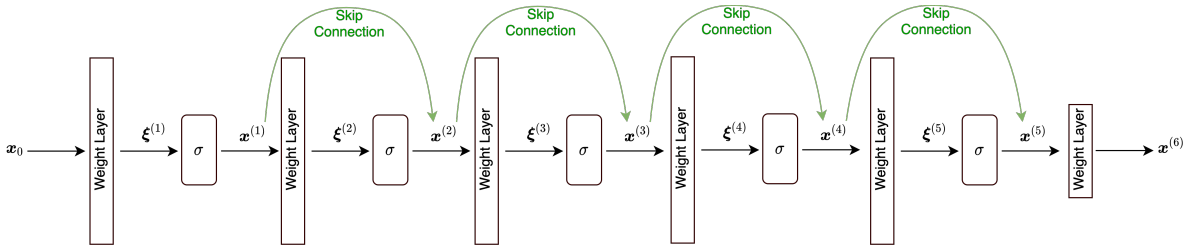


Figure 3.2: ResNet of depth 6 with skip connections.

Consider an MLP with depth 6 (as shown in Figure 3.2) with a fixed width H for each hidden layer. We add skip connections between the hidden layers in the following manner

$$x_i^{(l)} = \sigma(W_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)}) + x_i^{(l-1)}, \quad 2 \leq l \leq L. \tag{3.4}$$

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

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

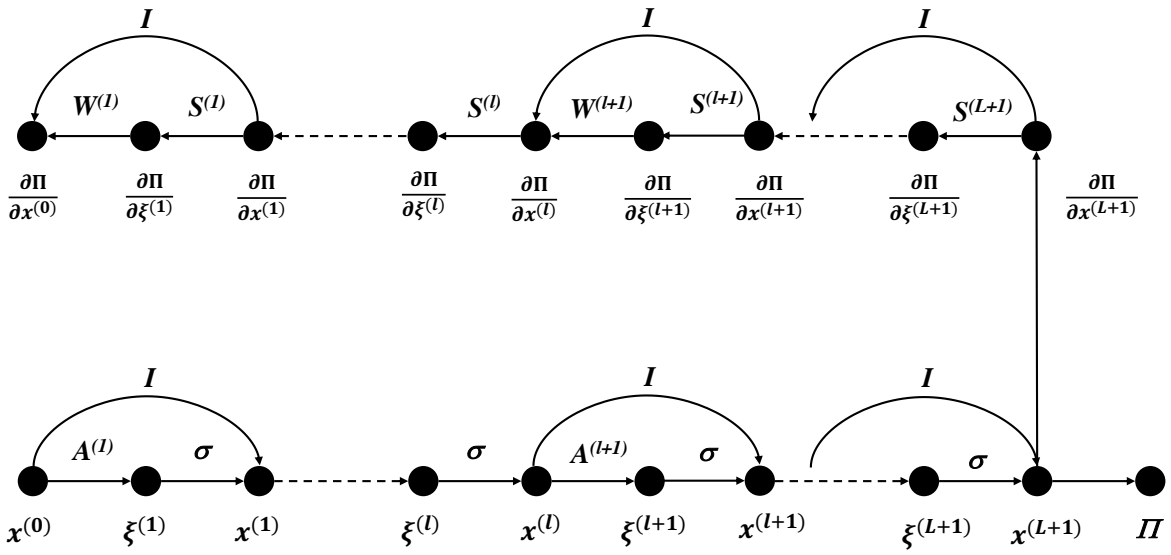


Figure 3.3: Computational graph for forward and backpropagation in a Resnet.

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 Figure 3.3. 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

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{\Sigma}^{(l)} \prod_{m=l+1}^{L+1} (\boldsymbol{I} + \boldsymbol{W}^{(m)T} \boldsymbol{\Sigma}^{(m)}) \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(L+1)}}. \tag{3.5}$$

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

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{\Sigma}^{(l)} \big(\boldsymbol{I} + \sum_{m=l+1}^{L+1} \boldsymbol{W}^{(m)\ T} \boldsymbol{\Sigma}^{(m)} + \text{ higher order terms} \big) \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(L+1)}}. \tag{3.6}$$

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

Remark 3.2.1. *The above analysis can be extended to cases when H is not fixed, but the analysis is not as clean. See [8] 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 (3.4), which we can rewrite as

$$\frac{\boldsymbol{x}^{(l)} - \boldsymbol{x}^{(l-1)}}{\Delta t} = \frac{1}{\Delta t} \sigma(\boldsymbol{W}^{(l)} \boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)}) = \frac{1}{\Delta t} \sigma(\boldsymbol{\xi}^{(l)}) \tag{3.7}$$

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 (3.7) as

$$\frac{\boldsymbol{x}^{(l)} - \boldsymbol{x}^{(l-1)}}{\Delta t} = \boldsymbol{V}(\boldsymbol{x}^{(l-1)}; \boldsymbol{\theta}^{(l)}). \tag{3.8}$$

Now consider a first-order system of (possibly non-linear) ODEs, where given $\boldsymbol{x}(0)$ and

$$\dot{\boldsymbol{x}} \equiv \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{V}(\boldsymbol{x}, t) \tag{3.9}$$

we want to find $\boldsymbol{x}(T)$. 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 (3.9) 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 (3.8). 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 $i = 1, \cdots, N_{\text{train}}$.
- 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 $i = 1, \cdots, N_{\text{train}}$.
- 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 [4]. Consider a system of ODEs given by

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

Given $\boldsymbol{x}(0)$, we wish to find $\boldsymbol{x}(T)$. In [4], the RHS, i.e., $\boldsymbol{V}(\boldsymbol{x}, t)$, is defined using a feed-forward neural network with parameters $\boldsymbol{\theta}$ (see Figure 3.4). 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 (3.10) is solved using a suitable time-marching scheme, such as forward Euler, Runge-Kutta, etc.

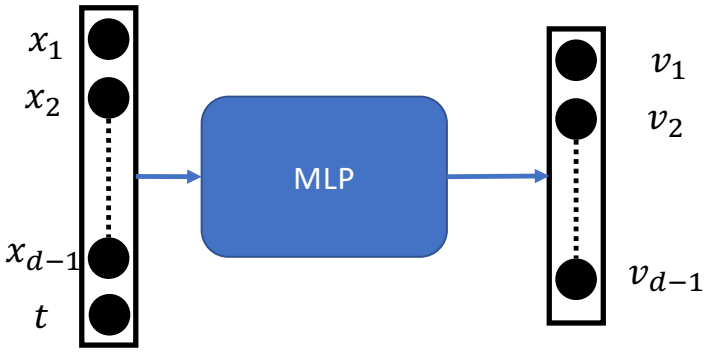


Figure 3.4: Feed-forward neural network used to model the right hand side in a Neural ODE. The number of dependent variables $= d - 1$.

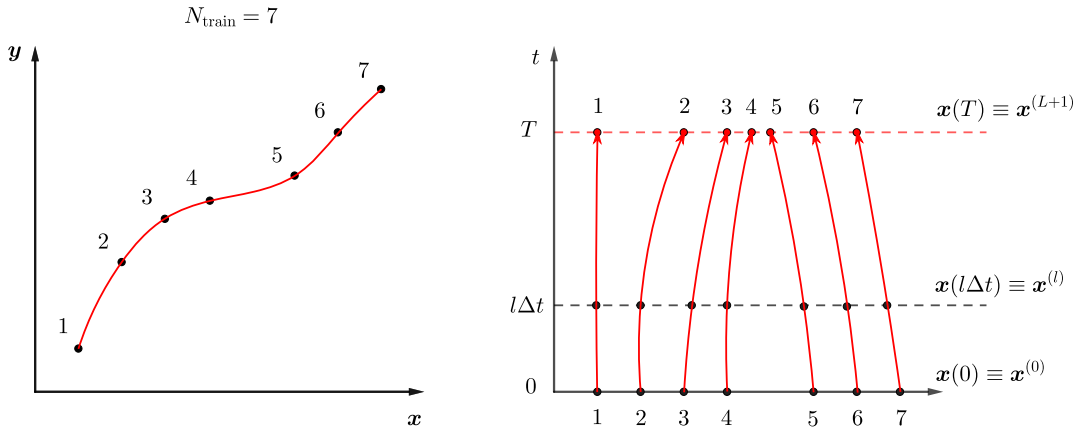


Figure 3.5: Analogy between regression problems and Neural ODEs.

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 (3.10) 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 (3.10) 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.

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

Solving PDEs with MLPs

A number of numerical methods exist to solve PDEs. Some of these are:

- Finite difference methods
- Finite volume methods
- Finite element methods
- Spectral Galerkin and collocation methods
- 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 find $u(x)$ in the interval $x \in (0, l)$ such that

$$\begin{aligned} a\frac{\mathrm{d}u}{\mathrm{d}x} - \kappa\frac{\mathrm{d}^2u}{\mathrm{d}x^2} &= f(x), \quad x \in (0, \ell) \\ u(0) &= 0 \\ u(\ell) &= 1 \end{aligned} \tag{4.1}$$

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

$$\begin{aligned} \boldsymbol{a} \cdot \nabla \boldsymbol{u}(\boldsymbol{s}) - \kappa \Delta \boldsymbol{u}(\boldsymbol{s}) &= \boldsymbol{f}(\boldsymbol{s}), \quad \boldsymbol{s} \in \Omega \\ \boldsymbol{u}(\boldsymbol{s}) &= \boldsymbol{g}(\boldsymbol{s}), \quad \boldsymbol{s} \in \partial\Omega \end{aligned} \tag{4.2}$$

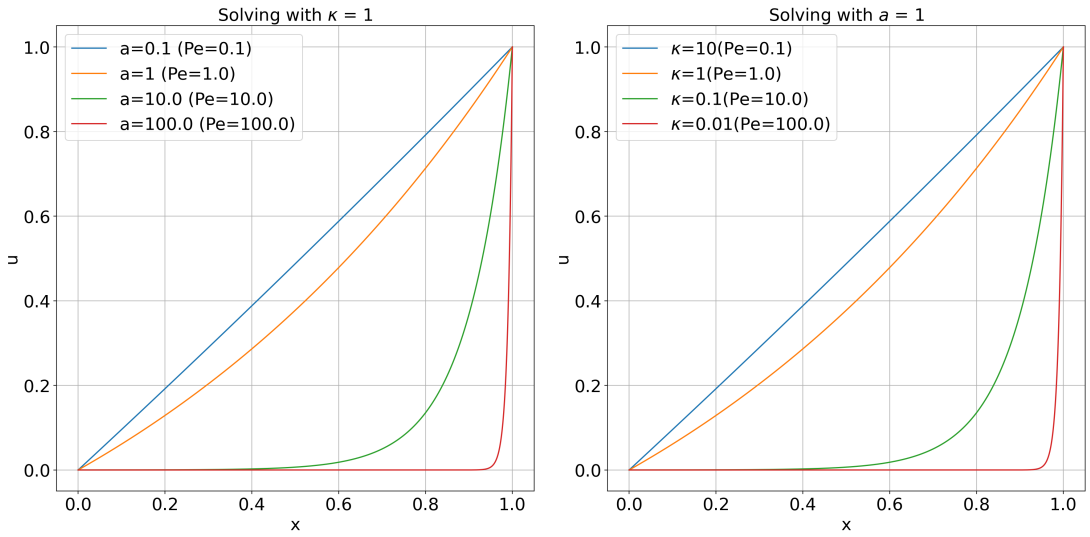
Note that the model problem is a linear PDE (ODE in the one-dimensional case). Replacing the velocity a by u leads to the viscous Burgers equation.

The solution to (4.1) 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 Figure 4.1. 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 $\delta \approx \mathrm{Pe} \times l$.

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



(a) Fixed κ

(b) Fixed a

Figure 4.1: Exact solution of (4.1) with $\ell = 1$.

4.1 Finite difference method

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 (4.1) leads to:

1. Discretize the domain into $N + 1$ points, with $x_i = ih$, $0 \leq i \leq N$ 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 the approximation 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, $1 \leq i \leq N-1$

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

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

$$\begin{aligned} \beta u_1 + \gamma u_2 &= -\alpha u_0 + f_1 \\ \alpha u_{i-1} + \beta u_i + \gamma u_{i+1} &= f_i, \quad \forall\ 2 \leq i \leq N-2 \\ \alpha u_{N-2} + \beta u_{N-1} &= -\gamma u_N + f_{N-1} \end{aligned} \tag{4.3}$$

Combining all the $N-1$ equations in (4.3), we get the following linear system

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{4.4}$$

where the tridiagonal matrix $\boldsymbol{K}$ and the other vectors in (4.4) are defined as

$$\begin{aligned} \boldsymbol{K} &= \begin{bmatrix} \beta & \gamma & & 0 \\ \alpha & \ddots & \ddots & \\ & \ddots & \ddots & \gamma \\ 0 & & \alpha & \beta \end{bmatrix} \in \mathbb{R}^{(N-1)\times(N-1)}, \\ \boldsymbol{u} &= \begin{bmatrix} u_1 & u_2 & \cdots & u_{N-2} & u_{N-1} \end{bmatrix}^\top \in \mathbb{R}^{N-1}, \\ \boldsymbol{f} &= \begin{bmatrix} -\alpha u_0 + f_1 & f_2 & f_3 \cdots & f_{N-2} & f_{N-1} & -\gamma u_N + f_{N-1} \end{bmatrix}^\top \in \mathbb{R}^{N-1} \end{aligned}$$

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 (4.4). 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 with uses MLPs to solve PDEs.

4.2 Spectral collocation method

Spectral collocation methods seek 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) Is smooth enough so that derivatives can be evaluated.

   (c) Easy to evaluate.

   (d) Derivatives that are easy to evaluate.

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

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

   The first few Chebyshev polynomials are shown in Figure 4.2. 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 evaluate a recurrence relation for derivatives of all orders.

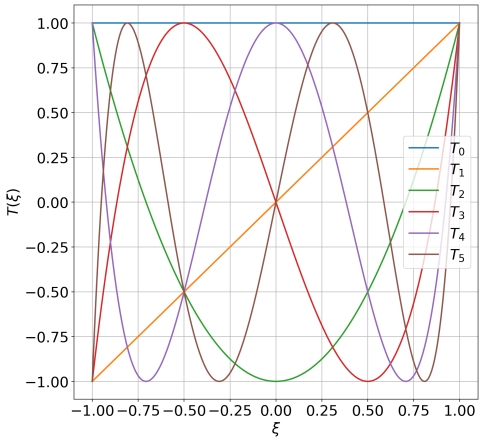


Figure 4.2: First few Chebyshev polynomials.

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

$$u(x) = \sum_{n=0}^{N} u_n \phi_n(x) \tag{4.5}$$

   where u_n are the basis coefficients. For our toy problem (4.1) (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

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}x}(x) &= \sum_{n=0}^{N} u_n \phi_n'(x) = \sum_{n=0}^{N} u_n 2T_n'(2x-1) \\ \frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x) &= \sum_{n=0}^{N} u_n \phi_n''(x) = \sum_{n=0}^{N} u_n 4T_n''(2x-1) \end{aligned} \tag{4.6}$$

4. Use the boundary conditions of the PDE. For the specific case of (4.1),

$$\begin{aligned} u(0) = 0 &\implies \sum_{n=0}^{N} u_n \phi_n(0) = \sum_{n=0}^{N} u_n T_n(-1) = 0, \\ u(1) = 1 &\implies \sum_{n=0}^{N} u_n \phi_n(1) = \sum_{n=0}^{N} u_n T_n(1) = 1 \end{aligned} \tag{4.7}$$

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 found in step 3. in the PDE evaluated at these $N-1$ nodes

$$\begin{aligned} & a \sum_{n=0}^{N} u_n \phi_n'(x_i) - \kappa \sum_{n=0}^{N} u_n \phi_n''(x_i) = f(x_i) \\ \implies & \sum_{n=0}^{N} u_n \left(2a T_n'(2x_i - 1) - 4\kappa T_n''(2x_i - 1) \right) = f(x_i) \end{aligned} \tag{4.8}$$

This leads to an additional $N-1$ equations for the $N+1$ coefficients. Combining (4.7) and (4.8) leads to the following linear system

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{4.9}$$

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 (4.9) easier to solve. These include invertibility, positive-definiteness, sparseness, etc.

Remark 4.2.1. *The method is called a collocation method as the PDE is evaluated at the specific collocation/quadrature points x_i.*

Remark 4.2.2. *When working with a non-linear PDE, we will end up with a non-linear systems of algebraic equations for the coefficients $u_0, ..., u_N$, 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 (4.5) and compute its derivates. Then, the coefficients $\boldsymbol{u}$ are found by minimizing the following loss function

$$\begin{aligned} \Pi(\boldsymbol{u}) &= \Pi_{\text{int}}(\boldsymbol{u}) + \lambda \Pi_{\text{bc}}(\boldsymbol{u}) \\ \Pi_{\text{bc}}(\boldsymbol{u}) &= \left| \sum_{n=0}^{N} u_n \phi_n(0) - 0 \right|^2 + \left| \sum_{n=0}^{N} u_n \phi_n(1) - 1 \right|^2 \\ \Pi_{\text{int}}(\boldsymbol{u}) &= \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left| a \sum_{n=0}^{N} u_n \phi_n'(x_i) - \kappa \sum_{n=0}^{N} u_n \phi_n''(x_i) - f(x_i) \right|^2 \end{aligned} \tag{4.10}$$

This can be solved using any of the gradient-based methods we have seen in Chapter 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 (4.9). A few things to note here

- λ is a parameter used to scale the interior loss and boundary loss differently.
- The number of interior point 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.

4.3 Physics-informed neural networks (PINNs)

The idea of using neural networks to solve partial differential equations was introduced in 1990-2000s by Lagaris et al. [11]. With the renewed interest in using machine learning tools in solving PDEs, this idea was rediscovered in 2019 by Raissi et al. [24], 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 spectral collocation method described in the previous section.

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

$$u = \mathcal{F}(x; \boldsymbol{\theta}). \tag{4.11}$$

   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 Section 2.3.1).

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

   (c) Is it easy to evaluate? The answer is yes, due to a quick forward propagation pass.

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

2. Given the representation (4.11), we need to find $\boldsymbol{\theta}$ such that the PDE is satisfied in some suitable form. Compare this with spectral collocation approximation given by (4.5), where we need to determine the coefficients u_n. Note that while the dependence on the coefficients u_n in (4.5) is linear, the dependence on $\boldsymbol{\theta}$ in (4.11) 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 Figure 4.3. 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 Figure 4.3 is obtained by

simply setting $\Pi = \boldsymbol{x}^{(L+1)}$ in the graph shown in Figure 2.8. Further, once we recognize that $\frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{x}^{(L+1)}} = \mathbf{1}$, the identity matrix, we can easily read from this graph that

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

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 Figure 4.3). 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" is made.

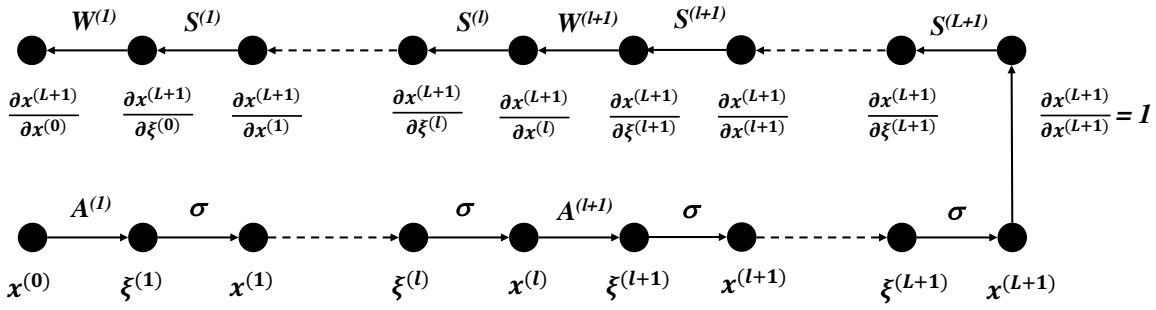


Figure 4.3: Extended graph to evaluate derivatives with respect to network input.

4. Insert the functional representation of the solution (4.11) 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 (4.1))

$$\begin{aligned} \Pi(\boldsymbol{\theta}) &= \Pi_{\text{int}}(\boldsymbol{\theta}) + \lambda_b \Pi_{\text{b}}(\boldsymbol{\theta}), \\ \Pi_{\text{b}}(\boldsymbol{\theta}) &= \left(\mathcal{F}(0;\boldsymbol{\theta}) - 0\right)^2 + \left(\mathcal{F}(1;\boldsymbol{\theta}) - 1\right)^2, \\ \Pi_{\text{int}}(\boldsymbol{\theta}) &= \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left(a\mathcal{F}'(x_i;\boldsymbol{\theta}) - \kappa\mathcal{F}''(x_i;\boldsymbol{\theta})_n - f(x_i)\right)^2. \end{aligned} \tag{4.12}$$

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:

- When we are able to find $\boldsymbol{\theta}^*$ 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.
- 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. That is, 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 constrained via the penalization term $\Pi_{\mathrm{b}}(\boldsymbol{\theta})$. 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 along the training have been proposed [29, 16, 3].

4.4 Extending PINNs to a more general PDE

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

$$\begin{aligned} L(\boldsymbol{u}(\boldsymbol{x})) &= \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\ B(\boldsymbol{u}(\boldsymbol{x})) &= \boldsymbol{g}(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{4.13}$$

where L is the differential operator, $\boldsymbol{f}$ is the known forcing term, B is the boundary operator, and $\boldsymbol{g}$ 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

$$\begin{aligned} \frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} + \nabla p - \mu \Delta \boldsymbol{v} &= \boldsymbol{f}, \quad \forall\ (\boldsymbol{s}, t) \in \Omega \\ \nabla \cdot \boldsymbol{u} &= 0, \quad \forall\ (\boldsymbol{s}, t) \in \Omega \\ \boldsymbol{v} &= \boldsymbol{0}, \quad \forall\ (\boldsymbol{s}, t) \in \partial\Omega_S \times [0, T] \\ \boldsymbol{v}(\boldsymbol{s}, 0) &= \boldsymbol{v}_0(\boldsymbol{s}), \quad \forall\ \boldsymbol{s} \in \Omega_S. \end{aligned} \tag{4.14}$$

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.

To design a PINN for (4.13), 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 (4.14), the input to the network would be $[s_1, s_2, s_3, t] \in \mathbb{R}^4$, while the output vector would be $\boldsymbol{u} = [v_1, v_2, v_3, p] \in \mathbb{R}^4$. The 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 $R_b(\boldsymbol{u}) = B(\boldsymbol{u}) - \boldsymbol{g}$.
   - 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 $\boldsymbol{u}^* = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta}^*)$

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 $N_v, N_b \to \infty$.
  - 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.

4.5 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 linear operators.

Note that if $\boldsymbol{u}$ is the exact solution, then

$$L(\boldsymbol{e}) = L(\boldsymbol{u}^* - \boldsymbol{u}) = L(\boldsymbol{u}^*) - L(\boldsymbol{u}) = L(\boldsymbol{u}^*) - \boldsymbol{f} = R(\boldsymbol{u}^*) \tag{4.15}$$

and

$$B(\boldsymbol{e}) = B(\boldsymbol{u}^* - \boldsymbol{u}) = B(\boldsymbol{u}^*) - B(\boldsymbol{u}) = B(\boldsymbol{u}^*) - \boldsymbol{g} = R_b(\boldsymbol{u}^*) \tag{4.16}$$

Thus, (4.15) (4.16) lead to a PDE for $\boldsymbol{e}$ driven by the residuals of the MLP solution,

$$\begin{aligned} L(\boldsymbol{e}) &= R(\boldsymbol{u}^*), \quad \text{in } \Omega \\ B(\boldsymbol{e}) &= R_b(\boldsymbol{u}^*), \quad \text{on } \Omega \end{aligned} \tag{4.17}$$

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

$$\|\boldsymbol{e}\|_{L^2(\Omega)} \leq C_1 \left(\|R(\boldsymbol{u}^*)\|_{L^2(\Omega)} + \|R_b(\boldsymbol{u}^*)\|_{L^2(\partial\Omega)} \right) \tag{4.18}$$

where C_1 is a stability constant that depends on the PDE, the domain Ω, etc. This is a condition that hols 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 practise we know and control $\Pi_{\text{int}}, \Pi_{\text{b}}$ 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,

$$\begin{aligned}\|R(\boldsymbol{u}^*)\|_{L^2(\Omega)} &= \left|m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2} + \|R(\boldsymbol{u}^*)\|_{L^2(\Omega)} - m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2}\right| \\ &\leq m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2} + \left|\|R(\boldsymbol{u}^*)\|_{L^2(\Omega)} - m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2}\right| \\ &\leq m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2} + C_2 (N_v)^{-\alpha}\end{aligned} \tag{4.19}$$

where m_Ω is the measure of the domain Ω, C_2 and $\alpha > 0$ will depend on the type of quadrature points chosen. In the equation above the first line is obtained by adding and subtracting the term $m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2}$. The second line is obtained by using the triangle inequality. The third line is a statement of error in approximating an integral with a finite sum of values evaluated at quadrature points.

Similarly for the boundary residual

$$\|R_b(\boldsymbol{u}^*)\|_{L^2(\partial\Omega)} \leq m_{\partial\Omega} \Pi_{\text{b}}(\boldsymbol{\theta}^*)^{1/2} + C_3 (N_b)^{-\beta} \tag{4.20}$$

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

Using (4.18), (4.19) and (4.20), we get

$$\|\boldsymbol{e}\|_{L^2(\Omega)} \leq C_1 \left(\underbrace{m_\Omega \Pi_{\text{int}}(\boldsymbol{\theta}^*)^{1/2} + m_{\partial\Omega} \Pi_{\text{b}}(\boldsymbol{\theta}^*)^{1/2}}_{\text{reduced by } N_\theta \uparrow} + \underbrace{C_2 (N_v)^{-\alpha} + C_3 (N_b)^{-\beta}}_{\text{reduced by } N_v, N_b \uparrow} \right) \tag{4.21}$$

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 [18].

4.6 Data assimilation using PINNs

The problem of data assimilation is often encountered in the 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 principal (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. 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 some 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 $\boldsymbol{u}^* = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta}^*)$.

Chapter 5

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, which are very useful when handling inputs which are images. These tasks include classifying images into categories, performing semantic segmentation on images, and transforming images from one type to another.

5.1 Functions and images

Consider a function $u(\boldsymbol{x})$ defined on $\boldsymbol{x} \in [a, b] \times [c, d] \subset \mathbb{R}^2$. Then we can visualize the discretized version of this function as an image $\boldsymbol{U} \in \mathbb{R}^{N_1 \times N_2}$, where

$$U[i, j] = u(ih, jh), \quad 1 \leq i \leq N_1,\ 1 \leq j \leq N_2,\ h = \text{ pixel size}. \tag{5.1}$$

Note that the image in (5.1) 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, would 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 represent the output of the linear transform and the j index represents the input. If the operation was the 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.

5.2 Convolutions of functions

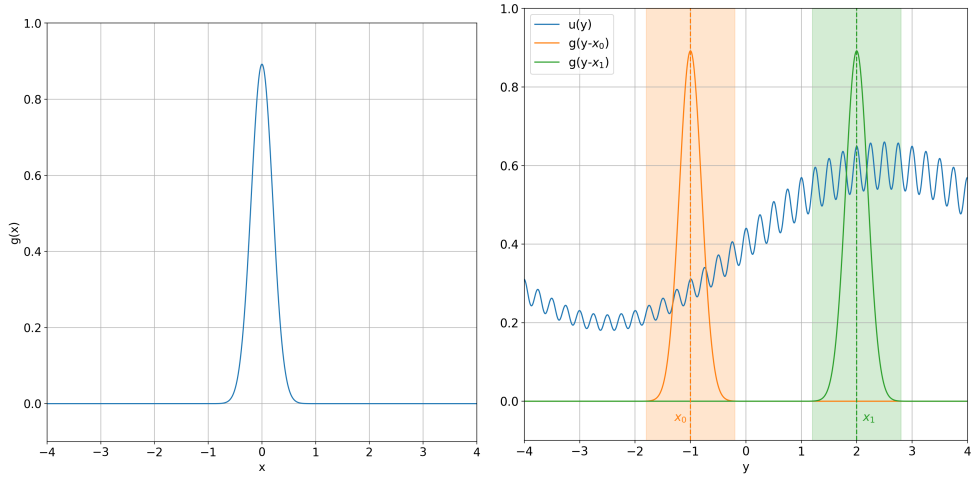
The convolution operator maps functions to functions. Consider the function $u(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^d$, and a sufficiently smooth kernel function $g(\boldsymbol{x})$ which decays as $|\boldsymbol{x}| \to \infty$. Then the convolution operator is given by

$$\overline{u}(\boldsymbol{x}) = \int_{\mathbb{R}^d} g(\boldsymbol{y} - \boldsymbol{x}) u(\boldsymbol{y}) \mathrm{d}\boldsymbol{y} \tag{5.2}$$

We can interpret the convolution operator as sampling u by varying $\boldsymbol{x}$. For example, in 1D, let $u(x)$ and $g(x)$ be as shown in Figure 5.1, 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. So as the kernel moves, it samples u in different windows. Note that the same operation is applied regardless of the value of x. Lets now consider a few typical kernel functions.



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

Figure 5.1: Sampling with shifted kernel in 1D.

5.2.1 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 5.1(a) for $\sigma = 0.2$. Some important properties of this kernel are:

- 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 frequency cut-off, as scales finer than σ are filtered out by the convolution. This smoothing effect is demonstrated in Figure 5.2

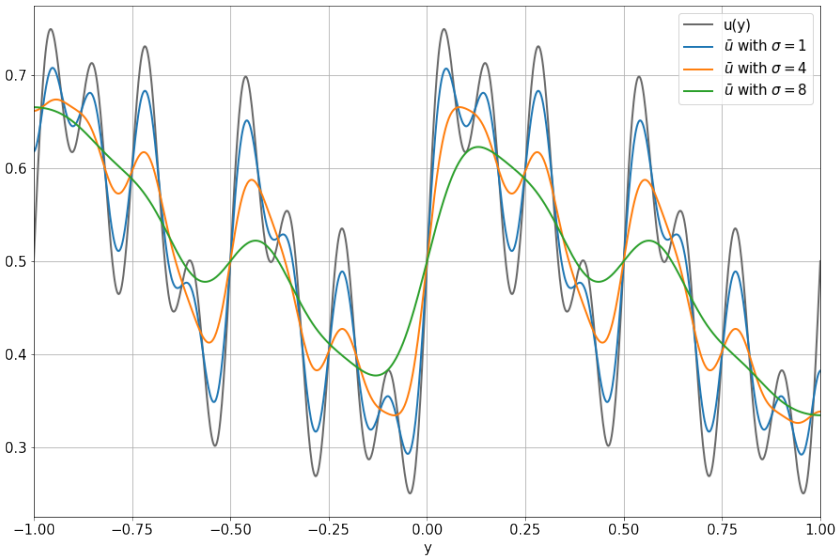


Figure 5.2: 1D convolution with Gaussian kernel.

5.2.2 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

$$\begin{aligned}\overline{u}(\boldsymbol{x}) &= \frac{\partial}{\partial x_1}\left(\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\rho(|\boldsymbol{y}-\boldsymbol{x}|)u(\boldsymbol{y})dy_1dy_2\right)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{\partial\rho(|\boldsymbol{y}-\boldsymbol{x}|)}{\partial x_1}u(\boldsymbol{y})dy_1dy_2\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\underbrace{\left(-\frac{\partial\rho(|\boldsymbol{y}-\boldsymbol{x}|)}{\partial y_1}\right)}_{\text{required kernel}}u(\boldsymbol{y})dy_1dy_2\end{aligned} \tag{5.3}$$

This kernel is shown in both 1D and 2D in Figure 5.3. Note that the action of this kernel look 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.

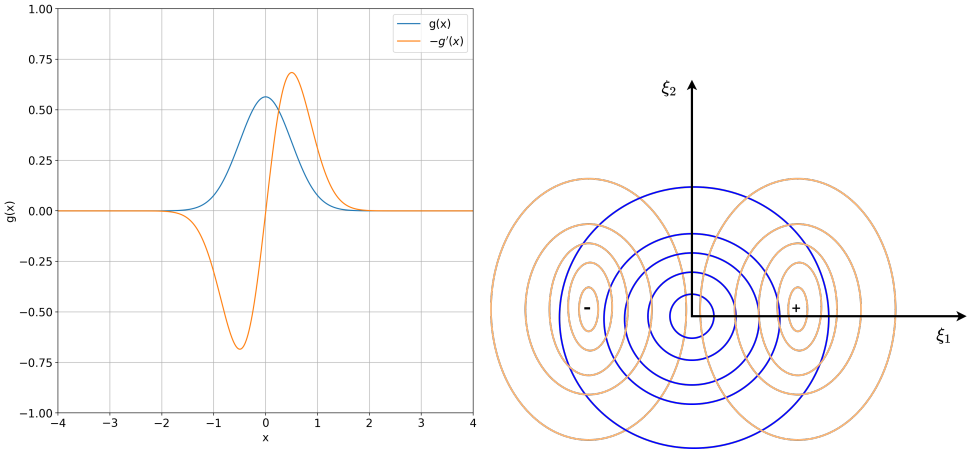
5.3 Discrete convolutions

To evaluate the discrete convolution in 2D, consider (5.2) and discretize it using quadrature. Then, we will have

$$\overline{U}[i,j] = \sum_{m=1}^{N_1}\sum_{n=1}^{N_2} g[m-i, n-j]U[m,n] \tag{5.4}$$

where we have absorbed the measure h^2 into the definition of the kernel. As in the continuous case, we will assume that g vanishes after a certain distance

$$g[p,q] = 0, \quad |p|, |q| > \bar{N} \text{ (measure of width of the kernel).}$$



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

Figure 5.3: Derivative kernel. The blue curve denotes the Gaussian kernel and the orange curve denotes the derivative.

Thus, the limits of the sum are reduced by exclusing all the pixels over which the convolution will be zero,

$$\overline{U}[i,j] = \sum_{m=i-\bar{N}}^{i+\bar{N}} \sum_{n=j-\bar{N}}^{j+\bar{N}} g[m-i, n-j]U[m,n]. \tag{5.5}$$

Let $m' = m - i$ and $n' = n - j$. Then

$$\overline{U}[i,j] = \sum_{m'=-\bar{N}}^{\bar{N}} \sum_{n'=-\bar{N}}^{\bar{N}} g[m', n']U[i+m', j+n']. \tag{5.6}$$

This 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 weights of the convolution layer, with the number of weight being $(\bar{N}+1)^2$.

Let's consider some examples:

- A smoothing kernel 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-direction and y-direction are given by

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

- Similarly, the second derivatives anlong the x and y-directions are given by kernels of the form

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

- While the Laplacian is given by the kernel of the type

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

Remark 5.3.1. *We can have different $\bar{N}$ in different directions. That is, we can have kernels with different widths along each direction.*

5.4 Connection to finite differences

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. Then we use the notation $U[i, j] = u(x_1^i, x_2^j)$. Using Taylor series expansion about (i, j)

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

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.

5.5 Convolution layers

The key things to remember are:

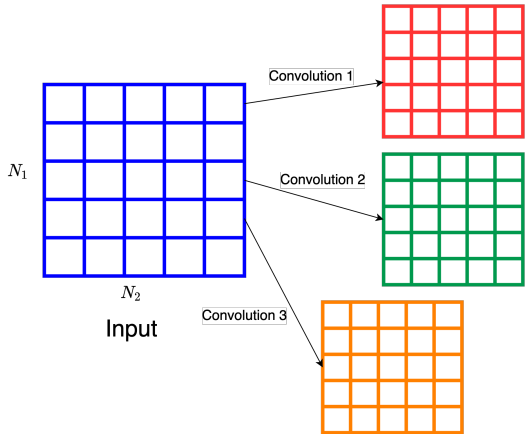
- Each convolution layer consists of several discrete convolutions, each with its own kernel.
- The weights of the kernel, which determine its action (smoothing, first derivative, second derivative etc.), are learnable parameters and are determined when training the network. Thus the way to think about 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 $N_1 \times N_2$ image as an input. Then, we will have multiple convolutions in a convolution layer, each of which will generate a different image, as shown in Figure 5.4(a). The trainable parameters of this layer are the weights of each convolution kernel. Assuming the width of the kernels is $\bar{N}$ in each direction, and there are P kernels, then the number of trainable weights will be $P \times (2\bar{N}+1)^2$.

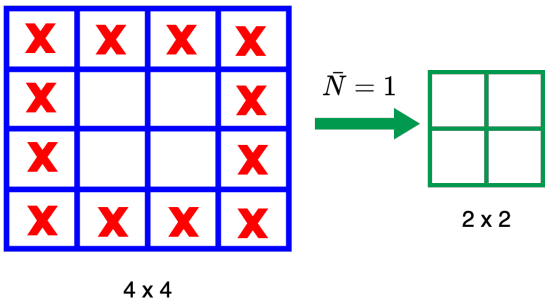
Next let us consider the size of the output image after applying a single kernel operation. Note that we will not be able to apply the kernel on the boundary pixels since there are no pixel-values available beyond the image boundary (see Figure 5.4(b)). Thus, we will have to skip $\bar{N}$ pixels at each boundary when applying the kernel, leading to an output image of shape $(N_1 - \bar{N} + 1) \times (N_2 - \bar{N} + 1)$.

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 Figure 5.4(c).

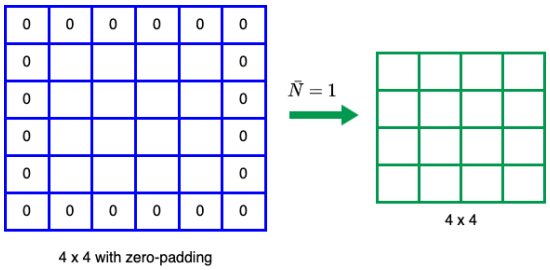
Another feature of convolutions is known as the *stride* which determines the number of pixels by which the kernel is shifted as we move over the image. In the examples above, the stride was 1 in both directions. In practice, we can choose a stride > 1 which will further shrink the size of the output 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 Figure 5.4(d)).



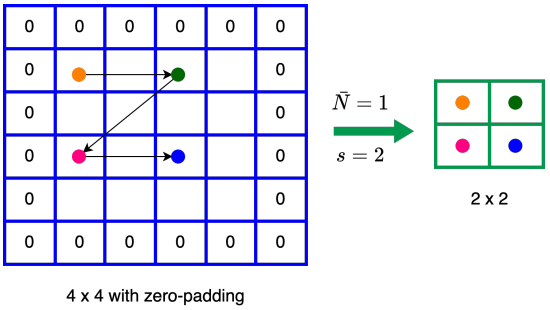
(a) Action of a convolution layer



(b) Convolution without padding



(c) Convolution with zero-padding



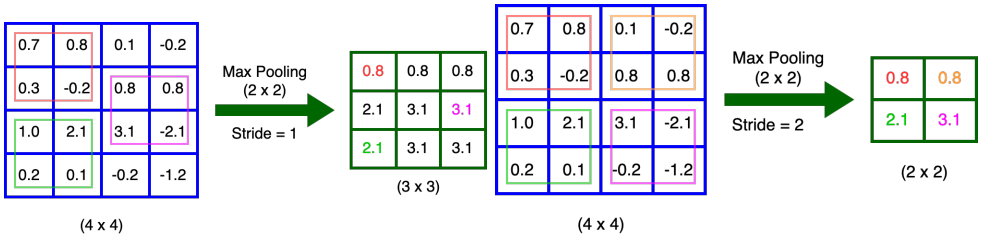
(d) Convolution with zero-padding and stride 2

Figure 5.4: Action of a convolution layer/kernel.

5.5.1 Average and Max Pooling

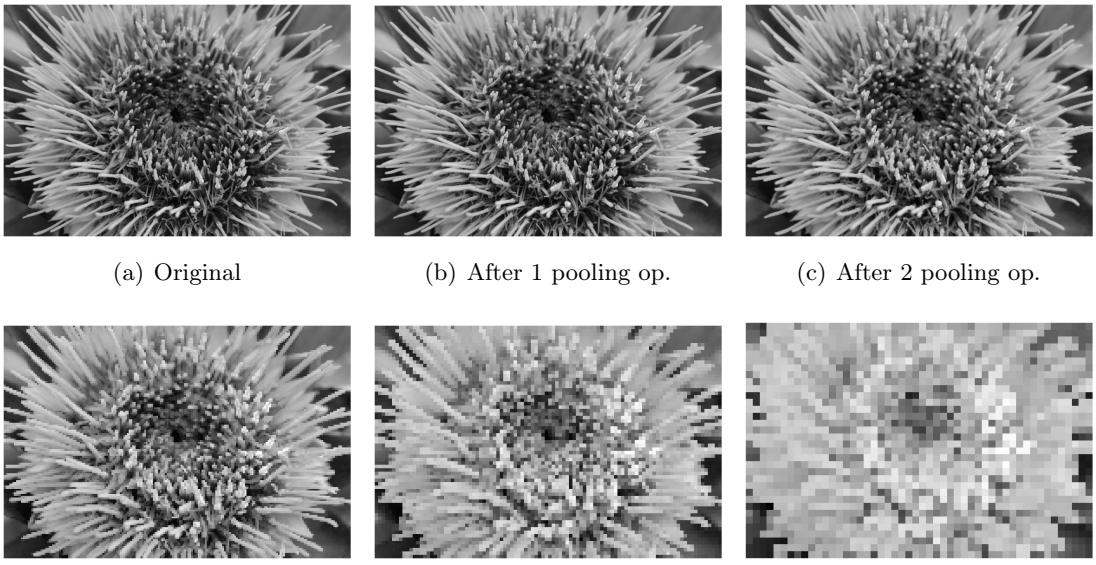
Pooling operations are generally used to reduce the size of an image, and allowing you to step through different scales of the image. If applied on an image of size $N \times N$ over patches of size $S \times S$, the new image will have dimensions $\frac{N}{S} \times \frac{N}{S}$, where S is the stride of the pooling operation. This is shown in Figure 5.5 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 Figure 5.5 (b) but not for 5.5 (a).

Also, we show in Figure 5.6 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.



(a) Stride 1 (b) Stride 2

Figure 5.5: Max pooling applied to an image over patches of size (2×2).



(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.

Figure 5.6: Max pooling applied repeatedly to an image over patches of size (2×2) with stride 2.

5.5.2 Convolution for inputs with multiple channels

Assume that the input to a convolution layer is of size $N_1 \times N_2 \times C$, where C is the number of channels in the input image. Then a convolution layer apply 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 a kernel of width $k = 2\bar{N} + 1$, then the kernel will be of the shape $k \times k \times C$, i.e., the kernel will have $k \times k$ weights for each of the C input channels of the input image. Each convolution will act on the input to give an output image of shape $M_1 \times M_2 \times 1$. The output of each convolution are stacked together to give the final output of the convolution layer. This can be written as,

$$\bar{U}[i,j,k] = \sum_{m=-\bar{N}}^{\bar{N}} \sum_{n=-\bar{N}}^{\bar{N}} \sum_{c=1}^{C} g_k[m,n,c] U[i+m, j+n, c], \quad 1 \le i \le M_1,\ 1 \le j \le M_2,\ 1 \le k \le P,$$

where g_k is the kernel of the k-th convolution in the layer. Note that the total number of trainable parameters will be $(2\bar{N}+1) \times (2\bar{N}+1) \times P \times C$. This is the type of convolutional layer most frequently encountered in a convolutional neural network, which is described in the following section.

5.6 Convolution Neural Network (CNN)

Now let's put all the elements together to form the full network. Consider an image classification problem. Then the CNN will be given by $\boldsymbol{y} = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta})$ where $\boldsymbol{x} \in \mathbb{R}^{N_1 \times N_2 \times N_3}$ is the input image with N_3 channels, while $\boldsymbol{y} \in \mathbb{R}^C$ is the 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 $\boldsymbol{y}$ are typically one-hot encoded.

The possible architecture of this network is shown in Figure 5.7. 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). The final fully connected layer reduced 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 $\boldsymbol{y}$. Since we are solving a classification problem, the loss function is taken to be the cross-entropy function

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

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

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 Figure 5.7, the convolution layers can be interpreted as encoding information about the input image, while the fully connected layers can be interpreted as

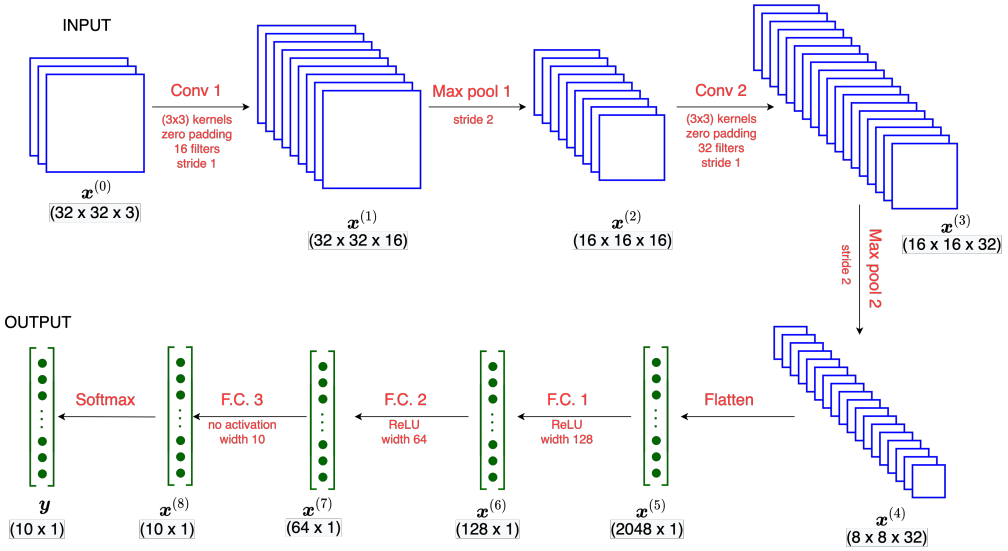


Figure 5.7: Example of a CNN architecture for an image classification problem, for 10 classes.

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. Sometime, activation is also applied to the output of the convolution layer along with a bias

$$x^{(l+1)}[i,j,k] = \sigma(\sum_{m,n,c} g_k[m,n,c]x^{(l)}[i+m,j+n,c] + b_k)$$

where a single bias b_k is used for a given output channel k.

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 flatten input would be extremely large. In that case, we would have 2 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.

5.7 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

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

Consider 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 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, we will need to shift the rows of the outer-product by 2

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

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 Figure 5.8(a), 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 in the Figure), which leads to checker-boarding, which is undesirable. Checker-boarding refers to pixel-to-pixel variations in the values of the output image.

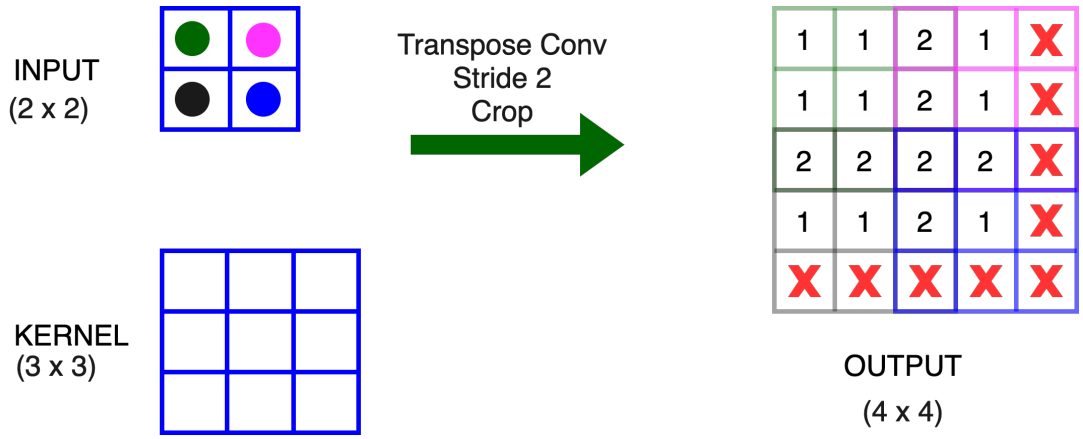
5. One way to avoid checker-boarding, is by ensuring that the filter size is an integer multiple of the stride. Let us repeat the previous example but with a (2×2) kernel. The operation is illustrated in Figure 5.8(b)In this case, we do not need to pad (crop) and each output pixel has an equal contribution.

We make some remarks:

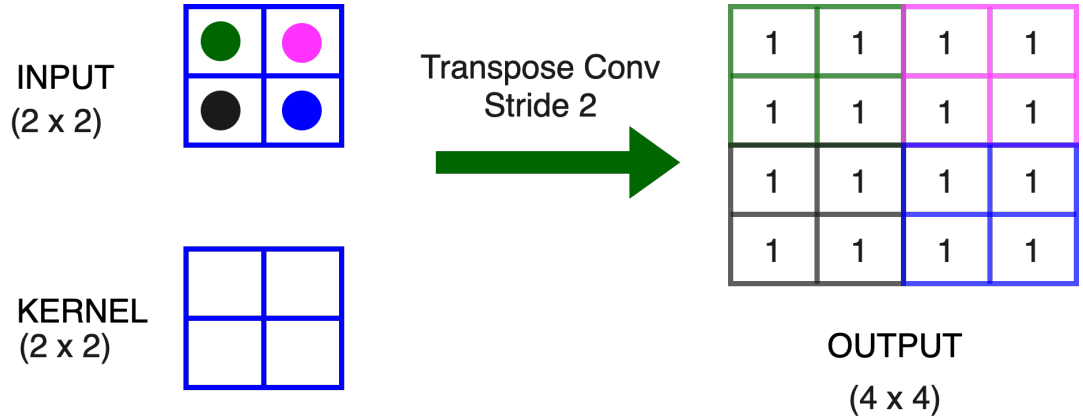
1. Transpose convolution layers are also called fractionally-strided layers, because for every one step in the input, we take greater than one step in the output. This is the opposite of what happens in a convolution layer. In a convolution layer, we take step greater than one in the input image, for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

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.



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



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

Figure 5.8: 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.

5.8 Image-to-image transformations

Image-to-image to 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 (RHS) field to the solution of the PDE.

We will discuss a particular type of network for such transformations, which is known as U-Nets [26]. In a U-Net (see Figure 5.9), there a down 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 increase. After we reach the coarsest level, we have an upward branch that scales up the image and reduced the number of channels using 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. These connections are similar to what are used in ResNets. 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 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.

Remark 5.8.1. *The U-net architecture shares many common features with the V-cycle that is typically used in multigrid preconditioners.*

Remark 5.8.2. *We can also think of a the U-net as an encode-decoder network with the additional feature of including skip connections.*

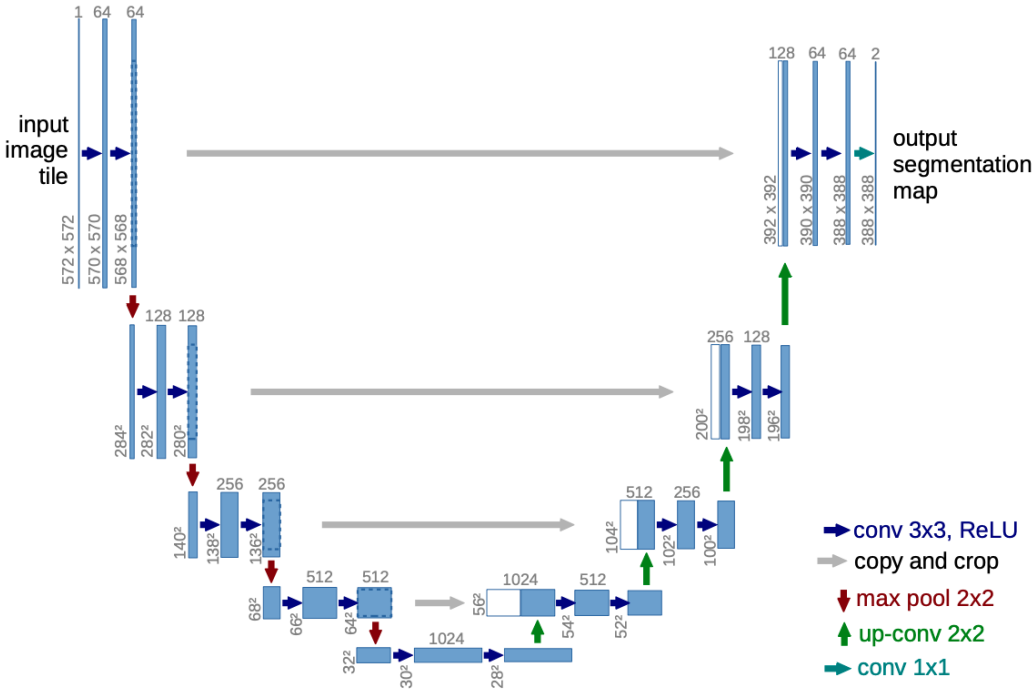


Figure 5.9: Example of a U-Net taken from [26].

Chapter 6

Operator Networks

6.1 The problem with PINNs

Recall that a typical MLP $\boldsymbol{y} = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta})$ 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 Chapter 4, a PINN is a network of the form $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta})$ taking as input the independent variable $\boldsymbol{x}$ of the underlying PDE and giving the solution $\boldsymbol{u}(\boldsymbol{x})$ (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

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

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.

6.2 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

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

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 Figure 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 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 $g(\boldsymbol{x})$.

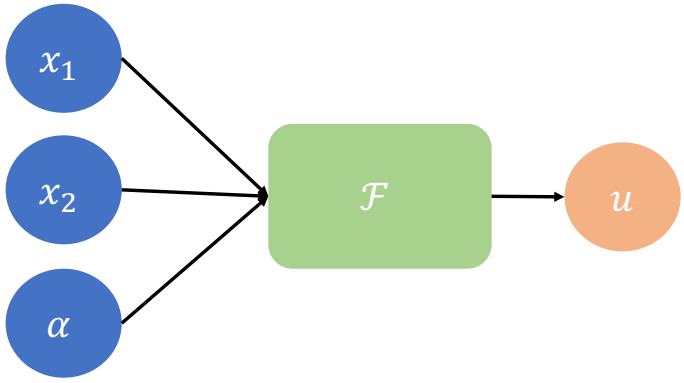


Figure 6.1: Schematic of a PINN with a parameterized input.

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.**

6.3 Operators

Consider a class of functions $\boldsymbol{a}(\boldsymbol{y}) \in A$ such that $\boldsymbol{a} : \Omega_Y \to \mathbb{R}^D$. 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
$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= f, \quad \boldsymbol{x} \in \Omega \\ u &= g, \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{6.2}$$
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, $u = \mathcal{N}(f)(\boldsymbol{x})$. The input and the output to the operator are related by the equation above where it is assumed that κ and g are given and fixed.

2. Consider the PDE
$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= f, \quad \boldsymbol{x} \in \Omega \\ u &= g, \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{6.3}$$
which is the same as the previous PDE but we are assuming 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 equation above where it is assumed that f and g are given and fixed.

3. Once again, consider the same PDE but with conductivity and the boundary condition being allowed to change

$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= f(x), \quad \boldsymbol{x} \in \Omega \\ u(x) &= g(x), \quad \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{6.4}$$

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 above where it is assumed that f is given and fixed.

4. Now consider the equations of linear isotropic elasticity posed on a three-dimensional domain $\Omega \subset \mathbb{R}^3$,

$$\begin{aligned} \nabla \cdot \big(\lambda(\nabla \cdot \boldsymbol{u})\boldsymbol{I} + 2\mu\nabla^S(\boldsymbol{u})\big) &= \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\ \boldsymbol{u}(\boldsymbol{x}) &= \boldsymbol{g}(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega. \end{aligned} \tag{6.5}$$

Consider the operator defined by $\boldsymbol{u}(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{f})(\boldsymbol{x})$. Here the input function, $\boldsymbol{f} : \Omega \to \mathbb{R}^3$, and the output function $\boldsymbol{u} : \Omega \to \mathbb{R}^3$. The two are related by the equations above where $\lambda, \mu, \boldsymbol{g}$ are given and fixed.

5. Now, consider a different PDE. In particular, the advection-diffusion-reaction equation,

$$\begin{aligned} \frac{\partial u}{\partial t} + \boldsymbol{a} \cdot \nabla u - \kappa\nabla^2 u + u(1-u) &= f, \quad (\boldsymbol{x}, t) \in \Omega \times (0, T] \\ u(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t), \quad (\boldsymbol{x}, t) \in \partial\Omega \times (0, T] \\ u(\boldsymbol{x}, 0) &= u_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega. \end{aligned} \tag{6.6}$$

We want to find the operator $\mathcal{N}$ 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 $\boldsymbol{a}, \kappa, f, g$ given and fixed.

Remark 6.3.1. *It is often useful to determine whether an operator is linear or non-linear. 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 next section. These types of networks are often referred to as Operator Networks. They 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 in the next section.

6.4 Deep Operator Network (DeepONet) Architecture

Operator networks were first proposed by Chen and Chen [5], where they considered only shallow networks with a single hidden layer. This idea was rediscovered and extended to deep architectures more recently in [14] 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 Figure 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 $\boldsymbol{x}$ 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 $u(\boldsymbol{x})$

$$u(\boldsymbol{x}) \approx \widetilde{\mathcal{N}}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta}) = \sum_{k=1}^{p} \beta_k(\boldsymbol{a}) \tau_k(\boldsymbol{x}). \tag{6.7}$$

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

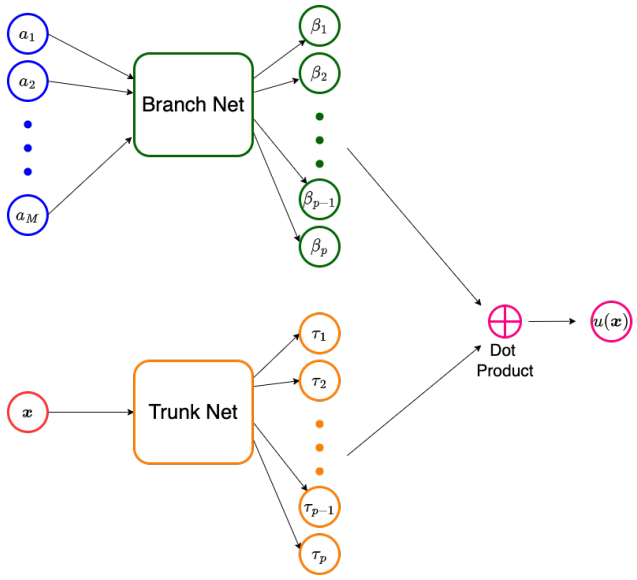


Figure 6.2: Schematic of a DeepONet

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 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 (6.7) 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 that of what is used 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 points 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 $N_1 \approx 10^4$, while the output is typically small $p \approx 10^2$, this network can be thought of as an encoder.

   When points for sampling the input function are chosen on a uniform grid, the appropriate architecture for the branch network comprises convolutional layer layers. In that case this network maps an image of large dimension ($N_1 \approx 10^4$) 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 experssivity of the 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.

6.5 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

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

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

7. Once trained, then given any new $a \in A$ samples 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 6.5.1. *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 6.5.2. *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 Figure 6.3.*

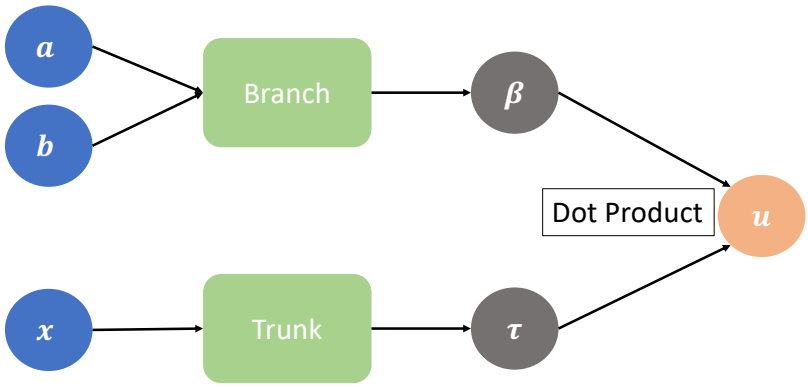


Figure 6.3: Schematic of a DeepONet with two input functions.

Remark 6.5.3. *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 Figure 6.3.*

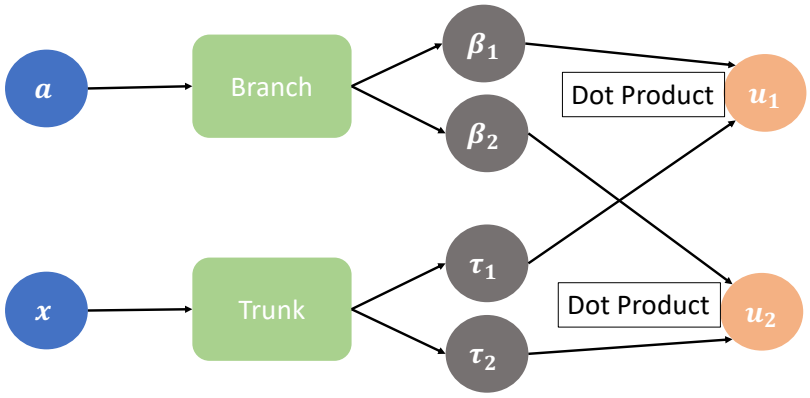


Figure 6.4: Schematic of a DeepONet with two output functions.

6.6 Error Analysis for DeepONets

There is a universal approximation theorem for a shallow version of DeepONets by Chen and Chen [5]

Theorem 6.6.1. *Suppose Ω_X and Ω_Y are compact sets in $\mathbb{R}^D$ (or more generally a Banach space) and $\mathbb{R}^d$, respectively. Let $\mathcal{N}$ be a nonlinear, continuous operator mapping $V \subset C(\Omega_Y)$ into $C(\Omega_X)$. Then given $\epsilon > 0$, there exists a DeepONet with M sensors and a single hidden layer of width P in the branch and trunk nets such that*

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

for a large enough P and M.

This result has been extended to a deeper version of the network in [14], and generalized further by removing the compactness assumptions on the spaces [12].

Recently in [20], the authors have developed an estimate of the error in a DeepONet that clearly pinpoints the different sources of error in a DeepONet. This estimate states, the error measured in the $L^2(\Omega)$ norm is bounded as

$$\max_{a\in A} \|\widetilde{\mathcal{N}}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta}) - \mathcal{N}(a)(\boldsymbol{x})\|_{L^2(\Omega)} \leq C\left(\epsilon_h + \sqrt{\epsilon_t} + \epsilon_s + M^{-\alpha_1} + (N_2)^{-\alpha_2}\right) \tag{6.8}$$

where ϵ_h is the error made by the numerical solver used to generate the approximate target solutions $u^{(i)}$ in the training set (as compared to the exact solutions), ϵ_t is the final training error/loss attained, while ϵ_s bounds the distance of any $a \in A$ from the set of functions $\{a^{(i)}\}_{i=1}^{N_1}$ used to construct the construct the training set, i.e., an estimate of how well the training samples covers the input space A. Further, since the input function is evaluated at M finite sensor nodes, while the output is evaluated at N_2 output nodes, this will lead to an additional discretization (or quadrature) error which is given by the last two terms in (6.8). Note that this is similar to the error estimates obtained for PINNs in (4.21).

6.7 Physics-Informed DeepONets

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

$$\begin{aligned} \nabla \cdot (\kappa u) &= f \text{ in } \Omega \\ u &= g \text{ on } \partial\Omega \end{aligned} \tag{6.9}$$

where κ and g are prescribed. To construct the operator $\mathcal{N}$ that maps f to u, we need to solve the PDE externally. 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 Physic-Informed DeepONets proposed in [30]. So for the above model PDE, the loss additional physics-based loss would would be,

$$\Pi_p(\boldsymbol{\theta}) = \frac{1}{\bar{N}_1}\frac{1}{\bar{N}_2}\sum_{i=1}^{\bar{N}_1}\sum_{k=1}^{\bar{N}_2}\left|\nabla_{\boldsymbol{x}} \cdot \left(\kappa\nabla_{\boldsymbol{x}}\widetilde{\mathcal{N}}(\boldsymbol{x}^{(k)}, \boldsymbol{f}^{(i)}; \boldsymbol{\theta})\right) - f^{(i)}(\boldsymbol{x}^{(k)})\right|^2. \tag{6.10}$$

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

$$\Pi_d(\boldsymbol{\theta}) = \frac{1}{N_1 N_2}\sum_{i=1}^{N_1}\sum_{k=1}^{N_2}|\widetilde{N}(\boldsymbol{x}^{(k)}, \boldsymbol{f}^{(i)}; \boldsymbol{\theta}) - u^{(i)}(\boldsymbol{x}^{(k)})|^2. \tag{6.11}$$

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

$$\Pi(\boldsymbol{\theta}) = \Pi_d(\boldsymbol{\theta}) + \lambda\Pi_p(\boldsymbol{\theta}), \tag{6.12}$$

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 is 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.

6.8 Fourier Neural Operators - Architecture

We now introduce and discuss Fourier Nerual Operators (FNOs) [13]. We discuss their architecture in this section, and then discuss other aspects in the following section.

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 so that the extended version maps a scalar valued function to another scalar valued function.

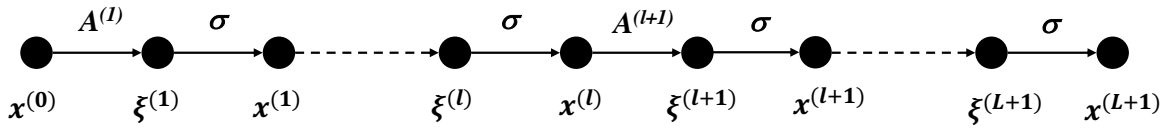


Figure 6.5: Computational graph for a feed-forward MLP.

In Figure 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 $\boldsymbol{x}^{(L)} = y$ 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 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 $a(\boldsymbol{x}) : \Omega \mapsto \mathbb{R}^1$. Similarly the output is the function $u(\boldsymbol{x}) : \Omega \mapsto \mathbb{R}^1$. This leads us to the computational graph shown in Figure 6.6.

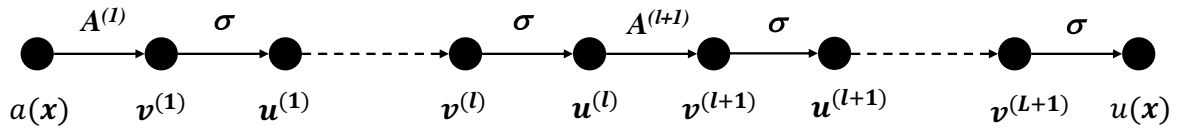


Figure 6.6: Computational graph for a feed-forward Fourier Neural Operator (FNO) network.

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 functions with H components. That is,

$$\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(L)}, \boldsymbol{u}^{(1)}, \cdots, \boldsymbol{u}^{(L)} : \Omega \mapsto \mathbb{R}^H. \tag{6.13}$$

As shown in Figure 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 $A^{(1)}$, 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,

$$\boldsymbol{v}^{(1)}(\boldsymbol{x}) = A^{(1)}(a)(\boldsymbol{x}), \tag{6.14}$$

where

$$v_i^{(1)}(\boldsymbol{x}) = W_i^{(1)} a(\boldsymbol{x}) + b_i^{(1)}, \qquad i = 1, \cdots, H. \tag{6.15}$$

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

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

$$v^{(L+1)}(\boldsymbol{x}) = A^{(L+1)}(\boldsymbol{u}^{(L)})(\boldsymbol{x}), \tag{6.16}$$

where

$$v^{(L+1)}(\boldsymbol{x}) = W_i^{(L+1)} u_i^{(L)}(\boldsymbol{x}) + b^{(L+1)}, \qquad i = 1, \cdots, H. \tag{6.17}$$

Here $W_i^{(L+1)}$ 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. It is a simple extension of the activation function applied to the point-wise values of the input function. That is,

$$\boldsymbol{u}^{(n)}(\boldsymbol{x}) = \sigma(\boldsymbol{v}^{(n)})(\boldsymbol{x}), \tag{6.18}$$

where

$$u_i^{(n)}(\boldsymbol{x}) = \sigma(v_i^{(n)}(\boldsymbol{x})), \qquad i = 1, \cdots, H. \tag{6.19}$$

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

$$\boldsymbol{v}^{(n+1)}(\boldsymbol{x}) = A^{(n+1)}(\boldsymbol{u}^{(n)})(\boldsymbol{x}), \tag{6.20}$$

where

$$\begin{aligned} v_i^{(n+1)}(\boldsymbol{x}) &= W_{ij}^{(n+1)} u_j^{(n)}(\boldsymbol{x}) + b_i^{(n+1)} && (6.21) \\ &\quad + \int_\Omega \kappa_{ij}^{(n+1)}(\boldsymbol{y} - \boldsymbol{x}) u_j^{(n)}(\boldsymbol{y}) d\boldsymbol{y}, \qquad i = 1, \cdots, H. && (6.22) \end{aligned}$$

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,

$$v_i(x_1, x_2) \;=\; \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(y_1 - x_1, y_2 - x_2) u_j(y_1, y_2)\, dy_2 dy_1, \qquad i = 1, \cdots, H. \tag{6.23}$$

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

Remark 6.8.1. *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 6.8.2. *It is instructive to list all the trainable entities in a FNO. First we list all the trainable parameters:*

$$W_i^{(1)}, W_{ij}^{(2)}, \cdots, W_{ij}^{(L)}, W_i^{(L+1)}; b_i^{(1)}, b_i^{(2)}, \cdots, b_i^{(L)}, b^{(L+1)}. \tag{6.24}$$

Thereafter, all the trainable kernel functions

$$\kappa_{ij}^{(n)}(\boldsymbol{z}), \qquad n = 2, \cdots, L. \tag{6.25}$$

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.

6.9 Discretization of the Fourier Neural Operator

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

$$a, \boldsymbol{v}^{(1)}, \boldsymbol{u}^{(1)}, \cdots, \boldsymbol{v}^{(L)}, \boldsymbol{u}^{(L)}, v^{(L+1)}, u. \tag{6.26}$$

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 on 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,

$$a[m, n] = a(x_{1m}, x_{2n}), \quad m = 1 \cdots, N_1,\, n = 1 \cdots, N_2. \tag{6.27}$$

where

$$x_{1m} \;=\; (m-1) \times \frac{L_1}{N_1 - 1} \tag{6.28}$$

$$x_{1n} \;=\; (n-1) \times \frac{L_2}{N_2 - 1}. \tag{6.29}$$

The same representation will be used for all other functions.

We now have to consider the discrete version of all operations 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 $A^{(1)}$. The discretized version is

$$\boldsymbol{v}^{(1)}[m, n] = A^{(1)}(a)[m, n], \tag{6.30}$$

where

$$v_i^{(1)}[m, n] = W_i^{(1)} a[m, n] + b_i^{(1)}, \qquad i = 1, \cdots, H. \tag{6.31}$$

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

$$v^{(L+1)}[m, n] = A^{(L+1)}(\boldsymbol{u}^{(L)})[m, n], \tag{6.32}$$

where

$$v^{(L+1)}[m, n] = W_i^{(L+1)} u_i^{(L)}[m, n] + b^{(L+1)}, \qquad i = 1, \cdots, H. \tag{6.33}$$

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

$$\boldsymbol{u}^{(n)}[m, n] = \sigma(\boldsymbol{v}^{(n)})[m, n], \tag{6.34}$$

where

$$u_i^{(n)}[m, n] = \sigma(v_i^{(n)}[m, n]), \qquad i = 1, \cdots, H. \tag{6.35}$$

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

$$\boldsymbol{v}^{(p+1)}[m, n] = A^{(p+1)}(\boldsymbol{u}^{(p)})[m, n], \tag{6.36}$$

where

$$v_i^{(p+1)}[m, n] = W_{ij}^{(p+1)} u_j^{(p)}[m, n] + b_i^{(p+1)} \tag{6.37}$$

$$+ \sum_{r=1}^{N_1} \sum_{s=1}^{N_2} \kappa_{ij}^{(p+1)}[r - m, s - n] u_j^{(p)}[r, s] h_1 h_2, \qquad i = 1, \cdots, H, \tag{6.38}$$

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 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 $O(N_1^2 N_2^2 H) = O(N^2 H^2)$, 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."

6.10 The Use of Fourier Transforms

Consider a periodic function $u(x_2, x_2)$ define 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,

$$u(x_1, x_2) \approx \sum_{m=-N_1/2}^{N_1/2} \sum_{n=-N_2/2}^{N_2/2} \hat{u}[m, n] e^{2\pi \mathrm{i}(\frac{m x_1}{L_1} + \frac{n x_2}{L_2})}. \tag{6.39}$$

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:

$$\hat{u}[r,s] = \frac{1}{L_1 L_2} \int_0^{L_1} \int_0^{L_2} u(x_1,x_2) e^{-2\pi \mathrm{i}(\frac{rx_1}{L_1}+\frac{sx_2}{L_2})} dx_1 dx_2. \tag{6.40}$$

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 (6.23). 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,

$$\begin{aligned}
v_i(x_1,x_2) &= \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(y_1-x_1, y_2-x_2) \sum_{m,n} \hat{u}_j[m,n] e^{2\pi \mathrm{i}(\frac{my_1}{L_1}+\frac{ny_2}{L_2})} \, dy_2 dy_1 \\
&= \sum_{m,n} \hat{u}_j[m,n] \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(y_1-x_1, y_2-x_2) e^{2\pi \mathrm{i}(\frac{my_1}{L_1}+\frac{ny_2}{L_2})} \, dy_2 dy_1 \\
&= \sum_{m,n} \hat{u}_j[m,n] \int_{-x_1}^{L_1-x_1} \int_{-x_2}^{L_2-x_2} \kappa_{ij}(z_1,z_2) e^{2\pi \mathrm{i}(\frac{m(z_1+x_1)}{L_1}+\frac{n(z_2+x_2)}{L_2})} \, dz_2 dz_1 \\
&= \sum_{m,n} \hat{u}_j[m,n] e^{2\pi \mathrm{i}(\frac{mx_1}{L_1}+\frac{nx_2}{L_2})} \int_0^{L_1} \int_0^{L_2} \kappa_{ij}(z_1,z_2) e^{2\pi \mathrm{i}(\frac{mz_1}{L_1}+\frac{nz_2}{L_2})} \, dz_2 dz_1 \\
&= L_1 L_2 \sum_{m,n} \hat{u}_j[m,n] \hat{\kappa}_{ij}[-m,-n] e^{2\pi \mathrm{i}(\frac{mx_1}{L_1}+\frac{nx_2}{L_2})}.
\end{aligned} \tag{6.41}$$

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 (6.40). 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 (6.39) at a finite set of grid points. Further, it means that we have to approximate the integral in the Fourier transform (6.40). This alternate form is given by

$$\hat{u}[r,s] = \frac{h_1 h_2}{L_1 L_2} \sum_{m=1}^{N_1} \sum_{n=1}^{N_2} u[m,n] e^{-2\pi \mathrm{i}(\frac{r x_{1m}}{L_1} + \frac{s x_{2n}}{L_2})}. \tag{6.42}$$

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 (6.39) and (6.42) 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.

Chapter 7

Probabilistic Deep Learning

So far, we have considered regression and classification problems, where for a given input $\boldsymbol{x}$ we need to compute a *single* output $\boldsymbol{y}$. However, this may not be enough for many problems of interest. In fact, there may be *many* $\boldsymbol{y}$'s for a given $\boldsymbol{x}$. For example,

1. $\boldsymbol{y}$ and $\boldsymbol{x}$ might be measured with some *random noise*.
2. $\boldsymbol{y}$ and $\boldsymbol{x}$ might be *inherently stochastic*. For instance, $\boldsymbol{y}$ could be the pressure measured in a turbulent flow at some point $\boldsymbol{x}$ in space.
3. *inverse problems* can have multiple solutions. For instance, the forward/direct problems would be determining the temperature field given the head conductivity, while the inverse problem could be determining the conductivity field given the (possibly noisy) temperature field.

Thus, we need to formulate a probabilistic framework to use deep learning algorithms to solve such problems. Recall, that our deterministic model was given by $\boldsymbol{y} = \boldsymbol{\mathcal{F}}(\boldsymbol{x}; \boldsymbol{\theta})$. In the probabilistic setup, $\boldsymbol{y}$, $\boldsymbol{x}$ and $\boldsymbol{\theta}$ are treated as *random variables*.

Before we can work with random variables we need to understand some key elements of the theory of probability that are necessary in defining random variables.

7.1 Key elements of Probability Theory

A **random experiment** is described by a procedure and a set of one or more observations/measurements. For example,

1. Observe a switch and determine whether it is *on* or *off*.
2. Toss a coin 3 times and note the sequence of heads H or tails T.
3. Toss a coin 3 times and count the number of times H appears. Note that this is the same experiment as earlier but the measurement is different.
4. Spin a spinner, and measure the final angle in radians.

The **outcome** is the results of the experiment that cannot be broken down into smaller parts. The **sample space**, denoted by $\mathbb{S}$, is the set of all possible outcomes of an experiment. For each of the four random experiments observed above, we have

1. $\mathbb{S} = \{\text{on},\ \text{off}\}$.

2. $\mathbb{S} = \{TTT,\ TTH,\ THT,\ HTT,\ THH,\ HTH,\ HHT,\ HHH\}$.

3. $\mathbb{S} = \{0,\ 1,\ 2,\ 3\}$.

4. $\mathbb{S} = \{\psi\ :\ \psi \in (0, 2\pi]\}$.

Note that there is a fundamental difference between the first 3 experiments, where $\mathbb{S}$ is discrete and countable, and the last experiment where the $\mathbb{S}$ is uncountable.

An **event** is a collection of outcomes, i.e., a subset of $\mathbb{S}$. Typically the outcomes in an event satisfy a condition. Let's see some examples for the above experiments

1. $A = \{\text{on}\}$ or $A = \{\text{on},\ \text{off}\} = \mathbb{S}$.

2. A are all outcomes with at least 2 H, i.e., $A = \{THH,\ HTH,\ HHT,\ HHH\}$.

3. A are all outcomes with at least 2 H, i.e., $A = \{2,\ 3\}$. If we define B to be all outcomes with 4 H, then no outcome would satisfy this condition, i.e., $B = \emptyset$ the null set.

4. A are all outcomes with $\psi > \pi/4$, i.e, $A = \{\psi\ :\ \psi \in (\pi/4, 2\pi]\}$.

An **event class** $\mathscr{E}$ is a collection of all event (sets) over which probabilities can be defined. When $\mathbb{S}$ is countable, $\mathscr{E}$ is all subsets of $\mathbb{S}$. When $\mathbb{S}$ is not countable, $\mathscr{E}$ is the *Borel* field (or Borel algebra), which is the collection of all open and closed sets in $\mathbb{S}$.

The **probability law** is a rule that assigns a probability to all sets in an event class $\mathscr{E}$. We list the **axioms of probability**, which are the requirements of a probability law.

Consider a sample space $\mathbb{S}$ for an experiment and the corresponding event class $\mathscr{E}$. Let $P : \mathscr{E} \mapsto [0, 1]$ satisfy

1. $P[A] \geq 0$ for all $A \in \mathscr{E}$.

2. $P[\mathbb{S}] = 1$.

3. If $A_1, A_2, ...$ are events such that $A_i \cap A_j = \emptyset$ for all $i \neq j$, i.e., the events are *mutually exclusive*, then
$$P[\bigcup_{i=1}^{\infty} A_i] = \sum_{i=1}^{\infty} P[A_i].$$

Any assignment P that satisfies the above conditions is said to be a *valid probability law.* Note that probability is like mass. It is non-negative (axiom 1), conserved (total mass is always 1, axiom 2), and for distinct points the total mass is obtained by adding individual masses (axiom 3).

If $\mathbb{S}$ is countable, then it is sufficient to define a probability law for all elements of $\mathbb{S}$, i.e., for all elementary outcomes, while making sure that the probabilities are non-negative and add up to 1 (the first two axioms). Let us try to assign probability laws for the first three examples which have a countable $\mathbb{S}$ using these criteria.

1. For some $p \in [0, 1]$, define $P[\text{on}] = p$; $P[\text{off}] = 1 - p$.

2. For a fair die with no memory, $P[a_i] = 1/8$, where $a_i \in \mathbb{S}, i = 1, \cdots, 8$.

3. For a fair die with no memory, $P[0] = 1/8,\ P[1] = 3/8,\ P[2] = 3/8,\ P[3] = 1/8$.

Remark 7.1.1. *As an exercise, verify that the axioms are satisfied for each of these cases.*

For a continuous sample space, it is sufficient to define a probability law for all open and closed intervals, while ensuring axioms 1 and 2. Let us consider the fourth example which has an uncountable $\mathbb{S}$. If the spinner is completely unbiased, then the probability is uniformly distributed. Then for $b \geq a$, we say that $P[(a, b]] = (b - a)/(2\pi)$. Note that the probability of singleton sets in a continuous sample space is zero (for any distribution).

Remark 7.1.2. *From this point on, whenever we talk about the sample space $\mathbb{S}$, we will implicitly assume that we are referring to the triplet $(\mathbb{S}, \mathscr{E}, P)$. This triplet is also known as a "measure space".*

7.2 Random Variables

A **random variable** X is a function defined from $\mathbb{S}$ to the real line with the property that the set $A_b = \{\xi \in \mathbb{S} : X(\xi) \leq b\}$ belongs to $\mathscr{E}$ for all $b \in \mathbb{R}$. Note that in the measure theoretic language, we are requiring X to be a *measurable function*. Also note that according to the definition of A_b, we are enforcing the requirement that we should be able to evaluate $P[A_b]$ for all $b \in \mathbb{R}$.

Let us define random variables (RVs) for the above examples:

1. For $\mathbb{S} = \{\text{on}, \text{off}\}$ with $P[\text{on}] = p$, $P[\text{off}] = 1 - p$, define the RV
$$X(\xi) = \begin{cases} 0 & \text{if } \xi = \text{off} \\ 1 & \text{if } \xi = \text{on}. \end{cases} \tag{7.1}$$
This is also known as a *Bernoulli Random Variable*.

2. For $\mathbb{S} = \{TTT,\ TTH,\ THT,\ HTT,\ THH,\ HTH,\ HHT,\ HHH\}$ with $P[a_i] = 1/8$ for all $a_i \in \mathbb{S}$, define
$$X(\xi) = \text{Number of heads in } \xi. \tag{7.2}$$
Note that this is the random event that was described in Experiment 3.

3. This random event is already a random variable.

4. For the spinner experiment with $\mathbb{S} = \{\psi\ :\ \psi \in (0, 2\pi]\}$ with $P[(a, b]] = (b-a)/(2\pi)$, define
$$X(\psi) = \frac{\psi}{2\pi}. \tag{7.3}$$

If X is defined on a discrete sample space, it is called a *discrete random variable*, while if it is defined on a continuous sample space, it is called a *continuous random variable*.

As described above, a random variable inherits its probabilistic interpretation from the measure space used to define it. In the following sections we define the probabilistic interpretation of a random variable.

7.2.1 Cumulative distribution function

The **cumulative distribution function** (cdf) of a random variable X is given by

$$F_X(x) = P[\xi : X(\xi) \leq x]$$

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 $F_X(x) = P[\emptyset] = 0$
- if $0 \leq x < 1$, then $F_X(x) = P[\{\text{off}\}] = 1 - p$
- if $x \geq 1$, then $F_X(x) = P[\{\text{on}, \text{off}\}] = 1$

The full cdf is shown in Figure 7.1(a).

2. For the RV defined by (7.2)

   - if $x < 0$, then $F_X(x) = P[\emptyset] = 0$
   - if $0 \leq x < 1$, then $F_X(x) = P[\#ofH = 0] = P[\{TTT\}] = 1/8$
   - if $1 \leq x < 2$, then $F_X(x) = P[\#ofH = 0, 1] = 1/8 + 3/8 = 4/8$
   - if $3 \leq x < 3$, then $F_X(x) = P[\#ofH = 0, 1, 2] = 1/8 + 3/8 + 3/8 = 7/8$
   - if $x \geq 3$, the $F_X(x) = P[\#ofH = 0, 1, 2, 3] = 1$

   The full cdf is shown in Figure 7.1(b).

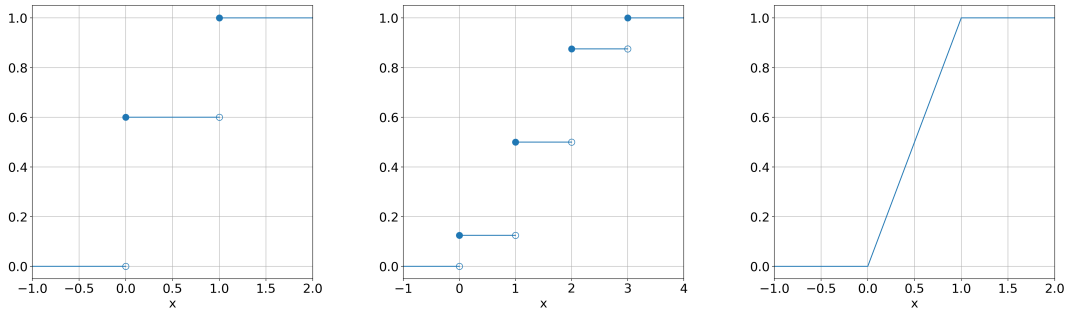
3. This random variable is the same as Example 2.

4. For the spinner experiment with the RV defined by (7.3)

$$F_X(x) = P[\psi : X(\psi) \leq x] = P[\{\psi : \psi \leq 2\pi x\}]$$

   - if $x < 0$, then $F_X(x) = P[\emptyset] = 0$
   - if $0 \leq x < 1$, then $F_X(x) = P[\{\psi \in (0, 2\pi x]\}] = \frac{2\pi x}{2\pi} = x$
   - if $x \geq 1$, then $F_X(x) = P[\{\psi \leq 2\pi\}] = 1$

   The full cdf is shown in Figure 7.1(c).



(a) Bernoulli (b) 3 coin tosses (c) Spinner

Figure 7.1: Examples of cumulative distribution functions

Let us discuss some properties of F_X:

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 Figure 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*

$$\begin{aligned} F_X(b) = P[X \leq b] &= P[(X \leq a) \cup (a < X \leq b)] \\ &= P[X \leq a] + P[a < X \leq b] \qquad \textit{(mutually exclusive events)} \\ &= F_X(a) + P[a < X \leq b]. \end{aligned}$$

Thus,

$$P[a < X \leq b] = F_X(b) - F_X(a).$$

7.2.2 Probability density function

We define the **probability density function** (pdf). For a continuous F_X, it is defined as

$$f_X(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_X(x) \tag{7.4}$$

which enjoys the following properties inherited 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. Furthermore,
$$\int_{-\infty}^{\infty} f_X(y)\mathrm{d}y = \lim_{x\to\infty} F_X(x) - \lim_{x\to-\infty} F_X(x) = 1.$$

6. For a very small $h > 0$, we have the interpretation
$$P[a < X \leq a + h] = \int_a^{a+h} f_X(y)\mathrm{d}y \approx h f_X(a).$$
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.

7.2.3 Examples of Important RVs

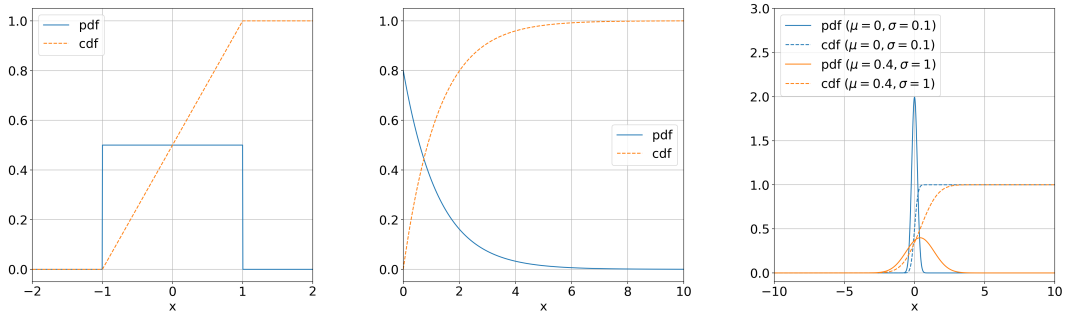
Let us look at some important random variables and the associated cdf, pdf (also see Figure 7.2):

1. **Uniform RV:** 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:** used to model lifetime of devices/humans after a critical event. In this case, X represents the time to failure and $P[X > x] = e^{-\lambda x}$ where $\lambda > 0$ is a model parameter which denotes the rate of failure. Thus,
$$F_X(x) = P[X \leq x] = 1 - P[X > x] = 1 - e^{-\lambda x},$$
and
$$f_X(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_X(x) = \lambda e^{-\lambda x}.$$

3. **Gaussian RV:** used to model natural things like height, weight, etc. In fact, through the Central Limit Theorem, this is also the distribution given by an aggregate of many RVs. The pdf is given by
$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$
which is parameterized by the *mean* μ which denotes the center of this distributions, and the *variance* σ^2 which denotes its spread. The corresponding cdf is given by
$$F_X(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right], \quad \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\mathrm{d}t.$$

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

Figure 7.2: Continuous random variables

7.2.4 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

$$\sigma_X := \mathrm{STD}[X] = \sqrt{\mathrm{VAR}[X]}.$$

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 of the pdf 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 a bit of algebra gives us

$$\text{VAR}[X] = \int_{-\infty}^{\infty} (x-\mu)^2 \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \mathrm{d}x = \sigma^2.$$

7.2.5 Pair of RVs

In probabilistic ML we deal with multiple random variables. For example, the input, the output and the weights might all be RVs. Thus we need to extend concepts from a single RV to a vector of RVs. We do this in this section by first considering a pair of RVs. Most of the concepts defined for a pair of RVs carry forward to a vector of RVs.

A pair of RVs is a mapping from the measure space, with event class $\mathscr{E}$, of the form

$$\boldsymbol{X} : \mathscr{E} \to \mathbb{R}^2,$$

where the mapping can be discrete or continuous. We will sometimes use the notation $\boldsymbol{X} = (X, Y)$. For example, we can spin the spinner twice and measure $\psi_1 \in (0, 2\pi]$, $\psi_2 \in (0, 2\pi]$. In this case, we can define the two RVs

$$X(\psi_1) = \frac{\psi_1}{2\pi}, \quad Y(\psi_2) = \frac{\psi_2}{2\pi}.$$

Events for $\boldsymbol{X}$ are sets in $\mathbb{R}^2$. To compute probability of events, we need to define the **joint cdf** $F_{XY} : \mathbb{R}^2 \to \mathbb{R}$, where

$$F_{XY}(x, y) = P[X \leq x, Y \leq y] = P[\xi \in \mathbb{S} : X(\xi) \leq x, Y(\xi) \leq y].$$

Analogous to single RVs

- Joint cdfs are non-increasing functions of x, y. In other words, for $x \geq x'$ and $y \geq y'$

$$F_{XY}(x, y) \geq F_{XY}(x', y').$$

- $\lim_{x \to -\infty} F(x, y) = 0$, $\lim_{y \to -\infty} F(x, y) = 0$, $\lim_{x,y \to \infty} F(x, y) = 1$.
- We can calculate

$$P[x_1 < X \leq x_2, y_1 < Y \leq y_2] = F_{XY}(x_2, y_2) + F_{XY}(x_1, y_1) - F_{XY}(x_1, y_2) - F_{XY}(x_2, y_1).$$

For X, Y jointly continuous, we can define the **joint pdf** as

$$f_{XY}(x, y) = \frac{\partial^2 F_{XY}(x, y)}{\partial x \partial y}$$

which enjoys the following properties

- $f_{XY}(x,y) = 0$ as $x \to \pm\infty$ or $y \to \pm\infty$.
- $F_{XY}(x,y) = \int_{-\infty}^{x}\int_{-\infty}^{y} f_{XY}(r,s)\mathrm{d}r\mathrm{d}s$.
- $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f_{XY}(r,s)\mathrm{d}r\mathrm{d}s = 1$.
- $P[x_1 < X \le x_2, y_1 < Y \le y_2] = \int_{x_1}^{x_2}\int_{y_1}^{y_2} f_{XY}(x,y)\mathrm{d}x\mathrm{d}y$.
- $P[\boldsymbol{X} \in B] = \int\int_B f_{XY}(x,y)\mathrm{d}x\mathrm{d}y$.

Let us look at some important joint random variables :

1. **Joint uniform RV:** for some region $(a,b]\times(c,d]$, the joint pdf is given by
$$f_{XY}(x,y) = \begin{cases} \frac{1}{(b-a)(d-c)} & \text{if } (x,y)\in(a,b]\times(c,d] \\ 0 & \text{otherwise} \end{cases}. \tag{7.5}$$
2. **Joint Gaussian RV:** the joint pdf is given by
$$f_{XY}(x,y) = \frac{1}{\sqrt{(2\pi)^2\det(\boldsymbol{\Sigma})}}\exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right]$$
where $\boldsymbol{x} = (x,y)$, $\boldsymbol{\mu} = (\mu_x,\mu_y)$ is the mean, and $\boldsymbol{\Sigma}$ is called the covariance matrix
$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}.$$
The covariance matrix is symmetric and positive definite.

We can define the **marginal PDF** of the RV X, which is the pdf of X assuming Y attains all possible values
$$f_X(x) = \int_{-\infty}^{\infty} f_{XY}(x,y)\mathrm{d}y.$$
Similarly, the marginal of Y is
$$f_Y(y) = \int_{-\infty}^{\infty} f_{XY}(x,y)\mathrm{d}x.$$
The RVs X and Y are said to be **independent** if $f_{XY}(x,y) = f_X(x)f_Y(y)$.

Question 7.2.1. *Show that the joint uniform RVs with joint pdf* (7.5) *are independent.*

Consider the function $g(\boldsymbol{X})$, which can be scalar-, vector-, or tensor-valued, then its expected value is given by
$$\mathbb{E}[g(\boldsymbol{X})] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(\boldsymbol{x})f_{XY}(x,y)\mathrm{d}x\mathrm{d}y$$
as long as the integral is defined. For instance:

- For $g(\boldsymbol{X}) = X$, we have $\mathbb{E}[g(\boldsymbol{X})] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x f_{XY}(x,y)\mathrm{d}x\mathrm{d}y$.
- For $g(\boldsymbol{X}) = \boldsymbol{X}$, we have a vector valued expectation $\mathbb{E}[g(\boldsymbol{X})] = [\mathbb{E}[X],\mathbb{E}[Y]]$.
- For $g(\boldsymbol{X}) = X + Y$, we have $\mathbb{E}[g(\boldsymbol{X})] = \mathbb{E}[X] + \mathbb{E}[Y]$.

The **covariance** of $\boldsymbol{X}$ is given by

$$\mathrm{COV}[\boldsymbol{X}] = \mathbb{E}[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}]) \otimes (\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])]$$

where

$$\begin{aligned}
\mathrm{COV}[\boldsymbol{X}]_{11} &= \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathrm{VAR}[X] \\
\mathrm{COV}[\boldsymbol{X}]_{22} &= \mathbb{E}[(Y - \mathbb{E}[Y])^2] = \mathrm{VAR}[Y] \\
\mathrm{COV}[\boldsymbol{X}]_{12} &= \mathrm{COV}[\boldsymbol{X}]_{21} = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])].
\end{aligned}$$

X and Y are said to be **uncorrelated** if $\mathrm{COV}[\boldsymbol{X}]_{12} = 0$. Furthermore, $\mathrm{COV}[\boldsymbol{X}]_{12} = 0$ for independent RVs. **Caution: $\mathbf{COV}[\boldsymbol{X}]_{12} = 0$ does not imply the RVs are independent!**

Finally, we are interested in looking at the pdf of Y when we know $X = \hat{x}$. A good guess would be $f_{XY}(\hat{x}, y)$. However, this need not be a pdf as it need not integrate to unity over y. This leads us to the **conditional pdf** of Y when we know $X = \hat{x}$,

$$f_{Y|X}(y|\hat{x}) = \frac{f_{XY}(\hat{x}, y)}{\int_{-\infty}^{\infty} f_{XY}(\hat{x}, y)\mathrm{d}y} = \frac{f_{XY}(\hat{x}, y)}{f_X(\hat{x})}.$$

Similarly, we can write the conditional pdf of X given $Y = \hat{y}$ as

$$f_{X|Y}(x|\hat{y}) = \frac{f_{XY}(x, \hat{y})}{f_Y(\hat{y})}.$$

We note that the extension of a regression problem to probabilistic framework leads us to determining the conditional distribution of the output given an instance of the input. This will be discussed in Section 7.4.

7.3 Unsupervised probabilistic deep learning algorithms

We begin with a vector of RVs $\boldsymbol{X}$ with N_X components with a pdf given by f_X. Let's assume that we are given a dataset of samples $\{\boldsymbol{x}_i\}$ sampled from the density f_X, which we denote by $\boldsymbol{x}_i \sim f_X$. For instance, these samples could correspond to RGB images of cars, with a resolution of 512×512. Note that this would mean that the samples would lie in a space with dimension $N_X = 512 \times 512 \times 3$, which is quite large! We can treat each pixel of the images as a RV, taking values given by the pixel intensities (across all 3 channels). Thus, these images can be seen as samples of a N_X-dimensional RV with some unknown density f_X. Also, because of the inherent structure of the objects (i.e. the cars) in these images, the various components of the multidimensional RV can be expected to be highly correlated, leading to a non-trivial form of f_X. This correlation also implies that it might be possible to reduce the dimension of $\boldsymbol{X}$ from N_X to a smaller number and thus make the representation simpler.

We are interested in using the given finite set of samples $\{\boldsymbol{x}_i\}$ to learning the density f_X of the data, and generating new samples from the learned distribution. Such methods are known as *generative algorithms*. Although a number of generative algorithms are available, we focus on a specific type of deep learning algorithm known as Generative Adversarial Networks, or GANs for short.

7.3.1 GANs

GANs were first proposed by Goodfellow et al. [6] 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 $\mathcal{S} = \{\boldsymbol{x}_i \in \Omega_X \subset \mathbb{R}^{N_X} : 1 \leq i \leq N_{\text{train}}\}$. We assume the samples are realizations of some RV $\boldsymbol{X}$ with density f_X, i.e., $\boldsymbol{x}_i \sim f_X$. We want to train a GAN to learn f_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

$$\boldsymbol{g}(.;\boldsymbol{\theta}_g) : \Omega_Z \to \Omega_X, \quad \boldsymbol{g} : \boldsymbol{z} \mapsto \boldsymbol{x} \tag{7.6}$$

where $\boldsymbol{\theta}_g$ are the trainable parameters and $\boldsymbol{z} \in \Omega_Z \subset \mathbb{R}^{N_Z}$ is the realization of a RV $\boldsymbol{Z}$ following a simple distribution, such as an uncorrelated multivariate Gaussian with density

$$f_Z(\boldsymbol{z}) = \frac{1}{\sqrt{(2\pi)^2 \det(\boldsymbol{\Sigma})}} \exp\left[-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}-\boldsymbol{\mu})\right] \quad \text{with} \quad \boldsymbol{\mu} = \boldsymbol{0}, \quad \boldsymbol{\Sigma} = \boldsymbol{I}.$$

Typically, $N_Z \ll N_X$ 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 $\boldsymbol{g}$ can be an MLP with input dimension N_Z and output dimension N_X. 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 Figure 5.9.) In either case, for a fixed $\boldsymbol{\theta}_g$, the generator $\boldsymbol{g}$ transforms the RV $\boldsymbol{Z}$ to another RV, $\boldsymbol{X}^g = \boldsymbol{g}(\boldsymbol{Z};\boldsymbol{\theta}_g)$ with density f_X^g, which corresponds to the latent density f_Z pushed-forward by $\boldsymbol{g}$. We want to choose $\boldsymbol{\theta}_g$ such that f_X^g is close to the unknown target distribution f_X.

The critic network is of the form

$$d(.;\boldsymbol{\theta}_d) : \Omega_X \to \mathbb{R} \tag{7.7}$$

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 d can be an MLP with input dimension N_X 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 Figure 5.7 but with a scalar output and without an output function.

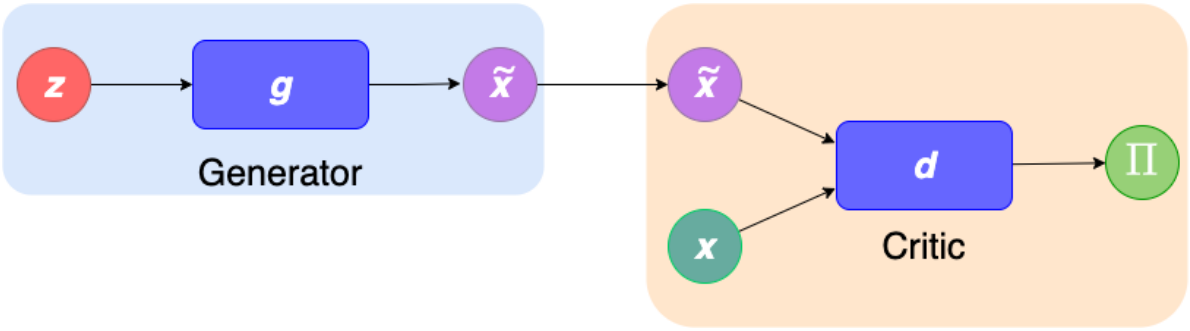


Figure 7.3: Schematic of a GAN

The schematic of the GAN along with the inputs and outputs of the sub-networks is shown in Figure 7.3. The generator and critic play adversarial roles. The critic is trained to distinguish

between true samples coming from f_X and fake samples generated by $\boldsymbol{g}$ with the density f_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_X.

We define the objective function describing a *Wasserstein GAN (WGAN)* [2], which has better robustness and convergence properties compared to the original GAN. The objective function is given by

$$\Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} d(\boldsymbol{x}_i; \boldsymbol{\theta}_d)}_{\text{critic value on real samples}} - \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} d(\boldsymbol{g}(\boldsymbol{z}_i; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d)}_{\text{critic value on fake samples}} \tag{7.8}$$

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 $\boldsymbol{g}$ 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, critic is trained to maximize objective function. In other words, we want to solve the problem

$$\boldsymbol{\theta}_d^*(\boldsymbol{\theta}_g) = \arg\max_{\boldsymbol{\theta}_d} \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) \quad \text{for any } \boldsymbol{\theta}_g. \tag{7.9}$$

Note that the optimal parameters of the critic will depend on $\boldsymbol{\theta}_g$. Now to fool the critic, the generator $\boldsymbol{g}$ tries to minimize the objective function,

$$\boldsymbol{\theta}_g^* = \arg\min_{\boldsymbol{\theta}_g} \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d^*). \tag{7.10}$$

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

$$\Pi_c(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) - \frac{\lambda}{\bar{N}} \sum_{i=1}^{\bar{N}} \left(\left\| \frac{\partial d}{\partial \hat{\boldsymbol{x}}}(\hat{\boldsymbol{x}}_i; \boldsymbol{\theta}_d) \right\| - 1 \right)^2 \tag{7.11}$$

where $\hat{\boldsymbol{x}}_i = \alpha \boldsymbol{x}_i + (1-\alpha)\boldsymbol{g}(\boldsymbol{z}_i; \boldsymbol{\theta}_g)$ and α is sampled from a uniform RV in $(0, 1)$. The additional term in (7.11) 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 1-Lipschitz function. For further details on this term, we direct the interested readers to [7].

The iterative Algorithm 1 is used train $\boldsymbol{g}$ and d 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 to 6 in practice.

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, d becomes better at detecting fake samples and $\boldsymbol{g}$ becomes better at creating samples that can fool the critic.

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

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 [2]. This is equivalent to saying

$$\mathbb{E}_Z[\ell(\boldsymbol{g}(\boldsymbol{Z}; \boldsymbol{\theta}_g^*))] \longrightarrow \mathbb{E}_X[\ell(\boldsymbol{X})], \tag{7.12}$$

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 $\boldsymbol{g}$ 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 $\boldsymbol{x} = \boldsymbol{g}(\boldsymbol{z}; \boldsymbol{\theta}_g^*)$. Furthermore, due to the weak convergence described above, the statistics (mean, variance, etc) of the generated samples will convergence 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 $\boldsymbol{g}$ 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 [2, 28]. The Wasserstein-1 distance is a popular metric used to measure discrepancies between two probability measures.*

3. *Since the dimension N_Z of the latent variable is typically much smaller than the dimension N_X 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 [21, 22].*

7.4 Supervised probabilistic deep learning algorithms

Recall the deterministic problem where given the labelled/pairwise dataset

$$\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : \boldsymbol{x}_i \in \Omega_X \subset \mathbb{R}^{N_X},\ \boldsymbol{y} \in \Omega_Y \subset \mathbb{R}^{N_Y}\}_{i=1}^{N_{\text{train}}} \tag{7.13}$$

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 RVs $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively. Further, let the paired samples in (7.13) be drawn from the unknown joint distribution f_{XY}. Then given a realization $\boldsymbol{X} = \hat{\boldsymbol{x}}$, we wish to use $\mathcal{S}$ to determine the conditional distribution $f_{Y|X}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ and generate samples from it.

There are several popular approaches to solve this probabilistic problem, such as Bayesian neural networks, variational inference, dropouts or deep Boltzman machines. But we will focus on an extension of GANs which also addresses these type of problems.

7.4.1 Conditional GANs

Conditional GANs were first proposed in [17] to learn conditional distributions. We will discuss a special variant of these models known as conditional Wasserstein GANS (cWGANs) which were developed in [1], and used to solve a number of physics-based (inverse) problems in [25].

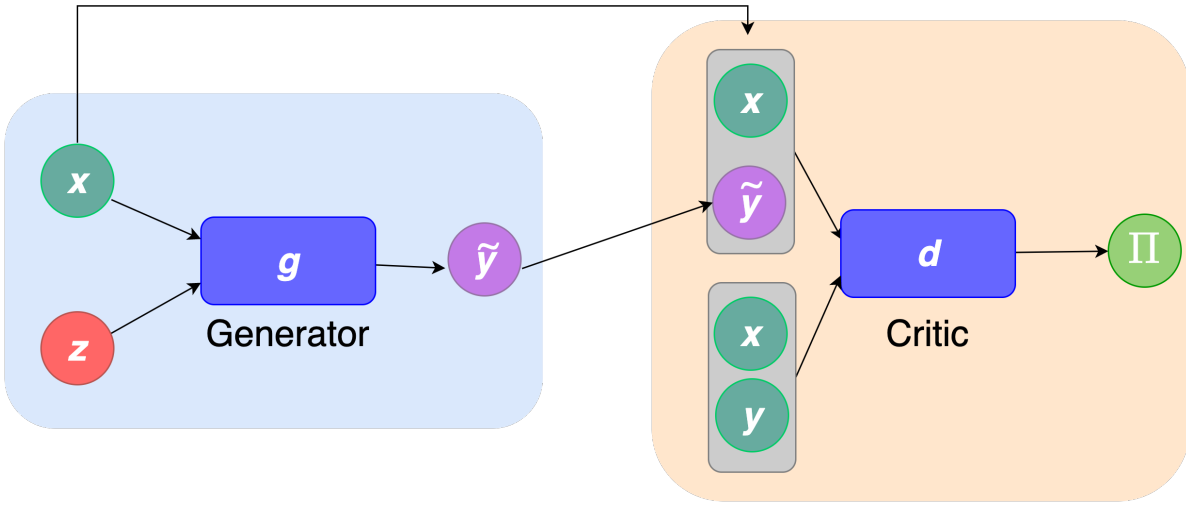


Figure 7.4: Schematic of a conditional GAN

The schematic of a conditional GAN is depicted in Figure 7.4. The generator is a network of the form

$$\boldsymbol{g}(.;\boldsymbol{\theta}_g) : \Omega_Z \times \Omega_X \to \Omega_Y, \quad \boldsymbol{g} : (\boldsymbol{z}, \boldsymbol{x}) \mapsto \boldsymbol{y} \tag{7.14}$$

where $\boldsymbol{z} \sim f_Z$ is the latent variable. Note that unlike a GAN, the generator in a conditional GAN also takes as input $\boldsymbol{x}$. For a given value of $\boldsymbol{X} = \hat{\boldsymbol{x}}$, sampling $\boldsymbol{z} \sim f_Z$ will generate many samples of $\boldsymbol{y}$ from some induced conditional distribution $f^{\boldsymbol{g}}_{Y|X}(\boldsymbol{y}|\hat{\boldsymbol{x}})$. The goal is to prescribe the parameters $\boldsymbol{\theta}_g$ such that $f^{\boldsymbol{g}}_{Y|X}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ approximates the true conditional $f_{Y|X}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ for (almost) every value of $\hat{\boldsymbol{x}}$.

The critic is a network of the form

$$d(.;\boldsymbol{\theta}_d) : \Omega_X \times \Omega_Y \to \mathbb{R} \tag{7.15}$$

which is trained to distinguish between paired samples $(\boldsymbol{x}, \boldsymbol{y})$ generated from the true joint distribution f_{XY} and the fake pairs $(\boldsymbol{x}, \hat{\boldsymbol{y}})$ where $\hat{\boldsymbol{y}}$ is generated by $\boldsymbol{g}$ given (real) $\boldsymbol{x}$.

The objective function for a cWGAN is given by

$$\Pi(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} d(\boldsymbol{x}_i, \boldsymbol{y}_i; \boldsymbol{\theta}_d)}_{\text{critic value on real pairs}} - \underbrace{\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} d(\boldsymbol{x}_i, \boldsymbol{g}(\boldsymbol{z}_i, \boldsymbol{x}_i; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d)}_{\text{critic value on fake pairs}} . \tag{7.16}$$

As earlier, the critic is trained to maximize the objective function (given by (7.9)) while the generator is trained to minimize it (given by (7.10)). Further, a stabilizating gradient penalty term needs to be included when optimizing the critic (see [25]). The generator and critic are trained using the alternating steepest descent algorithm described for GANs.

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 [1] that the generated conditional distribution $f^g_{Y|X}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ converges in a weak sense to the target condition distribution $f_{Y|X}(\boldsymbol{y}|\hat{\boldsymbol{x}})$ (on average) for a given $\boldsymbol{X} = \hat{\boldsymbol{x}}$.

Bibliography

[1] J. Adler and O. Öktem, *Deep bayesian inversion*. https://arxiv.org/abs/1811.05910, 2018.

[2] M. Arjovsky, S. Chintala, and L. Bottou, *Wasserstein generative adversarial networks*, in Proceedings of the 34th International Conference on Machine Learning, D. Precup and Y. W. Teh, eds., vol. 70 of Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia, 06–11 Aug 2017, PMLR, pp. 214–223.

[3] R. Bischof and M. Kraus, *Multi-objective loss balancing for physics-informed deep learning*. http://rgdoi.net/10.13140/RG.2.2.20057.24169, 2021.

[4] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural ordinary differential equations*. https://arxiv.org/abs/1806.07366, 2018.

[5] T. Chen and H. Chen, *Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems*, IEEE Transactions on Neural Networks, 6 (1995), pp. 911–917.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial nets*, in Advances in neural information processing systems, 2014, pp. 2672–2680.

[7] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, *Improved training of wasserstein gans*, in Advances in neural information processing systems, 2017, pp. 5767–5777.

[8] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[9] P. Kidger and T. Lyons, *Universal Approximation with Deep Narrow Networks*, in Proceedings of Thirty Third Conference on Learning Theory, J. Abernethy and S. Agarwal, eds., vol. 125 of Proceedings of Machine Learning Research, PMLR, 09–12 Jul 2020, pp. 2306–2327.

[10] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*. https://arxiv.org/abs/1412.6980v9, 2017.

[11] I. Lagaris, A. Likas, and D. Papageorgiou, *Neural-network methods for boundary value problems with irregular boundaries*, IEEE Transactions on Neural Networks, 11 (2000), pp. 1041–1049.

[12] S. Lanthaler, S. Mishra, and G. E. Karniadakis, *Error estimates for DeepONets: a deep learning framework in infinite dimensions*, Transactions of Mathematics and Its Applications, 6 (2022).

[13] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Fourier neural operator for parametric partial differential equations*. https://arxiv.org/abs/2010.08895, 2020.

[14] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, *Learning nonlinear operators via deeponet based on the universal approximation theorem of operators*, Nature Machine Intelligence, 3 (2021), pp. 218–229.

[15] A. L. Maas, A. Y. Hannun, A. Y. Ng, et al., *Rectifier nonlinearities improve neural network acoustic models*, in Proc. ICML, vol. 30, 2013.

[16] L. McClenny and U. Braga-Neto, *Self-adaptive physics-informed neural networks using a soft attention mechanism*. https://arxiv.org/abs/2009.04544, 2020.

[17] M. Mirza and S. Osindero, *Conditional generative adversarial nets*. https://arxiv.org/abs/1411.1784, 2014.

[18] S. Mishra and R. Molinaro, *Estimates on the generalization error of physics-informed neural networks for approximating PDEs*, IMA Journal of Numerical Analysis, (2022).

[19] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, *Robust stochastic approximation approach to stochastic programming*, SIAM Journal on Optimization, 19 (2009), pp. 1574–1609.

[20] D. Patel, D. Ray, M. R. A. Abdelmalik, T. J. R. Hughes, and A. A. Oberai, *Variationally mimetic operator networks*. https://arxiv.org/abs/2209.12871, 2022.

[21] D. V. Patel and A. A. Oberai, *Gan-based priors for quantifying uncertainty in supervised learning*, SIAM/ASA Journal on Uncertainty Quantification, 9 (2021), pp. 1314–1343.

[22] D. V. Patel, D. Ray, and A. A. Oberai, *Solution of physics-based bayesian inverse problems with deep generative priors*, Computer Methods in Applied Mechanics and Engineering, 400 (2022), p. 115428.

[23] A. Pinkus, *Approximation theory of the mlp model in neural networks*, Acta Numerica, 8 (1999), pp. 143–195.

[24] M. Raissi, P. Perdikaris, and G. Karniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, Journal of Computational Physics, 378 (2019), pp. 686–707.

[25] D. Ray, H. Ramaswamy, D. V. Patel, and A. A. Oberai, *The efficacy and generalizability of conditional gans for posterior inference in physics-based inverse problems*. https://arxiv.org/abs/2202.07773, 2022.

[26] O. Ronneberger, P. Fischer, and T. Brox, *U-net: Convolutional networks for biomedical image segmentation*, in Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, eds., Cham, 2015, Springer International Publishing, pp. 234–241.

[27] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, *Implicit neural representations with periodic activation functions.* https://arxiv.org/abs/2006.09661, 2020.

[28] C. Villani, *Optimal Transport: Old and New*, Grundlehren der mathematischen Wissenschaften, Springer Berlin Heidelberg, 2008.

[29] S. Wang, Y. Teng, and P. Perdikaris, *Understanding and mitigating gradient flow pathologies in physics-informed neural networks*, SIAM Journal on Scientific Computing, 43 (2021), pp. A3055–A3081.

[30] S. Wang, H. Wang, and P. Perdikaris, *Learning the solution operator of parametric partial differential equations with physics-informed deeponets*, Science Advances, 7 (2021).

[31] L. Wu, C. Ma, and W. E, *How sgd selects the global minima in over-parameterized learning: A dynamical stability perspective*, in Advances in Neural Information Processing Systems, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., vol. 31, Curran Associates, Inc., 2018.

[32] M. Yang, K. Yu, C. Zhang, Z. Li, and K. Yang, *Denseaspp for semantic segmentation in street scenes*, in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 3684–3692.

[33] D. Yarotsky and A. Zhevnerchuk, *The phase diagram of approximation rates for deep neural networks.* https://arxiv.org/abs/1906.09477, 2019.