The Mathematics of Deep Neural Networks

Raghda Wael Ezzeldin H. Aly

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Mathematics

Eastern Mediterranean University
August 2024
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Ali Hakan Ulusoy
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Masters of Science in Mathematics.

Prof. Dr. Nazim Mahmudov
Chair, Department of Mathematics

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Masters of Science in Mathematics.

Prof. Dr. Benedek Nagy
Supervisor

Examining Committee

1. Prof. Dr. Gergely Kovacs
2. Prof. Dr. Benedek Nagy
3. Assoc. Prof. Dr. Müge Saadetoğlu

ABSTRACT

Machine learning models built with deep neural networks (DNN) have gained immense popularity in recent years. However successful they might seem, there is a substantial collective blind spot when it comes to a rigorous understanding of these models, their practical limitations, and the explainability and credibility of the results they generate. Establishing a theoretical framework to ensure the robustness of deep learning algorithms is one of the most active research areas in applied mathematics. The first step along this research path is understanding how these models "think" and "reason", which -unlike how humans think- can entirely be described in mathematical terms. The aim of this thesis is to present the mathematics underlying neural network models and deep learning algorithms. It covers the mathematical foundation of artificial neurons and feed-forward neural networks and presents important mathematical results that support their expressive power and approximation capabilities. The thesis then focuses on the learning algorithm used to train these models and covers the main challenges these algorithms face. The thesis also presents the foundation of more advanced DNNs, highlighting the diversity of current architectures. Finally the thesis examines what is meant by the "black box" nature of these models and their vulnerability to adversarial attacks. By examining how these models process data and make decisions, this thesis aims to emphasize the need for rigorous scrutiny and expert involvement when developing, testing and employing DNN models.

Keywords: Machine Learning, Deep Neural Network, Learning Algorithm, Stochastic Gradient Descent, Backpropagation, Interpretability, Adversarial Attack.

ÖZ

Derin sinir ağları (DNN) ile oluşturulan makine öğrenimi modelleri son yıllarda büyük bir popülerlik kazanmıştır. Ne kadar başarılı görünseler de, bu modellerin, pratik sınırlamalarının ve ürettikleri sonuçların açıklanabilirliği ve güvenilirliğinin tam olarak anlaşılması söz konusu olduğunda önemli bir kolektif kör nokta vardır. Aslında derin öğrenme algoritmalarının sağlamlığını sağlayacak teorik bir çerçeve oluşturmak, uygulamalı matematiğin en aktif araştırma alanlarından biridir. Bu araştırma yolundaki ilk adım, insanların düşündüğünden farklı olarak tamamen matematiksel terimlerle tanımlanabilen bu modellerin nasıl "düşündüğünü" ve "akıl yürüttüğünü" anlamaktır. Bu tezin amacı derin sinir ağı modellerinin ve öğrenme algoritmalarının altında yatan matematiği sunmaktır. Yapay nöronların ve ileri beslemeli sinir ağlarının matematiksel temellerini kapsar ve bunların ifade gücünü ve yaklaşım yeteneklerini destekleyen önemli matematiksel sonuçlar sunar. Tez daha sonra bu modelleri eğitmek için kullanılan, geri yayılımı ve stokastik gradyan inişini kapsayan öğrenme algoritmasına ve bu algoritmaların karşılaştığı bazı temel zorluklara odaklanmaktadır. Tez aynı zamanda daha gelişmiş DNN modellerinin ve mimarilerinin temellerini sunmakta ve bunların bazı teorik sınırlamalarını tartışmaktadır. Bu tez, bu modellerin verileri nasıl işlediğini ve karar verdiğini inceleyerek, DNN modellerini geliştirirken, test ederken ve kullanırken dikkatli inceleme ve uzman katılımı ihtiyacını vurgulamayı amaçlamaktadır.

Anahtar Kelimeler: Makine Öğrenimi, Derin Sinir Ağı, Öğrenme Algoritması, Stokastik Gradyan İnişi, Geriye Yayılım.

DEDICATION

... Dedicated to Dede, my grandfather in heaven. You always said I would become a great writer, scientist and artist. I haven't gotten there yet, but I think I am on my way, thanks to you, to the stories you told me, the worlds you opened my eyes to, and more importantly to the intelligent, curious, critical thinker you raised, your daughter, my mother, who in her turn continued to follow your path, teaching her children to always thrive for knowledge and pursue the truth, no matter how hard it gets.

ACKNOWLEDGMENTS

I want to express my deepest gratitude towards my research partner and dear friend Mohamed O. H. Mahmoud. Your dedication to our work, the speed with which you learnt the computational and mathematical aspects that our research needed and the way you kept and keep pushing me forward is a huge gift.

And of course, I want to express my gratitude to my dear supervisor, Prof. Dr. Benedek Nagy, who helped me expand my knowledge in the computer science field and tough me to always question the validity, credibility and usefulness of what many others have taken for granted.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
FNN	Feed-forward Neural Network
MLP	Multi-layer Perceptron
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent

Chapter 1

INTRODUCTION

Exploring artificial intelligence requires exploring and refining our understanding of the core principles that define intelligence. When assessing human intelligence, several theories and tests have been developed, however when it comes to machines, where the "reasoning" process can entirely be described mathematically, it suffices to examine the mathematics behind the scenes to get a sense of how intelligent these machines really are.

A simple electronic calculators for instance has circuits which contain instructions for addition/subtraction and multiplication/division. With these permanent, hard-coded instructions it can compute with an accuracy and speed that by far surpasses the capacity of an average human being. Indeed, some tasks which are "among the most difficult mental undertakings for a human being are among the easiest for a computer." [17]

However, the human brain surpasses the capabilities of today's most advanced machines in areas of intuitive thinking and abstraction. "A person's everyday life requires an immense amount of knowledge about the world. Computers need to capture this same knowledge in order to behave in an intelligent way. One of the key challenges in artificial intelligence is how to get this informal knowledge into a computer" [17]. Since it is impractical and even impossible to hard-code all the knowledge, formal and informal, that guides our reasoning process, the main idea

behind Deep Neural Networks—or Deep Learning—is to mimic the biological structure of the brain, so that nothing needs to be hard-coded.

Despite their recent widespread popularity, a fundamental mathematical understanding of DNN models is still in a preliminary state. Echoing Ali Rahimi's sentiment, "Machine learning has become a form of alchemy." There is indeed a substantial collective blind spot when it comes to a rigorous understanding of these models, their practical limitations, and the explainability and credibility of the results they generate. In many fields where these models are applied, such as self-driving cars, medical diagnosis, and robotics, we definitely require a significant level of control and predictability. "The development of a theoretical foundation to guarantee the success of these algorithms constitutes one of the most active and exciting research topics in applied mathematics" [20].

Objective: The objective of this thesis is to provide a theoretical analysis of the mathematics at the base of DNNs. By systematically examining both the expressive power and the mathematical principles governing DNNs, this thesis aims to offer insights into their capabilities and limitations. This understanding is crucial for advancing the field of machine learning and for informing the development of more robust and efficient neural network models.

Scope: This thesis is divided into several chapters, each addressing a specific aspect of the mathematics of DNNs:

1. **Mathematical Foundations of Feed-Forward Neural Networks**: This chapter starts with the biological inspiration behind artificial neurons, and provides a historical background including key milestones in the field. It then covers the essential mathematical foundations of FNN, beginning with the model of a single artificial neuron.

2. **Expressive Power and Approximation of FNN**: This chapter explores the theoretical limitations of single perceptron models and methods to overcome these limitations. It then presents the Universal Approximation Theorem and the Kolmogorov-Arnold Superposition Theorem, discussing their implications on neural networks.

3. **Learning Algorithms: Challenges and Limitations**: This chapter introduces supervised, unsupervised and semi-supervised learning. It provides an analysis of supervised learning algorithms, including gradient descent, backpropagation, and stochastic gradient descent, along with challenges such as overfitting, underfitting, vanishing and exploding gradients.

4. **Advanced Neural Networks**: This chapter explores Convolutional Neural Networks (CNNs) and their foundational components, convolutional cells. It also discusses recurrent cells, which form the core of Recurrent Neural Networks (RNNs). Additionally, the chapter provides an overview of Transformer models and discusses their primary critiques.

5. **Interpretability and Robustness**: The final chapter discusses what is meant by the "black box" nature of these models and the implications of such lack of transparency. It also sheds light on the vulnerability of these models to adversarial attacks.

By addressing the mathematics of DNNs, this thesis aims to contribute to the development of reliable machine learning models. The mathematics used in current DNN models is not particularly complex, and one might wonder whether, just like complex numbers were developed for solving problems that would otherwise be impossible to solve, a new field of mathematics could emerge to provide us with stronger tools that can more rigorously explain how the human brain runs the most complex software that is the human mind.

Chapter 2

MATHEMATICAL FOUNDATIONS OF FEED-FORWARD NEURAL NETWORKS

In this chapter, we cover the essential mathematical foundations of feed-forward neural networks, beginning with the biological inspiration behind artificial neurons, and a historical background of neural network development. Then, we delve into the mathematical model of a perceptron and examine various activation functions. Finally, we present a comprehensive understanding of FNNs, including their structure, notation, and what is meant by model parameters. This foundation sets the base allowing us to explore in the subsequent chapter, the expressive and approximation power of neural networks .

2.1 Mathematical Model of an Artificial Neuron (Perceptron)

2.1.1 Biological Inspiration

In the human brain, there exist over 100 billion neurons [54], each forming connections with other neurons through synapses, resulting in more than 100 trillion neural connections [24]. This complexity is astonishingly comparable to over a thousand times the number of stars in our galaxy.

Simply put, "a neuron is a cell in the nervous system whose function it is to receive and transmit information." [54] They receive electrical signals from other cells via their dendrites and when the signal is strong enough, passing a certain level or threshold, the neuron fires and transmits an electrical signal to other neurons it is connected to. (see

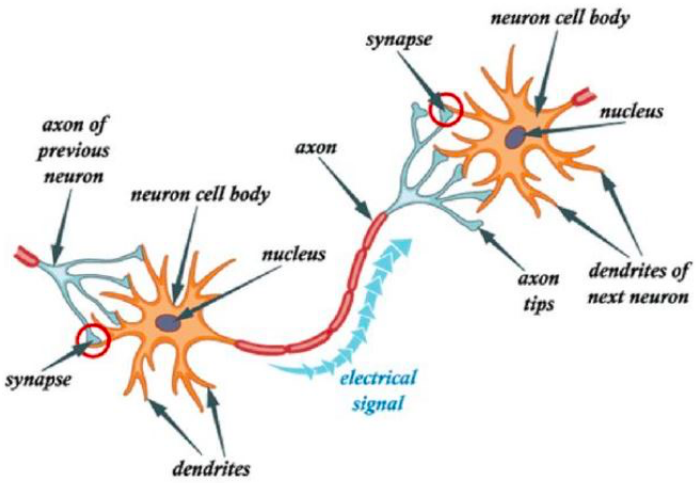

Figure 2.1: Structure of a Biological Neuron [26]

Figure 4.1). "Some neurons have hundreds or even thousands of dendrites, and these dendrites may themselves be branched to allow the cell to receive information from thousands of other cells." [54]

An important thing to note is that a biological neuron operates in an all or nothing manner, meaning, it either fires completely, or does not fire at all. We will later see that this is not the case for most artificial neurons, except the ones that have the Heaviside function as their activation functions, such as in the example we will present later. First we give a historical background for the development of the field of deep learning.

2.1.2 Historical Background

The concept of the first artificial neuron was introduced by McCulloch and Pitts in 1943 [47]. Their model was simply intended to explain how biological neuron work. The first multi-layered network that underwent training with artificial neurons was introduced by Rosenblatt in 1958 [20]. Since that time, numerous, more advanced neural network architectures and training algorithms have been developed.

In 1974, the backpropagation algorithm was developed by Paul Werbos, which

significantly advanced the training of multi-layered neural networks as it allowed for efficient computaion of gradients [60]. This was further popularized in 1986 by Rumelhart, Hinton, and Williams [46].

In 1985, Geoffrey Hinton and Terry Sejnowski introduced a type of stochastic recurrent neural network called Boltzmann Machines, capable of learning deep representations of data [28].

The year 1986 saw the introduction of Multilayer Perceptrons MLPs and Recurrent Neural Networks (RNNs), which allowed for the modeling of sequential data and being able to capture temporal dependencies [46]. In 2012, the concept of Dropout was introduced by Geoffrey Hinton and his team, providing a technique that prevents overfitting by randomly dropping some units during training [53].

In 2014, Ian Goodfellow and his colleagues proposed Generative Adversarial Networks (GANs), a novel approach to generative modeling using two neural networks competing against each other to produce realistic data [17].

The introduction of architectures like the Transformer in 2017 by Vaswani et al. [58] was a game-changer for natural language processing (NLP). It allowed models to become more efficient at handling sequential data, leading to the creation of large-scale language models like BERT and GPT.

Building on this foundation of advanced neural network architectures, it is essential to understand the basics of how individual components, such as neurons, operate within these models. In the next section we will present an introductory example to illustrate

how a single artificial neuron operates, how it processes input signals and when and how it "fires"- produces an output.

2.1.3 Introductory Example: Single Artificial Neuron

Suppose we have a single artificial neuron (or perceptron) that receives three different signals, x_1, x_2, x_3, each corresponding to the grades of a student's exams in a given course. We want the neuron to fire only if the student has passed the course. **Input**

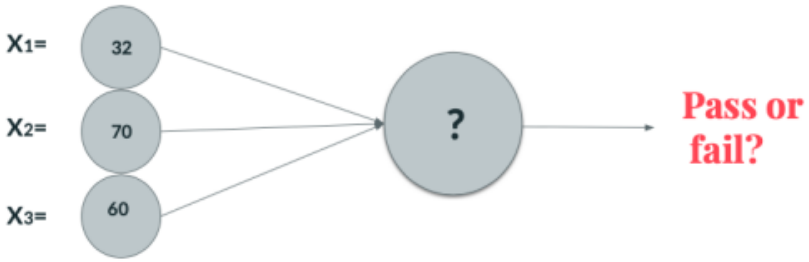

Figure 2.2: Single Artificial Neuron or Perceptron

and Output: The neuron should process this input, which in this case is a 3x1 input vector, containing the grades x_1, x_2, x_3 and output either a 1 (pass) or 0 (fail). It is not hard to see the computation that this artificial neuron should do.

Weights and Bias: To determine if the student has passed we need to know the weights of each exam as well as the passing grade. Suppose the weights for x_1, x_2, x_3 are w_1, w_2, w_3 respectively, the first thing to do is to compute the weighted sum.

$$\sum_{i=1}^{3} x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3$$

Whether the result is a pass or a fail depends on the passing grade. To incorporate that, we can add a constant called a bias b and set it's value to the opposite of the passing grade and evaluate the result. In general, the bias can be thought of as the threshold, after which the neuron should fire.

$$\sum_{i=1}^{3} x_i w_i - b = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

Activation using Heaviside Function: Now as a final step, this neuron should be activated (output a 1) if the result is a positive number and output 0 otherwise. Very simply, we plug our result in an activation function that does precisely that.

Definition 2.1 (Heaviside Function): The heaviside function or unit step function $H(x)$ is defined as:

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Domain: all real numbers, i.e., $(-\infty, \infty)$. **Range:** the set $\{0, 1\}$.

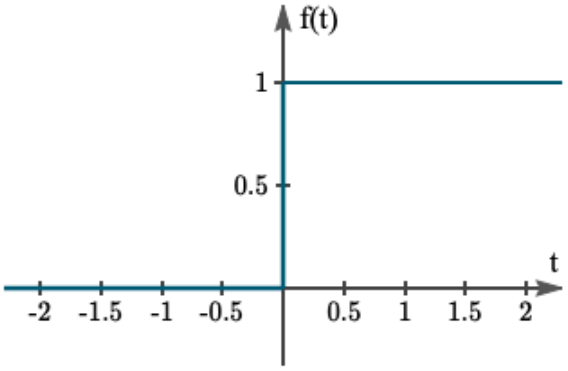

Figure 2.3: Heaviside Function (Unit Step Function)

With this function used for activation, our artificial neuron (or perceptron) is now ready to process the grades of students, firing only if the student has passed.

$$\hat{y} = H(\sum_{i=1}^{3} x_i w_i + b) = H(x_1 w_1 + x_2 w_2 + x_3 w_3 + b)$$

The Heaviside function is only one of a multitude of activation functions which can be

used in DNNs. Even though the Heaviside function makes artificial neurons behave like biological neurons (either fire or not fire) it is actually not a widely used activation function, especially in advanced models. In a later section, we will explore more functions that are popularly used today.

2.1.4 Definition of an Artificial Neuron

An artificial neuron receives information from the layer before it, processes the information and produces an output (or not) which in its turn is fed to other neurons in the subsequent layer.

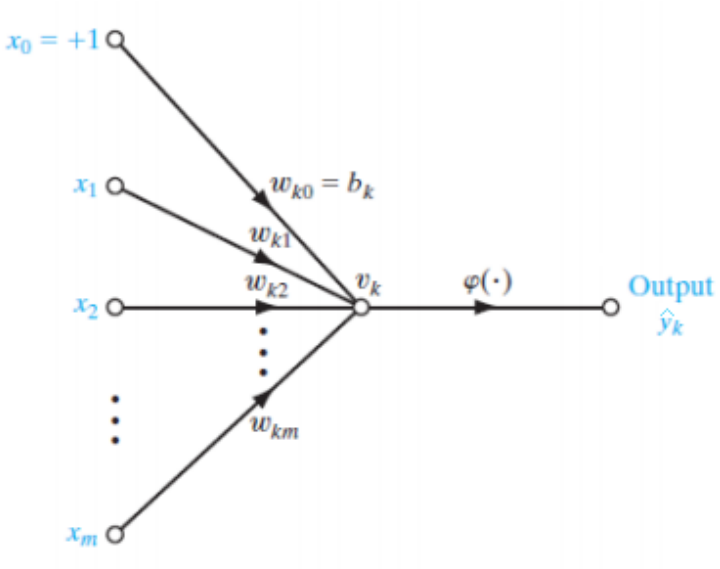

Figure 2.4: Signal-flow Graph Representation of an Artificial Neuron

- $x_i, 1 \leq i \leq m$ are the input signals,
- $x_0 = 1$ is the bias coefficient,
- $w_{ki}, 1 \leq i \leq m = (w_{k1}, w_{k2}, \ldots, w_{km}) \in \mathbb{R}^m$ are the synaptic weights of the k^{th} neuron,
- $w_{k0} = b_k \in \mathbb{R}$ is the offset (bias) of the k^{th} neuron,
- $v_k \in \mathbb{R}$ is a linear combination of the input and the bias vector,

- $\phi : \mathbb{R} \to \mathbb{R}$ is the activation function, and
- $\hat{y}_k \in \mathbb{R}$ is the model's output.

Rules of the signal flow graph: [57]

1. The signal can only go in one direction, following the arrows.

2. There are two types of edges: synaptic edges, where the relationship between input and output is linear, and activation edges, where there is a non-linear relationship between input and output.

3. The signal at a vertex is equal to the signals arriving at this vertex.

4. Edges emanating from the same vertex transmit the same signal.

Definition 2.2 (Artificial Neuron): An artificial neuron [34] connected with the input vector $[x_1, x_2, \ldots, x_m]$, with weights w_1, w_2, ..., $w_m \in \mathbb{R}$, bias $b \in \mathbb{R}$, and activation function $\phi : \mathbb{R} \to \mathbb{R}$ is a function $f : \mathbb{R}^m \to \mathbb{R}$ given by: $f(x_1, x_2, \ldots, x_m) = \phi\left(\sum_{i=1}^{m} x_i w_i + b\right)$

The values of v_k and $\hat{y}_k$ are obtained using the following formulas:

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k = \mathbf{w}_k^T \mathbf{x} + b_k$$

$$\hat{y}_k = \phi(v_k) = \phi(\mathbf{w}_k^T \mathbf{x} + b_k)$$

- $x^T = (x_1, x_2, \ldots, x_m) \in \mathbb{R}^m$ are the input signals,
- $w_k^T = (w_{k1}, w_{k2}, \ldots, w_{km}) \in \mathbb{R}^m$ are the synaptic weights,
- $b_k \in \mathbb{R}$ is the offset (bias),

To include b_k in the summation (like in the signal-flow diagram 2.4), we simple add an

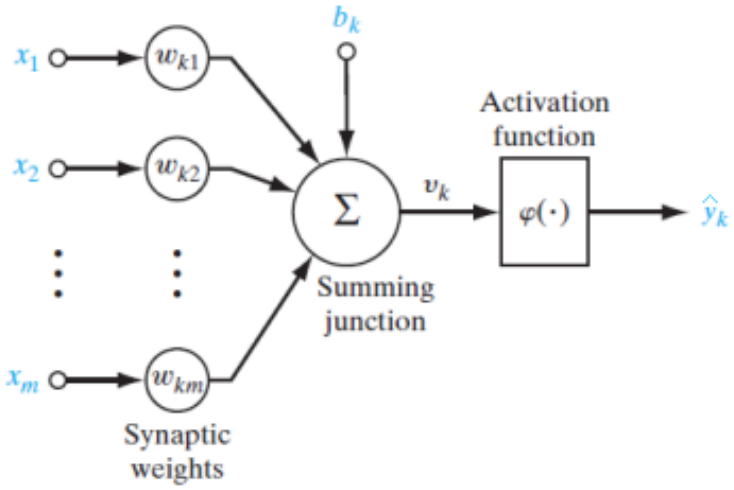

Figure 2.5: The K-block Diagram of a Neuron [57]

element x_0 to the input vector and initialize its value to 1. This will give us an $m+1$ dimensional vector $x^* = [x_0, x_1, x_2, ..., x_m]$. We now also have to add an element w_{k0} to our new weight vector w^* and set $w_{k0} = b_k$. This way we can combine b_k in the summation when calculating v_k:

$$v_k = \sum_{j=0}^{m} w_{kj}^* x_j^* = (w_k^*)^T x^*$$

- $(x^*)^T = (x_0, x_1, x_2, \ldots, x_m) \in \mathbb{R}^{m+1}$ and $x_0 = 1$
- $(w_k^*)^T = (w_{k0}, w_{k1}, w_{k2}, \ldots, w_{km}) \in \mathbb{R}^{m+1}$ and $w_{k0} = b_k$

2.1.5 Activation Functions

There are many types of activation functions used in DNN. Depending on the task at hand, some are more suitable than others. Currently, among the widely-used activation functions is the rectified linear unit (ReLU), the Sigmoid function and the Hyperbolic Tangent (Tanh) [34]. We will define these functions below and also state their derivatives as they are used in the process of backpropagation which we will explore in a later section.

1. Rectified linear unit (ReLU):

$$\phi(v) = \max(0, v), \quad \frac{d\phi(v)}{dv} = \begin{cases} 1, & \text{if } v > 0 \\ 0, & \text{if } v \leq 0 \end{cases}$$

   ReLU is by far the most extensively used activation function because its simple piecewise linear structure allows for efficient computation, making the training process faster compared to more complex functions. It also mitigates the vanishing gradient problem (which we will examine later) by enabling gradients to flow more effectively during backpropagation, which is crucial for learning deeper networks.

2. Sigmoid function:

$$\phi(v) = \frac{1}{1+e^{-v}}, \quad \frac{d\phi(v)}{dv} = \frac{e^{-v}}{(1+e^{-v})^2}$$

   The sigmoid function is ideal for models that are used to predict probabilities, as its output ranges between 0 and 1. In addition, its derivative offers a smooth gradient that avoids abrupt changes in output values.

3. Hyperbolic tangent (tanh) function:

$$\phi(v) = \tanh(v), \quad \frac{d\phi(v)}{dv} = 1 - \tanh^2(v)$$

   The Hyperbolic Tangent (tanh) function is commonly used in recurrent neural networks and in the hidden layers of feed-forward neural networks. It is particularly favored for its ability to center the data, making the mean of the activations closer to zero, which can be useful in the training process.

In addition to the previous functions, "there exists a zoo of activation functions" [34] which include -but are not limited to- the functions list below:

1. Identity function:

$$\phi(v) = v, \quad \frac{d\phi(v)}{dv} = 1$$

The identity function is commonly used in linear regression models and in the output layer of neural networks for regression tasks. It is also used when it is required for the output to be the same as the input signal.

2. ArcTan function:

$$\phi(v) = \arctan(v), \quad \frac{d\phi(v)}{dv} = \frac{1}{1+v^2}$$

3. Softsign function:

$$\phi(v) = \frac{v}{1+|v|}, \quad \frac{d\phi(v)}{dv} = \frac{1}{(1+|v|)^2}$$

4. Inverse square root unit (ISRU) function:

$$\phi(v) = \frac{v}{\sqrt{1+\alpha v^2}}, \quad \frac{d\phi(v)}{dv} = \frac{1}{(1+\alpha v^2)^{\frac{3}{2}}}$$

5. Softplus function:

$$\phi(v) = \log(1+e^v), \quad \frac{d\phi(v)}{dv} = \frac{e^v}{1+e^v}$$

6. Bent Identity function:

$$\phi(v) = \frac{\sqrt{v^2+1}-1}{2} + v, \quad \frac{d\phi(v)}{dv} = \frac{v}{2\sqrt{v^2+1}} + 1$$

7. Sinusoid function:

$$\phi(v) = \sin(v), \quad \frac{d\phi(v)}{dv} = \cos(v)$$

8. Gaussian function:

$$\phi(v) = e^{-v^2}, \quad \frac{d\phi(v)}{dv} = -2ve^{-v^2}$$

9. Leaky rectified linear unit (Leaky ReLU):

$$\phi(v) = \begin{cases} v, & \text{if } v > 0 \\ \alpha v, & \text{if } v \leq 0 \end{cases}, \quad \frac{d\phi(v)}{dv} = \begin{cases} 1, & \text{if } v > 0 \\ \alpha, & \text{if } v \leq 0 \end{cases}$$

where α is a small positive constant.

Deciding which activation function to use often depends on the nature of the task, the data and the architecture of the network. Some functions are more robust against certain problem faced during training. For instance ReLU activation function can mitigate the vanishing gradient problem [42], which we will later present (Chapter 4). With a clear understanding of the mathematical model of a perceptrons, we have grasped the building blocks of artificial neurons and we are ready to explore the mathematics needed to build deep neural networks.

2.2 Feed Forward Neural Networks (FNN)

The natural brain is inherently able to do parallel processing of information and operate in a non-linear fashion. The purpose of deep neural networks is to mimic this process. These models have the advantage of being able to conduct parallel processing, and use the data to get better at generalizing and predicting, which is described as the process of "learning."

An artificial neural network (ANN) is comparable to the human brain with regards to the following:

1. The network structure allows the model to acquire knowledge inherent in the information through the learning process.

2. It uses the strengths of the connections between neurons (synaptic weights) to store the acquired information and knowledge. [57] A neural network can be represented by a graph consisting of vertices (neurons) and directed edges between them (see

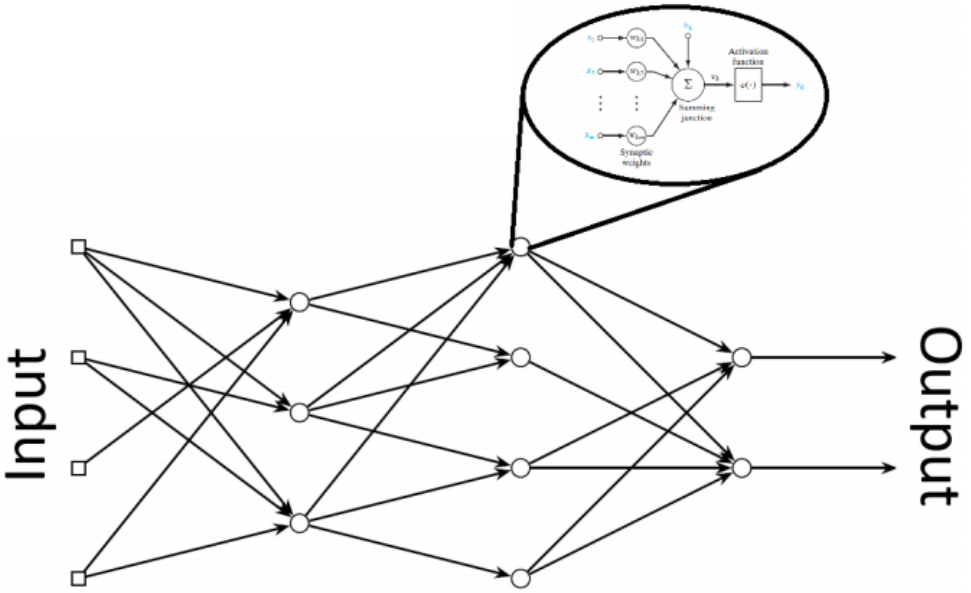

Figure 2.6: Feed-forwardward Neural Network [57]

figure 2.6), where:

1. Each neuron (vertex) is connected to the previous layer with linear synaptic edges, and to the subsequent layer through an activation edge.

2. The activation potential of the neuron is defined as the weighted sum of the input signals.

3. The activation edge takes as input the activation potential and inserts it in an activation function to produce an output.

Feed Forward Neural Networks are the simplest form of neural network architecture where artificial neurons are connected and organized into layers. As the name suggests, in Feed Forward models, information flows strictly in one direction – from input to output – with no feedback or loops, unlike other architectures which we will examine in later sections.

2.2.1 Terminology

- **Perceptron**, in its original formulation, typically has a single output node which conducts binary operations (see Introductory Example 2.1.3)
- **Single-layer Perceptron (SLP)** extends the concept of the basic perceptron to allow for multiple output neurons. It can be used for both binary and multi-class classification tasks.
- **Fully Connected Neural Network (FC)** is a network where every neuron in one layer is connected to all neurons in the next layer.
- **Multilayer Perceptron (MLP)** is a fully connected network that consists of at least two layers (excluding the input layer).
- **Deep Neural Network DNN** has many hidden layers between the input and output layers. The term "deep" refers to the number of layers through which the data passes. An MLP with only one hidden layer is not a DNN.

Figure 2.7: Two-layer MLP Processing a 2-Feature Input Vector: 1 Hidden Layer with 2 Neurons (green), and 1 Output Layer with 1 Neurons (red)

After introducing more notations, we will present the formal mathematical definition of neural networks NN.

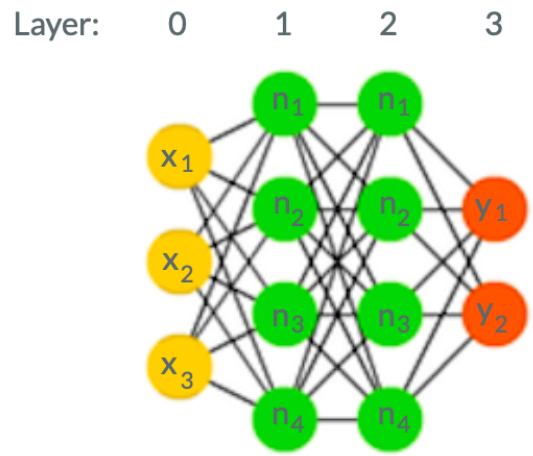

Figure 2.8: Three-layer MLP or DNN with Two Hidden Layers

2.2.2 Notation

Within a NN model, vectors are used to represent each layer's input and output vectors as well as biases, while matrices are use to represent a single layer's weights.

x: An input with m features is represented by an input vector **x** with m dimensions (or elements), each element corresponding to an input feature:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

where x_j is the j-th input feature.

y: The actual output of a given input (the values we want out model to learn) is represented either by a scalar (if the output is single valued) or by a vector y with n dimensions (or elements):

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

where y_i is the value of the i-th row of the actual output vector.

$\mathbf{W}^{(\ell)}$: The weight of the ℓ^{th} layer of a neural network with n neurons in the output layer and m neurons in the input layer is giver by the $n \times m$ dimensions matrix $\mathbf{W}^{(\ell)}$:

$$\mathbf{W}^{(\ell)} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{bmatrix}$$

where w_{ij} is the weight connecting the j-th neuron in the input layer to the i-th neuron in the output layer: for instance, row 1 represents the weights associated with the first output neuron, while column 1 represents the weights from all neurons associated with the first input value from the previous layer.

$\mathbf{b}^{(\ell)}$: The bias vector associated with the ℓ-th layer which gets added to the weighted sum. It is represented with a vector $b^{(\ell)}$ with n dimensions, where each element b_i in the bias vector corresponds to a bias for the i-th neuron in the output layer.

$$\mathbf{b}^{(\ell)} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$\mathbf{v}^{(\ell)}$: The result of the matrix multiplication of weights $\mathbf{W}^{(\ell)}$ and input $\mathbf{x}$ plus the bias $\mathbf{b}^{(\ell)}$. It represents the linear combination of inputs, weights, and biases before

applying the activation function in the ℓ-th layer. For a layer with n neurons, $\mathbf{v}^{(\ell)}$ can be represented as:

$$\mathbf{v}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{x} + \mathbf{b}^{(\ell)}$$

where $\mathbf{v}^{(\ell)}$ is:

$$\mathbf{v}^{(\ell)} = \begin{bmatrix} v_1^{(\ell)} \\ v_2^{(\ell)} \\ \vdots \\ v_n^{(\ell)} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} w_{1i}^{(\ell)} x_i + b_1^{(\ell)} \\ \sum_{i=1}^{m} w_{2i}^{(\ell)} x_i + b_2^{(\ell)} \\ \vdots \\ \sum_{i=1}^{m} w_{ni}^{(\ell)} x_i + b_n^{(\ell)} \end{bmatrix}$$

Where n: Represents the index of the neuron within the ℓ-th layer of the network.

$\phi^{(\ell)}$: The activation function applied to $\mathbf{v}^{(\ell)}$.

$a_n^{(\ell)}$: The output of the n-th neuron in the ℓ-th layer after applying the activation function to the weighted sum of its inputs from the previous layer.

y vs. $\hat{\mathbf{y}}$: y denotes the actual value of the output that corresponds to a given input vector (the correct value that we want our model to learn to predict) while $\hat{y}$ denotes the final model's output, which is the model's prediction of the actual value y.

$$\hat{\mathbf{y}} = \phi(v) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

By understanding and using this notation, we can clearly describe the operations and transformations that occur within a neural network.

2.2.3 Definition of a Neural Network

Definition 2.3 (Neural Network): "Let $d \in \mathbb{N}$ be the dimension of the input layer, L the number of layers, $N_0 := d$, N_ℓ, $\ell = 1, \ldots, L$, the dimensions of the hidden and last layer, $\phi : \mathbb{R} \to \mathbb{R}$ a (non-linear) activation function, and, for $\ell = 1, \ldots, L$, let T_ℓ be the affine-linear functions:

$$T_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}, \quad T_\ell x = W^{(\ell)} x + b^{(\ell)},$$

with $W^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ being the weight matrices and $b^{(\ell)} \in \mathbb{R}^{N_\ell}$ the bias vectors of the ℓ-th layer. Then:

$$f : \mathbb{R}^d \to \mathbb{R}^{N_L}, \quad f(x) = T_L \phi(T_{L-1} \phi(\dots \phi(T_1(x)))), \quad x \in \mathbb{R}^d,$$

is called a *(deep) neural network* of *depth L*" [34] .

2.2.4 Model Parameters

The main types of model parameters are weights, biases, and, in some cases, additional parameters specific to certain types of layers or architectures.

Definition 2.4 (Weights): Weights are the core parameters in a neural network. They are the coefficients applied to the input features as they pass through the artificial neurons of the network. Each connection between neurons in adjacent layers has an associated weight. The set of all weights in the network determines how input data is transformed at each layer.

Definition 2.5 (Biases): Biases are additional parameters added to the weighted sum of inputs before applying the activation function. This process allows the activation function to be shifted either to the left or to the right.

There are more parameters which will be discussed in later sections. Overall, these parameters are essential components of neural networks, determining how inputs are transformed and how the network learns from data.

In a **fully connected FC** neural network, every neuron (N_ℓ neurons) in the ℓ^{th} layer is connected to $N_{\ell-1}$ neurons, corresponding to the output from the previous layer. The number of weights in the layer ℓ is $N_\ell N_{\ell-1}$. Additionally each neuron in the layer has

a bias. Thus the total number of parameters is given by: $P(N) = \sum_{\ell=1}^{L}(N_\ell N_{\ell-1} + N_\ell)$

In the introductory example of this section, our pass or fail perception we could easily find the values of the weights (weight of each exam) and bias (passing grade), because we could interpret their role clearly in relation to the task. This is almost never the case in larger neural network models where the number of parameters can be in the billions and the "correct" or "best" weights and biases that describe the true relationship between each input feature and the final output of the model is not known. In fact, what is meant by training the model is letting it adjust its parameters so that it can better capture the relationship between input and output (in the case of supervised learning) or discover patterns and structure in the data (unsupervised learning).

Initialization of Weights and Bias

Before we start the training process, we have to initialize the weights and biases to some values. In fact, there is no solid theoretical foundation on what the values of the weights and biases should be set to before the training process. Weights and biases are usually initialized randomly and are often drawn from random distributions, like uniform or normal distributions, whereas biases are typically initialized to zero or small random values [35]. Later in Chapter 4, we will explore some results on weights and bias initialization that optimize the learning process.

2.2.5 Model Hyperparameters

Hyperparameters are the parameters that are decided upon prior to the start of the learning process. These parameters control the behavior of the training algorithm. Unlike model parameters (e.g., weights), which are learned during training, hyperparameters must be specified before training and can significantly affect how the model performs [17].

- **Number of Layers**: this refers to the number of hidden layers.
- **Number of Neurons per Layer**: This specifies the number of neurons in each hidden layer.
- **Activation Functions**: The functions used to introduce non-linearity into the network, such as ReLU, sigmoid, or tanh.
- **Kernel Size**: In CNNs, this refers to the size of the filter used in convolutional layers. (We will explore CNNs in a later chapter.)
- **Dropout Rate**: Used to prevent overfitting, this is the rate of neurons that are dropped during training [53].

There is no definitive theoretical foundation that prescribes the exact number of layers and neurons, the best activation functions, kernel sizes, and dropout rates for every neural network task. However, there are empirical studies that provide some insights into choosing these hyperparameters. Generally, researchers rely on experimentation, domain knowledge, and recent advancements in hyperparameter optimization to determine a suitable of hypermarameters. For instance, the dropout rate, introduced by Srivastava et al., is typically chosen based on empirical performance, with common values ranging from 0.2 to 0.5 [53].

With a clear understanding of artificial neural networks, their parameters and hyperparameters, we are ready to explore their expressive and approximation power.

Chapter 3

EXPRESSIVE POWER AND APPROXIMATION OF FNN

We will start this chapter by highlighting the limitations of a single perceptron model and examine some techniques that expand its modelling capacity. We will then present two powerful results: the Universal Approximation Theory and Kolomogrov Superposition Theory. These theories show that advanced neural network architectures with hidden layers, such as MLPs, are able to approximate more complex, non-linear functions.

3.1 Limitations of Perceptrons

In the introductory example presented in Chapter 2, a single perceptron model could perfectly capture the relationship between the grades (three-dimensional input vector), and the difference between their weighted average and the passing grade. This is because the relationship between them is linear, and the perceptron architecture allows it to create linear decision boundaries (planes in three-dimensional space and hyper-planes in higher dimensions). When the decision boudary is a line (or hyperplane) we say that the function is perceptron-computable. However, when the decision boundary is non-linear, which is very common in the real world, a single perceptron is unable to separate the data points.

Example 1: Non-linear Models Can a perceptron accurately model the following non-linear relationship?

$$y = 3x_1^2 x_2^3 x_3 \tag{3.1}$$

Answer: Using a linear mapping of the input vector, a single perceptron cannot model this relationship.

Example 2: Spherical Decision Boundary Is it possible for a perceptron to classify the data points described by the following relationship?

$$y = \begin{cases} 1 & \text{if } x_1^2 + x_2^2 + x_3^2 > 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

Answer: No, because the decision boundary defined by $x_1^2 + x_2^2 + x_3^2 = 1$ is a sphere, not a hyper-plane.

Example 3: Boolean Functions Can a perceptron model the boolean functions AND, OR and XOR? Let's examine each function separately to determine it's decision boundary. The AND operation yields TRUE (1) when all inputs are True. These

x_1	x_2	x_1 **AND** x_2
1	1	1
1	0	0
0	1	0
0	0	0

Figure 3.1: Truth Table of AND Function of Two Variables

results can be graphically visualized, with the red dots representing 0's and green dots representing 1's as follows: **Answer:** Yes, a perceptron can model the AND function [8]. The graph above illustrates a line that correctly separates these two

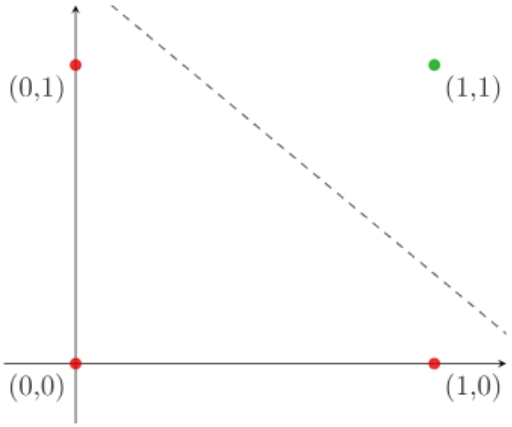

Figure 3.2: AND Function Graph

output states. This line represents a hyperplane in $\mathbb{R}^2$, and there are infinitely many solutions that can correctly represent the AND function.

Now let's take a look at the OR function of two variables:

x_1	x_2	x_1 **OR** x_2
1	1	1
1	0	1
0	1	1
0	0	0

Figure 3.3: Truth Table of OR Function of Two Variables

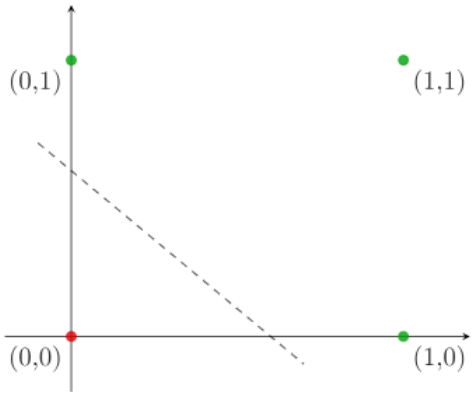

Figure 3.4: OR Function Graph

Answer: The OR function can also be accurately modeled using a perceptron since a line can separate the two output states [8].

Finally let's take a look at the XOR function of two variables:

x_1	x_2	x_1 **XOR** x_2
1	1	0
1	0	1
0	1	1
0	0	0

Figure 3.5: Truth Table of XOR Function of Two Variables

Below is the graphical representation of the XOR truth table:

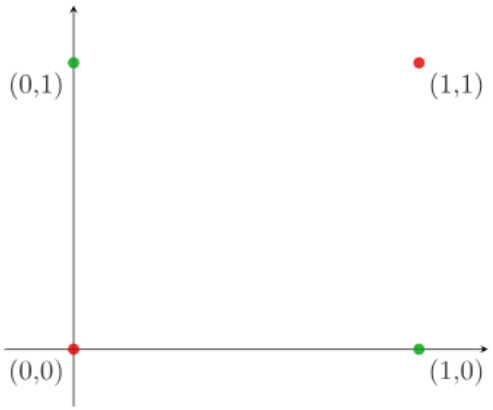

Figure 3.6: XOR Function Graph

Answer: A perceptron cannot draw a line to separate the two states. This demonstrates that the XOR classification problem is not linearly separable.

Examples 1, 2 and 3 illustrate a fundamental limitation of perceptrons: their inability to handle non-linear separations in n-dimensional space.

3.2 Overcoming Perceptron Limitations

3.2.1 Input Vector Transformation

In Example 1 (3.1) the function we want to model is a product of polynomials, which a perception is unable to model, however if we apply a vector transformation, combined with the correct activation function, a perceptron can theoretically model this relationship with 100% accuracy as follows:

Given the input vector:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

We can apply the natural logarithm function $\ln : (0, \infty) \to \mathbb{R}$ to the input vector $\mathbf{x}$ to get:

$$\mathbf{x}^* = \begin{bmatrix} \ln(x_1) \\ \ln(x_2) \\ \ln(x_3) \end{bmatrix}$$

This new input will be processed by the neuron and we will get:

$$\mathbf{v} = w_1 \ln(x_1) + w_2 \ln(x_2) + w_3 \ln(x_3) + b$$

Now let's chose the exponential function as activation:

$$\phi(x) = \exp(x)$$

The model's prediction $\hat{y}$ is then given by:

$$\hat{y} = \exp(w_1 \ln(x_1) + w_2 \ln(x_2) + w_3 \ln(x_3) + b) = \exp(b) x_1^{w_1} x_2^{w_2} x_3^{w_3}$$

If we set $b = \ln(3)$, $w_1 = 2$, $w_2 = 3$, and $w_3 = 1$, we will have the exact weights and biases that correspond to the model such that:

$$\hat{y} = y = 3x_1^2 x_2^3 x_3$$

We will see in a later chapter that this is not always practically possible due to the exploding and vanishing gradients problems that arise during training.

3.2.2 Expanding the Hypothesis Space

Informally, a model's capacity refers to its ability to learn and represent a diverse set of functions [17]. One way to increase a model's capacity can be to expand it's hypothesis space.

Definition 3.1 (Hypothesis space): A model's hypothesis space is "the set of

functions that the learning algorithm is allowed to select as being the solution" [17]

We can in fact also expand the capacity of a single perceptron by including more functions in its hypothesis space. One way to do that is to add polynomial features to the input vector . First let's look at the linear regression model, where the input is a single value x:

$$\hat{y} = b + wx$$

The hypothesis space of this model is the set of all linear functions. We can increase the model's capacity by including polynomial features such that if the original input is x, we can create an expanded feature vector $\mathbf{x}$ as:

$$\mathbf{x}^* = \begin{bmatrix} x & x^2 & x^3 & \dots & x^n \end{bmatrix}^T$$

The neuron then processes this expanded feature vector, allowing it to capture non-linear relationships. Our output then becomes:

$$\hat{y} = b + w_1 x + w_2 x^2 + \dots w_n x^n$$

Although this new model implements a n-degree polynomial function of its inputs, the output remains a linear function of the model's parameters $w_1, w_2, \dots w_n$. We will see later why the linear relationship between output and weights is important in the training process of the model.

Let's recall Example 2 (3.1):

$$y = \begin{cases} 1 & \text{if } x_1^2 + x_2^2 + x_3^2 > 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

We can transform the input vector $[x_1, x_2, x_3]$into a larger vector:

$$\mathbf{x}^* = \begin{bmatrix} x_1 & x_1^2 & x_2 & x_2^2 & x_3 & x_3^2 \end{bmatrix}^T$$

If the model was more complex such as:

$$y = \begin{cases} 1 & \text{if } 2x_1^2 x_2 + 3x_2^2 - 4x_2^2 x_3^2 > 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

Here we would need to include all possible terms of the form: $x_1^n x_2^m x_3^p$ where $n, m, p \in \{0, 1, 2\}$ in the perceptron's hupothesis space. This gives a total of $3^3 = 27$ terms to include in our transformed input vector x^*. We would then have:

$$\hat{y} = \sum_{k=0}^{2} \sum_{l=0}^{2} \sum_{m=0}^{2} w_{k,l,m} \cdot x_1^k \cdot x_2^l \cdot x_3^m$$

Note: The bias is incorporated in the model with $w_{0,0,0} = b$.

In the previous example, because we know the function we want to model, we can easily see what functions need to be added in the perceptron's hypothesis space. However, this is never the case in tasks for which we build neural networks. After all, we wouldn't need a neural network in the first place if we have a formula that computes output from input.

One might assume that by including a large enough variety of functions (higher degree polynomials, trigonometric, exponential, logarithmic...) in a perceptron's hypothesis space, the perceptron can theoretically model any kind of mathematical relationship, given that its hypothesis space allows it. However, this is not true because "machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform." We will see in later sections the problems that arise by having an unnecessarily large hypothesis space (overfitting).

It is important to note that the inability to handle non-linear separations (without any input vector transformations) is a limitation of a single perceptron model, however this is not the case for MLPs which don't depend only on one neuron, but consists of layers of neurons working together. As we will see in the next section, the modeling and approximation ability of MLPs is much more advanced than that of a single perceptron.

3.2.3 Using Multilayer Perceptrongs (MLP)

In the first section of this chapter (3.1), we saw that a single perceptron is unable to model the XOR function of two variables. However, if we allow ourselves to use more artificial neurons and create a simple FNN which contains a single hidden layer with two neurons, we can in fact model the XOR function [17]. Here the network contains

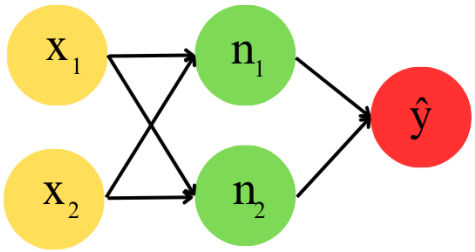

Figure 3.7: MLP with One Hidden Layer Containing Two Neurons

two functions chained together the first function transforms the input vector $[x_1, x_2]$ to $[n_1, n_2]$ and the second one transforms $[n_1, n_2]$ into $\hat{\mathbf{y}}$. The output of the function is still a linear model but instead of being applied the the input vector it is applied to the activation output of the hidden layer.

In our input space, we have four different cases for $[x_1, x_2]$: $[0,0]$, $[0,1]$, $[1,0]$ and $[1,1]$. Let W_1 and b_1 be the weight matrix and bias vector associated with the hidden layer respectively:

$$W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

And let W_2 and b_2 be the weight matrix and bias vector associated with the output layer respectively:

$$W_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$b_2 = 0$$

For the activation functions we will use $\phi_1(x) = ReLu(x) = max(0,x)$ for the first layer and $\phi_2(x) = x$ for the second layer.

Now we can compute the output of each of our four data points and compare it with the actual values given by the XOR truth table.

1. For $[0,0]$:

$$v_1 = W_1 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + b_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$a_1 = \phi_1(v_1) = \begin{bmatrix} \max(0,0) \\ \max(0,-1) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$v_2 = W_2^T a_1 + b_2 = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0 = 0$$

$$\hat{y} = \phi_2(v_2) = 0$$

2. For $[0,1]$:

$$v_1 = W_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + b_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$a_1 = \phi_1(v_1) = \begin{bmatrix} \max(0,1) \\ \max(0,0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$v_2 = W_2^T a_1 + b_2 = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 = 1$$

$$\hat{y} = \phi_2(v_2) = 1$$

3. For $[1,0]$:

$$v_1 = W_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + b_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$a_1 = \phi_1(v_1) = \begin{bmatrix} \max(0,1) \\ \max(0,0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$v_2 = W_2^T a_1 + b_2 = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 = 1$$

$$\hat{y} = \phi_2(v_2) = 1$$

4. For $[1,1]$:

$$v_1 = W_1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + b_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$a_1 = \phi_1(v_1) = \begin{bmatrix} \max(0,2) \\ \max(0,1) \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$v_2 = W_2^T a_1 + b_2 = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + 0 = 0$$

$$\hat{y} = \phi_2(v_2) = 0$$

This network which contains only one hidden layer managed to successfully output the rights values of the XOR function for every possible pair of inputs. This demonstrates how adding one or more hidden layers -introducing additional weights and biases- to transform the single perceptron into a multi-layer perceptron allows the network to learn more complex representations of the input data. However, this does not expand the hypothesis space of a single artificial neuron, but of the network as a whole. In this example, we could specify the values of weights and biases and demonstrated that it does in fact work, however in a real life problem, we could have millions and even billions of weights and biases (parameters) and billions of examples, so it is impossible to simply guess the correct values of the parameters as we did here [17]

In the next section we will examine two powerful results: The Universal Approximation Theory and Kolomogrov Superposition Theory. These theories imply that more advanced neural network architectures with hidden layers, such as MLPs, overcome many of the limitations of a single perceptron and allows a neural network to approximate more complex, non-linear functions.

3.3 Universal Approximation Theorem

The Universal Approximation Theorem asserts that "a feedforward network with a linear output layer and at least one hidden layer with any 'squashing' activation function (such as the logistic sigmoid activation function) can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$, from one

finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units." [17] [29] [7] Kolmogorov's Superposition Theorem supports this by showing that multivariate functions can be broken down into compositions of simpler functions. This implies that even a relatively simple neural network has the potential to approximate complex functions.

3.4 Kolmogorov-Arnold Superposition Theorem

Historically, the initial excitement for neural networks faded when researchers studying perceptrons demonstrated their significant limitations in the types of functions they could represent [40]. Interest in neural networks returned when Hecht-Nielsen [44] observed that the Kolmogorov-Arnold Representation (or Superposition) Theorem, which was motivated by Hilbert's 13th problem, implies that neural networks with four layers are universal for compact intervals.

Hilbert's 13th Problem

Hilbert's 13th problem, posed by David Hilbert in 1900, asked whether there exist solutions to the general seventh-degree equation that cannot be expressed as superpositions (composition) of continuous functions of two variables [27].

> *Let $f(x,y,z) = u$, where u is a solution of the seventh-degree equation $u^7 + xu^3 + yu^2 + zu + 1 = 0$. Prove that f cannot be expressed as a composition of a finite number of functions of two variables.*

The problem initially focused on whether solutions to certain algebraic equations could be expressed using a finite number of algebraic functions of fewer variables. However, it was later generalized to the question of whether any continuous function of multiple variables can be expressed as a composition of continuous functions of fewer variables.

Generalization of Hilbert's 13th Problem Statement

> *"Show that there exists a continuous function of three variables that cannot be expressed using a finite number of continuous functions of two variables."*

If Hilbert's conjecture were true, some (nonlinear) three-input systems could not be decomposed into one- or two-input systems. However, the conjecture was proved false in 1957 by Vladimir Arnold, who at the time was a student of Andrey Kolmogorov.

3.4.1 The Kolmogorov–Arnold Representation Theorem (1957)

In 1956, Andrey Kolmogorov proved that every continuous function of several variables can be represented as a finite composition of continuous functions of two variables [33]. In 1957, he proved the following result:

> *"Every continuous function $f\colon [0,1]^n \to \mathbb{R}$ of n variables can be represented as a superposition of continuous functions of one variable and addition."*

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \psi_{pq}(x_p) \right) \tag{3.5}$$

where Φ_q and ψ_{pq} are continuous functions of a single variable [32].

3.4.2 Arnold's Generalization and Answer to Hilbert's 13th Problem (1958)

Building on Kolmogorov's work, Vladimir Arnold demonstrated that only two-variable functions were necessary to represent any continuous function of three variables. This significant refinement of Kolmogorov's theorem further strengthened the understanding of functional decomposition and answered Hilbert's question for the class of continuous functions [2].

3.4.3 Implications on Neural Networks

Kolmogorov's Superposition Theorem and Arnold's results support the theoretical capabilities of DNNs and have practical implications for their design and training:

- **Layered Structure of DNNs**: Kolmogorov's work implies that deeper networks

with multiple layers can achieve better approximations of complex functions by successively applying non-linear transformations.

- **Function Approximation in High Dimensions**: In high-dimensional spaces, the ability to approximate functions using compositions of lower-dimensional functions becomes even more crucial. DNNs leverage this by using hierarchical layers, where each layer extracts features at different levels of abstraction. Kolmogorov's theorem provides a mathematical foundation for this hierarchical approach, as it ensures that complex functions in high-dimensional spaces can be broken down into simpler, more manageable components.
- **Network Depth**: The theorem suggests that deeper networks (with more layers) can potentially approximate more complex functions, as they can represent compositions of simpler functions.
- **Non-linearity**: By using a non-linear activation (ReLU, sigmoid, tanh ...), neural networks can approximate non-linear relationships in the data, aligning with the principle that non-linear functions can be composed from simpler functions.

Kolmogorov's Superposition Theorem provides a deep theoretical understanding of why neural networks, particularly DNNs, are capable of approximating complex functions. This theorem bridges the gap between mathematical theory and practical application.

Chapter 4

LEARNING ALGORITHMS CHALLENGES AND LIMITATIONS

This part of the thesis focuses on the learning mechanisms of DNNs. It delves into the mathematical tools that underlie the training process of neural networks, namely stochastic gradient descent (SGD) and backpropagation. This chapter also explains the two general types of learning and discusses several challenges faced during the learning process including overfitting, underfitting, vanishing and exploding gradients, hyperparameter tuning, shedding light on the limitations of current learning algorithms.

4.1 Calculus Concepts

We assume the reader is familiar with derivatives, partial derivatives, function composition and the chain rule.

Definition 4.1 (Scalar function): is a function $f : \mathbb{R}^n \to \mathbb{R}$ that maps a set of input variables to a single real number (a scalar).

Definition 4.2 (Gradient vector): of a scalar function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$, is defined as the vector of partial derivatives of f with respect to each component of $\mathbf{x}$. Formally, the gradient vector $\nabla f(\mathbf{x})$ is given by:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

where $\frac{\partial f}{\partial x_i}$ denotes the partial derivative of f with respect to the i-th component of $\mathbf{x}$.

Definition 4.3 (The Euclidean norm): also known as the L_2 norm, of a vector $\mathbf{v}$ with components $v_1, v_2, \ldots, v_n$ is defined as follows:

$$\|\mathbf{v}\|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

In general, the L_2 norm of a vector $\mathbf{v}$ is calculated by taking the square root of the sum of the squares of its components.

Definition 4.4 (Directional derivative): The **directional derivative** of a function $f\colon \mathbb{R}^n \to \mathbb{R}$ in the direction of a unit vector $\mathbf{u}$ represents the rate at which the function f changes in the direction of $\mathbf{u}$ [17]. Specifically, it is the derivative of the function $f(\mathbf{x}+\alpha\mathbf{u})$ with respect to α, evaluated at $\alpha = 0$. By applying the chain rule, we find that:

$$\frac{\partial}{\partial \alpha} f(\mathbf{x}+\alpha\mathbf{u})\bigg|_{\alpha=0} = \mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x})$$

This can be rewritten as:

$$\|\mathbf{u}\|_2 \|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 \cos\theta$$

where θ is the angle between $\mathbf{u}$ and the gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$.

4.2 Types of Learning

4.2.1 Supervised Learning

Supervised learning is the most widely used machine learning algorithm [43] and it refers to algorithms that learn input to output mapping. The model learns by seeing many examples of correct pairs of input x and output label y.

Definition 4.5 (An Example): An example, also refereed to as a data point is "a collection of features that have been quantitatively measured from some object or event that we want the machine learning system to process." [17].

Each example is made up of a collection of n features represented as an input vector $\mathbf{x} \in \mathbb{R}^n$. This vector **x** can be representing a/an:

- **Image**: by flattening its pixel values, either in grayscale or color, into a single one-dimensional array
- **Text**: by encoding each character or word as a numerical value
- **Sound**: by sampling the audio signal at regular intervals and storing the amplitude values of these samples sequentially
- **Set of n features** that describe an object, a person or anything that has features that can be represented as an n-dimentional vector.

In the context of supervised learning, the input vector **x** is associated with a corresponding label or target **y** which can be a vector or a value. The value of the output **y** tells us something about this input vector, a classification (pass or fail, cat or dog or bird, male or female), a percentage (success rate, risk rate), or any result that can be represented numerically and that we are interested in modeling.

The goal is for the model to process the given set of examples and learn the relationship between the input vector and the output value, such that after the training is complete, given an input value **x** which it hasn't seen before (outside of our training dataset), the model can accurately predict the corresponding output value. In short, given pairs of input-output data (training set), the task of the network is to learn the "correct" input-output mapping from the set of examples it processes. There are two main types of supervised learning: regression and classification.

4.2.1.1 Regression

Regression involves predicting a single value which can take infinite possible values

Table 4.1: Examples of Regression Applications

Input (x)	**Output (y)**	**Applications**
House size, Location, Year built	Price	Real Estate
Salary, Debt, Savings, Assets	Risk Rating	Banking
Environmental Data	Weather forecasting	Meteorology

4.2.1.2 Classification

Classification involves predicting only a finite number of possible outputs, called classes or categories, which could be non-numerical.

Table 4.2: Examples of Classification Applications

Input (x)	**Output (y)**	**Applications**
Email	Spam?	Spam Detection/Filtering
Tumor Size	Benign/Malignant?	Cancer Detection
Image	Cat/Dog/Bird?	Image Recognition

4.2.2 Unsupervised Learning

After supervised learning, unsupervised learning is the second most used form of machine learning [43]. In this type of learning algorithm we are given unlabeled data, meaning in our examples there is no output **y** associated which each input vector **x**. It is called unsupervised because the goal is not to supervise the algorithm to give the "correct" answer for every given input, classify or predict a certain value, but to find some structure or pattern in the data. For instance the algorithm might decide to assign the unlabelled data into two -or more- different groups or clusters: this is called clustering algorithm [43].

4.2.2.1 Clustering Algorithm

The network must then develop its behavior based on inputs and outputs without knowing the expected responses. This type of algorithm is used for example in news platforms, where the algorithm, on its own, finds the articles that mention the same words and groups them into clusters. It is also used in genetics to classify people based on their DNA, or in marketing to group customers based on their profiles. As

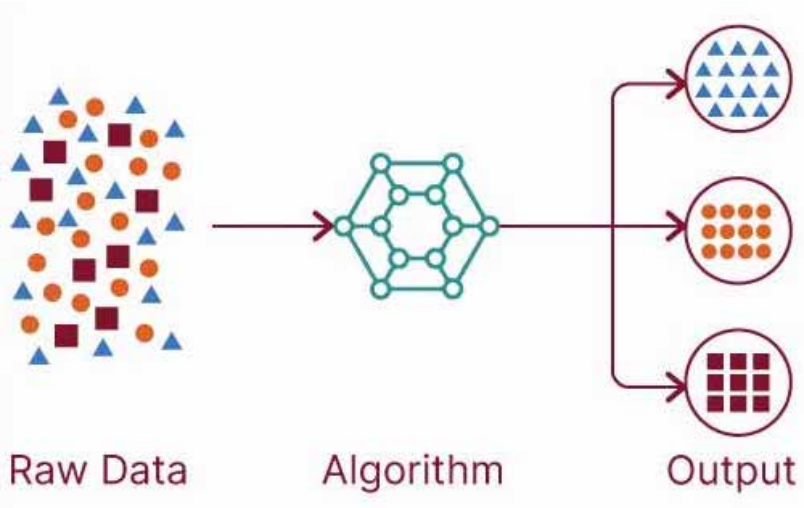

Figure 4.1: Visualizing Clustering Algorithms

opposed to classification and regression (supervised) which are predictive data mining methods, clustering (unsupervised) is a descriptive method.

4.2.2.2 Other Types of Unsupervised Learning

There are also many other types of unsupervised learning such as:

1. Anomaly Detection: Used to detect unusual data points, used in fraud detection
2. Dimensionality Reduction: Compress a big data set into a much smaller data set while loosing as little information as possible.

4.2.3 Semi-supervised Learning

Semi-supervised learning merges aspects of both supervised and unsupervised learning. These models are trained using a limited quantity of labeled data in combination with a larger quantity of unlabeled data. This method leverages the strengths of supervised learning for exact labeling while also utilizing unsupervised learning to find patterns and structures from the larger, unlabeled dataset.

4.3 Train and Test Data

In supervised learning algorithms, the dataset is split into two primary parts: the training set and the test set. The training set is used to build and train the model, while the test set is used to assess the model's performance. Understanding the distinction between these two subsets and their roles in the training process is very important for building effective models.

4.3.1 Training Data

The training dataset is the subset of the dataset used to train the model. It should be representative of the overall dataset, covering the various scenarios and variations the model is expected to encounter in real-world applications.

4.3.2 Test Data

Test data is used to assess the accuracy of the trained model. This data is not used during the training process and serves as a way to evaluate the ability of the model to generalize given new, unseen examples. By comparing the model's predictions on the test data to the actual values, we can estimate its accuracy, precision, recall, and other performance metrics.

It is essential that the test data remains separate from the training data to provide an unbiased evaluation of the model's performance. Using the same data for both training and testing can lead the model to performing well on the training dataset while performing poorly on unseen datasets. This is called overfitting, and we will explore it in more details in a later section.

4.3.3 Splitting the Data

The process of dividing the data into training and test sets can be done in various ways. A common method is to randomly partition the dataset, allocating a certain percentage for training and the rest for testing. For instance, a typical split might involve using 70-80% for training and 20-30% for testing.

4.3.4 Statistical Learning Theory and i.i.d. Assumptions

Statistical learning theory offers insights that explain why observations from the training set can influence the performance on the test set. If training and test sets are collected randomly without any specific criteria, improving test set performance becomes challenging. However, assumptions about the data collection process can provide a solution.

The data used for training and testing is assumed to be generated by a probability

distribution over datasets, referred to as the data-generating process. A common set of assumptions, known as the i.i.d. (independent and identically distributed) assumptions, is often applied. These assumptions indicate that the examples within each dataset are independent of one another and that both the training and test sets originate from the same underlying probability distribution. This shared distribution is referred to as the data-generating distribution, represented as p_{data} [17].

With these assumptions, we can establish "that the expected training error of a randomly selected model is equal to the expected test error of that model" [17]. In practice, however, the training set is sampled first, then it's used to adjust the model parameters to minimize training error, and then we evaluate the model on the test set. This typically results in the expected test error being greater than or equal to the expected training error.

In a nutshell, the success of a machine learning algorithm is assessed by its ability to:

1. Minimize the training error.
2. Reduce the gap between training and test errors.

These two factors are central to addressing the challenges of underfitting and overfitting [17] which we will explore in a later section. First, we will explore one the most widely used approach for training deep neural networks: Gradient Based Optimization

The gradient descent algorithm is primarily used in supervised learning but also has applications in certain unsupervised learning tasks where optimization is required.

4.4 Gradient Descent

4.4.1 Gradient Based Optimization

Gradient descent is an optimization technique that aims to reduce a function's value by repeatedly moving in the direction of the steepest descent, which is determined by the negative gradient. At its core, gradient descent aims to find the minimum of a function $f(\theta)$, where θ represents the parameters of the function. The goal is to adjust θ such that $f(\theta)$ is minimized. In optimization, when we aim to minimize a function, we refer to it as loss function (also cost or error function). [17] Given a

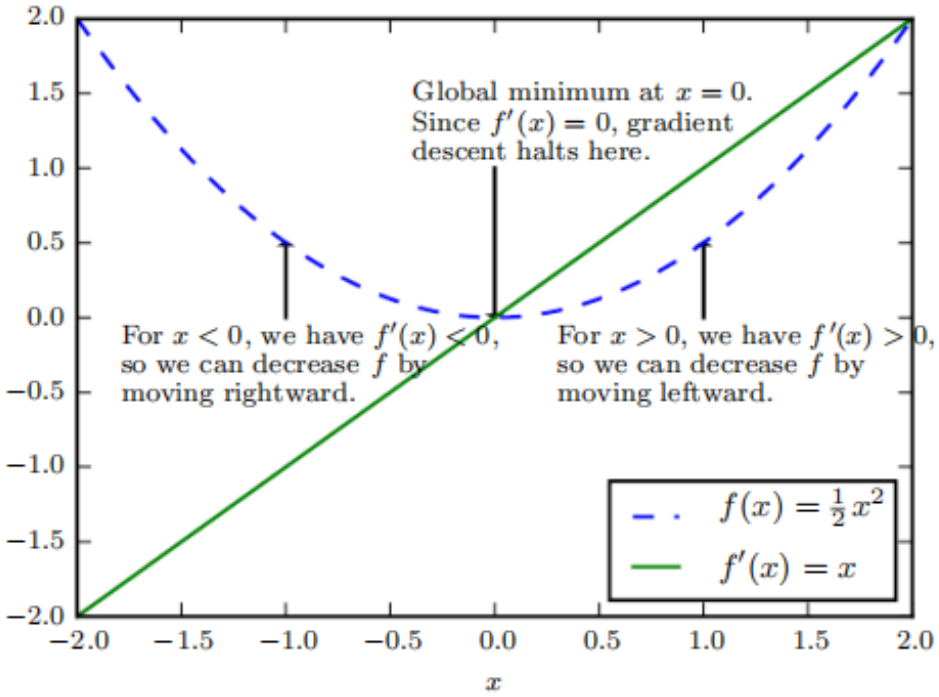

Figure 4.2: Visualizing Gradient Descent [17]

function $y = f(x)$ the derivative of this function denoted $f'(x)$ or as $\frac{dy}{dx}$ gives the gradient or slope of $f(x)$ at the point x. This value specifies how to scale a small change in the input so that we can obtain the corresponding change in the output. [17]

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

"The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y." [17]

Note: For critical points (or stationary points), where $f'(x) = 0$, the derivative does not provide any information about which direction to move in order to minimize the function. These points are either local or global minima, maxima or saddle points.

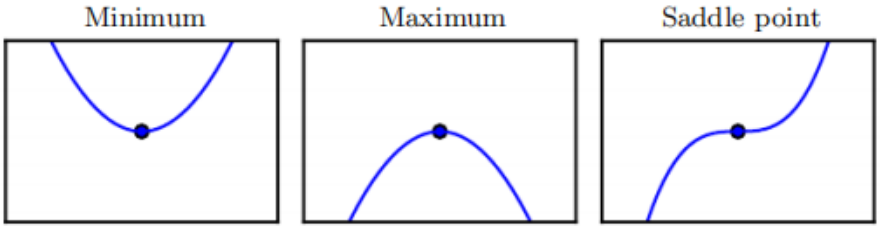

Figure 4.3: Critical Points: Min (left), Max (center) and Saddle Point (left) [17]

In theory, when optimizing the cost or loss function, we aim to find the global maximum, or a global minimum (if the function has more than one) which is the absolute lowest value of f(x). In the case of deep learning loss functions, this is a very hard, sometimes impossible task, since optimization algorithms might struggle to locate a global minimum if the function has numerous local minima or flat regions. [17] Figure 4.4 illustrates this problem in a function with only one input x, however in deep learning, the functions we aim to minimize -the loss function- is multidimensional and the number of parameters that affect the value of the loss function are sometimes in the billions!

When training a DNN model we usually settle for solutions that aren't truly minimal,

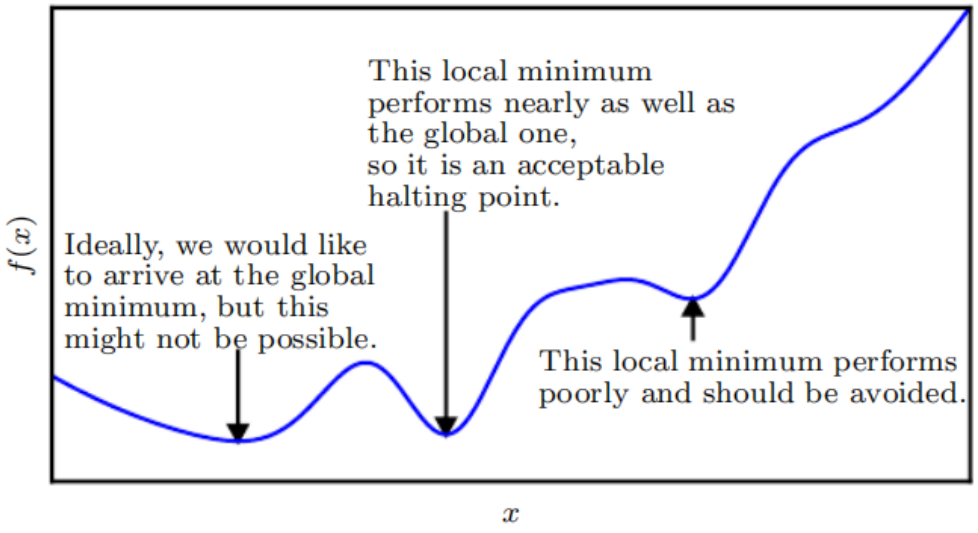

Figure 4.4: 1-D Function with Multiple Local Minima or Plateaus [17]

given that they provide significantly low loss function values. The goal from training is for the model to learn which direction and by how much it should changes the values of its parameters (weights, biases) to minimize the loss function. Hence, partial derivatives are used.

To minimize f, we need to identify the direction in which f decreases most rapidly. This can be achieved by using the directional derivative:

$$\min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} \|\mathbf{u}\|_2 \|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 \cos\theta$$

Since $\|\mathbf{u}\|_2 = 1$ and ignoring factors that are independent of $\mathbf{u}$, this expression can be simplified as follows:

$$\min_{\mathbf{u}} \cos\theta$$

The minimum value is obtained when $\cos\theta = -1$ which happened when $\theta = \pi$, in other words when $\mathbf{u}$ is oriented exactly opposite to the gradient. Therefore, the gradient indicates the direction of the steepest ascent, while the negative gradient indicates the

direction of the steepest descent. Thus **gradient descent** (or **steepest descent**) is a method that involves moving in the direction of the negative gradient in order to decrease f. In the next section we will explore the different functions used as loss functions and then dive into how the process of gradient descent is applied.

4.4.2 Loss functions in Neural Networks

Loss functions play a critical role in training DNNs since they measure how well the model's predictions are compared to the actual target values. By minimizing the loss function during training, the model learns to make more accurate predictions.

Loss functions are typically functions of the predicted outputs, (which in their turn, are functions of model parameters) and the actual target values. In order for the idea of minimization to make sense, the loss function $L\colon \mathbb{R}^n \to \mathbb{R}$ has to have only one (scalar) output [17].

4.4.2.1 Mean Squared Error (MSE)

Mean Squared Error is widely used for regression tasks in feed-forward and convolutional neural networks.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the actual value associated with the input x_i, $\hat{y}_i$ is the model's output (predicted value), and N is the number of examples in the sample.

4.4.2.2 Cross-Entropy Loss

Also known as log loss, the cross-entropy Loss, is commonly applied in classification tasks in FNNs, CNNs, and Recurrent Neural Networks (RNNs).

For binary classification:

$$\text{Binary Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

For multi-class classification:

$$\text{Categorical Cross-Entropy Loss} = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

where y_{ij} is a binary indicator (0 or 1) if class label j is the correct classification for sample i, $\hat{y}_{ij}$ is the predicted probability of sample i being in class j, and C is the number of classes.

4.4.2.3 Hinge Loss

Hinge Loss is primarily used for training Support Vector Machines (SVMs) but can also be used in FNNs and CNNs for binary classification tasks.

$$\text{Hinge Loss} = \frac{1}{N}\sum_{i=1}^{N}\max(0, 1-y_i\hat{y}_i)$$

where y_i is the actual class label (-1 or 1), and $\hat{y}_i$ is the predicted value.

4.4.2.4 Negative Log-Likelihood Loss (NLL)

Negative Log-Likelihood Loss is often used with softmax activation in the output layer for classification tasks, particularly in RNNs.

$$\text{NLL} = -\sum_{i=1}^{N}\log(\hat{y}_{i,y_i})$$

where $\hat{y}_{i,y_i}$ is the predicted probability of the correct class for sample i.

4.4.3 Example of Gradient Descent

Consider a simple neural network model with no hidden layers, taking three inputs x_1, x_2, x_3 and producing one output y. The activation function is ReLU, defined as $\text{ReLU}(v) = \max(0, v)$.

The model can be described by the following equations:

$$v = \mathbf{w}^T \mathbf{x} + b$$
$$\hat{y} = \text{ReLU}(v)$$

where $\mathbf{w} = [w_1, w_2, w_3]$ is the weight vector, $\mathbf{x} = [x_1, x_2, x_3]$ is the input vector, b is the bias, and $\hat{y}$ is the predicted output.

Assume the loss function is the mean squared error (MSE). In the case where N=1 (we are updating the parameters based on only one example), we get:

$$L = (y - \hat{y})^2$$

where y is the actual output.

To perform gradient descent, we need to compute the gradients of the loss function with respect to the weights and the bias.

First, compute the gradient of the loss with respect to the predicted output $\hat{y}$:

$$\frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y})$$

Next, compute the gradient of the loss with respect to v:

$$\frac{\partial L}{\partial v} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial v}$$

Since $\hat{y} = \text{ReLU}(v)$, the gradient of ReLU is:

$$\frac{\partial \hat{y}}{\partial z} = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$$

Thus,

$$\frac{\partial L}{\partial v} = -2(y - \hat{y}) \cdot \text{ReLU}'(v)$$

where $\text{ReLU}'(v)$ is the derivative of ReLU.

Now, compute the gradients of the loss with respect to the weights w_1, w_2, and w_3 and the bias b:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial v} \cdot \frac{\partial v}{\partial w_1} = -2(y - \hat{y}) \cdot \text{ReLU}'(v) \cdot x_1$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial v} \cdot \frac{\partial v}{\partial w_2} = -2(y - \hat{y}) \cdot \text{ReLU}'(v) \cdot x_2$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial v} \cdot \frac{\partial v}{\partial w_3} = -2(y - \hat{y}) \cdot \text{ReLU}'(v) \cdot x_3$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial v} \cdot \frac{\partial v}{\partial b} == -2(y - \hat{y}) \cdot \text{ReLU}'(v)$$

With these individual gradients, we can update each weight and the bias as follows:

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

$$w_2 \leftarrow w_2 - \eta \frac{\partial L}{\partial w_2}$$

$$w_3 \leftarrow w_3 - \eta \frac{\partial L}{\partial w_3}$$

$$b \leftarrow b - \eta \frac{\partial L}{\partial b}$$

where η is the learning rate.

4.4.4 Training Hyperparameters

In the previous chapter we examined the model parameters and hyperparameters, which determine what the network looks like, pre-training. To train the model, we also need to specify hyperparameters which influence the learning process itself.

- **Learning Rate**: A key hyperparameter that determines the size of the steps the

model takes when adjusting its parameters (weights and biases) to minimize the error. For faster convergence and to avoid getting trapped in local minima, the learning rate is usually updated during training, either using a learning rate schedule or using adaptive learning rate methods [45]. The initial learning rate can be set to a system default or selected using various techniques [51]. A learning rate schedule modifies the learning rate during training, typically between epochs or iterations, using parameters like decay and momentum. "Common learning rate schedules include time-based, step-based, and exponential methods" [45]. Adaptive learning rates address the need to manually choose hyperparameters for each learning session.

- **Decay** helps stabilize learning by preventing oscillations that occur when a constant high learning rate causes the learning process to overshoot a minimum. The time-based learning rate schedule adjusts the learning rate based on the previous iteration's learning rate. The formula incorporating decay is:

$$\eta_{n+1} = \frac{\eta_n}{1 + dn}$$

where η is the learning rate, d is a decay parameter, and n is the iteration step.

The step-based learning rate schedule changes the learning rate at predefined steps. The formula is:

$$\eta_n = \eta_0 d^{\lfloor \frac{1+n}{r} \rfloor}$$

where η_n is the learning rate at iteration n, η_0 is the initial learning rate, d indicates the rate change (e.g., 0.5 for halving), and r is the drop rate (e.g., every 10 iterations).

Exponential learning schedules use a decreasing exponential function:

$$\eta_n = \eta_0 e^{-dn}$$

where d is a decay parameter.

- **Momentum**: momentum accelerates learning when the error cost gradient moves consistently in one direction and helps, thus leading to faster converging [48].
- **Batch Size**: The number of training examples used in a single iteration of model training.
- **Number of Epochs**: The number of times the entire training dataset is passed forward and backward through the neural network.
- **Optimizer**: The algorithm used to minimize the loss function, such as SGD, Adam, or RMSprop.

4.5 Backpropagation

Backpropagation is a crucial algorithm for efficiently computing gradients in neural networks, enabling the training process to update weights and biases effectively using gradient descent methods. This technique is widely used beyond neural networks, offering a powerful tool for computing derivatives in a variety machine learning and optimization problems [35].

In FNN information is flowing forward in the network. This process is known as forward propagation. During training, forward propagation continues until it produces a scalar loss $L(\theta)$.

The backpropagation algorithm (Rumelhart et al., 1986a) enables information about the loss function to propagate backward through the network to calculate the gradient. Although finding an analytical expression for the gradient is straightforward, its numerical calculation can be computationally intensive.

Backpropagation algorithm offered a practical and efficient way [46] to calculate

gradients as follows:

1. **Layer-wise Gradient Calculation**: Backpropagation computes gradients layer by layer. This approach significantly reduces computational complexity. It calculates the gradient at each layer and uses these gradients to update weights and biases in the preceding layer [46].

2. **Utilizing the Chain Rule**: The gradient of the loss function with respect to the input of a neuron can be computed as the product of the gradient of the loss with respect to the neuron's output and the derivative of the neuron's activation function with respect to its input. This allows for efficient computation of gradients from the output layer back to the input layer [17].

3. **Error Propagation**: Backpropagation propagates errors backward from the output layer to the input layer ensuring that each weight and bias is updated in a way that minimizes the overall error. This makes the gradient calculation both practical and scalable, even for deep networks with many layers [35].

4. **Weight and Bias Updates**: After computing the gradients, these values are used to update the network's weights and biases. This cycle of forward propagation, gradient computation, and weight adjustment continues iteratively until the network converges to a solution that "minimizes" the error [28].

4.5.1 Example of Backpropagation

We will now demonstrate how to perform backpropagation in a simple neural network taking inputs x_1 and x_2 and producing the single output $\hat{y}$. Suppose the network has 1 hidden layer containing 2 neurons. The target model is $y = 3x_1 - x_2$. For simplicity, let all activation functions be the identity function ($a_n^{(\ell)} = v_n^{(\ell)}$) and let's initialize all weights to 1 and biases to 0.

This example will show the forward pass, loss calculation, and step-by-step

backpropagation, followed by the weight and bias updates using a learning rate of 0.01.

Network Structure

1. Input Layer: 2 inputs $[x_1, x_2]$
2. Hidden Layer: 2 neurons $[a_1^{(1)}, a_2^{(1)}]$
3. Output Layer: 1 output $\hat{y}$

Weights and biases:

- Weights from Input Layer to Hidden Layer 1:

$$w_1, w_2, b_1 \text{ for neuron } a_1^{(1)}$$
$$w_3, w_4, b_2 \text{ for neuron } a_2^{(1)}$$

- Weights from Hidden Layer 1 to Output Layer:

$$w_5, w_6, b_3$$

- Biases: All biases are set to 0.

We will pass an example $[x_1, x_2] = [1, 2]$ into the network as follows:

4.5.1.1 Forward Pass

Input layer to hidden layer:

$$a_1^{(1)} = w_1 x_1 + w_2 x_2 + b_1 = 1(1) + 1(2) + 0 = 3$$
$$a_2^{(1)} = w_3 x_1 + w_4 x_2 + b_2 = 1(1) + 1(2) + 0 = 3$$

Hidden layer to output layer:

$$\hat{y} = w_5 a_1^{(1)} + w_6 a_2^{(1)} + b_3 = 1(3) + 1(3) + 0 = 6$$

4.5.1.2 Loss Calculation

We will use MSE as our loss function and in this demonstration, we will update the weights based on only one example and thus $N = 1$. We get:

$$L = (y - \hat{y})^2$$

For this example: $y = 3x_1 - x_2 = 3(1) - 1 = 1$ so that:

$$L = (1-6)^2 = (-5)^2 = 25$$

4.5.1.3 Backward Pass: Output Layer to Hidden Layer

$$\frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y})$$

$$\frac{\partial \hat{y}}{\partial w_5} = a_1^{(1)} = x_1 + x_2$$

$$\frac{\partial \hat{y}}{\partial w_6} = a_2^{(1)} = x_1 + x_2$$

$$\frac{\partial \hat{y}}{\partial b_3} = 1$$

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_5} = -2(y - \hat{y}) \cdot (x_1 + x_2) = -2(-5)(3) = 30$$

$$\frac{\partial L}{\partial w_6} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_6} = -2(y - \hat{y}) \cdot (x_1 + x_2) = 30$$

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y}) = 10$$

4.5.1.4 Output Layer Weight and Bias Updates

Using the gradient descent update rule $\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$, where η is the learning rate (set to 0.01), the updated weights are:

$$w_5 \leftarrow w_5 - 0.01 \cdot \frac{\partial L}{\partial w_5}$$
$$= 1 - 0.01 \cdot 30 = 0.7$$

$$w_6 \leftarrow w_6 - 0.01 \cdot \frac{\partial L}{\partial w_6}$$
$$= 1 - 0.01 \cdot 30 = 0.7$$

$$b_3 \leftarrow b_3 - 0.01 \cdot \frac{\partial L}{\partial b_3}$$
$$= 0 - 0.01 \cdot 10 = -0.1$$

4.5.1.5 New Loss Calculation

After updating the weights and bias in the output layer, we use them to calculate the new $\hat{y}_{\text{new}}$. Given:

$$w_5 = 0.7, \quad w_6 = 0.7, \quad b_3 = -0.1$$

Now, the new $\hat{y}$ is calculated as follows:

$$\hat{y}_{\text{new}} = w_5 a_1^{(1)} + w_6 a_2^{(1)} + b_3$$

Substitute the activations $a_1^{(1)}$ and $a_2^{(1)}$ with $x_1 + x_2 = 1 + 2 = 3$:

$$\hat{y}_{\text{new}} = 0.7 \cdot 3 + 0.7 \cdot 3 + (-0.1) = 2.1 + 2.1 - 0.1 = 4.1$$

Now, calculate the loss:

$$L = (y - \hat{y}_{\text{new}})^2 = (1 - 4.1)^2 = (-3.1)^2 = 9.61$$

4.5.1.6 Hidden Layer Weight and Bias Update

Now we need to do another backward pass to update the weights and biases in the hidden layer, using the new values for $\hat{y}_{\text{new}}, w_5, w_6$ and b_3 :

1. Output Layer Gradients:

$$\frac{\partial L}{\partial \hat{y}_{\text{new}}} = -2(y - \hat{y}_{\text{new}}) = -2(1 - 4.1) = 6.2$$

2. Gradients w.r.t Hidden Layer Activations:

$$\frac{\partial L}{\partial a_1^{(1)}} = \frac{\partial L}{\partial \hat{y}_{\text{new}}} \cdot w_5 = 6.2 \cdot 0.7 = 4.34$$

$$\frac{\partial L}{\partial a_2^{(1)}} = \frac{\partial L}{\partial \hat{y}_{\text{new}}} \cdot w_6 = 6.2 \cdot 0.7 = 4.34$$

3. Update Weights in Hidden Layer:

$$\frac{\partial L}{\partial w_1} = 4.34 \cdot x_1 = 4.34 \cdot 1 = 4.34$$

$$w_1 \leftarrow w_1 - 0.01 \cdot 4.34 = 1 - 0.0434 = 0.9566$$

$$\frac{\partial L}{\partial w_2} = 4.34 \cdot x_2 = 4.34 \cdot 2 = 8.68$$

$$w_2 \leftarrow w_2 - 0.01 \cdot 8.68 = 1 - 0.0868 = 0.9132$$

4. Update Bias in Hidden Layer:

$$b_1 \leftarrow b_1 - 0.01 \cdot 4.34 = 0 - 0.0434 = -0.0434$$
$$b_2 \leftarrow b_2 - 0.01 \cdot 4.34 = 0 - 0.0434 = -0.0434$$

So, the final weights and biases in the hidden layer after updating with the new $\hat{y}_{\text{new}}$ are:

$$w_1 = 0.9566, \quad w_2 = 0.9132, \quad b_1 = -0.0434$$
$$w_3 = 0.9566, \quad w_4 = 0.9132, \quad b_2 = -0.0434$$

4.5.1.7 Forward Pass with Updated Weights and Biases

Now, let's perform the forward pass with the updated weights and biases using the same input $[x_1, x_2] = [1, 2]$:

1. Calculate activations in the hidden layer:

$$a_1^{(1)} = w_1 \cdot x_1 + w_2 \cdot x_2 + b_1 = 0.9566 \cdot 1 + 0.9132 \cdot 2 + (-0.0434) = 2.7396$$

$$a_2^{(1)} = w_3 \cdot x_1 + w_4 \cdot x_2 + b_2 = 0.9566 \cdot 1 + 0.9132 \cdot 2 + (-0.0434) = 2.7396$$

2. Calculate output $\hat{y}$:

$$\hat{y}_{\text{new}} = 0.7 \cdot 2.7396 + 0.7 \cdot 2.7396 + (-0.1) = 1.91772 + 1.91772 - 0.1 = 3.73544$$

3. Calculate the new loss:

$$L_{\text{new}} = (y - \hat{y}_{\text{new}})^2 = (1 - 3.73544)^2 = (-2.73544)^2 = 7.4826$$

After updating the weights and biases we have reduced the loss from 25 to ≈ 7.5

This completes the step-by-step backpropagation for this simple network, using only one example. The process can be repeated iteratively, taking one example after the other, updating the weights and biases after every iteration until the loss function is minimized to an acceptable level.

However, updating weights and biases per example can be inefficient for several reasons:

1. **High Variability:** The gradient estimates can have high variability which can cause the model to jump around rather than converge smoothly.
2. **Slow Convergence:** The model might require many iterations to reach a satisfactory solution, especially if the data contains a lot of outliers.
3. **Computational Inefficiency:** The model has to perform many small updates, which increases the overhead and reduces the efficiency of the training process.
4. **Risk of Overfitting:** Since the model is updated with every single example, there is a higher risk of overfitting to specific data points, especially if the data is not representative of the overall distribution. This can lead to poor generalization

performance on unseen data.

"The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient" [17].

4.6 Stochastic Gradient Descent (SGD)

"Nearly all of deep learning is powered by one very important algorithm: stochastic gradient descent (SGD) [which is] an extension of the gradient descent algorithm" [17].

While per-example updates can be inefficient and slow, Stochastic Gradient Descent improves the training process by updating weights and biases based on minibatches, leading to faster convergence, better generalization, and more efficient computation.

Typically, the loss function breaks down into a sum of individual loss functions over the training examples. For example, the negative conditional log-likelihood of the training data can be written as:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}}[L(x,y,\theta)] = \frac{1}{m}\sum_{i=1}^{m} L(x^{(i)},y^{(i)},\theta)$$

where L is the per-example loss function. For these additive loss functions, the gradient descent method requires that we compute the following:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x^{(i)},y^{(i)},\theta)$$

The computational cost of this process is $O(m)$. As the size of the training set increases to billions of examples, the time required for a single gradient step becomes significantly long.

SGD offers an efficient solution by approximating the gradient using a small sample of data. In each iteration of the algorithm, a minibatch $B = \{x^{(1)}, \ldots, x^{(m')}\}$ is uniformly sampled from the training set. The size of the minibatch m' is usually kept small, typically ranging from 1 to a few hundred examples. Importantly, m' remains constant even as the training set size m increases. This allows the algorithm to handle large training sets with billions of examples by computing updates based on just a small subset of data.

The gradient estimate is calculated as:

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

using examples from the minibatch B. The SGD algorithm then updates the parameters by moving in the direction of the estimated gradient:

$$\theta \leftarrow \theta - \eta g$$

where η is the learning rate.

Stochastic Gradient Descent (SGD) offers a more efficient approach to training by addressing the issues associated with per-example updates:

1. **Batch Updates:** Instead of updating the weights and biases after each individual example, SGD changes the model parameters based on a relatively small, randomly selected subset of the training data called a *minibatch*. This reduces the noise in the gradient estimates, leading to more stable and consistent updates.
2. **Faster Convergence:** By using minibatches, SGD reduces the variability in the gradient estimates, leading to smoother and faster convergence. Although the

updates are still somewhat noisy compared to full-batch gradient descent, they are more efficient than per-example updates and help the model converge more quickly.

3. **Computational Efficiency:** By processing minibatches, SGD reduces the computational overhead and makes better use of parallel processing capabilities, speeding up the training process.
4. **Better Generalization:** The inherent noise in the gradient estimates from minibatches helps the model escape local minima and explore a larger portion of the parameter space. This stochasticity can lead to better generalization on unseen data, as the model is less likely to overfit to specific examples.

"The optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function quickly enough to be useful." [17].

4.6.1 Example of Stochastic Gradient Descent

Let's consider the same neural network used in the backpropagation example, with 2 inputs, 1 hidden layer with 2 neurons, and 1 output $y = 3x_1 - x_2$.

Let's suppose we have a minibatch containing 3 examples:

- Example 1: $[x_1, x_2] = [1, 1]$
- Example 2: $[x_1, x_2] = [2, 3]$
- Example 3: $[x_1, x_2] = [1, 3]$

4.6.1.1 Forward Pass for Each Example

Example 1: $[x_1, x_2] = [1, 1]$

$$a_1^{(1)} = w_1 \cdot 1 + w_2 \cdot 1 + b_1 = 1 + 1 + 0 = 2$$

$$a_2^{(1)} = w_3 \cdot 1 + w_4 \cdot 1 + b_2 = 1 + 1 + 0 = 2$$

$$\hat{y} = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 1 \cdot 2 + 1 \cdot 2 + 0 = 4$$

Example 2: $[x_1, x_2] = [2, 3]$

$$a_1^{(1)} = w_1 \cdot 2 + w_2 \cdot 3 + b_1 = 2 + 3 + 0 = 5$$

$$a_2^{(1)} = w_3 \cdot 2 + w_4 \cdot 3 + b_2 = 2 + 3 + 0 = 5$$

$$\hat{y} = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 1 \cdot 5 + 1 \cdot 5 + 0 = 10$$

Example 3: $[x_1, x_2] = [1, 3]$

$$a_1^{(1)} = w_1 \cdot 1 + w_2 \cdot 3 + b_1 = 1 + 3 + 0 = 4$$

$$a_2^{(1)} = w_3 \cdot 1 + w_4 \cdot 3 + b_2 = 1 + 3 + 0 = 4$$

$$\hat{y} = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 1 \cdot 4 + 1 \cdot 4 + 0 = 8$$

4.6.1.2 Loss Calculation for Each Example

We will use MSE as our loss function and so we will calculate the loss per example and then compute the average.

Example 1:

$$y = 3(1) - 1 = 2, \quad \hat{y} = 4$$
$$L_1 = (2 - 4)^2 = (-2)^2 = 4$$

Example 2:

$$y = 3(2) - 3 = 3, \quad \hat{y} = 10$$
$$L_2 = (3 - 10)^2 = (-7)^2 = 49$$

Example 3:

$$y = 3(1) - 3 = 0, \quad \hat{y} = 8$$

$$L_3 = (0-8)^2 = (-8)^2 = 64$$

Total Loss:

$$L = \frac{1}{3}\sum_{i=1}^{3} L_i = \frac{1}{3}(4+49+64) = \frac{117}{3} = 39$$

4.6.1.3 Backward Pass and Gradient Calculation

We will now calculate the gradients for the weights and biases in the network based on the loss from each example. We will then average these gradients to perform the update.

Gradient Calculation for Output Layer

Example 1:

$$\frac{\partial L_1}{\partial \hat{y}_1} = -2(y - \hat{y}) = -2(2-4) = 4$$
$$\frac{\partial L_1}{\partial w_5} = \frac{\partial L_1}{\partial \hat{y}_1} \cdot a_1^{(1)} = 4 \cdot 2 = 8$$
$$\frac{\partial L_1}{\partial w_6} = \frac{\partial L_1}{\partial \hat{y}_1} \cdot a_2^{(1)} = 4 \cdot 2 = 8$$
$$\frac{\partial L_1}{\partial b_3} = \frac{\partial L_1}{\partial \hat{y}_1} = 4$$

Example 2:

$$\frac{\partial L_2}{\partial \hat{y}_2} = -2(y - \hat{y}) = -2(3-10) = 14$$
$$\frac{\partial L_2}{\partial w_5} = \frac{\partial L_2}{\partial \hat{y}_2} \cdot a_1^{(1)} = 14 \cdot 5 = 70$$
$$\frac{\partial L_2}{\partial w_6} = \frac{\partial L_2}{\partial \hat{y}_2} \cdot a_2^{(1)} = 14 \cdot 5 = 70$$
$$\frac{\partial L_2}{\partial b_3} = \frac{\partial L_2}{\partial \hat{y}_2} = 14$$

Example 3:

$$\frac{\partial L_3}{\partial \hat{y}_3} = -2(y - \hat{y}) = -2(0 - 8) = 16$$
$$\frac{\partial L_3}{\partial w_5} = \frac{\partial L_3}{\partial \hat{y}_3} \cdot a_1^{(1)} = 16 \cdot 4 = 64$$
$$\frac{\partial L_3}{\partial w_6} = \frac{\partial L_3}{\partial \hat{y}_3} \cdot a_2^{(1)} = 16 \cdot 4 = 64$$
$$\frac{\partial L_3}{\partial b_3} = \frac{\partial L_3}{\partial \hat{y}_3} = 16$$

4.6.1.4 Average Gradient and Weight Update

We average the gradients for w_5, w_6, and b_3 over all three examples:

$$\frac{\partial L}{\partial w_5} = \frac{8 + 70 + 64}{3} = \frac{142}{3} \approx 47.33$$
$$\frac{\partial L}{\partial w_6} = \frac{8 + 70 + 64}{3} = \frac{142}{3} \approx 47.33$$
$$\frac{\partial L}{\partial b_3} = \frac{4 + 14 + 16}{3} = \frac{34}{3} \approx 11.33$$

Using the gradient descent update rule:

$$w_5 \leftarrow w_5 - 0.01 \cdot 47.33 \approx 1 - 0.4733 = 0.5267$$

$$w_6 \leftarrow w_6 - 0.01 \cdot 47.33 \approx 1 - 0.4733 = 0.5267$$

$$b_3 \leftarrow b_3 - 0.01 \cdot 11.33 \approx 0 - 0.1133 = -0.1133$$

4.6.1.5 New Loss Calculation

Example 1: $[x_1, x_2] = [1, 1]$

$$a_1^{(1)} = w_1 \cdot 1 + w_2 \cdot 1 + b_1 = 1 + 1 + 0 = 2$$
$$a_2^{(1)} = w_3 \cdot 1 + w_4 \cdot 1 + b_2 = 1 + 1 + 0 = 2$$
$$\hat{y} = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 0.5267 \cdot 2 + 0.5267 \cdot 2 - 0.1133 = 1.9935$$
$$y = 3(1) - 1 = 2, \quad \hat{y} = 1.9935$$
$$L_1 = 0.00004225$$

Example 2: $[x_1, x_2] = [2, 3]$

$$a_1^{(1)} = w_1 \cdot 2 + w_2 \cdot 3 + b_1 = 2 + 3 + 0 = 5$$

$$a_2^{(1)} = w_3 \cdot 2 + w_4 \cdot 3 + b_2 = 2 + 3 + 0 = 5$$

$$\hat{y} = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 0.5267 \cdot 5 + 0.5267 \cdot 5 - 0.1133 = 5.1537$$

$$y = 3(2) - 3 = 3, \quad \hat{y} = 5.1537$$

$$L_2 = 4.6384$$

Example 3: $[x_1, x_2] = [1, 3]$

$$a_1^{(1)} = w_1 \cdot 1 + w_2 \cdot 3 + b_1 = 1 + 3 + 0 = 4$$

$$a_2^{(1)} = w_3 \cdot 1 + w_4 \cdot 3 + b_2 = 1 + 3 + 0 = 4$$

$$\hat{y} = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 0.5267 \cdot 4 + 0.5267 \cdot 4 - 0.1133 = 4.1003$$

$$y = 3(1) - 3 = 0, \quad \hat{y} = 4.1003$$

$$L_3 = 16.81246$$

Total Loss:

$$L = \frac{1}{3} \sum_{i=1}^{3} L_i = \frac{1}{3}(L_1 + L_2 + L_3) \approx 7.15$$

4.6.1.6 Hidden Layer Weight and Bias Update

Now, let's use the updated output layer values to compute the gradients for the hidden layer and update the weights and biases similarly.

Example 1:

$$\frac{\partial L_1}{\partial \hat{y}_1} = -2(y - \hat{y}) = -2(2 - 1.9935) = -0.013$$

$$\frac{\partial L_1}{\partial a_1^{(1)}} = \frac{\partial L_1}{\partial \hat{y}_1} \cdot w_5 = -0.013 \cdot 0.5267 \approx -0.0068471$$

$$\frac{\partial L_1}{\partial a_2^{(1)}} = \frac{\partial L_1}{\partial \hat{y}_1} \cdot w_6 = -0.013 \cdot 0.5267 \approx -0.0068471$$

$$\frac{\partial L_1}{\partial w_1} = \frac{\partial L_1}{\partial a_1^{(1)}} \cdot x_1 = -0.0068471 \cdot 1 \approx -0.0068472$$

$$\frac{\partial L_1}{\partial w_2} = \frac{\partial L_1}{\partial a_1^{(1)}} \cdot x_2 = -0.0068471 \cdot 1 \approx -0.0068471$$

$$\frac{\partial L_1}{\partial b_1} = \frac{\partial L_1}{\partial a_1^{(1)}} = -0.0068471$$

Example 2:

$$\frac{\partial L_2}{\partial \hat{y}_2} = -2(y - \hat{y}) = -2(3 - 5.1537) = 4.3074$$

$$\frac{\partial L_2}{\partial w_1} \approx 4.537$$

$$\frac{\partial L_2}{\partial w_2} \approx 6.806$$

$$\frac{\partial L_2}{\partial b_1} = \approx 2.268$$

Example 3:

$$\frac{\partial L_3}{\partial \hat{y}_3} = -2(y - \hat{y}) = -2(0 + 4.1003) = 8.2006$$

$$\frac{\partial L_2}{\partial w_1} \approx 4.319$$

$$\frac{\partial L_3}{\partial w_2} \approx 12.958$$

$$\frac{\partial L_3}{\partial b_1} = \approx 4.319$$

Average Gradient for Hidden Layer

$$\frac{\partial L}{\partial w_1} = \frac{\frac{\partial L_1}{\partial w_1} + \frac{\partial L_2}{\partial w_1} + + \frac{\partial L_3}{\partial w_1}}{3} \approx 2.95$$
$$\frac{\partial L}{\partial w_2} = \frac{\frac{\partial L_1}{\partial w_2} + \frac{\partial L_2}{\partial w_2} + + \frac{\partial L_3}{\partial w_2}}{3} \approx 6.59$$
$$\frac{\partial L}{\partial b_1} = \frac{\frac{\partial L_1}{\partial b_1} + \frac{\partial L_2}{\partial b_1} + + \frac{\partial L_3}{\partial b_1}}{3} \approx 2.19$$

Using the gradient descent update rule:

$$w_1 \leftarrow w_1 - 0.01 \cdot 2.95 \approx 1 - 0.007823 = 0.9705$$

$$w_2 \leftarrow w_2 - 0.01 \cdot 6.59 \approx 1 - 0.0224707 = 0.9341$$

$$b_1 \leftarrow b_1 - 0.01 \cdot 2.19 \approx 0 - 0.007553 = -0.0219$$

Similarly we will do the calculations for w_3, w_4, and b_2 and we will get:

$$w_3 \leftarrow w_1 - 0.01 \cdot 2.95 \approx 1 - 0.007823 = 0.9705$$

$$w_4 \leftarrow w_2 - 0.01 \cdot 6.59 \approx 1 - 0.0224707 = 0.9341$$

$$b_2 \leftarrow b_1 - 0.01 \cdot 2.19 \approx 0 - 0.007553 = -0.0219$$

4.6.1.7 Forward Pass with Updated Weights and Biases

With these new weights lets do a forward pass for each of our examples and compute the Loss so we verify that it has indeed decrease.

News weights and biases:

$$w_1 = 0.9705, \qquad w_2 = 0.9341, \qquad b_1 = -0.0219$$
$$w_3 = 0.9705, \qquad w_4 = 0.9341, \qquad b_2 = -0.0219$$
$$w_5 = 0.5267, \qquad w_6 = 0.5267, \qquad b_3 = -0.1133$$

Example 1: $[x_1, x_2] = [1, 1]$

Hidden Layer Activations:

$$a_1^{(1)} = w_1 \cdot 1 + w_2 \cdot 1 + b_1 = 0.9705 \cdot 1 + 0.9341 \cdot 1 + (-0.0219) = 1.8827$$

$$a_2^{(1)} = w_3 \cdot 1 + w_4 \cdot 1 + b_2 = 0.9705 \cdot 1 + 0.9341 \cdot 1 + (-0.0219) = 1.8827$$

Output:

$$\hat{y}_1 = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 0.5267 \cdot 1.8827 + 0.5267 \cdot 1.882 + (-0.1133) = 1.8699$$

Target:

$$y_1 = 3 \cdot 1 - 1 = 2$$

Loss:

$$L_1 = (y_1 - \hat{y}_1)^2 = (2 - 1.8699)^2 = 0.0169$$

Example 2: $[x_1, x_2] = [2, 3]$

Hidden Layer Activations:

$$a_1^{(1)} = w_1 \cdot 2 + w_2 \cdot 3 + b_1 = 0.9705 \cdot 2 + 0.9341 \cdot 3 + (-0.0219) = 4.722$$

$$a_2^{(1)} = w_3 \cdot 2 + w_4 \cdot 3 + b_2 = 0.9705 \cdot 2 + 0.9341 \cdot 3 + (-0.0219) = 4.722$$

Output:

$$\hat{y}_2 = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 0.5267 \cdot 4.722 + 0.5267 \cdot 4.722 + (-0.1133) = 4.8603$$

Target:

$$y_2 = 3 \cdot 2 - 3 = 3$$

Loss:

$$L_2 = (y_2 - \hat{y}_2)^2 = (3 - 4.8603)^2 = 3.4607$$

Example 3: $[x_1, x_2] = [1, 3]$

Hidden Layer Activations:

$$a_1^{(1)} = w_1 \cdot 1 + w_2 \cdot 3 + b_1 = 0.9705 \cdot 1 + 0.9341 \cdot 3 + (-0.0219) = 3.751$$

$$a_2^{(1)} = w_3 \cdot 1 + w_4 \cdot 3 + b_2 = 0.9705 \cdot 1 + 0.9341 \cdot 3 + (-0.0219) = 3.751$$

Output:

$$\hat{y}_3 = w_5 \cdot a_1^{(1)} + w_6 \cdot a_2^{(1)} + b_3 = 0.5267 \cdot 3.751 + 0.5267 \cdot 3.751 + (-0.1133) = 3.838$$

Target:

$$y_3 = 3 \cdot 1 - 3 = 0$$

Loss:

$$L_3 = (y_3 - \hat{y}_3)^2 = (0 - 3.838)^2 = 14.73$$

Average Loss

The average loss over the three examples is calculated as:

$$L_{\text{avg}} = \frac{1}{3}(L_1 + L_2 + L_3)$$

$$\approx 6.0693$$

This is a significant reduction from the initial loss of 39 calculated with the initialized weights and biases.

4.7 Computational Limitations

4.7.1 Overfitting and Underfitting

Recall in Chapter 3, we discussed what is meant by a model's capacity and how its capacity is affected by the hypothesis space. We also stated that any model will perform best, if it's capacity matches the complexity of the task.

"Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with." [17]. Models with lower capacity than needed will fail to solve the given tasks and result in under-fitting, while models with unnecessarily large capacity may overfit. Figure 4.5 illustrates this concept. "We compare a linear, quadratic, and degree-9 predictor on a problem where the true underlying function is quadratic" [17] . The linear function fails to represent the relationship between input and output, resulting in underfitting. The degree-9

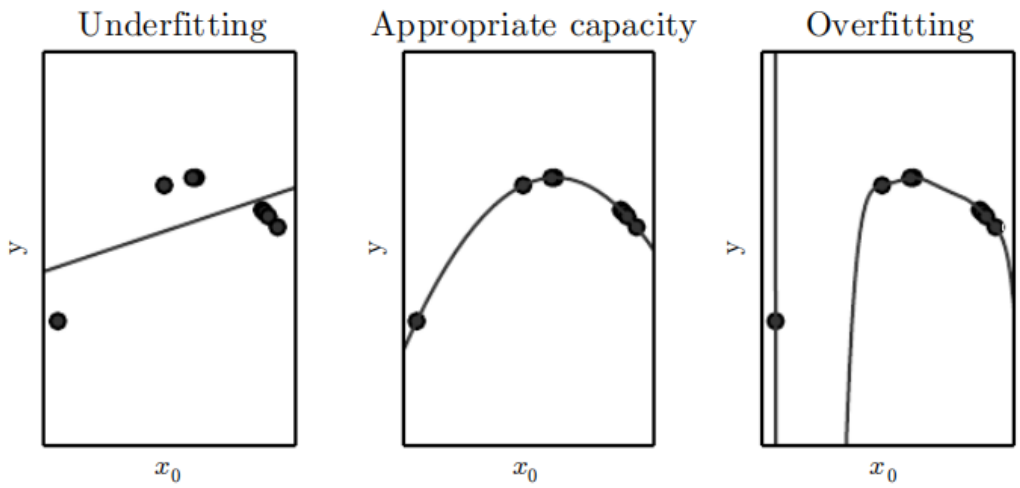

Figure 4.5: (Left) A Linear Function: Underfitting. (Center) A Quadratic Function Accuratley Captures the Relationship. (Right) A Polynomial of Degree 9: Overfitting.

predictor can represent the correct function, but it can also fit many other functions that pass exactly through the training points, as there are more parameters than training examples. This situation makes it unlikely to choose a solution that generalizes well since there are numerous vastly different solutions resulting in overfitting.

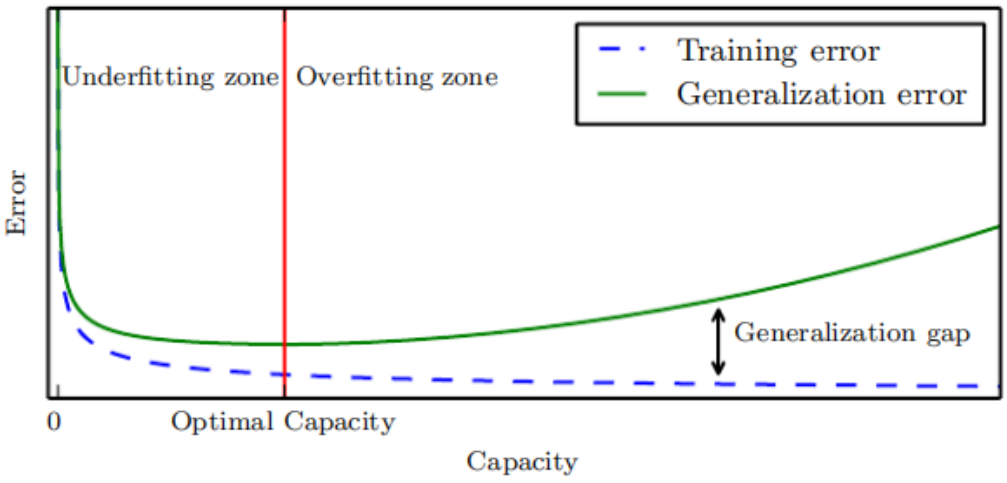

Figure 4.6: Relationship between Capacity and Error

Looking at the figure 4.6, we can see that at the low-capacity end of the graph (left), the value of the training error is high, as well as the generalization error which corresponds to the underfitting scenario. When the capacity is increased, the training error is reduced, but the difference between training error and generalization error widens. This widening difference becomes more significant than the reduction in training error, leading to the overfitting scenario, where the model's capacity surpasses the optimal level. [17]

4.7.2 Vanishing and Exploding Gradients

The vanishing and exploding gradient are challenges that can arise when training deep neural networks using gradient descent, affecting their ability to learn effectively.

Vanishing Gradient Problem

The vanishing gradient happens when the gradients of the loss function with respect to the model parameters become very small as they are propagated backward through the layers of the network. This makes it difficult for the weights in the earlier layers to update effectively, which slows down the learning process.

Example: Consider a neural network with the sigmoid function as activation. During backpropagation, the gradients are multiplied by the derivatives of the activation functions at each layer. Since the derivative of the sigmoid function is always less than 1, multiplying many such small numbers together causes the gradient to shrink exponentially as it moves backward through the layers. This can result in gradients that are so small that the weights stop updating, effectively halting learning in the early layers of the network.

Exploding Gradient Problem

The exploding gradient issue happens when the gradients increase exponentially as they move backward through the network. This can cause the model parameters to update with excessively large values, leading to unstable behavior and divergence

during training.

Example: Imagine a deep neural network with weights that are initialized to relatively large values. During backpropagation, the gradients might be multiplied by these large weights, causing the gradients to grow exponentially. As a result, the weight updates become excessively large, and the model's parameters can overflow, causing numerical instability and making it impossible to learn anything useful from the data.

To mitigate these issues, several techniques can be employed:

- **Weight Initialization:** In 2010 Glorot et al. [16] introduced the **Xavier** or **Glorot** initialization method, which addresses the issue of vanishing and exploding gradients that often occur when training deep neural networks (we will explore it more depth in a later section). By initializing weights from a distribution with zero mean:

$$W \sim \mathscr{U}\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right)$$

  where $\mathscr{U}$ denotes the uniform distribution.

  And a variance of:

$$\text{Var}(W) = \frac{2}{n_{\text{in}}+n_{\text{out}}}$$

  where n_{in} is the number of input units in the weight tensor, and n_{out} is the number of output units, the Xavier initialization method ensures that the variance of the activations remains consistent across layers.

  In 2012, He et al. introduced a new method for initializing weights specifically designed for rectified linear unit (ReLU) activation functions, known as the "He initialization" or "Kaiming initialization" [22].

They proposed initializing the weights from a scaled normal distribution to address the issue of vanishing and exploding gradients in very deep networks. The weights are drawn from a normal distribution with a mean of 0 and a variance of $\frac{2}{n}$, where n is the number of input units to a neuron.

The authors proved through extensive experiments that networks initialized using He initialization converge faster and achieve higher accuracy compared to other initialization methods.

Using methods like Xavier (Glorot) initialization or He initialization to set the initial weights in a way that keeps the gradients from becoming too small or too large.

- **Activation Functions:** Using activation functions like ReLU, which do not squash the input values into a small range as much as sigmoid or tanh functions, thus preserving the gradient magnitude better.
- **Gradient Clipping:** Restricting the gradients to a certain range during backpropagation to prevent them from exploding.

4.7.3 Computation Complexity and Data Requirements

While the thesis primarily focuses on the theoretical aspects, it is important to acknowledge that deep neural networks also face practical limitations such as computational complexity and data requirements.

DNNs require substantial computational resources for training, involving numerous matrix multiplications and gradient calculations. High-performance hardware like GPUs or TPUs is often necessary, making the process time-consuming, financially and energetically costly. Additionally, memory requirements are significant, and managing these resources efficiently can be challenging. Inference time can also be a

concern, particularly for real-time applications, as high latency may impact performance.

Moreover, deep neural networks typically need vast amounts of high-quality data to train effectively, making data collection and annotation a significant challenge. The quality and diversity of the data are crucial for optimal model performance, necessitating extensive preprocessing to clean and normalize the data. Additionally, handling imbalanced datasets and ensuring representativeness are critical tasks that add to the complexity of the data preparation process.

Chapter 5

ADVANCED NEURAL NETWORKS

Now that we have a solid foundation of how feed-forward neural networks are built and trained we will explore more advanced architectures and structures. The fundamental training method (gradient descent, backpropagation) is the same for as in FNNs as the difference lies in how these methods are applied to the specific architectures.

We will first present a general classification of DNN models and then dive into the building blocks of CNNs and RNNs, in order to get an idea of what distinguishes these advanced models from FNNs. In the final section will present the idea behind Transformer models and the main concerns around their reliability, given that we know how these models actually "learn".

5.1 Variety of DNN architectures

Whether it's image recognition, language processing, or time-series analysis, there's a ANN designed to better perform in that specific area. These networks can be broadly categorized into four types based on their structure and how they process information:

1. **Feed-forwardward Networks (FNN):**
    - Multilayer Perceptron (MLP)
    - Convolutional Neural Network (CNN)
    - Radial Basis Function Network (RBFN)
    - Autoencoder
    - Generative Adversarial Network (GAN)

2. **Recurrent Networks:**
    - Simple Recurrent Neural Network (RNN)
    - Long Short-Term Memory (LSTM)
    - Gated Recurrent Unit (GRU)
3. **Attention-based Networks:**
    - Transformer Network
4. **Self-Organizing Networks:**
    - Self-Organizing Map (SOM)

This list is by no means comprehensive as there are also other specialized networks which do not neatly fit in one of the above categories. The purpose of presenting this list is to highlight the diversity and variety of DNN models. In the next sections we will present more information about CNNs, RNNs and Transformers. These networks have more complex structures and use other types of artificial neurons which we will explore. There are much more types of artificial neurons which will not be covered in this chapter (see Appendix A).

5.2 Convolutional Neural Networks (CNN)

With FNN we saw input cells, hidden cells and output cells. These are the basic type of artificial neurons or cells in a neural network.

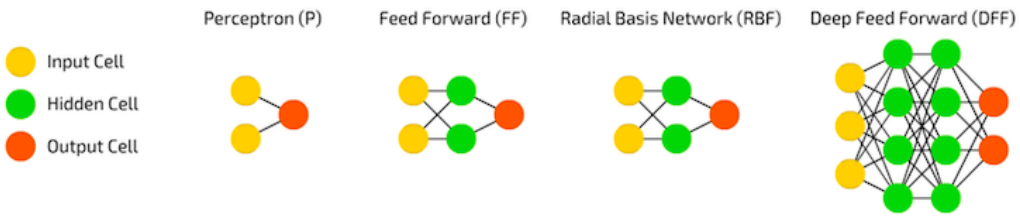

Figure 5.1: Simple NN Structures

There are other types of cells that allow for the creation of convolutional neural networks. These cells are called kernel and convolution cells which we will examine in the next section.

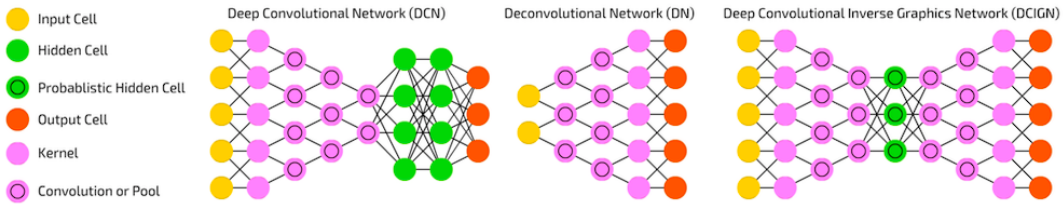

Figure 5.2: Convolutional Neural Networks

Convolutional Neural Network is one of the most widely used types of feed-forward neural network [20]. As opposed to MLPs, a CNN has hidden layers called convolutional layers. Just like the hidden layers in MLPs, these convolutional layers also transform the input and produce an output, however, this transformation is not done through a matrix multiplication of weights, bias vector addition and an activation function, it is done through a convolution operation. This operation allows the network to pick up on patterns which is very useful in image analysis.

5.2.1 Motivation

For inputs with high dimensions that are spatially related, using fully connected neural networks (FCs) often leads to an excessive number of parameters [20]. In image classification tasks, for instance, adjacent pixels frequently share information, and this spatial relationship should be considered in the network architecture. Consequently, it makes sense to design NNs with local receptive fields that gather information from nearby inputs. Additionally, in image processing, effective classifiers remain consistent under various transformations, such as image translation or rotation. Therefore, it is logical to integrate these invariances directly into the

network architecture.

5.2.2 Convolution of Sequences (Discrete)

The nature of the convolution—whether it is continuous or discrete—depends on the domains of the functions involved:

- **Continuous Convolution:** If the functions f and g are defined over continuous domains, the convolution is represented as an integral. This type of convolution is appropriate for continuous-time signals where the functions can take any value within a given range.
- **Discrete Convolution:** If the functions a and b are defined over discrete domains, the convolution is represented as a sum. This is suitable for discrete-time signals or data sequences where the functions take values at specific, discrete points.

In CNNs, discrete convolution is used. CNNs apply discrete convolution operations to input data, typically images, to extract features and learn patterns. The use of discrete convolution allows CNNs to effectively process and analyze spatial hierarchies in data by learning filters (or kernels) that respond to specific visual features.

Definition 5.1 (Convolution of Sequences): The discrete convolution of two sequences a and b is a summation that produces a third sequence c. It is defined as:

$$c[k] = \sum_{i=0}^{m-1} a[i] \cdot b[k-i]$$

for $k = 0, 1, \ldots, m+n-2$, where m and n are the lengths of a and b, respectively. If the index $k-i$ is outside the bounds of b, the value of $b[k-i]$ is taken to be zero.

Example of Discrete Convolution

For $a = (a_0, a_1, a_2)$ and $b = (b_0, b_1, b_2)$, we compute the convolution as:

$$c[0] = a_0 \cdot b_0$$

$$c[1] = a_0 \cdot b_1 + a_1 \cdot b_0$$

$$c[2] = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0$$

$$c[3] = a_1 \cdot b_2 + a_2 \cdot b_1$$

$$c[4] = a_2 \cdot b_2$$

Thus, the resulting list c is:

$$c = (c[0], c[1], c[2], c[3], c[4])$$

Substituting the expressions, we get:

$$c = (a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, a_1b_2 + a_2b_1, a_2b_2)$$

5.2.3 Visual Representation

Here is a visual illustration of discrete convolution by Grant Sanderson [62] the creator of "3Blue1Brown" education chanell. In this example we will let $a = (1,2,3)$ and $b = (4,5,6)$. We can visualize the computation of the convolution by flipping the list b, putting it to the bottom left of the list a, such that a_0 and b_0 are aligned.

To get $c[0]$ we simply multiply a_0 and b_0:

$$c[0] = a_0 \cdot b_0 = 1 \cdot 4 = 4$$

Then we slide the b list one step towards the right so that a_0 and b_1 are aligned, as are a_1 and b_0. To get $c[1]$, we multiply the top number with the bottom one, and add the two results together:

$$c[1] = 1 \cdot 5 + 2 \cdot 4 = 13$$

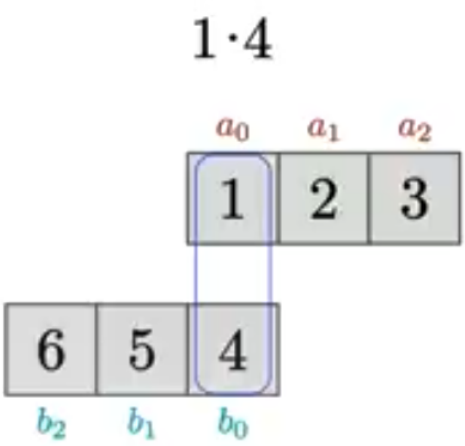

Figure 5.3: We Start with the First Terms a_0 and b_0 aligned

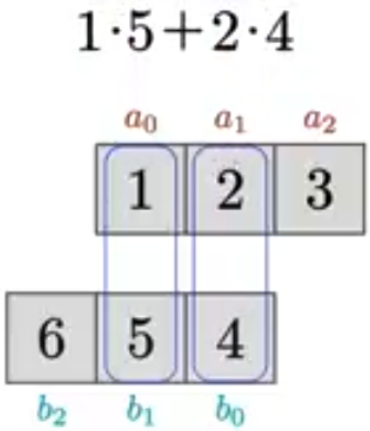

Figure 5.4: Aligning a_0 and b_1

We slide the list once more to get the next value in the convolution sequence:

$$c[2] = 1 \cdot 6 + 2 \cdot 5 + 3 \cdot 4 = 28$$

$$c[3] = 2 \cdot 6 + 3 \cdot 5 = 27$$

$$c[4] = 3 \cdot 6 = 18$$

Thus, the resulting list c is:

$$c = (1,2,3) * (4,5,6) = (4,13,28,27,18)$$

5.2.4 Moving Average Analogy

This visual representation [62] helps to see the relation between convolution operation and calculating the moving average in a sequence. For instance if we let a be a sequence of numbers and b be a sequence that include probabilities that add up to 1. Suppose b has five probabilities that are all same such that:

$$b = [0.2, 0.2, 0.2, 0.2, 0.2]$$

Note: Flipping b in this example results in the same sequence, since the sequence is symmetric about the centre. If we do this convolution of the two lists using this sliding

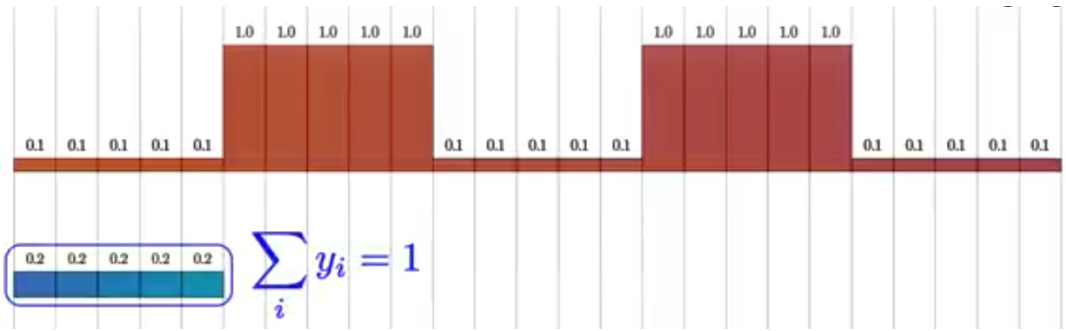

Figure 5.5: Convolution of a Large Sequence a (orange) with a Smaller Sequence b (blue)

method, once the smaller list of value b entirely overlaps with the bigger list a and we start computing the values after each step taken, what we will get at each iteration is

the value of the moving average of the sequence a. The process gives a smoothed out version of the original sequence.

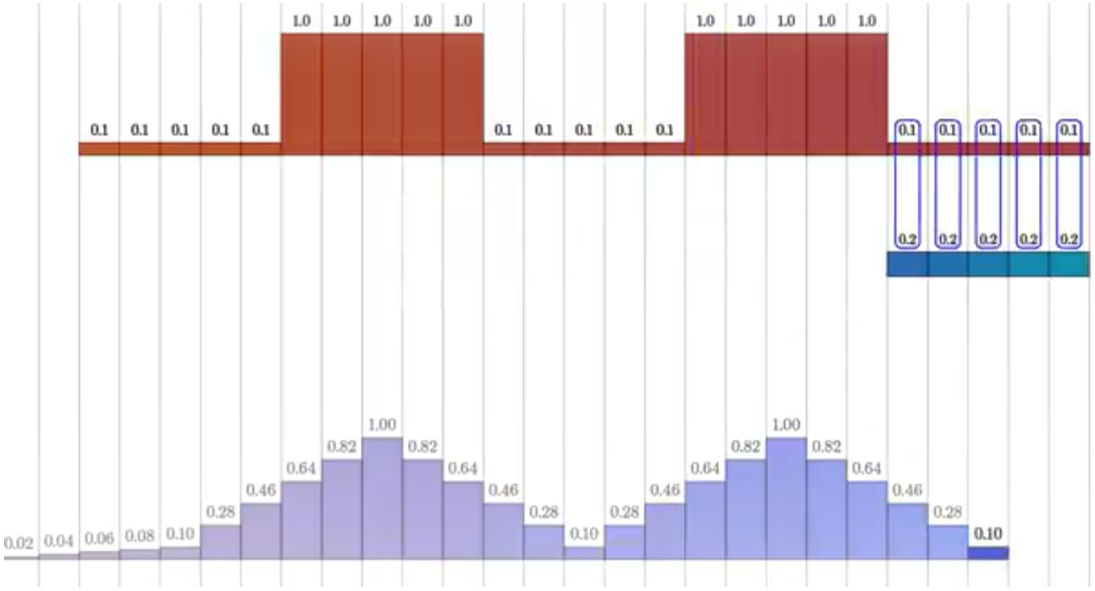

Figure 5.6: Moving Average of a with Equal Weights

If we pick different values for the sequence b, giving more weight to the central values, we would get a different moving average.

For instance if we let $b = [0.1, 0.2, 0.4, 0.2, 0.1]$, we will get this new graph: The idea of a moving average is to gather information from nearby inputs so that a particular "region" in the data is evaluated together. In the next section we will explore convolution matrices to see how they help in pattern recognition in images.

Definition 5.2 (Kernel): A **kernel** (or filter) is a small matrix used for blurring, sharpening, edge detection, and other image processing operations [17]. It is applied to an image by convoluting it with the kernel matrix. A kernel is typically a square matrix, such as:

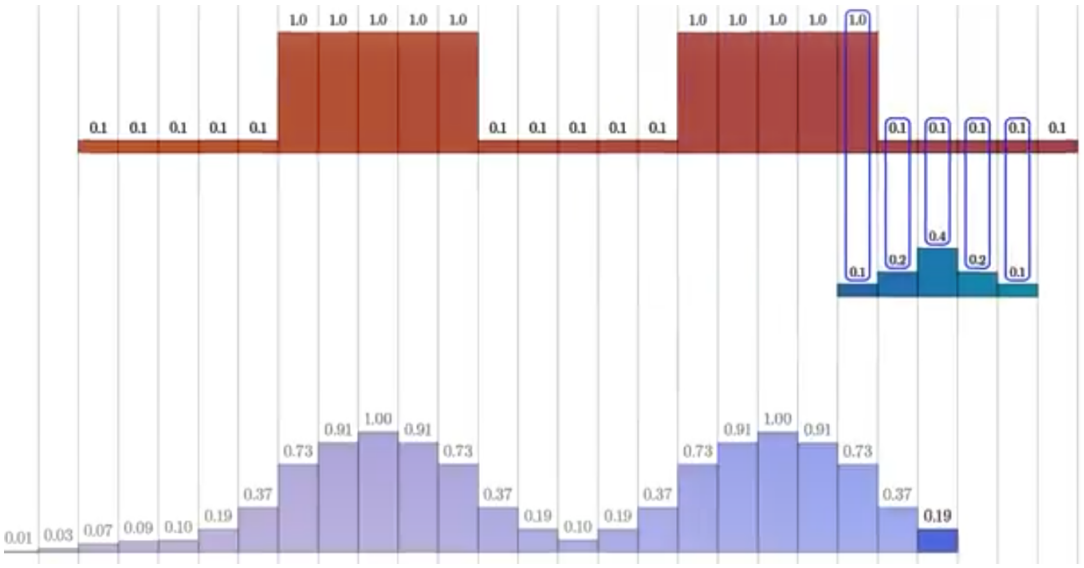

Figure 5.7: Moving Average of a with More Weight Given to the Central Values

$$K = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix}$$

where each element k_{ij} represents the weight of the kernel at position (i, j).

Smaller kernels are often preferred for capturing local features, while larger kernels can capture more global patterns [50].

Definition 5.3 (Convolution Matrix): A **convolution matrix** is the result of applying the kernel to an image (or another matrix) through the process of convolution [17] . The convolution of a kernel K with an input matrix I produces an output matrix O where each element is computed as the sum of element-wise products of the kernel and the corresponding submatrix of the input.

For an input matrix I of size $m \times n$ and a kernel K of size $p \times p$, the output matrix O will have dimensions $(m-p+1) \times (n-p+1)$. The element O_{ij} of the output matrix

is given by:

$$O_{ij} = \sum_{u=0}^{p-1} \sum_{v=0}^{p-1} K_{uv} \cdot I_{(i+u)(j+v)}$$

where i and j are the coordinates of the current position in the output matrix, and u and v iterate over the dimensions of the kernel.

5.2.5 Numerical Example

For an input matrix I of size 4×3 and a kernel K of size 2×2, the output matrix O will have dimensions $(4-2+1) \times (3-2+1) = 3 \times 2$. The element O_{ij} of the output matrix is given by:

$$O_{ij} = \sum_{u=0}^{1} \sum_{v=0}^{1} K_{uv} \cdot I_{(i+u)(j+v)}$$

where i and j are the coordinates of the current position in the output matrix, and u and v iterate over the dimensions of the kernel [17]. Let's consider an input matrix I and a kernel K as follows:

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$$K = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix}$$

We compute the convolution O as follows:

$$\begin{aligned} O[0,0] &= (1 \cdot 1) + (2 \cdot 0) + (4 \cdot 1) + (5 \cdot -1) \\ &= 1 + 0 + 4 - 5 \\ &= 0 \end{aligned}$$

$$
\begin{aligned}
O[0,1] &= (2\cdot 1)+(3\cdot 0)+(5\cdot 1)+(6\cdot -1)\\
&= 2+0+5-6\\
&= 1\\
O[1,0] &= (4\cdot 1)+(5\cdot 0)+(7\cdot 1)+(8\cdot -1)\\
&= 4+0+7-8\\
&= 3\\
O[1,1] &= (5\cdot 1)+(6\cdot 0)+(8\cdot 1)+(9\cdot -1)\\
&= 5+0+8-9\\
&= 4\\
O[2,0] &= (7\cdot 1)+(8\cdot 0)+(10\cdot 1)+(11\cdot -1)\\
&= 7+0+10-11\\
&= 6\\
O[2,1] &= (8\cdot 1)+(9\cdot 0)+(11\cdot 1)+(12\cdot -1)\\
&= 8+0+11-12\\
&= 7
\end{aligned}
$$

Thus, the resulting output matrix O is:

$$
O = \begin{bmatrix} 0 & 1 \\ 3 & 4 \\ 6 & 7 \end{bmatrix}
$$

5.2.6 Convolutionl Cell For Image Analysis

We can extend the moving average analogy, to see how a convolutional matrix can allow us to blur or sharpen an image, and also detect vertical and horizontal edges. In the figure 5.8, we applied a 3x3 kernel where all entries have the same value of 1/9, adding up to 1. At each iteration, we slide the kernel onto the image pixels and multiply each of these values by the corresponding pixel that it sits on top. When we

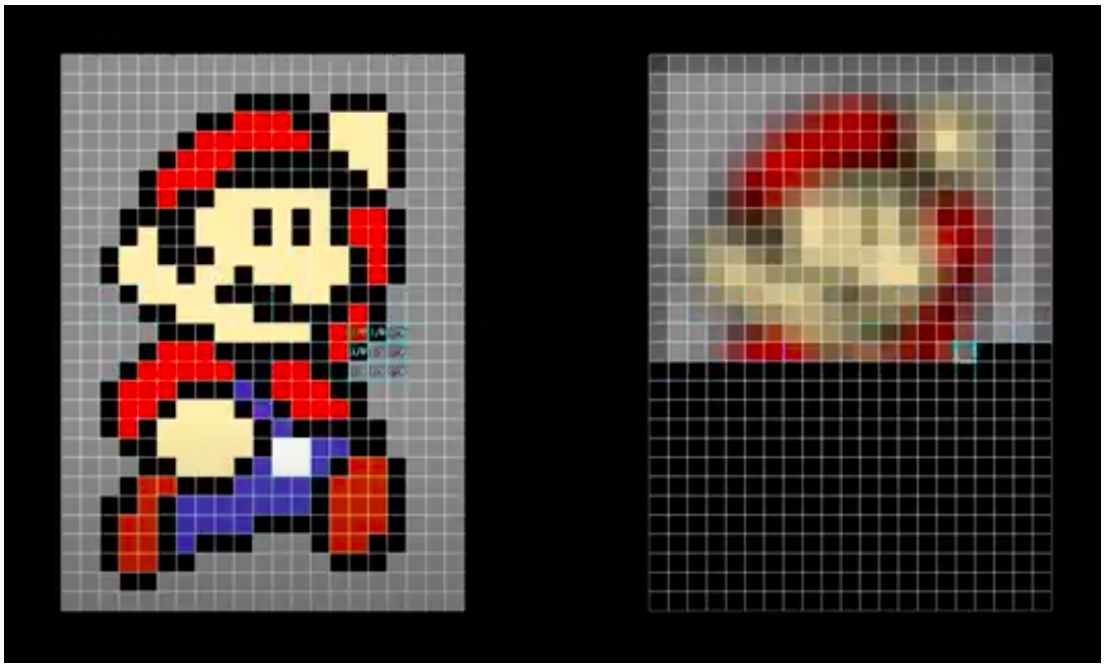

Figure 5.8: Blurring and Image with CNN

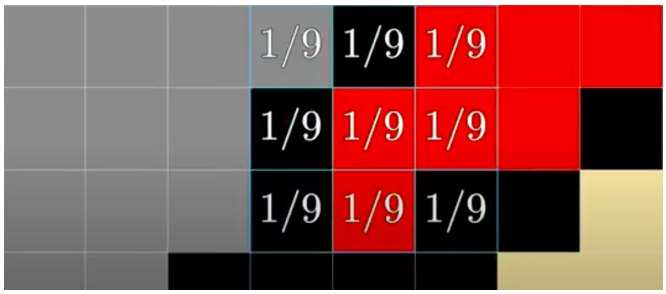

Figure 5.9: 3x3 Bluring Kernel

add these values together, it gives the average color of this particular region which corresponds to the outputted pixel (on the right of figure 5.8) This gives the blurring effect that we observe. Mathematically, we say that the image on the right is a convolution of the image on the left with the matrix:

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Now what happens if we use a different kernel, say:

$$\begin{bmatrix} \frac{1}{4} & 0 & -\frac{1}{4} \\ \frac{1}{5} & 0 & -\frac{1}{5} \\ \frac{1}{4} & 0 & -\frac{1}{4} \end{bmatrix}$$

Since some of the entries are negative, the values of the output can also be negative, and so if we decide to display the positive output values in blue and the negative values in red, we will be able to detect vertical edges. The rational is that in areas where there

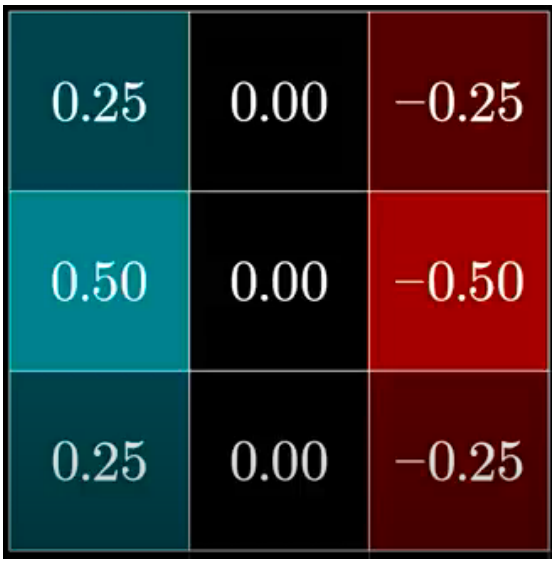

Figure 5.10: Visualizing the Kernel

is no major change of colors, the result will be close to 0, however, when there is a significant variation in the pixel value we will be able to see it. Similarly if we rotate

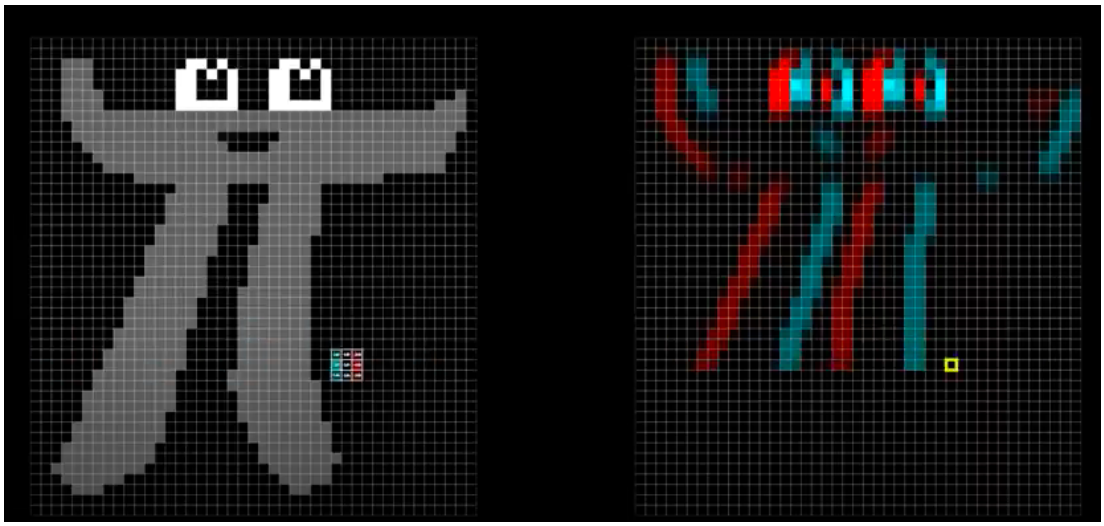

Figure 5.11: Detecting Vertical Edges [62]

this kernel around, we will be able to detect horizontal edges. This demonstrates how

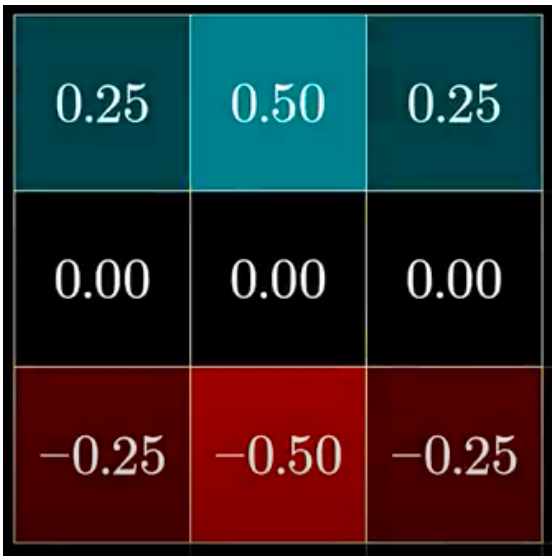

Figure 5.12: Rotating the Kernel

choosing a different kernel gives us a different image processing effect.

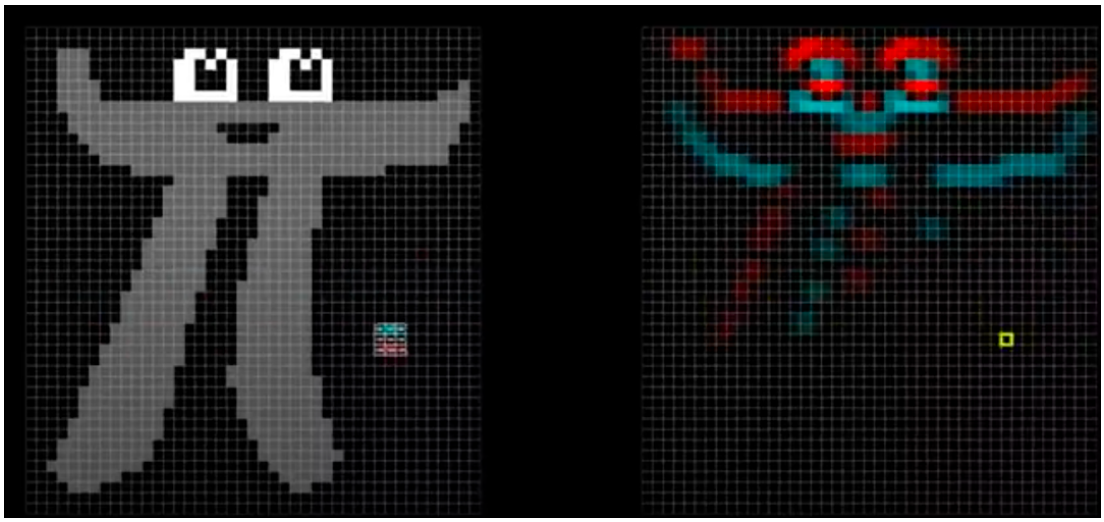

Figure 5.13: Detecting Horizontal Edges

5.3 Recurrent Neural Network (RNN)

So far, we have examined types of feed-forward networks, where information flows in one direction from input to output. We will now explore a new type of cells called recurrent cells, which constitute the building blocks of Recurrent Neural Networks (RNN), a non-feedforward type of model.

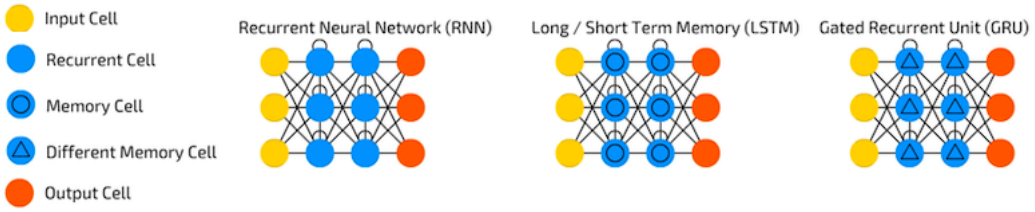

Figure 5.14: Recurrent Neural Networks

5.3.1 Motivation

Recurrent cells (or recurrent units) allow a network to have "memory" which is important in tasks involving sequences of data, such as words in a sentence or sentences in a paragraph. To understand a sentence for example, a network needs to remember the words that came before to make sense of the words that come after. In

other applications such as time series prediction, for a network to be able to predict future stock prices, if needs to remember past prices.

Recurrent cells provide a solution to this type of problems due to their looping structure that lets them process input data sequentially, while retaining information from previous steps in the sequence. This loop helps the network "remember" previous data points.

5.3.2 Recurrent Cell

In a recurrent cell, the output of a neuron is directly fed back to its own input [57].

$$y_i(n+1) = f_i(x_i(n) + w \cdot y_i(n))$$

where $y_i(n)$ is the output of neuron i at time step n, $x_i(n)$ is the input to neuron i at time step n, and w is the feedback weight [57].

The input signal is denoted as $x_j(n)$, the internal signal as $x'_j(n)$, and the output signal as $y_k(n)$, where n represents the discrete time variable. In the following figure, notice the different between the forward operator A and the feedback operator B.

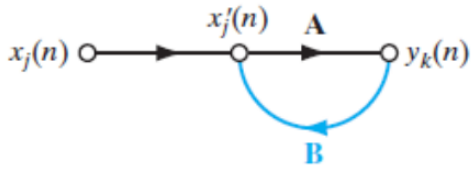

Figure 5.15: Elementary Feedback Signal Flow

For simplicity let $x_j(n) \rightarrow x'_j(n)$ be the identity operator. We will also suppose that the system is linear, i.e. there are only synaptic edges and the operators are also linear [57], we get :

$$y_k(n) = A[x'_j(n)] \quad \text{and} \quad x'_j(n) = x_j(n) + B[y_k(n)]$$

By combining the two equations and eliminating $x'_j(n)$, we derive:

$$y_k(n) = A[x_j(n) + B[y_k(n)]] = A[x_j(n)] + AB[y_k(n)]$$

Simplifying further, we obtain:

$$y_k(n) = \frac{A}{1 - AB}[x_j(n)]$$

5.3.3 Example of Elementary Feedback

Considering A as a fixed weight w, and B as the unit delay operator z^{-1} [57]:

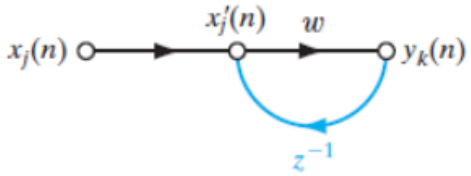

Figure 5.16: Elementary Feedback Example

$$y_k(n) = \frac{w}{1 - wz^{-1}}[x_j(n)] = w \sum_{l=0}^{\infty} w^l z^{-l}[x_j(n)]$$

Using the z-transform property $z^{-l}[x_j(n)] = x_j(n-l)$, we get:

$$y_k(n) = w \sum_{l=0}^{\infty} w^l x_j(n-l) = \sum_{l=0}^{\infty} w^{l+1} x_j(n-l)$$

We consider the system's stability based on the weight w:

- If $|w| < 1$, the output signal $y_k(n)$ converges, indicating the system is stable.
- For $|w| = 1$, $y_k(n)$ diverges. Here the system is unstable and the divergence is linear.
- If $|w| > 1$, $y_k(n)$ diverges. In this case, the system is unstable and the divergence

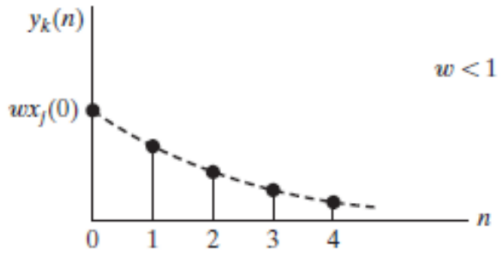

Figure 5.17: Values of $y_k(n)$ when $|w| < 1$

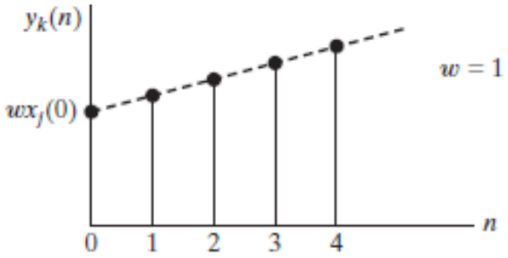

Figure 5.18: Values of $y_k(n)$ when $|w| = 1$

is exponential.

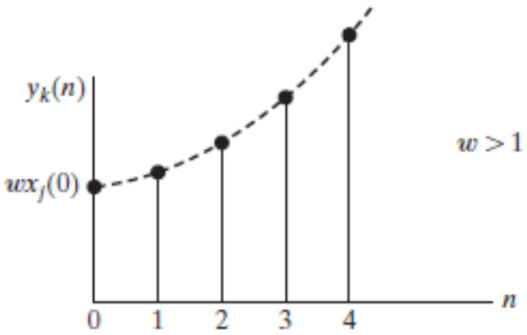

Figure 5.19: Values of $y_k(n)$ when $|w| > 1$

5.3.4 Feedback Replacement

Feedback mechanisms can often be replaced with feed-forwardward connections to simplify the network structure and maintain stability, especially when the weight w is less than 1 ($|w| < 1$). This ensures that the system remains stable and does not

diverge. For $|w| < 1$, if N is sufficiently large and $|w|$ small enough, the output can be approximated by a finite sum:

$$y_k(n) \approx \sum_{l=0}^{N-1} w^{l+1} x_j(n-l)$$

This approximation corresponds to a network where stable systems can replace

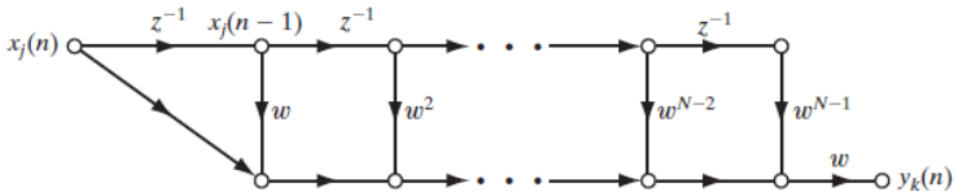

Figure 5.20: Feedback Replacement

feedback with feed-forwardward edges.

5.4 Transformers: What Do These Models Learn?

Currently, one of the most advanced types of deep learning model architecture are Transformers. Introduced in 2017 by Vaswani et al. in the seminal paper "Attention is All You Need" [58], these models are good for tasks that involve sequential data, such as natural language processing (NLP). The famous GPT (Generative Pre-trained Transformer) is a type of transformer model.

Transformers capture long-range dependencies and relationships within the data using a mechanism called self-attention. We will not get into the technical details of how self-attention mechanism works, however, we will present some views on the intelligence, or rather unintelligence of these models.

Linguist Emily Bender and computer scientist Timnit Gebru published a paper in 2021

that describes language models based on Transformers as “stochastic parrots”. They described it as “a system for haphazardly stitching together sequences of linguistic forms it has observed in its vast training data, according to probabilistic information about how they combine, but without any reference to meaning.” [4] Anyone who has used models like GPT would have probably come across an absurd response that support the "stochastic parrots" viewpoint.

Despite their seemingly impressive capabilities, deep learning LLMs sometimes fail in tasks that require genuine understanding and inference. While, they can correctly answer questions based on direct patterns they learnt from the data, they struggle with tasks that involve reversing relationships or handling unusual scenarios. [25] This is shown in examples where LLMs fail to infer relationships that humans can easily understand, such as realizing that if A is the father of B, then B is the son (or daughter) of A.

In 2023, Lukas Berglund et al. published a paper titled "The Reversal Curse: LLMs trained on ‘A is B’ fail to learn ‘B is A’", where they explained: "If a human learns the fact, ’Valentina Tereshkova was the first woman to travel to space’, they can also correctly answer, ’Who was the first woman to travel to space?’ This is such a basic form of generalization that it seems trivial. Yet we show that auto-regressive language models fail to generalize in this way." [5]

These shortcoming highlights that LLMs do not truly "understand" or "reason" but rather manipulate language based on patterns learned during training and attempt to generate plausible-sounding text without any true comprehension. Additionally, these LLMs struggle with outlier problems, which are situations that deviate from the norm

and require reasoning beyond learned patterns.

While some in the AI community, such as GPT CEO Sam Altman, dismiss these concerns, many researchers believe that these limitations might undermine the long-term promises of AI, especially regarding the development of general intelligence. It is indeed concerning that there is no guarantee that these models will not fail in real-world applications where true understanding is crucial.

Chapter 6

INTERPRETABILITY AND ROBUSTNESS

After understanding the mathematical base of how deep learning models are trained, and what these models try to accomplish (minimizing a loss function) one can wonder if these models can be described as intelligent. In the following sections, we will present two challenges that highlight the lack of intelligence of these models.

6.1 "Black Box" Nature of DNNs

The term "black box" refers to the idea that, while DNN models can make highly accurate predictions or classifications, the inner workings or decision-making processes of these models are often a complete mystery to us. This lack of transparency can be problematic, especially in critical applications like healthcare, finance, and autonomous systems, where understanding the reasoning behind a decision is crucial.

The "black box" nature of DNNs arises because these models have complex architectures with many layers and parameters. Each layer transforms the data in ways that are not easily understood by humans, making it difficult to trace how specific features of the input contribute to the final output.

In models with billions of parameters, it is virtually impossible to make sense of the significance or meaning of a chosen weight or bias in any given neuron. Even if we attempt to do that in a simple model with a relatively small number of parameters, it is almost never the case that we can interpret the weights and biases that the learning

algorithm learnt, in any meaningful way.

Let's take as an example the handwritten digit detection model created by Michael Nielsen and presented in his book "Neural networks and deep learning" [39]. This network was trained using the MNIST database [38] which contains thousands of examples of 28x28 pixel grey-scale images of handwritten digits. The examples are represented by an input vector containing 784 features, for the 784 pixels. The model contains two hidden layers with 16 neurons per layer, and finally an output layer containing 10 neurons, each representing a digit. The activation function used is the sigmoid function, outputting a value between 0 and 1. After undergoing training, Nielsen's model "can recognize digits with an accuracy over 96 percent" [39]. Considering this relatively high accuracy, let's examine how this model "reasons". Number of Weights = $784 \cdot 16 + 26 \cdot 16 + 16 \cdot 10$

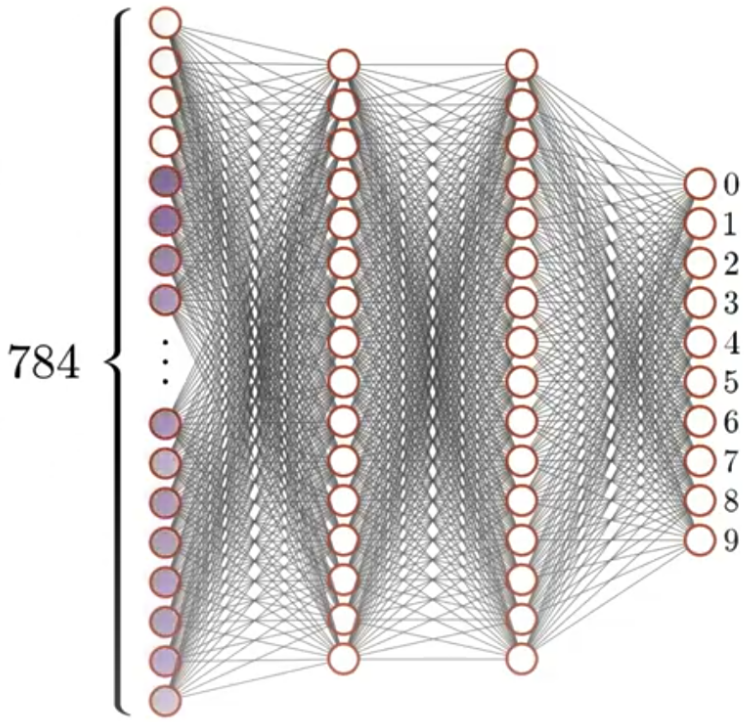

Figure 6.1: Michael Nielsen's Model for Handwritten Digit Detection Containing 2 Hidden Layers with 16 Neurons Each and 10 Neurons in the Output Layer

Number of Biases = $16 + 16 + 10$

In this relatively simple network, we still have 13,002 parameters. Although this number is still large, given the task at hand, we can still imagine how this neural network might "reason."

When we teach children how to recognize digits, we explain how we piece together separate components to make up a digit. These components are in their turn made up

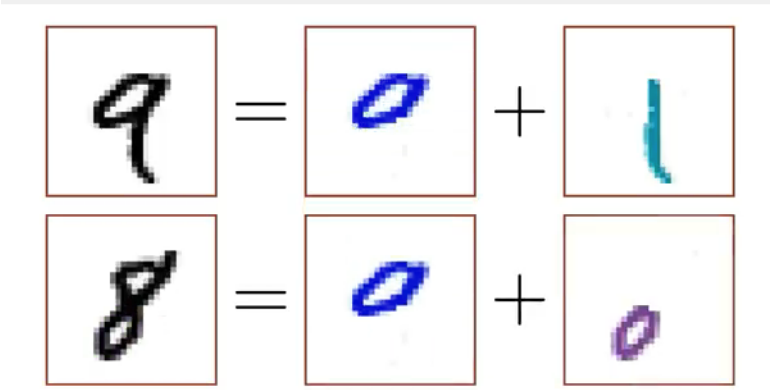

Figure 6.2: (Top) One Circle on Top of a Line Segment is a 9. (Bottom) Two Circles on Top of Each other Make an 8

of sub-components: If this neural network is to "reason" in a similar manner, then we

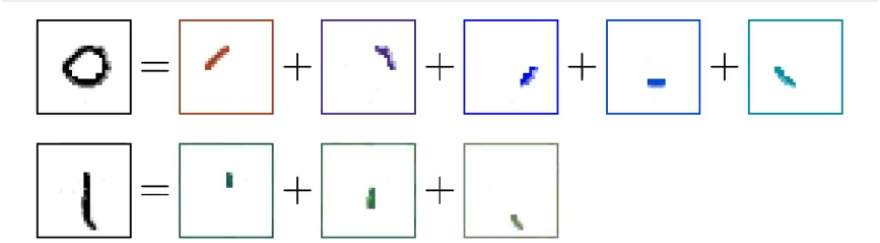

Figure 6.3: Breaking Down a Handwritten Digit into Its Sub-components

could hope that the first layer learns to pickup on all sub-components (little edges), and then the second layer, can put these sub-components together to find the main components (circles, lines) and finally the last layer could put these components together to determine the correct digit (see figure 6.4). For instance one neuron in the

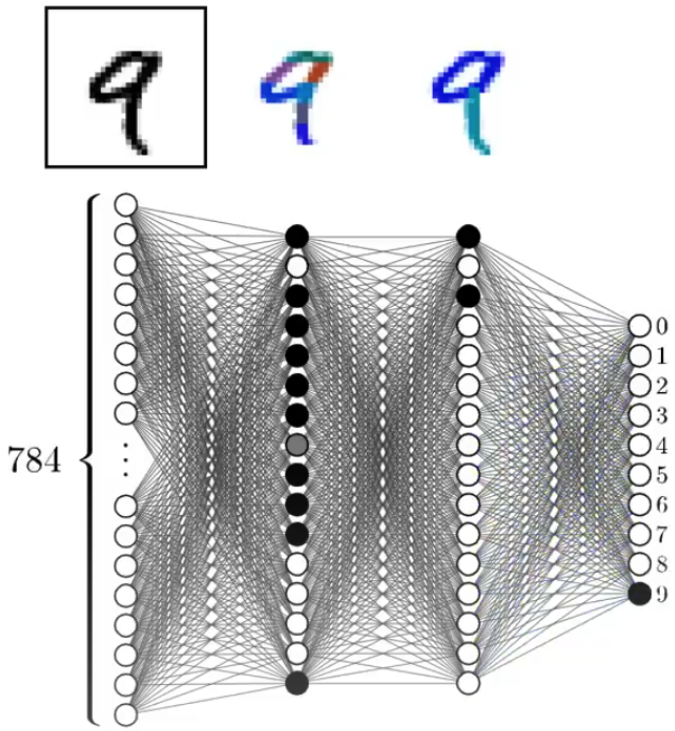

Figure 6.4: A Reasonable Way a Neural Network Can Detect Handwritten Digits

first layer might be in charge of detecting horizontal edges in a certain region of the image. Mathematically this can be done by setting all the weights associated to pixels which are outside the focus region to 0, while having positive weights for the pixels in the region, and negative weights around the region. Thus the weighted sum is largest when the middle pixels are bright but the surrounding pixels are darker. We can then determine what is the minimum value (bias) we want this sum to reach before this neuron fires, communicating that a horizontal edge does in fact exist in this region. If the network did in fact reason in such a way, then we can easily interpret what each

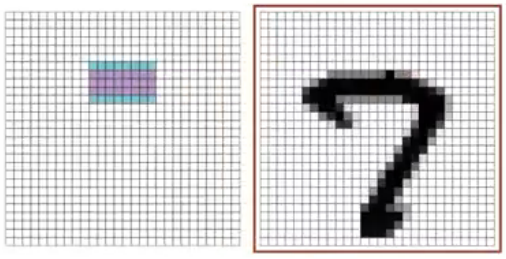

Figure 6.5: Detecting Edges by Assigning Positive Weights (red) to Pixels in Region of Interest and Negative Weights (blue) to Surrounding Area, and Setting All Other Weights to 0 (blank) [61]

weight's role is and how it helps the network detect the digit. However, this is not at all how this particular network works.

Using the trained model's parameter values, we can visualize the weights of each neuron of the first layer using the same visual representation in figure 6.5. When we actually do that [61], instead of finding that each neuron is picking up on an isolated edge, the weights associated with each pixel look almost completely random. This

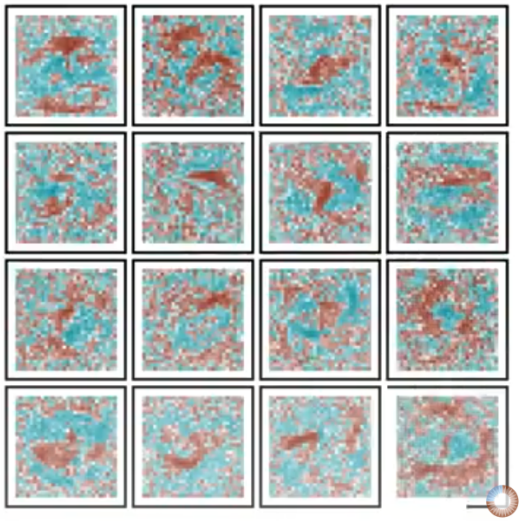

Figure 6.6: Visual Representation of the Weights in the 16 Neurons of the First Hidden Layer [61]

indicates that unlike how children reason, this neural network has managed to find itself a comfortable local minimal that somehow enables it to correctly detect handwritten digits with 96% accuracy using values for weights and biases which no one can interpret in any meaningful way.

One might argue that there is no need for interpretability, so long as the model is accurate. We will demonstrate a simple experiment why this is not true.

If we show a person a random image of pixels, and ask it what this digit is, the person will either say "I don't know" or even better "This is not a digit." If Michael Nielsen's model was smart, then if we show it this random image of pixels, it should communicate uncertainty either by not activating any neuron in the output layer or possibly activating many or all neurons evenly. This experiment was indeed conducted by Grant Sanderson [61] and the result is illustrated in figure 6.7. The

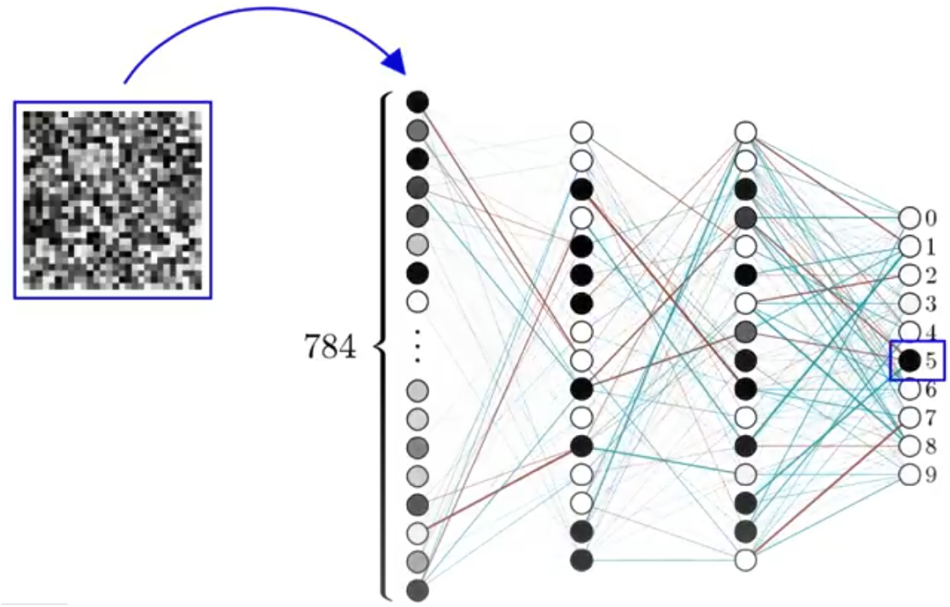

Figure 6.7: Inputting a Random Pixel Image

model confidently gives a wrong answer of 5, as if it is as sure that this random image is a 5, as an actual image of a five is a 5. This example demonstrates how the lack of interpretability leads to unpredictability.

The fact that deep neural network models are being employed in self-driving cars and medical diagnostics should raise many red flag to anyone who understand how these models really work. If we don't understand why a 28x28 random pixel image looks like a "5" to a model that is accurate at 96%, how can we know for a fact that a self-driving car would always recognize when it has to stop before killing a person?

6.2 Adversarial Attacks

In 2015, Goodfellow et al. found that even with state-of-the-art DNNs, applying small, intentional perturbations to an example from the dataset, "can fool DNNs into making incorrect predictions with high confidence" [36]. By small we mean that the perturbed example or input can be almost indistinguishable from the original one, yet be classified incorrectly by the network.

A couple of years later, Jlawei Su et al. published "One Pixel Attack for Fooling Deep Neural Networks" where they presenting very alarming results: "67.97% of the natural images in Kaggle CIFAR-10 test dataset and 16.04% of the ImageNet (ILSVRC 2012) test images can be perturbed to at least one target class by modifying just one pixel with 74.03% and 22.91% confidence on average." [55] These are very concerning findings, which highlight how easy it is to fool a DNN model. In fields where image recognition technology can impact the future of a person (self-driving cars, medical field) there should be no room for such errors.

In fact in 2021, Xingjun Ma et al, found that medical DNNs are potentially even more

susceptible to adversarial attacks compared to models designed to process natural images. They proposed two possible reasons for that: firstly, the nature of the medical images, which contain complex textures unlike natural images, lead to more vulnerable regions. They explain, "the rich biological textures in medical images sometimes distract the DNN model into paying extra attention to areas that are not necessarily related to the diagnosis." [36] Thus, any small perturbation in these regions can cause a big change in the model output. Secondly, and more importantly they argue that advanced DNNs used in large-scale natural image processing can be overparameterized for simple medical image analysis tasks. Their analysis reveals that the deep representations of medical images are relatively straightforward compared to the structures learned from natural images. This suggests that, in the context of medical imaging, DNN models are identifying basic patterns and these simple patterns can be captured without the need for highly complex deep networks.

Chapter 7

CONCLUSION

Today, people from a variety of fields are building and trusting neural networks to guide their professional, scientific and/or personal endeavors. Given the increasing reliance on these models, especially in critical fields where the well-being of people is at risk, ensuring their integrity and reliability is crucial.

DNNs have definitely dazzled the world with models such as OpenAI's ChatGPT. Digging deep into the theoretical foundations of these models, it is indeed dazzling to see how from chaos, order can seemingly emerge. Although the information processing mechanism and "learning" process of DNN models can almost entirely be described using linear algebra, probability and calculus, there is no theoretical explanation of what it is that these models actually learn: No one understands how they "reason." Indeed, however powerful, the effectiveness of deep neural networks often comes at the cost of interpretability. "If AI is so complex it can't be explained, there are areas where it shouldn't be used, " argues Human rights lawyer Susie Alegre in an article published by The Guardian.

In order to dissect the inner deepest workings of these models, this thesis has examined the mathematical foundation behind the structure of feed-forward neural networks, the implications of the Universal Approximation Theorem, and the challenges inherent in the learning algorithms used to train these models. Additionally, we have discussed advanced neural network architectures and emerging

models like Transformers. These discussions highlights the complexity and diversity of DNN models, yet also point to the gaps in our current understanding, particularly concerning the lack of interpretability and their vulnerability to adversarial attacks.

The purpose of this thesis is to bridge the gap between the theoretical understanding of these models and their practical applications, in order to draw the line between when DNNs can be trusted and when they shouldn't be. We argue that however powerful they might seem, these models are very much unintelligent and their unintelligence is simply hidden behind the immense size of the models' parameters.

In fact, the mathematics currently applied in DNN models may be insufficient for fully capturing the nuances of intelligent behavior as exhibited by the human mind. There is potential for new mathematical frameworks, theories or even entirely new mathematical objects to emerge, offering deeper insights into how intelligent systems reason, thus allowing us to achieve a more profound understanding of intelligence itself. By building on the insights presented here, future research can further our understanding of both artificial and natural intelligence, ultimately contributing to the creation of more sophisticated and reliable artificially intelligent models.

REFERENCES

[1] Anthony, M., & Barlett, P. L. (2009). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.

[2] Arnold, V. I. (1958). On functions of three variables. *Doklady Akademii Nauk SSSR*, 114, 679-681.

[3] Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., ... & Herculano-Houzel, S. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5), 532-541.

[4] Bender, E. M., McMillan-Major, A., Gebru, T., & Shmitchell, S. (2021). On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (pp. 610-623). ACM.

[5] Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, Owain Evans, "The Reversal Curse: LLMs trained on 'A is B' fail to learn 'B is A'", *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[6] Brownlee, J. (2019). "How to Configure the Learning Rate When Training Deep Learning Neural Networks". *Machine Learning Mastery*. Retrieved 4 January 2021.

[7] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2, 303–314.

[8] Deep Mind. (2023, March 26). The Universal Approximation Theorem. Retrieved from https://www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/#Structure_of_Neural_Networks

[9] De Silva, G. (2020). Exploring the world of artificial neural networks - A beginner's overview. *ResearchGate*. https://doi.org/10.13140/RG.2.2.14790.14406

[10] ERCIM News. (2023). Explainable AI, Number 134, July.

[11] ERCIM News. (2024). Large Language Models, Number 136, January.

[12] Fausett, L. (1994). *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall.

[13] Feng, Z. (2010). *Hilbert's 13th Problem*. PhD Thesis, Department of Mathematics, University of Pittsburgh.

[14] Ferenc, H., & Istvan, G. (2008). Matematikai analízis mérnök informatikus mesterszak eloadásjegyzet.

[15] Funahashi, K. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks, 2*(3), 183-192.

[16] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 249-256).

[17] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[18] Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*.

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

[20] Grohs, P., & Kutyniok, G. (2023). *Mathematical Aspects of Deep Learning*. Cambridge University Press.

[21] Haykin, S. O. (2008). *Neural Networks and Learning Machines* (3rd ed.). Pearson.

[22] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026-1034).

[23] Hecht-Nielsen, R. (1987). Kolmogrov's mapping neural network existence theorem. In *Proceedings of the IEEE First International Conference on Neural*

Networks (Vol. III, pp. 11-13). IEEE Press.

[24] Herculano-Houzel, S. (2009). The human brain in numbers: A linearly scaled-up primate brain. *Frontiers in Human Neuroscience, 3*, 31.

[25] Hern, A. (2024). Why AI's Tom Cruise problem means it is 'doomed to fail'. *The Guardian*. Retrieved from https://www.theguardian.com/technology/article/2024/aug/06/ai-llms

[26] Healthline. (n.d.). An easy guide to neuron anatomy with diagrams. Retrieved from https://www.healthline.com/health/neurons

[27] Hilbert, D. (1902). Mathematical problems. *Bulletin of the American Mathematical Society, 8*, 437-479.

[28] Hinton, G. E., & Sejnowski, T. J. (1985). Learning and relearning in Boltzmann machines. In *Parallel distributed processing: Explorations in the microstructure of cognition*, 1, 282-317.

[29] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 359–366.

[30] Kinsley, H., & Kukieła, D. (2020). *Neural Networks from Scratch in Python*.

[31] Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

[32] Kolmogorov, A. N. (1957). On the representation of continuous functions of three variables by superpositions of continuous functions of two variables. *Doklady Akademii Nauk SSSR*, 114, 953-956.

[33] Kolmogorov, A. N. (1956). On the representation of continuous functions of several variables by superpositions of continuous functions of one variable and addition. *Doklady Akademii Nauk SSSR*, 108, 179-182.

[34] Kutyniok, G. (2022). The mathematics of artificial intelligence. In *Proceedings of the International Congress of Mathematicians*.

[35] LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (2012). Efficient backprop. In *Neural Networks: Tricks of the Trade* (pp. 9-48). Springer, Berlin, Heidelberg.

[36] Ma, X., Niu, Y., Gu, L., Wang, Y., Zhao, Y., Bailey, J., & Lu, F. (2021). Understanding adversarial attacks on deep learning-based medical image analysis systems. Pattern Recognition

[37] Marteney, J. (2020). *Arguing Using Critical Thinking*. Academic Senate for California Community Colleges. Chapter 12: The Foundations of Critical Thinking, Section 12.2: Defining Intelligence.

[38] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of handwritten digits," *Modified National Institute of Standards and Technology*, Available: http://yann.lecun.com/exdb/mnist/, 1998.

[39] M. A. Nielsen, *Neural Networks and Deep Learning: A Visual Introduction to Deep Learning*. Determination Press, 2015.

[40] Minsky, M., & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

[41] Murphy, K. (2021). *Probabilistic Machine Learning: An Introduction*. MIT Press. Retrieved 10 April 2021.

[42] Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (pp. 807-814).

[43] Ng, A. *Supervised Machine Learning: Regression and Classification*. Coursera, Stanford Online and Deeplearning.ai. Retrieved from https://www.coursera.org/learn/machine-learning, n.d.

[44] Hecht-Nielsen, R. *Kolmogorov's Mapping Neural Network Existence Theorem*. In *Proceedings of the IEEE International Conference On Neural Networks III*, New York, IEEE Press, 1987, pp. 11–14.

[45] Patterson, J., & Gibson, A. (2017). *Deep Learning: A Practitioner's Approach*. O'Reilly. ISBN 978-1-4919-1425-0.

[46] Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533-536.

[47] McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *University of Illinois, College of Medicine, Department of Psychiatry at the Illinois Neuropsychiatric Institute, University of Chicago, Chicago, U.S.A.*

[48] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1), 145-151.

[49] Scriven, M., & Paul, R. (1987). Defining critical thinking. 8th Annual International Conference on Critical Thinking and Education Reform. Retrieved from http://www.criticalthinking.org/pages/defining-critical-thinking/766

[50] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[51] Smith, L. N. (2017). "Cyclical learning rates for training neural networks". *arXiv:1506.01186 [cs.CV]*.

[52] Sprecher, D. (1965). On the structure of continuous functions of several variables. *Transactions of the American Mathematical Society, 115*, 340-355.

[53] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 1929-1958.

[54] Stangor, C., & Walinga, J. (2014). The neuron is the building block of

the nervous system. In *Introduction to Psychology - 1st Canadian Edition*. Retrieved from https://opentextbc.ca/introductiontopsychology/chapter/3-1-the-neuron-is-the-building-block-of-the-nervous-system/

[55] Su, J., Vargas, D. V., & Sakurai, K. (2017). One Pixel Attack for Fooling Deep Neural Networks. *IEEE Transactions on Evolutionary Computation*, 23(5), 828-841.

[56] Thomas, G. B., Weir, M. D., Hass, J., & Giordano, F. R. (2011). *Thomas' Calculus* (12th ed.). Boston: Pearson Addison Wesley.

[57] Tollner, D. (2018). Neural networks: An introduction. Seminar series on Neural networks, Budapest University of Technology and Economics.

[58] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems, 30*.

[59] Veen, F. van. (2020). *The Mostly Complete Chart of Neural Networks, Explained*. Retrieved from https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464.

[60] Werbos, P. J. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. Harvard University.

[61] Grant Sanderson. *Neural Networks.* 3Blue1Brown YouTube Series. Available

at: https://www.youtube.com/playlist?list=PLZHQObOWTQDMsr9KymGpF3_F2p2w5t4qF.

[62] 3Blue1Brown. (2022, Nov 18). But what is a convolution? [Video]. YouTube. https://www.youtube.com/watch?v=KuXjwB4LzSA

APPENDIX

Chart of Neural Networks [59]