YEDITEPE UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

AN OVERVIEW OF SOME MATHEMATICAL TECHNIQUES IN DEEP LEARNING

A Thesis Submitted

by

Elif Çalışkan

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

Mathematics

Supervisor

Prof. Dr. Mustafa Polat

Istanbul - 2024

AN OVERVIEW OF SOME MATHEMATICAL TECHNIQUES IN DEEP LEARNING

by

Elif Çalışkan

Approved by:

Prof. Dr. Mustafa Polat
(Yeditepe University) .....
(Thesis Supervisor)

Assist. Prof. Dr. Barış Efe
(Yeditepe University) .....

Assoc. Prof. Dr. Saadet Seher Özer
(İstanbul Technical University) .....

DATE OF APPROVAL: .... / .... / 20....

DECLARATION OF ORIGINALITY

I hereby declare that this thesis is my own work and that all information in this thesis has been obtained and presented following academic rules and ethical conduct. I have fully cited and referenced all material and results as required by these rules and conduct, and this thesis study does not contain any plagiarism. The necessary permissions have been obtained if any material used in the thesis requires copyright. No material from this thesis has been used to award another degree.

To the best of my knowledge and belief, it contains no material previously published or written by another person nor material accepted for the award of any other degree except where due acknowledgment has been made in the text.

I accept all kinds of legal liability that may arise in cases contrary to these situations.

Elif Çalışkan

ABSTRACT

AN OVERVIEW OF SOME MATHEMATICAL TECHNIQUES IN DEEP LEARNING

In this thesis, we will explore the mathematical aspects of deep learning. Our goal is to analyze the mathematical tools used in deep learning. We aim to understand mathematical challenges such as minimizing convex cost functions.

Firstly, by taking a brief look at the history of deep learning, we start to gain a better understanding of this field. Then, we delve into the principles, applications, and workings of deep learning and artificial neural networks. Next, we introduce activation functions that allow neural networks to depart from linearity and help them learn more complex relationships. After discussing sigmoid and ReLU functions and their advantages, we create a simple neural network and learn how to mathematically compute the weights and biases of neural networks.

Subsequently, we work with a handwritten example to understand how artificial neural networks function from a mathematical perspective. We learn to compute stochastic gradient descent to minimize the cost function and apply this method to train the network. Then, we apply the backpropagation method to determine the contribution of each weight and bias term to the error.

This thesis also includes MATLAB code implementing backpropagation and stochastic gradient descent, along with an image classification example. Additionally, we explore the three main types of layers in convolutional neural networks, understand the principles of convolutional layers, perform calculations, and use Sobel filters.

Finally, we conclude our thesis with an image classification example using the Keras CIFAR-10 dataset. In this example, by examining all layers of the neural network in detail, we understand its functionality and how it reaches the training stage.

ÖZET

DERİN ÖĞRENMEDEKİ BAZI MATEMATİKSEL TEKNİKLERE BİR BAKIŞ

Bu tezde, derin öğrenmenin matematiksel yönlerini inceleyeceğiz. Amacımız, derin öğrenmede kullanılan matematiksel araçları analiz etmektir. Dış bükey maliyet fonksiyonunun minimizasyonu gibi matematiksel zorlukları anlamayı hedefliyoruz. İlk olarak, derin öğrenmenin tarihine kısa bir göz atarak bu alana daha iyi bir anlayış kazanmaya başlıyoruz. Ardından, derin öğrenme ve yapay sinir ağlarına aşina olmak için çalışma prensiplerine, uygulamalarına ve yapay sinir ağlarına dalmaktayız. Daha sonra, sinir ağlarının doğrusallıktan çıkmasını sağlayan aktivasyon fonksiyonlarını tanıtıyoruz ve daha karmaşık ilişkileri öğrenmelerine yardımcı oluyoruz. Sigmoid ve ReLU fonksiyonlarını ve avantajlarını tartıştıktan sonra, basit bir sinir ağı oluşturuyoruz ve sinir ağlarının ağırlıklarını ve önyargılarını matematiksel olarak nasıl hesaplayabileceğimizi öğreniyoruz. Sonrasında, yapay sinir ağlarının matematiksel bir bakış açısından nasıl çalıştığını anlamak için el yazısı örneğiyle çalışıyoruz. Maliyet fonksiyonunu minimize etmek için stokastik gradyan inişini hesaplamayı öğreniyor ve ağı eğitmek için bu yöntemi uyguluyoruz. Daha sonra, her ağırlık ve bias teriminin hataya olan katkısını belirlemek için geri besleme yöntemi uyguluyoruz. Bu tez ayrıca, geri besleme ve stokastik gradyan yönteminin uygulandığı bir MATLAB kodu içerir ve bir görüntü sınıflandırma örneği sunar. Ayrıca, evrişimsel sinir ağlarının üç ana tür katmanlarını keşfediyor, evrişimli katmanların çalışma prensiplerini anlıyor, hesaplamalar yapıyor ve Sobel filtrelerini kullanıyoruz. Son olarak, Keras CIFAR-10 veri kümesi kullanarak bir görüntü sınıflandırma örneğiyle tezimizi sonlandırıyoruz. Bu örnekte, sinir ağının tüm katmanlarını detaylı bir şekilde inceleyerek, işlevselliğini anlıyor ve ağın eğitim aşamasına nasıl geldiğini öğreniyoruz.

DEDICATION

For their unwavering support and encouragement throughout my entire educational journey,
I dedicate this thesis to my cherished parents, Sevim Afşarlıoğlu Ç. and Nedim Çalışkan...

ACKNOWLEDGEMENTS

This thesis, titled "AN OVERVIEW OF SOME MATHEMATICAL TECHNIQUES IN DEEP LEARNING," is prepared by Elif Çalışkan. This study is a survey of research on some mathematical techniques in deep learning. Many individuals have assisted in the completion of this thesis, and I want to thank them sincerely. First and foremost, I offer my thanks to my thesis advisor, Mustafa Polat, and my academic mentors, İlknur Kuşbeyzi Aybar and Orhan Aybar, for their motivation and advice throughout the entire process. Additionally, I want to express my gratitude to my beloved mother, Sevim Afşarlıoğlu Çalışkan, my father, Nedim Çalışkan, my brothers, Safa Çalışkan and Arif Çalışkan, for their endless love and support, and Furkan Hatık for his unwavering support and patience throughout my academic journey, especially during challenging times. Their continuous encouragement greatly contributed to the completion of this thesis.

The scope and limitations of this study are evident. We acknowledge that this research on some mathematical techniques in deep learning has a specific focus and perspective. Additionally, we recognize that this study does not comprehensively cover all topics and that some subjects could be explored in more detail. Finally, I hope that this thesis will contribute to Yeditepe University, Faculty of Science, Department of Mathematics, as well as to the relevant research community and general readers. I hope that this study will enhance the understanding of deep learning and neural networks, providing valuable insights for future researchers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

b	Vector
$D^{[l]} \in \mathbb{R}^{n_l \times n_l}$	Diagonal matrix
R(x)	ReLU function
W	Matrix
$(x^{(k_i)})_{i=1}^{m}$	Minibatch
$\nabla Cost(p)$	Gradient
p	Single vector
η	Learning rate
$\sigma(x)$	Sigmoid curve
ANN	Artifical neural network
AI	Artifical intelligence
CNN	Convolutional neural network
DL	Deep learning
EMNIST	Extended modified national institute of standards and technology
GPU	Graphics processing unit
ReLU	Reactifier linear unit

1. INTRODUCTION

In this chapter, after briefly discussing the history of deep learning, we will delve into fundamental definitions such as what deep learning and neural networks are, as well as what the applications of deep learning entail. We will also make some comparisons.

1.1. HISTORY OF DEEP LEARNING

Deep learning (DL) stands out as one of the most favored technological terms, currently being heavily invested in by companies such as Facebook, Google, YouTube, Tesla, Spotify, and Microsoft. The primary reason for its surge in demand stems from the rebranding of artificial neural networks (ANN), which have been in existence since the 1960s. The earliest algorithms resembling deep learning were devised by Ivakhnenko and Lapa in 1965. Many advancements occurred in the 1980s (with Fukushima's convolutional neural networks) and 1990s (with LeCun's LeNet). However, computers were slow and data sets were small during those times, resulting in few applications for neural networks, and research declined during the 2000s. Only in recent years did neural networks witness a revival, fueled by enhanced computational power and greater data availability. The introduction of Graphics Processing Units (GPUs) and the exponential increase in data allowed for more widespread use. GPUs, for instance, increased processing speeds by 1000 times in a decade. Additionally, advancements in technology allowed for more data storage, and the internet facilitated the sharing of this data. Some developments include speech recognition, computer vision, and drug discovery. [3]

1.2. DEEP LEARNING AND NEURAL NETWORKS

Now let's focus on some fundamental definitions and comparisons.

1.2.1. What are the Neural Networks?

An artificial neural network (ANN), also referred to as a simulated neural network (SNN), is a fundamental component of artificial intelligence (AI) algorithms. These networks are pivotal in deep learning, a branch of machine learning techniques that draws inspiration from the human brain's functionality. In deep learning, algorithms replicate the intricate communication processes among biological neurons to process and analyze data. Consisting of layers of interconnected units, or neurons, neural networks handle intricate relationships to transfer and analyze information.

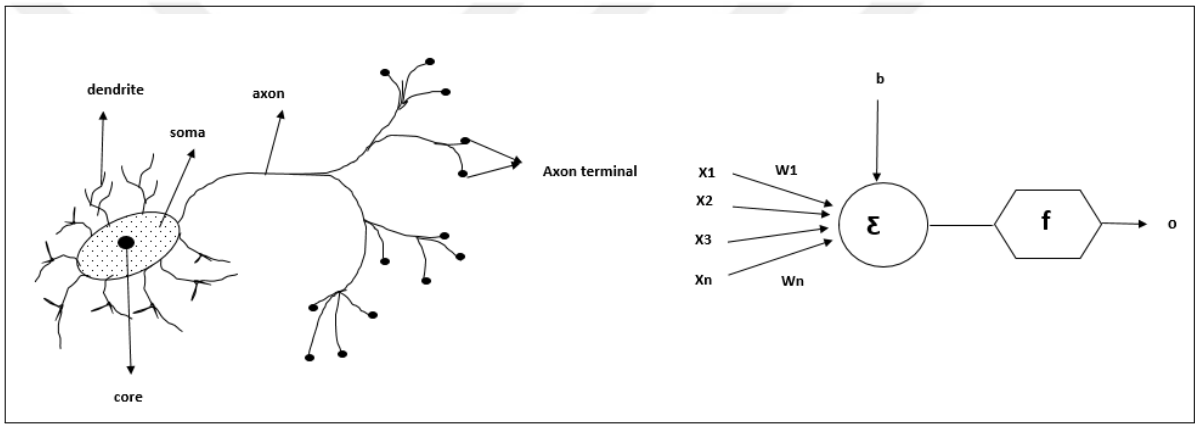


Figure 1.1. Similarity of neural networks with brain neural network

1.2.2. What is Deep Learning?

For a detailed description of deep learning, we refer to Nolan [4].

Deep learning falls within the category of machine learning techniques. These neural networks, consisting of three or more layers, aim to replicate the operations of the human brain.

Deep learning systems fuels everyday products and services, such as voice-controlled TVs, remote monitoring systems, and credit card fraud detection, along with emerging technologies such as autonomous vehicles.

Deep learning is a subset of machine learning, but there's more to it. Deep learning eliminates some of the dependence on human professionals. For example, let's say you have a series of

photos of pets and you want to group them by cat, dog, hamster, etc.

Deep learning algorithms can identify the most distinctive features (such as ears) when categorizing animals. However, this feature structure is typically created manually by human machine learning experts. Deep learning is thus a machine learning method that instructs computers to emulate the natural human ability of learning through examples.

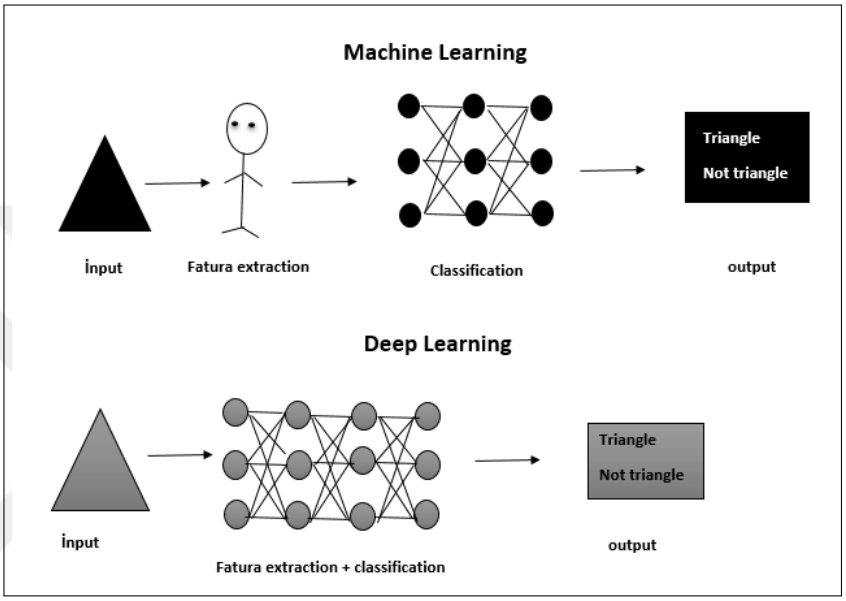


Figure 1.2. Learning in deep learning and machine learning algorithms

1.2.3. How Deep Learning Works?

Deep learning neural networks, also referred to as artificial neural networks, seek to replicate the human brain by utilizing a blend of input data, weights, and biases. They are commonly used to identify, classify, and describe objects within data.

Deep neural networks consist of multiple layers of interconnected nodes, with each layer building on top of the previous layer to refine and optimize predictions and classifications.

The input and output layers of a deep neural network are referred to as the **visible layers**. The deep learning model receives data for processing in the input layer, and it makes the final prediction or classification in the output layer.

1.2.4. Deep Learning Applications

For a detailed description of deep learning applications, we refer to Doğan, Ferdi [5].

- **Law Enforcement:** Deep learning improves investigative analysis by extracting patterns and evidence from audio, video, images, and documents. This technology enables law enforcement to process large data volumes quickly and accurately, enhancing decision-making and investigations.
- **Financial Services:** Financial institutions constantly employ predictive analytics to find fraud and assist their customers in managing their loan and investment documents.
- **Customer Service:** Chatbots, employed across diverse applications, services, and customer support platforms, represent a straightforward advancement in AI technology. For instance, consider Apple's Siri as a prime example.
- **Healthcare:** Image analysis tools aid medical imaging specialists and radiologists in examining and assessing a greater number of images in less time.

1.2.5. Artificial Neural Networks and Deep Learning Techniques

Deep learning, a subset of machine learning, seeks to extract information from large and complex datasets by using multi-layered neural networks. Neural networks consist of artificial neurons organized in a way that resembles the structure of the human brain. Each neuron receives input data, applies a process, and produces output.

Neural networks become capable of learning more intricate relationships and analyzing data at a deeper level as the number of layers increases, defining the fundamental essence of deep learning. The main reason for the use of the term "deep" in deep learning is because it relies on neural networks with three or more layers. Traditional neural networks typically consist of an input layer, an output layer, and one or two hidden layers, while deep learning neural networks are characterized by having numerous hidden layers.

Deep learning techniques, along with neural networks, are employed for automating complex tasks such as visual recognition, voice recognition, and language understanding. The success

of deep learning comes from the increased depth of neural networks, allowing them to understand more complex data structures.

1.2.6. Differences and Relationships: AI, Machine Learning, Deep Learning, and Neural Networks

Deep learning, a subset of machine learning, utilizes neural networks, while machine learning itself is a component of artificial intelligence.

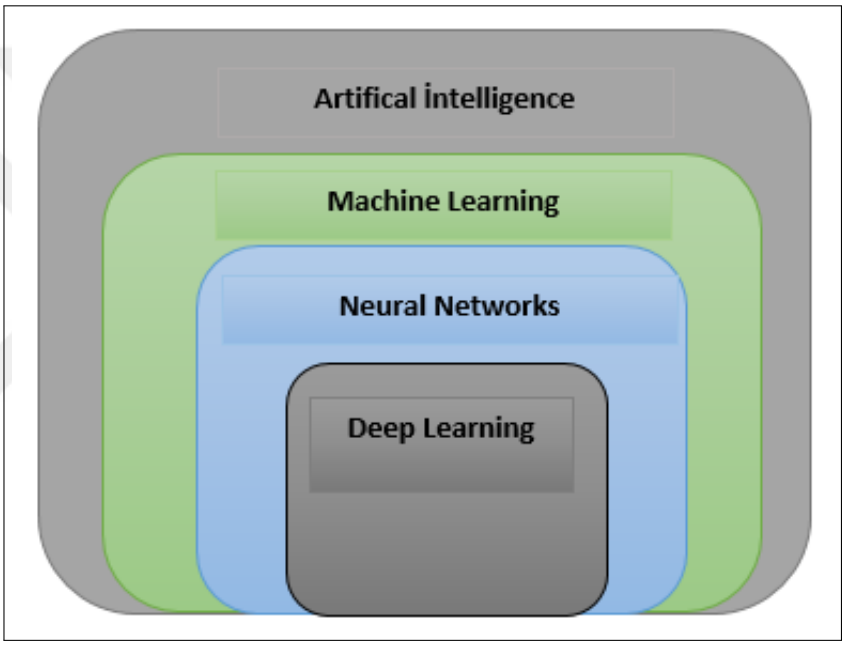


Figure 1.3. Hierarchical structure of learning models

2. ACTIVATION FUNCTIONS AND THEIR NOTATIONS

In this chapter, we discuss how activation functions contribute to expressing the nonlinear properties of neural networks, which enhances the network's ability to model and learn from more complex datasets. We also introduce other activation functions in addition to the most commonly used sigmoid and ReLU functions.

2.1. WHAT IS ACTIVATION FUNCTION?

Activation functions are functions added to artificial neural networks to enable them to learn complex patterns in data. Compared to neuron-based models of our brain, the activation function ultimately determines what is sent to the next neuron. Also called **"transfer function"**. Like in Figure 1.1

2.2. WHY ARE ACTIVATION FUNCTIONS ESSENTIAL IN NEURAL NETWORKS?

Activation functions are used to determine whether the output of a neural network is active or inactive, assigning the resulting value in the range of 0 to 1 or -1 to 1, depending on the function used.

2.3. WHAT ARE THE SIGMOID AND RELU FUNCTIONS?

A sigmoid function is a mathematical function known for its characteristic 'S'-shaped curve, also referred to as a sigmoid curve or logistic curve.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

and it is a non-linear function. The domain of the function is $[-\infty, \infty]$ and the range is [0,1]. ∞ is mapped to 1 and $-\infty$ is mapped to 0.

An advantageous property of the sigmoid function is that its derivative simplifies to a straightforward form

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2.2}$$

It is easy to compute and can be used in gradient-based learning algorithms. A Sigmoid function is differentiable anywhere in its range and continuous everywhere.

In contrast, the ReLU function is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs 0.

The Relu function is defined as follows

$$R(x) = \left\{ \begin{array}{ll} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{array} \right\} = max(0, x) \tag{2.3}$$

and it's a nonlinear function. Also called **rectifier linear unit**. The domain of the function is $[-\infty, \infty]$ and the range is $[0, \infty]$. ∞ is mapped to ∞ and $-\infty$ is mapped to 0.

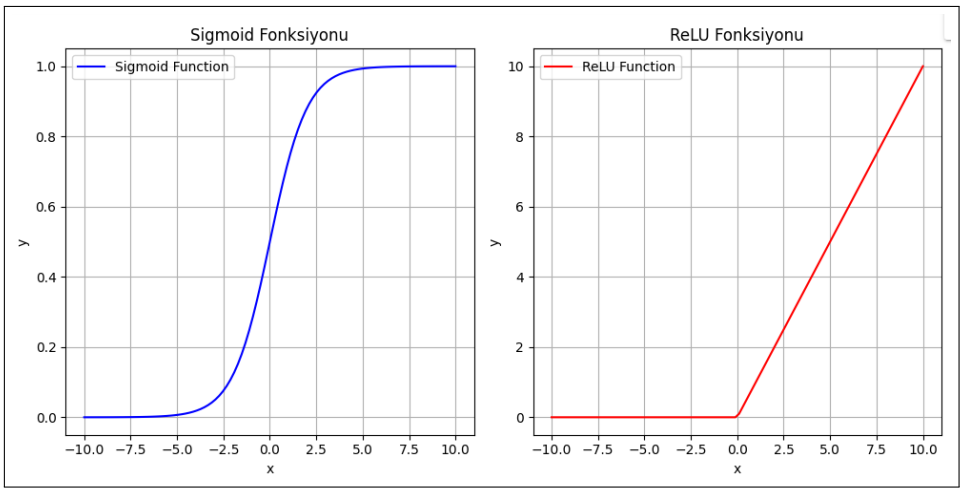


Figure 2.1. Graph of ReLu and Sigmoid function

2.4. WHAT MAKES RELU SUPERIOR TO THE SIGMOID FUNCTION IN DEEP NEURAL NETWORKS?

The sigmoid function is used because it outputs values between 0 and 1, making it suitable for models that predict probabilities. ReLU, on the other hand, is popular in deep learning due to its fast learning and lower gradient vanishing, making it ideal for image processing and pattern recognition. However, ReLU can cause some neurons to produce only zero values, which leads to "dead" neurons that are not updated during backpropagation, potentially causing overfitting. This issue can be mitigated by using variations of ReLU or other activation functions.

Table 2.1. Some examples of other activation functions

Function	**Notation**	**Range**
Identify	$identity(x) = x$	$(-\infty, \infty)$
Binary Step	$binaryStep(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$	(0, 1)
Tanh Function	$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	[-1, 1]
Softmax Function	$softmax(x) = \frac{e^x}{\sum_{i=1}^{n} e^x}$	(0, 1)
Leaky Relu	$leakyRelu(x) = max(0.01x, x)$	$(-\infty, \infty)$
PRelu	$prelu(x) = max(ax, x)$	$(0, \infty)$
Exponential linear unit (ELU)	$elu(x) = max(ax, x)$	$(-a, \infty)$
Gaussian	$gaussian(x) = e^{-x^2}$	(0, 1]
Scaled Exponential Linear Unit	$selu(x) = \lambda \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$	$(-\lambda\alpha, \infty)$ and $\alpha = 1.67$, $\lambda = 1.05$
Softplus	$softplus(x) = \ln(1 + e^x)$	$(0, \infty)$

3. THE CONSTRUCTION OF A NEURAL NETWORK

By the knowledge we have gained by chapter 2, we can now understand the construction of a neural network and set up layers of neurons.

3.1. THE GENERAL CONSTRUCTION OF A NEURAL NETWORK

The construction of a neural network is shown in the figure below.

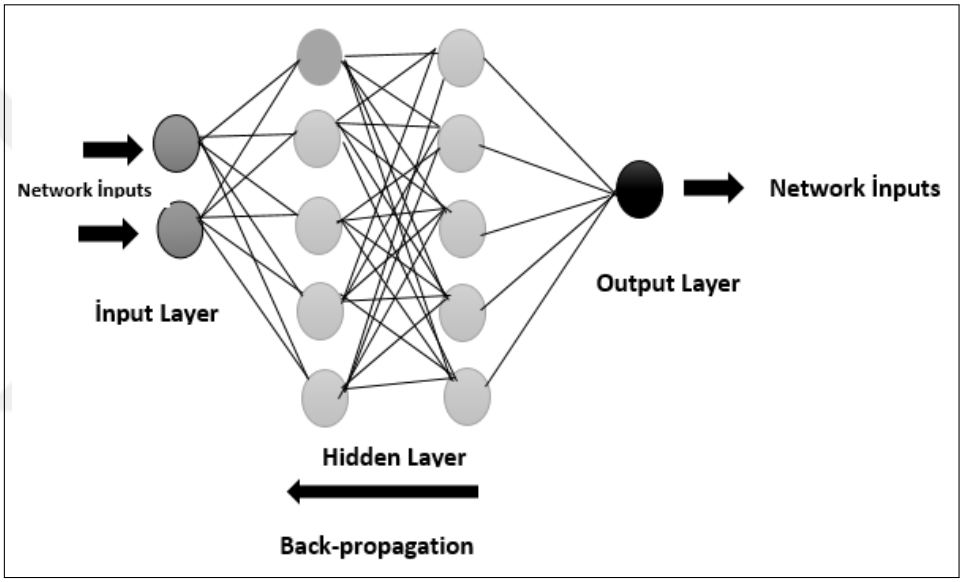


Figure 3.1. Structure of neural network

As illustrated in Figure 3.1, neural networks consist of layers of units. A neural network typically consists of an input layer, one or more hidden layers, and an output layer. These neural networks emulate the operations of the human brain. They receive real values from each neuron in the previous layer and produce real values that are passed forward in the network. The layers located between the input and output layers are referred to as **hidden layers**. Neurons in these layers perform intermediate computations

Every input is transmitted to every neuron within the initial layer. Neurons in a given layer are connected to neurons in the subsequent layer through channels. Each of these connections is assigned a numerical value known as a **weight**. The inputs are then multiplied by their corresponding weights, and the resulting products are summed to form the input to the

neurons in the hidden layer. Additionally, each neuron is assigned a value referred to as a **bias**, which is added to the sum of the inputs. This sum undergoes a threshold function known as the **activation function**. The outcome of the activation function determines whether each neuron becomes activated or remains dormant. Activated neurons then transmit data through channels to neurons in the subsequent layer. This process, known as **forward propagation**, facilitates the transmission of data throughout the network. Ultimately, within the output layer, the neuron with the highest value becomes active and determines the network's output.

3.2. WEIGHTS AND BIASES

By section 3.1, we know that each neuron in a layer forms a uniquely weighted combination of the values from the previous layer, biases it, and applies an activation function to it. Weights and biases are neural network parameters used to form a weighted combination of values from previous layers. Mathematically, the output vector of the following layer is represented as $\mathbf{a}$, where the real values produced through the neurons in that layer are combined into the vector $\mathbf{a}$.

$$\sigma(Wa + b). \tag{3.2.1}$$

Here, W **is represented as a matrix** and b **as a vector**. W encompasses the weights, while b encapsulates the biases.

As we perform matrix multiplication, the number of columns in W must match the number of neurons in $\mathbf{a}$ from the preceding layer.The number of rows in W corresponds to the number of neurons in the current layer. Likewise, the number of elements in b should be equal to the number of neurons in that layer. To understand the function of the ith neuron in equation (3.2.1), we can analyze the ith component:

$$\sigma\left(\sum_{i=0}^{n} w_i a_i + b_i\right), \tag{3.2.2}$$

where n is the quantity of inputs/outputs vector pairs in the model. We could simplify it as:

$$\sum_{i=0}^{n}\Big(w_i a_i + b_i\Big). \tag{3.2.3}$$

The sum is computed across all entries in a. For instance, if we possess a neural network consisting of 3 vector pairs, then we can write the equation as:

$$\sum_{i=0}^{3}\Big(w_i a_i + b_i\Big) = (w_0 a_0 + b_0) + (w_1 a_1 + b_1) + (w_2 a_2 + b_2) + (w_3 a_3 + b_3). \tag{3.2.4}$$

3.3. EXAMPLE OF AN ARTIFICIAL NEURAL NETWORK

We talked about the notation of activation functions and the structure of a neural network in the previous chapter. But in this part, we will show an example of an artificial neural network and how it works.This example has been created inspired by [6].

The example shown here is the EMNIST dataset. The EMNIST dataset extends the structure of MNIST, which is composed of only digits, to include a broader range of classification tasks with both letters and digits. Therefore, it is an appropriate dataset for recognizing handwritten letters and digits. EMNIST has different sub-datasets, each with varying numbers of training and test samples. The most suitable sub-dataset for creating examples of the letter "A" is EMNIST Letters. EMNIST Letters contains only 26 uppercase letters.

Suppose you have a handwritten letter **A**, as shown in the image below Figure 3.2.

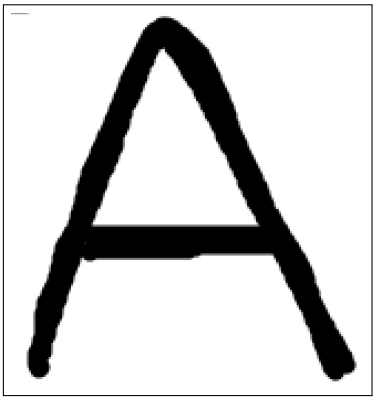


Figure 3.2. A handwritten letter **A** from the EMNIST dataset

Each letter in the EMNIST dataset has a size of 28x28 pixels. This means each letter has a

total of 28x28 = 784 pixels. Each pixel value falls within the range of 0.0 to 1.0, where 0.0 denotes a fully white pixel and 1.0 signifies a fully black pixel. Intermediate values denote different shades of gray, transitioning from lighter to darker tones.

In the neural network we use, the input layer has 784 neurons, and there are two hidden layers, each containing 16 neurons. The hidden layers extract features from the image and identify patterns. The output layer has 26 neurons, each representing a letter from A to Z.

The neural network takes signals from 784 input neurons to recognize the handwritten letter "A." The hidden layers work to identify edges and patterns in the image. The output layer has 26 neurons, and the neuron with the highest activation determines which letter is predicted. A cost function is used to avoid incorrect results, allowing the neural network to optimize. The cost function commonly used for classifying handwritten letters is known as cross-entropy or categorical cross-entropy. Cross-entropy operates in conjunction with the softmax activation function employed at the neural network's output layer. Softmax assigns a probability to each class, with the total probability summing to 1. Cross-entropy operates by increasing the probability of the correct class while decreasing the probabilities of incorrect classes. The cost function ensures the neural network minimizes incorrect results and optimizes for correct results. The neural network processes a 28x28 pixel image input to predict the correct letter.

In our network, we have $[(784*16)+(16*16)+(16*26)] = 13216$ weights and $[(16+16+26)] = 58$ biases. Total of 13274 parameters. In the input layer, the pixels in the image have values between 0.0 and 1.0. For EMNIST Letters, the input layer consists of pixels, and the hidden layers perform operations such as edge detection, like this:

In the second hidden layer, patterns such as triangles and lines are detected. For example, letters like Y and H contain lines, while on the other hand, the letter A features an upper triangle. We can think of the second hidden layer as a filter that detects patterns in the image. Thus, if two of these hidden neurons are activated, it could indicate that the letter is A. Of course, this is not the only possible arrangement. There are other ways to detect the letter A, but this configuration can lead to the correct conclusion that the letter is A.

It is important to note that, due to the intricate nature of neural networks, such explanations

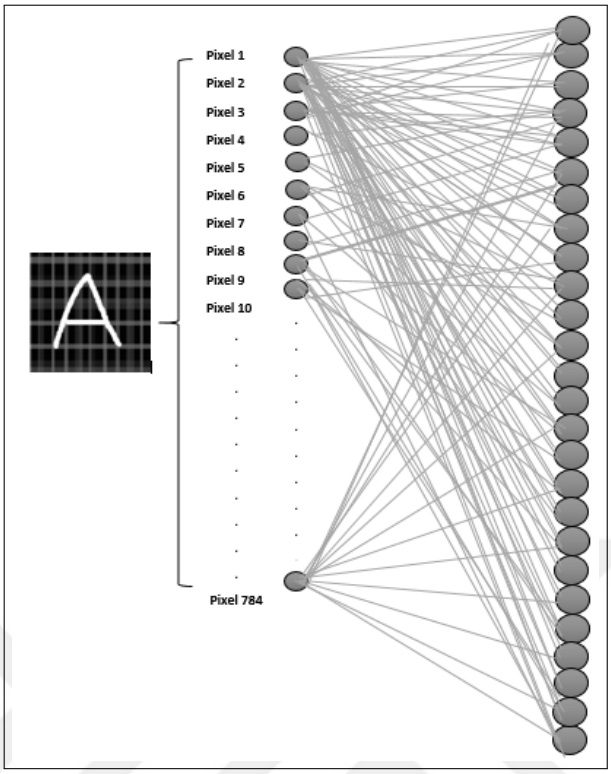


Figure 3.3. Layer of neural network for EMNIST

do not inherently indicate a particular arrangement or configuration. In practical applications, the outputs of neural networks may differ based on the training process, model architecture, and various other factors. For the EMNIST Letters dataset, the results generated by a neural network model can fluctuate depending on the training methodology and design employed.

You might ask: Why do we use 26 output neurons, and what are the disadvantages of using fewer output neurons? Here's the explanation: Using 26 output neurons to recognize handwritten letters is a common approach because it represents the 26 letters of the English alphabet. However, there are some challenges and consequences of using fewer output neurons. Representing 26 letters with fewer output neurons can lead to some letters representing more than one category simultaneously, which can reduce the model's ability to accurately identify each letter and increase the error rate.

When we perform the calculations, we see that each output neuron in the output layer gives a value between 0.0 and 1.0. Each neuron can be thought of as a function that takes the output from all the neurons in the previous layer and returns a value between 0.0 and 1.0. It's more accurate to think of the entire network as a function that takes 784-pixel values as input and

Table 3.1. Hidden layer pixels

First Piece	Second Piece	Third Piece	Fourth Piece
			

returns 26 values between 0.0 and 1.0 as output.

Suppose we have a mathematical calculation like this:

$$\begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ \dots & \dots & \dots & \dots & \dots \\ w_{k,0} & w_{k,1} & w_{k,2} & \dots & w_{k,n} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

$$a_0^{(1)} = \sigma\left(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + \cdots + w_{0,n}a_n^{(0)} + b_0\right) \tag{3.3.1}$$

$$a^{(1)} = \sigma\left(W^{(1)}a^{(0)} + b^{(1)}\right) \tag{3.3.2}$$

If we write (3.3.2) more generally, we will get,

$$a^{(l)} = \sigma\left(W^{(l)}a^{(l-1)} + b^{(l)}\right) \in \mathbb{R}^{n_l} \text{for} \quad l = 2, 3, \dots, L \quad \text{where} \quad L \text{ is the number of layers.} \tag{3.3.3}$$

Before starting to train the network, if all weights and biases are initialized completely randomly, the network might perform poorly on a given example. The network can sometimes

be confusing and produce incorrect outputs because it is essentially just doing something at random. To prevent this, we have a cost function.

Assuming we have N data points, or training samples, in $\mathbb{R}^{n_1}$, denoted as $[x^{(i)}]_{i=1}^{N}$, with corresponding target outputs $[y(x^{(i)})]_{i=1}^{N}$ in $\mathbb{R}^{n_L}$, we seek to minimize the quadratic cost function described as follows:

$$Cost = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|y(x^{(i)}) - a^{(L)}(x^{(i)})\|_2^2, \tag{3.3.4}$$

where $y(x^{(i)})$ is the correct output for the i-th training point, and $a^{(L)}(x^{(i)})$ is the output of the network for the i-th training point.

In other words, the cost function is the average sum of squared errors (or differences) between the correct output and the output of the network.

The cost function (3.3.4) gives you the exact result you want. For example, an incorrect output neuron should have a value of 0.0, while a correct output neuron should have a value of 1.0. Our goal is to minimize the cost function using various methods. This can be accomplished through the stochastic gradient descent method. The next section explains how to do this.

3.4. STOCHASTIC GRADIENT

In the prior section, we saw that training a neural network involves choosing weights and biases to minimize an objective function. These weights and biases, represented as matrices and vectors, can be consolidated into a single vector p, simplifying the optimization process.

Referring to our example, Figure 3.3 shows a network with 13,274 weights and biases, represented by the parameter vector $p \in \mathbb{R}^{13274}$. More generally, we denote $p \in \mathbb{R}^s$ to capture all parameters, with the cost function in (3.3.4) written as $Cost(p)$, or $Cost : \mathbb{R}^s \to \mathbb{R}$, to indicate its dependency on p. Our goal is to minimize this cost function using the ***steepest descent*** or ***gradient descent*** method, an iterative approach that computes a series of points in $\mathbb{R}^s$ to find the optimal p that minimizes the cost.

In a broader framework, we assume $p \in \mathbb{R}^s$ and define the cost function in (3.3.4) as $Cost(p)$,

expressed as $Cost : \mathbb{R}^s \to \mathbb{R}$, to highlight its dependence on the parameters. If Δp is minimal, then by ignoring higher-order terms of $\|\Delta p\|^2$, a Taylor series expansion of $Cost(p + \Delta p)$ around p produces

$$Cost(p + \Delta p) \approx Cost(p) + \sum_{r=1}^{s} \frac{\partial Cost(p)}{\partial p_r} \Delta p_r. \tag{3.4.1}$$

For simplicity, we will let $\nabla Cost(p) \in \mathbb{R}^s$ represent the vector consisting of partial derivatives, alternatively referred to as the **gradient**, so that

$$(\nabla Cost(p))_r = \frac{\partial Cost(p)}{\partial p_r}.$$

Then (3.4.1) can be written as

$$Cost(p + \Delta p) \approx Cost(p) + \nabla Cost(p)^T \Delta p. \tag{3.4.2}$$

The goal is to minimize the cost function's value. By the equation (3.4.2), the idea of choosing Δp is to make $\nabla Cost(p)^T \Delta p$ as negative as possible. We can see that if we choose Δp so that $\nabla Cost(p)^T \Delta p < 0$, then $\nabla Cost(p + \Delta p) < Cost(p)$.

To understand this, assume we have $\vec{W} = (w_0, w_1, w_2, \ldots, w_{13215}, w_{13216})$ and $-\nabla Cost(\vec{W}) = (0.31, 0.03, -1.25, \ldots, -0.37, 0.16)$ by our example. This tells us that w_0 should be increased somewhat, w_1 should be increased a little, w_2 should be decreased a lot, w_{13215} should be decreased a little, and w_{13216} should be increased a little, and so on.

To conclude, the way the gradient descent algorithm works is to compute the gradient $\nabla Cost$ iteratively and then to move in the opposite direction, essentially "falling down" the slope of the valley. We can visualize this as a ball rolling down a hill (like that in Figure 3.4), with the gradient pointing in the direction of the steepest slope.

This problem can be addressed by using the **Cauchy-Schwarz inequality**, which expresses that for any $f, g \in \mathbb{R}^s$, we have $|f^T g| \leq \|f\|_2 \|g\|_2$. So the minimum value that $f^T g$ can reach is $-\|f\|_2 \|g\|_2$, which occurs when $f = -g$. Thus, according to (3.4.2), we ought to

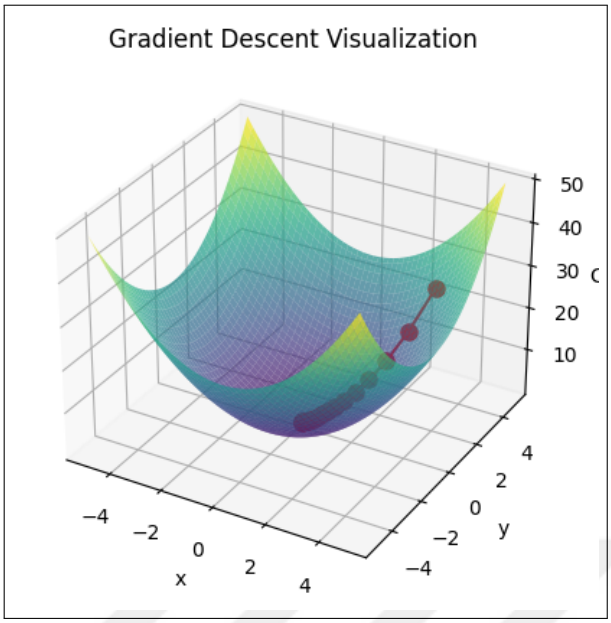


Figure 3.4. Gradient descent visualization

select Δp to align with the direction of $-\nabla Cost(p)$.

We recognize that (3.4.2) serves as a valid estimation only for minor Δp. Hence, we will restrict ourselves to a minimal adjustment in that direction. This results in the following update:

$$p \to p - \eta \nabla Cost(p). \tag{3.4.3}$$

Here, η denotes a small positive quantity known as the ***learning rate*** in this context. Equation (3.4.3) describes **the steepest descent method**. In this technique, we start with an initial vector and repeatedly apply equation (3.4.3) until we reach an acceptable solution or until the number of iterations surpasses the computational threshold.

The cost function includes the sum of terms that run over the training data. It can be concluded that $\nabla Cost(p)$ comprises individual partial derivatives aggregated across the training data. More descriptively,

$$C_{x^{(i)}} = \frac{1}{2}\|y(x^{(i)}) - a^{(L)}(x^{(i)})\|_2^2. \tag{3.4.4}$$

Then from (3.3.4), we have

$$\nabla Cost(p) = \frac{1}{N} \sum_{i=1}^{N} \nabla C_{x^{(i)}}(p). \tag{3.4.5}$$

When dealing with a vast amount of training data, calculating the gradient vector (3.4.5) by summing over the entire dataset becomes impractical, and performing (3.4.3) may be computationally prohibitive. Instead, we can employ a more efficient technique known as ***stochastic gradient descent***. This approach substitutes the mean of gradients calculated from all training instances with the gradient obtained from a randomly selected single training instance. A single step in this process can be outlined as follows:

1. Choose an integer i uniformly at random from $(1, 2, \ldots, N)$.

2. Then update

$$p \to p - \eta \nabla C_{x^{(i)}}(p). \tag{3.4.6}$$

In summary, during each iteration, the stochastic gradient method selects a random training point to approximate the entire training dataset. This frequent updating allows for faster convergence, and as more training points are encountered over time, it can lead to a meaningful reduction in the cost function with each iteration, improving the overall training efficiency. It's important to note that, despite using a small η, the update (3.4.6) doesn't ensure an overall reduction in the cost function.

The presented stochastic gradient method (3.4.6) represents the most basic form among numerous possibilities available. To elaborate further, the index i in (3.4.6) is selected through *sampling with substitution*. Once a training point is utilized, it's reinstated into the training set and remains eligible for use in subsequent iterations.

We have another option, which involves sampling without replacement, allowing us to iterate through each of the N training points in a random sequence.

Performing N steps is referred to as completing an *epoch*[1] and can be summarized as follows:

[1] An epoch in a neural network is the training of the neural network with all the training data for one cycle. For example, if you have a training dataset consisting of 1000 samples and your model processes this dataset 10 times during the training process, then in this case, 10 epochs have occurred.

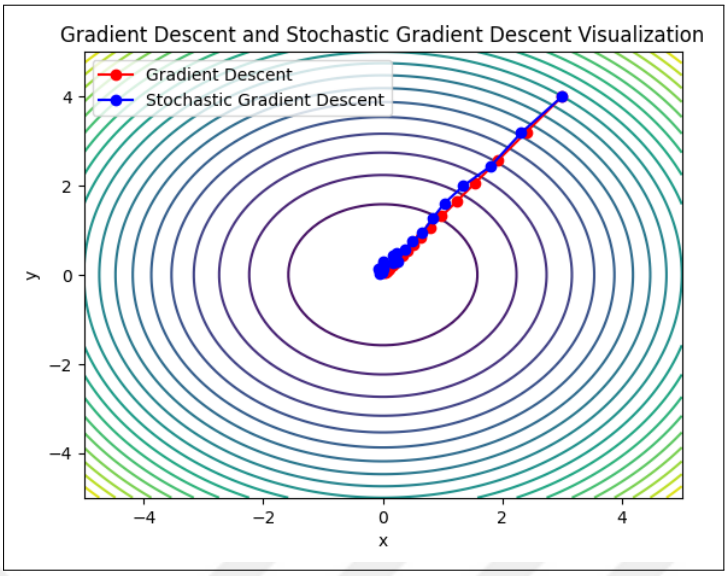


Figure 3.5. An example illustrating the dynamics of gradient descent and stochastic gradient descent for a function

1. Mix the integers (1,2,. . . ,N) into a new order, say $(k_1, k_2, \ldots, k_N)$.

2. For $i = 1, 2, \ldots, N$, then update

$$p \to p - \eta \nabla C_{x^{(k_i)}}(p). \tag{3.4.7}$$

If we view the stochastic gradient method as an estimation technique, it replaces the average of all training samples in (3.4.5) with a single randomly selected point. While this may increase update variability, it allows for more frequent updates and faster exploration of the parameter space, balancing accuracy and efficiency in gradient estimation.Suppose we have $m \in \mathbb{N}$, then we may employ the subsequent approximation:

1. Choose m integers $(k_1, k_2, \ldots, k_m)$, uniformly at random from (1,2,. . . ,N).

2. Then update

$$p \to p - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x^{(k_i)}}(p). \tag{3.4.8}$$

In this iteration, the set $(x^{(k_i)})_{i=1}^{m}$ is termed as ***minibatch***.[2] There is another way where we

[2] Minibatch processing involves splitting the dataset into small subsets, called minibatches, instead of using the entire dataset at once. For instance, with a dataset of 1000 samples and a minibatch size of 50, the model is updated with 50 samples per iteration. After 20 iterations, one epoch is completed.

assume $N = Km$ for some $K \in \mathbb{Z}$, without replacement alternative, we divide the training data set arbitrarily into K different mini-batches and cycle through them.

There are many choices that go into designing an algorithm, such as the size of a minibatch and the type of random selection. These decisions are frequently influenced by the requirements of high-performance computational frameworks. And there is also the option to change these throughout training in an effort to expedite convergence. In reality, accurately determining the learning rate is also an essential factor for practical implementations.

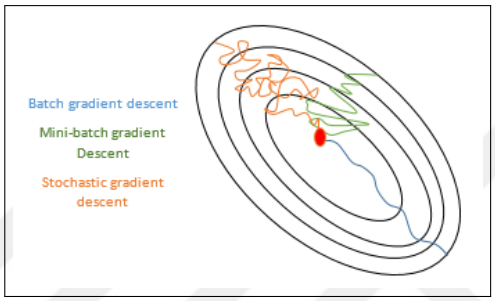


Figure 3.6. It demonstrates the convergence rate of stochastic gradient descent and minibatch gradient descent

Figure 3.6 is an evaluation comparing the convergence rate of stochastic gradient descent and minibatch gradient descent on a quadratic objective function.

The results in Figure 3.6 indicate that minibatch gradient descent converges faster than stochastic gradient descent.

3.5. BACKPROPAGATION

We are now ready to implement the stochastic gradient approach for training the artificial neural network. In this section, we will concentrate on the individual components of the weight matrix and bias vector, moving away from the broader parameter vector p mentioned earlier. Our objective is to calculate the partial derivatives of the cost function concerning each $w_{jk}^{[l]}$ and $b_j^{[l]}$.

The fundamental concept of the stochastic gradient approach is to approximate the gradient of the loss function by using the gradient calculated from an individual training instance.Since the cost function (3.3.4) is essentially a linear combination of the individual terms

corresponding to all training samples, its partial derivatives follow a similar pattern.

Therefore, for fixed training points, we consider $C_{x^{(i)}}$ of (3.4.4) in relation to the weights and biases and the dependence on $x^{(i)}$

$$C = \frac{1}{2}\|y - a^{[l]}\|_2^2 \tag{3.5.1}$$

The output of the artificial neural network is given by $a^{[l]}$. The weights and biases affect C solely through their influence on $a^{[l]}$. To derive a useful expression for the partial derivatives, it is logical to introduce two more sets of variables. Firstly, let

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l} \qquad \text{for} \quad l = 2, 3, \ldots, L. \tag{3.5.2}$$

This implies that $z^{[l]}$ represents the ***weighted input*** to neuron j in layer l. The basic equation (3.3.3) governing the flow of information within the network can be expressed in the following manner.

$$a^{[l]} = \sigma(z^{[l]}) \quad \text{for} \quad l = 2, 3, \ldots, L. \tag{3.5.3}$$

Secondly, we let $\delta^{[l]} \in \mathbb{R}^{n_l}$ and can be defined as

$$\delta^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} \qquad \text{for} \quad 1 \leq j \leq n_l \quad \text{and} \quad 2 \leq l \leq L. \tag{3.5.4}$$

Equation (3.5.4) is usually called the ***error*** of neuron j in layer l. This equation serves as an intermediary value that proves valuable for both analysis and computation. In general, it is unclear to what extent each neuron should be "blamed" for the discrepancy in the final output. At power level L, equation (3.5.4) does not directly quantify these discrepancies. However, this provides a measure of the *local* error for every neuron in the output layer. The idea of calling $\delta_j^{[l]}$ with (3.5.4) is that the error is occurring because the function cost is only at the minimum value. It seems that if all partial derivatives are zero. Therefore, the idea that

$\delta_j^{[l]} = 0$ is a sensible goal. To enhance accuracy, it is preferable to assume that $\delta_j^{[l]}$ indicates how the cost function responds to changes in the weighted inputs of neurons j in layer l, or simply $z_j^{[l]}$.

To understand the lemma that follows, we need to introduce the Hadamard or element-wise product of two vectors. Let $u, v \in \mathbb{R}^n$, then the Hadamard product of $u \circ v \in \mathbb{R}^n$ is defined as $(u \circ v)_i = u_i v_i$ for $1 \leq i \leq n$. Put differently, the Hadamard product results from multiplying corresponding entries of two vectors pairwise.

Equipped with this understanding, let's now introduce the following lemma. The following lemma is quoted from the book [4].

Lemma 3.5.1. *We have*

$$\delta^{[l]} = \sigma'(z^{[L]}) \circ (a^L - y), \tag{3.5.5}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \quad \textit{for } 2 \leq l \leq L-1, \tag{3.5.6}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]} \quad \textit{for } 2 \leq l \leq L, \tag{3.5.7}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \quad \textit{for } 2 \leq l \leq L. \tag{3.5.8}$$

Proof. We start with (3.5.5). The relation (3.5.3) with $l = L$ shows that $z_j^{[L]}$ and $a_j^{[L]}$ are related by $a^{[L]} = \sigma(z^{[L]})$, and hence

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}). \tag{3.5.9}$$

Also, from (3.5.1) we have

$$\frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \frac{1}{2} \sum_{k=1}^{n_L} (y_k - a_k^{[L]})^2 = -(y_j - a_j^{[L]}). \tag{3.5.10}$$

So, by the chain rule, we get

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j)\sigma'(z_j^{[L]}), \tag{3.5.11}$$

This represents the componentwise form of (3.5.5).

To demonstrate (3.5.6), we employ the chain rule to transition from $z_j^{[l]}$ to $[z_k^{[l+1]}]_{k=1}^{n_{l+1}}$. Utilizing the chain rule and referring to the definition (3.5.4), we get

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}. \tag{3.5.12}$$

Now, by (3.5.2) we know that $z_k^{[l+1]}$ and $z_j^{[l]}$ are related via

$$z_k^{[l+1]} = \sum_{s=1}^{n_l} w_{ks}^{[l+1]} \sigma(z_s^{[l]}) + b_k^{[l+1]}. \tag{3.5.13}$$

Thus,

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \sigma'(z_j^{[l]}). \tag{3.5.14}$$

By (3.5.12) this gives us

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{kj}^{[l+1]} \sigma'(z_j^{[l]}), \tag{3.5.15}$$

which can be rearranged as

$$\delta_j^{[l]} = \sigma'(z_j^{[l]}) \left((W^{[l+1]})^T \delta^{[l+1]} \right)_j. \tag{3.5.16}$$

This is the componentwise form of (3.5.6).

To show (3.5.7), we know that $z_j^{[l]}$ and $b_j^{[l]}$ are related from (3.5.2) and (3.5.3) as

$$z_j^{[l]} = \left(W^{[l]}\sigma\left(z^{[l-1]}\right)\right)_j + b_j^{[l]}. \tag{3.5.17}$$

Since, $z^{[l-1]}$ does not depend on $b_j^{[l]}$, we get

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \tag{3.5.18}$$

So, by the chain rule, we have

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} = \delta_j^{[l]}, \tag{3.5.19}$$

by the definition (3.5.4). Hence, this gives us (3.5.7).

To achieve the final result (3.5.8), we initiate from the componentwise rendition of (3.5.2),

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}, \tag{3.5.20}$$

which gives us

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]}, \quad \text{independently of } j, \tag{3.5.21}$$

and

$$\frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = 0, \quad \text{for } s \neq j. \tag{3.5.22}$$

As the neuron at position j in layer l exclusively uses the weights from the j-th row of the matrix $W^{[l]}$, it subsequently applies these weights in a linear fashion. Then, from the chain rule, (3.5.21) and (3.5.22) we get

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} a_k^{[l-1]} = \delta_j^{[l]} a_k^{[l-1]}, \tag{3.5.23}$$

The final step relied on the definition of $\delta_j^{[l]}$ from (3.5.4). Hence, this concludes the validation of the backpropagation equations. ■

Lemma (3.5.1) exhibits several characteristics. Remembering (3.3.2), (3.5.2), and (3.5.3), we can ascertain the output $a^{[L]}$ by conducting the *Forward Pass* throughout the network.

Once we do this, we observe from (3.5.5) that $\delta^{[L]}$ is readily accessible. Then, from (3.5.6), $[\delta^{[L-1]}, \delta^{[L-2]}, \ldots, \delta^{[2]}]$ can be calculated using *backward pass* through the network. You can access the partial derivatives from (3.5.7) and (3.5.8). Therefore, calculating gradients in this way is called ***backpropagation***.

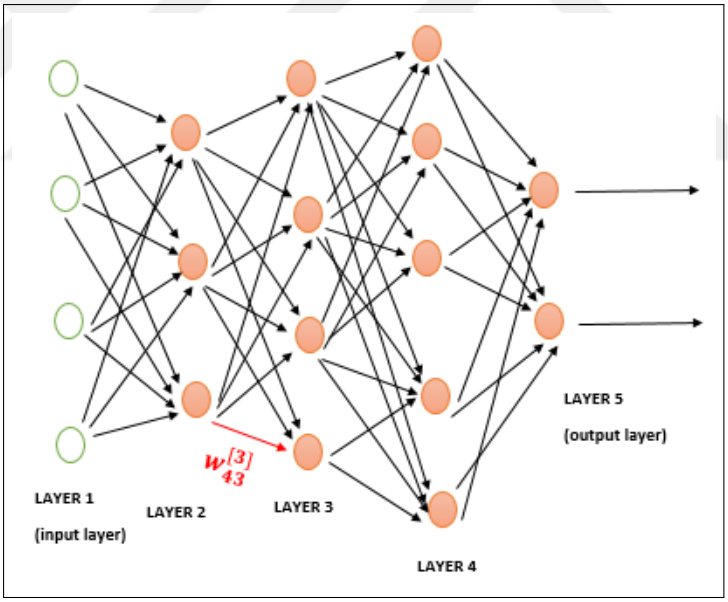


Figure 3.7. Sample network

To grasp the backpropagation formulas (3.5.7) and (3.5.8) in Lemma (3.5.1), it's beneficial to revisit the basic definition of partial derivatives. The term $\frac{\partial C}{\partial w_{jk}^{[l]}}$ indicates the sensitivity of C to a slight adjustment in $w_{jk}^{[l]}$. As an example, consider the following neural network in Figure 3.7. In the network, it highlights the weight $w_{43}^{[3]}$. Since altering this weight doesn't impact the outputs of prior layers, determining $\frac{\partial C}{\partial w_{43}^{[3]}}$ doesn't require knowledge of the partial derivatives of preceding layers. In finer detail, the activation flowing into the fourth neuron

in layer 3 is denoted as $z_4^{[3]}$, and $\delta_4^{[3]}$ quantifies how sensitive C is to this input. So, in regard to this neuron we have $w_{43}^{[3]} a_3^{[2]}$ + constant, then we get

$$\frac{\partial C}{\partial w_{43}^{[3]}} = \delta_4^{[3]} a_3^{[2]}. \tag{3.5.24}$$

Similarly, for the bias, we have a constant term added to $b_4^{[3]}$ that feeds into the neuron, providing us with

$$\frac{\partial C}{\partial b_4^{[3]}} = \delta_4^{[3]} \times 1. \tag{3.5.25}$$

We can avoid the need for Hadamard product notation in equations (3.5.5) and (3.5.6) by utilizing diagonal matrices.

Let $D^{[l]} \in \mathbb{R}^{n_l \times n_l}$ denote the **diagonal matrix** whose (i,i) entry is $\sigma'(z_i^{[l]})$. Thus, we can express $\delta^{[l]} = D^{[l]}(a^{[L]} - y)$ and $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$. This allows us to write:

$$\delta^{[l]} = D^{[l]}(W^{[l+1]})^T D^{[l+1]}(W^{[l+2]})^T \dots D^{[L-1]}(W^{[L]})^T D^{[L]}(a^{[L]} - y). \tag{3.5.26}$$

Equation (3.5.7) demonstrates that $\delta^{[l]}$ represents the gradient of the cost function concerning the bias in layer l. When interpreting $\frac{\partial C}{\partial w_{jk}^{[l]}}$ as the (j,k) element in a matrix of partial derivatives for layer l, equation (3.5.8) illustrates that this resulting matrix is obtained from the outer product of $\delta^{[l]}$ and $a^{[l-1]} \in \mathbb{R}^{n_l \times n_{l-1}}$.

With this understanding, we can proceed to outline the pseudocode for the backpropagation process, which trains the network over a predetermined number of stochastic gradient iterations, N_{iter}. During each iteration, a forward pass computes the final output $a^{[L]}$, followed by a backward pass that assesses errors and applies necessary corrections.

The following code is quoted from book [4].

Algorithm 3.1. Principal component analysis algorithm

Input: $Niter$ number of iterations, η learning rate, $W^{[1]}, \ldots, W^{[L]}$ initial weights, $b^{[1]}, \ldots, b^{[L]}$ initial biases
Output: $W^{[1]}, \ldots, W^{[L]}$ final weights, $b^{[1]}, \ldots, b^{[L]}$ final biases
for $i = 1$ upto $Niter$ (choose i uniformly random) **do**
    $x^{(i)}$ is the current training data sample.
    $a^{[1]} = x^{(i)}$
    for $l = 2$ upto L **do**
        $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
        $a^{[l]} = \sigma(z^{[l]})$
        $D^{[l]} = \mathrm{diag}(\sigma'(z^{[l]}))$
    end for
    $\delta^{[L]} = D^{[L]}(a^{[L]} - y(x^{[i]}))$
    for $l = L - 1$ downto 2 **do**
        $\delta^{[l]} = D^{[L]}(W^{[l+1]})^T \delta^{[l+1]}$
    end for
    for $l = L$ downto 2 **do**
        $W^{[l]} \rightarrow W^{[l]} - \eta \delta^{[l]} (a^{[l-1]})^T$
        $b^{[l]} \rightarrow b^{[l]} - \eta \delta^{[l]}$
    end for
end for

4. IMPLEMENTATION

In this chapter, we will implement the backpropagation algorithm and stochastic gradient method in Matlab. At the conclusion of the preceding chapter's segment, we will build a simple network to understand the pseudocode. Note that: This section's example is inspired by the paper [2]. The input data has been altered to generate a new graph.

4.1. BACKPROPAGATION IN PRACTICE WITH MATLAB

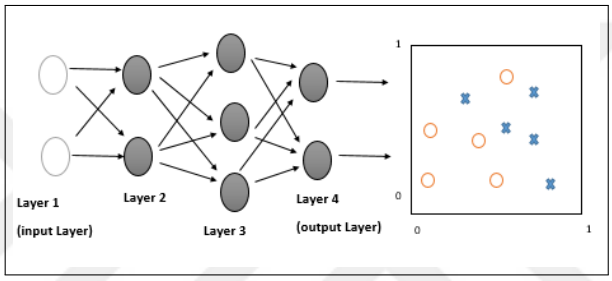


Figure 4.1. Network with four layers and labeled data points

The following Matlab code demonstrates how a network resembling the one depicted on the left can be implemented using the data displayed on the right. The code is written in such a way that it can be easily adapted to networks of different sizes. This code has been optimized for clarity and brevity, not for speed and efficiency. Specifically, the layer count has been repeated during forward and backward passes for the cache line. Weights and biases do not have the same dimensions, so they cannot be stored in a three-dimensional array. Instead, store the weights and biases in two-dimensional arrays and use *cell* or struct arrays to implement forward and backward passes through the **for** loop.

Listing 1 shows that the function **activate** computes the activation of the network for a given input through the activation function σ. Activation functions are forward and backward pass implementations.

The function **netbp** in Listing 2 includes an internal function named **cost**, which calculates a scaled variant of the cost function, as described in (3.3.4). Since these functions are nested, the training data can be accessed from the main function. Additionally,the nested **cost** function is invoked at each step of the stochastic gradient approach, even though it does not

directly participate in the forward or backward passes. This feature enables you to monitor the training process. To observe the changes, make sure to remove the semicolon at the end of line 35.

At the onset of the **netbp** algorithm, we first prepare the training dataset and initialize the network's weights and biases using the widely utilized pseudo-random number generator, **randn**. To simplify the training process, we chose a constant learning rate of **eta = 0.05** and set the total number of iterations to **Niter = 1e6**. The network training was carried out using the stochastic gradient descent (SGD) algorithm, as outlined in section 3.4. Additionally, the **randi(10)** function now generates a uniformly distributed random integer within the range of 1 to 10 at each iteration, randomly selecting a training data point for that step.

We further employed the **semilogy** function to graphically represent the evolution of the cost function with respect to the number of iterations performed by the stochastic gradient algorithm. During initial trials, we observed that the starting values of the weights and biases produced a cost function value of 6.3. However, after performing 10^6 iterations of the stochastic gradient descent method, the cost function value drastically decreased to 7.3×10^{-4}, signaling a significant performance gain. As illustrated in Figure 4.2, the cost function demonstrates a consistent downward trend as the iteration count grows. Once this initial rapid decline levels off (plateaus), the cost reduction continues at a much slower linear rate. Notably, the difference between consecutive cost values stabilizes, with the ratio between them falling within an approximate tolerance of 10^{-6}.

Algorithm 4.1. Matlab code listing 1 example

```
1
2 % Activate Evaluates sigmoid function
3 function y = activate(x, W, b)
4 % x is the input vector, W is the weight matrix
5 % b is the bias vector, y is the output vector
6 % The ith element of y is activate((Wx+b)_i)
7 % Where activate(z) = 1/(1+exp(-z))
8 y = 1/(1+exp(-(W*x+b)));
9 end
```

Algorithm 4.2. Matlab code listing 2 example

```
1
2  % NETB Uses backpropagation to train the network
3  function netbp
4  % Data
5  x_1 = [0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7];
6  x_2 = [0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6];
7  x_3 = [0.1,0.2,0.6,0.9,0.4,0.5,0.1,0.2,0.3,0.8];
8  y = [ones(1,5) zeros(1,5); zeros(1,5) ones(1,5)];
9  % Initializing the weights and biases
10 rng(5000);
11 W_2 = 0,5*randn(2,3); W3 = 0.5*randn(3,3) ; W4 = 0.5*randn(2,3);
12 b_2 = 0,5*randn(2,1); b3 = 0.5*randn(3,1) ; b4 = 0.5*randn(2,1);
13 % Forward and Back propagation
14 eta = 0.05;    % Learning rate eta
15 Niter = 1e6;  % Number of Stochastic Gradient iterations
16 % Cost function value at each iteration
17 savecost = zeros(Niter, 1);
18 for counter = 1: Niter
19     k = randi(10);
20     x = [x_1(k); x_2(k); x_3(k)];
21     % Forward pass
22     a_2 = activate(x, W_2, b_2);
23     a_3 = activate(a_2, W_3, b_3);
24     a_4 = activate(a_3, W_4, b_4);
25     % Backward pass
26     delta_4 = a_4.*(1-a_4).*(a_4-y(:, k));
27     delta_3 = a_3.*(1-a_3).*(W_4'*delta_4);
28     delta_2 = a_2.*(1-a_2).*(W_3'*delta_3);
29     % Stochastic Gradient
30     W_2 = W_2 - eta*delta_2*x';
31     W_3 = W_3 - eta*delta_3*a_2';
32     W_4 = W_4 - eta*delta_4*a_3';
33     b_2 = b_2 - eta*delta_2;
34     b_3 = b_3 - eta*delta_3;
35     b_4 = b_4 - eta*delta_4;
36     % Monitoring
37     new_cost = cost(W_2, W_3, W_4, b_2, b_3, b_4);
38     savecost(counter) = new_cost;
39 end
40 % Display cost to screen
41 save costvec
42 semilogy(1:1e4:Niter, savecost(1:1e4:Niter))
43     function costval = cost(W_2, W_3, W_4, b_2, b_3, b_4)
44         costvec = zeros(10, 1);
45         for i = 1:10
46             x = [x_1(i); x_2(i); x_3(i)];
47             a_2 = activate(x, W_2, b_2);
48             a_3 = activate(a_2, W_3, b_3);
49             a_4 = activate(a_3, W_4, b_4);
50             costvec(i) = norm(y(:, i) - a_4,2);
51         end
52         costval = norm(costvec, 2)^2;
53     end
54 end
```

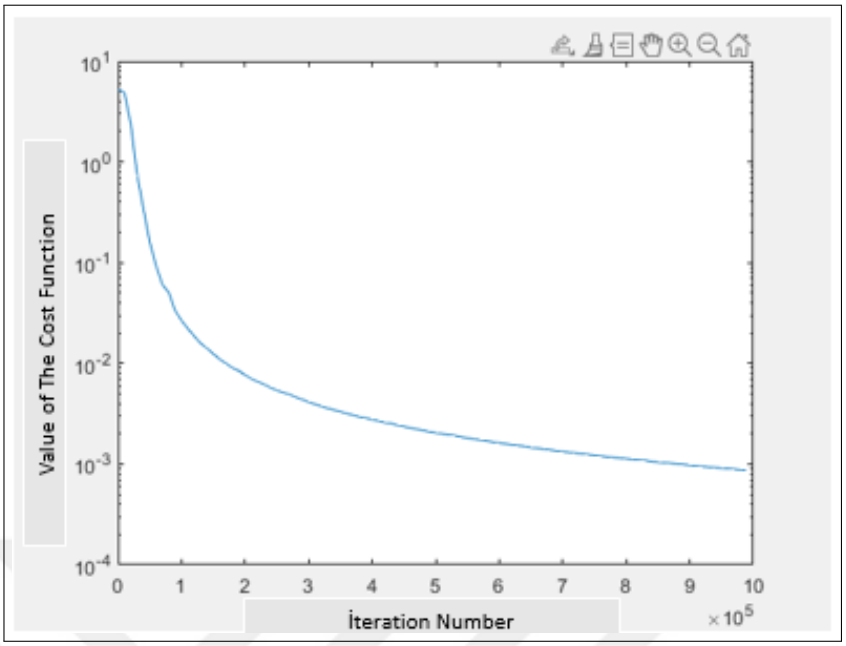


Figure 4.2. The y-axis represents the value of the cost function, while the x-axis represents the number of iterations performed using the stochastic gradient method

4.2. IMAGE CLASSIFICATION EXAMPLE

This part was inspired by the information in Chapter 10 of Jeff Heaton's book [7]. In this section, we explore the potential of neural networks and deep learning. We will train a neural network to classify images of various objects using the CIFAR-10 dataset provided by the Keras library. CIFAR-10 consists of 60,000 color images of size 32x32 pixels, distributed across 10 classes with each class containing 6,000 images. The dataset is split into 50,000 training images and 10,000 test images. Each image is labeled with one of 10 categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

To train our neural network, which is quite large and requires significant computation time, we will use TensorFlow and Google Colab. For constructing the neural network, we will utilize convolutional neural networks (CNNs), which are particularly effective for tasks involving image classification.

4.2.1. Convolutional Neural Networks

Convolutional neural networks (CNNs) are well-matched for the image classification task within a neural network. There are several differences according to ordinary neural networks; CNN include of neurons with weights and biases that can learn. Each neuron receives several inputs, performs a dot product, and may optionally apply a non-linear function afterwards. When using convolutional neural networks (CNNs), inputs are usually images, allowing us to embed specific properties into the architecture. This streamlines the forward function and reduces the network's parameters, enhancing efficiency and improving performance in tasks like image recognition. Here we tell shortly these layers and their properties. Then next chapter we will see how they can be assembled to form various architectures. Convolutional neural networks distinguish themselves from other neural networks due to their specialization in handling inputs such as images, speech, or audio signals.They categorize layers into three key classifications, which are:

- Convolutional Layer
- Pooling Layer
- Fully-Connected Layer

Convolutional Layer: The convolutional layer is the core structure block of CNN and it is where the majority of calculations occur. The primary role of the convolutional layer is to identify characteristics such as edges, lines, color patterns, and other visual elements. Filters can detect these characteristics. Increasing the number of filters in a convolutional layer enhances its ability to recognize diverse properties. The parameters of the convolutional layer are comprised of a collection of adaptable filters. Each filter, shaped like a square, traverses across the image, scanning its contents. Picture a grid, where each pixel represents a unique aspect of the image. In essence, the convolutional layer acts as a smaller grid, moving horizontally across each row of the image. Additionally, a hyperparameter dictates the dimensions of the square-shaped filter, determining its width and height.

Each pixel in a convolutional layer corresponds to a weight. Therefore, the total number of weights between a convolutional layer and the preceding layer or image region is as follows:

$$[FilterSize] \times [FilterSize] \times [\#ofFilters]$$

For example, if the filter size is 5 and there are 10 filters, then the total number of weights would be 250.

We will illustrate the operational principle of convolutional filters using Figure 4.3.

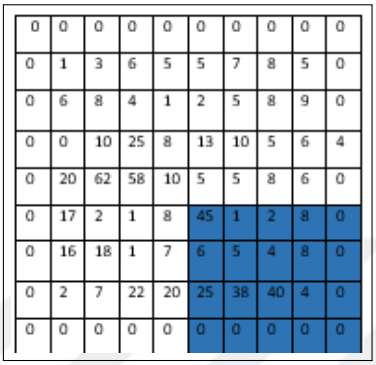


Figure 4.3. Work principle of convolutional layer

The preceding diagram illustrates a convolutional filter sized 4 with a padding of 1. Padding ensures the addition of zeros around the image processed by the filter, which helps prevent loss of information in edge pixels during the filtering process. Despite the actual image size being 8x7, additional padding enlarges it to 9x8. The stride indicates the step size of the convolutional filter as it moves across the image. Initially, the filter moves a certain stride to the right. When it reaches the right edge, it moves back and then steps down, repeating the process. This procedure enables filters to scan across the entire image.

Example 4.2.1. *This example is about the work principle calculus of convolutional filters in convolutional theory.*

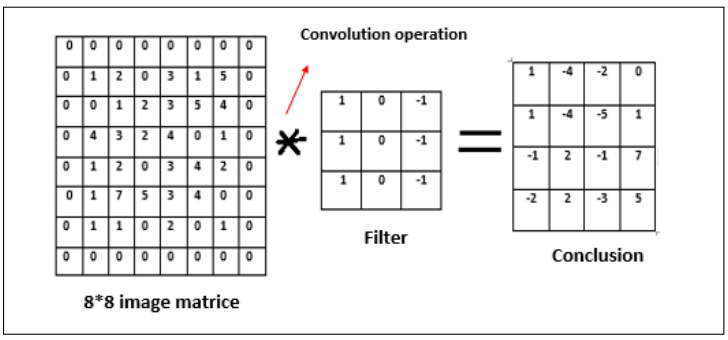


Figure 4.4. Work principle calculus of convolutional filters

Zeros are padded in the corners to ensure that edge information is not lost. Additionally, if you want a more effective filter for better results, you should consider using the Sobel filter, developed by Irwin Sobel [8].

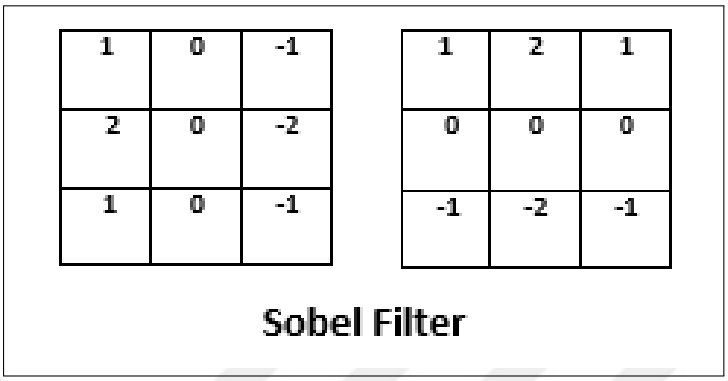
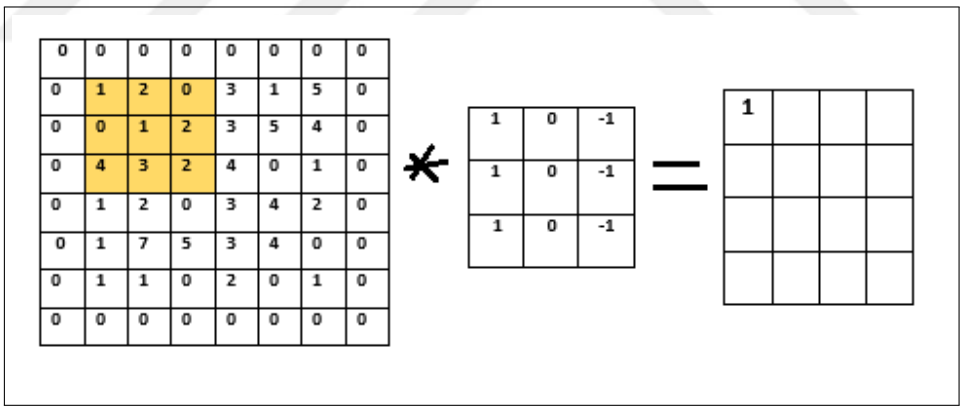


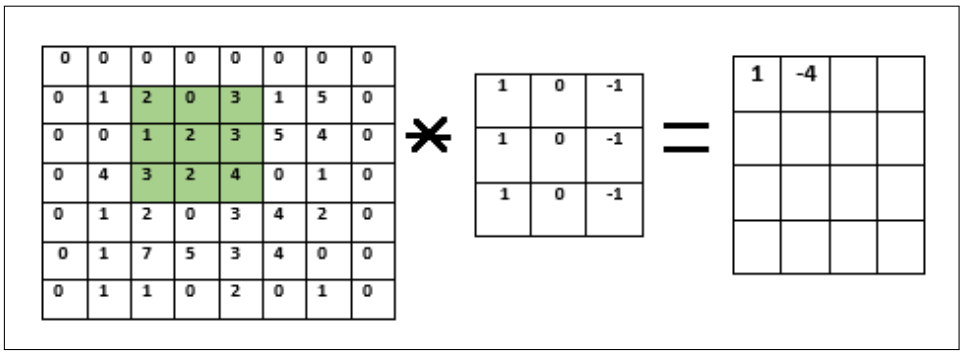
Figure 4.5. Sobel filter

Now we will step by step on how we can do this calculus.

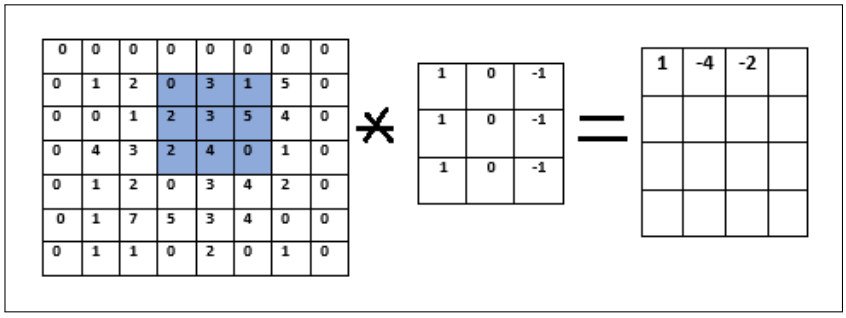
Step 1:



Step 2:

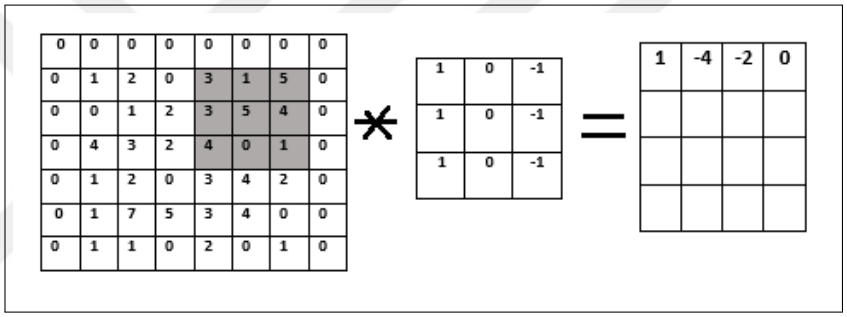


Step 3:



We are now moving on to the fourth step.

Step 4:



In the bottom row, we move from the zeros in the far left corner to the next column, continuing the process until we find the solution for the element in the second row and first column of the solution matrix.

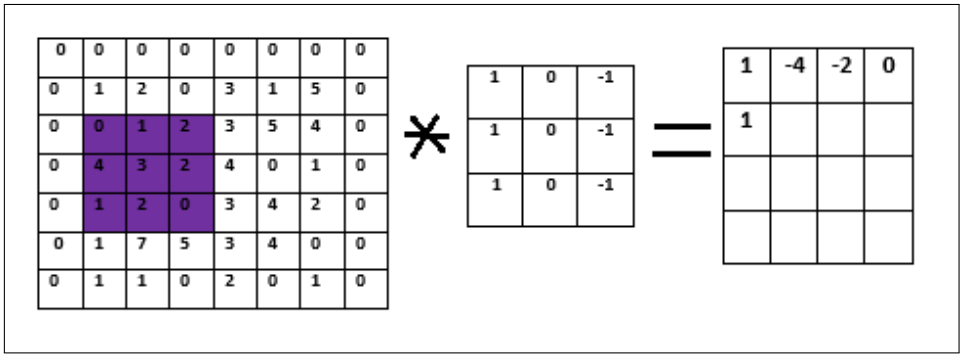


Figure 4.6. Step-by-step work principle calculus of convolutional filters

Then we continue this process for the other conclusions. Then we found a conclusion in Figure 4.4.

Pooling Layer: Pooling layers, also known as downsampling layers, are a form of nonlinear downsampling. They are used to reduce the dimensionality of feature maps while retaining important features. Pooling layers achieve this by summarizing the presence of features in patches of the feature map. There are two main types of pooling:

- Pooling Layer : The pooling layer averages the values within each region of the feature map. This reduces the data size while preserving important features. For each region, the average of all values is computed and assigned to a new feature map.
- The Max Pooling layer extracts the maximum value from each region of the feature map. This technique enhances prominent features and generates distinct feature maps. It identifies the maximum value within each region and assigns this value to a new feature map.

The primary distinction between these two layers lies in their approach to feature selection. The Max Pooling layer, by selecting the maximum value from each region of the feature map, tends to enhance crucial features more effectively, often leading to improved overall performance.

Fully-Connected Layer: While the pixel values of the input image are not directly linked to the output layer in partially connected layers, every node in the output layer establishes a direct connection with a node in the preceding layer in fully connected layers. These layers are commonly used for tasks such as classifying datasets, performing regression, or recognizing patterns.

Firstly, a linear transformation is applied by multiplying each input value by its associated weights and summing them. This produces a new output. Subsequently, this linear output undergoes processing with an activation function, typically ReLU, sigmoid, or tanh.

To obtain an output suitable for classification, the softmax activation function interprets the outputs as probability values among classes, normalizing them to sum up to 1.

4.2.2. Avoiding Overfitting

Overfitting occurs when the algorithm memorizes the results by working down to the lowest break on the training data and achieves success only on those data. When building the model, the aim is not for it to predict everything but to obtain general correctness, meaning finding a general pattern that can be applied to subsequent data as well. If the algorithm is too complex, instead of finding the pattern within the data, the learning process can result in memorizing the noise.

Some methods that can be used to prevent overfitting include: We can prevent overfitting by dividing the given data into training data and validation data. The training set is used to assess the model's generalization performance. This way, we can prevent the model from fitting too closely to the training set and exhibiting inadequate performance on the validation set. However, it is important to be cautious about conducting excessive experiments using the validation set and to implement proper measures to prevent the model from overfitting on the validation data.

Other methods can be listed as follows :

Gathering more data: Collecting more varied data improves the model's capacity to generalize and reduces the likelihood of overfitting.

Data Augmentation: Diversifying existing data allows the model to accommodate various variations.

Choosing a simpler model: Using a simpler model with fewer parameters can decrease overfitting.

Regularization techniques: Using techniques like L1 and L2 regularization helps control the model's overfitting.

Using Dropout: By randomly dropping units during training with a certain probability, dropout helps the network generalize better among different features.

Using Cross-validation: Splitting the data into training and validation sets and using cross-validation to objectively evaluate the model's performance.

However, these methods have their own advantages and disadvantages, which are as follows:

Dividing data into training and validation sets:

Advantages: Simple and easy to implement, usually yields quick results.

Disadvantages: If your dataset is small, the validation set may not be sufficiently representative. Additionally, the performance on the validation set can significantly influence the selection of model hyperparameters.

Other overfitting prevention methods:

Advantages: More specific and can help you better understand your model. For example, using dropout or regularization can make your model more robust.

Disadvantages: Some methods are more complex and may require more time to implement. Additionally, applying these methods in some cases can affect the model's performance or prolong training time.

To determine which method is best for you, it is recommended to evaluate both methods based on your available dataset, time, and resources. Especially for large and complex datasets, they can be used complementarily.

4.2.3. Methods for Activation and Cost

In sections 3.1 through 4.1, we utilized activation functions in the form of $\sigma(x) = \frac{1}{1+e^{-x}}$ and a quadratic cost function (3.3.4). However, in our image classification example, we will use *rectified linear unit*, also known as ReLU.

$$\sigma(x) = \begin{cases} 0 & \text{if } x \leq 0, \\ x & \text{if } x > 0, \end{cases} \tag{4.2.1}$$

as the activation function.

In this case, where the dataset $[x^{(i)}]_{i=1}^{N}$ is drawn from K unique classes, let $l_i \in 1, 2, \ldots, K$ represent the label assigned to the data point $x^{(i)}$. Rather than applying the quadratic loss

function (3.3.4), we can adopt the *softmax cross-entropy* loss method, as described below. Consider the output $a^{[L]}(x^{(i)})$, denoted by $v^{(i)}$ ($a^{[L]}(x^{(i)}) =: v^{(i)}$), obtained from the network, which is represented as a vector in $\mathbb{R}^K$. In this representation, the jth element of the vector is prominent when the image is predicted to belong to category j. Then, we may define the softmax operation as

$$(v^{(i)})_s \mapsto \frac{e^{v_s^{(i)}}}{\sum_{j=1}^{K} e^{v_j^{(i)}}} \quad \text{for } s = 1, \ldots, K. \tag{4.2.2}$$

We aim to encourage the softmax output for training point $x^{(i)}$ to approach unity as closely as possible in component l_i and to approach zero in all other components. By employing a logarithmic measure of error instead of a quadratic one, we derive the cost function

$$-\sum_{i=1}^{N} \log\left(\frac{e^{v_{l_i}^{(i)}}}{\sum_{j=1}^{K} e^{v_j^{(i)}}}\right). \tag{4.2.3}$$

4.3. IMAGE CLASSIFICATION EXPERIMENT

We will show how to train a Convolutional Neural Network (CNN) for image classification tasks using the Keras CIFAR-10 dataset with TensorFlow backend. The CIFAR-10 dataset consists of 32×32-pixel colored images, where each pixel contains three color channels (red, green, blue), representing each pixel as a 3-dimensional vector. Thus, an image is represented as a $32 \times 32 \times 3$-dimensional vector. The training dataset typically includes $50,000$ images, and the testing dataset includes $10,000$ images. Each image is 32×32 pixels and belongs to one of 10 distinct classes. The training images are used to train the model, while the testing images are used to evaluate the model's performance.

To better understand this, we can break down a typical deep learning architecture for a CIFAR-10 image classification model into blocks and provide explanations for each block. In this section, we present the sizes of inputs, outputs, and weights using a concise matrix format.

Block 1 includes a convolutional layer, a pooling layer, and an activation layer. Input tensor

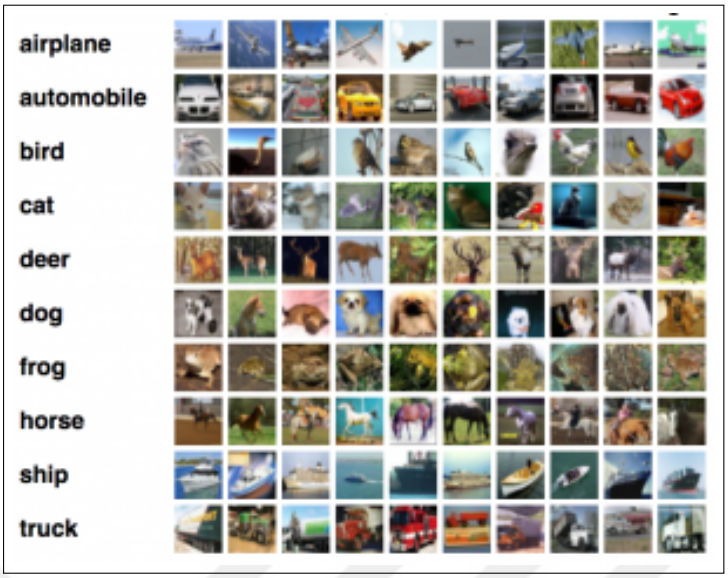


Figure 4.7. Sample images from CIFAR-10 [1]

dimensions are 32×32 pixels with 3 color channels (RGB), and output tensor dimensions are 16×16 pixels with 32 channels.

Convolutional Layer: Uses 32 filters of size 5×5, applied with stride 1. The weights are represented as a tensor of dimensions $(5, 5, 3, 32)$.

Pooling Layer (Max Pooling): Reduces each feature map's size by half, resulting in dimensions of 16×16.

Activation (ReLU): Converts negative values to zero after the convolution operation.

This block initializes the neural network by extracting fundamental features from the images before proceeding to deeper layers.

Block 2 comprises a convolutional layer followed by a pooling layer, which decreases the dimensions to $8 \times 8 \times 32$.

Convolutional Layer: Uses 32 filters of size 5×5. The weights are a tensor of shape $(5, 5, 32, 32)$, resulting in 32 feature maps of size 16×16 due to stride 1.

ReLU Activation: Converts negative values to zero.

Pooling Layer (Average Pooling): Reduces dimensions by half, resulting in output size of 8×8.

This block further extracts features while reducing dimensions in the neural network.

Block 3 involves convolution, activation, and pooling, reducing dimensions to $4 \times 4 \times 64$.

Convolutional Layer: Uses 64 filters of size 5×5, scanning across all 32 feature maps. The weights are a tensor of shape $(5, 5, 32, 64)$, maintaining 32 feature maps of size 8×8 due to stride 1.

ReLU Activation: Sets negative values to zero after convolution.

Pooling Layer (Average Pooling): Halves dimensions with stride 2, resulting in output of 4×4.

Block 3 extracts further features and reduces spatial dimensions, enhancing convolutional layer performance.

Block 4 does not use a pooling layer, opting instead for a convolutional layer followed by ReLU activation, reducing dimensions to $1 \times 1 \times 64$.

Convolutional Layer: Uses 64 filters of size 4×4, scanning across 64 feature maps. The weights are a tensor of shape $(4, 4, 64, 64)$.

ReLU Activation: Converts negative values to zero post-convolution.

This block significantly reduces dimensions without pooling, using 64 convolution filters across 64 feature maps to yield a single value per filter.

Block 5 incorporates a fully connected layer, producing output dimensions of $1 \times 1 \times 10$.

Fully Connected Layer: Utilizes weight matrix of shape $(64, 10)$. Fully connects inputs for higher-level classification.

Block 5 includes the final fully connected layer responsible for network output and classification. Integrating data fully yields ultimate output at network's final stage.

Softmax Operation at final stage converts each neural network output component into probability between zero and one. Facilitates expressing model results as probabilities. Softmax aids classification problems in assessing most likely class.

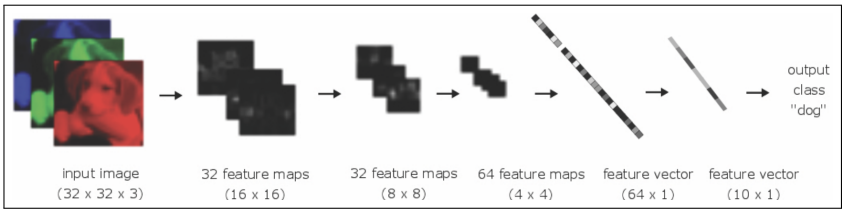


Figure 4.8. Architecture of the CNN used for image classification [2]

Our dataset comprises ten real numbers, generating a vector. In this model, we use mini-batches of size 32. Training spans 10 epochs with Adam optimization. All tasks were conducted on Google Colab. Training the network took around 12 minutes. Test accuracy reached approximately 72

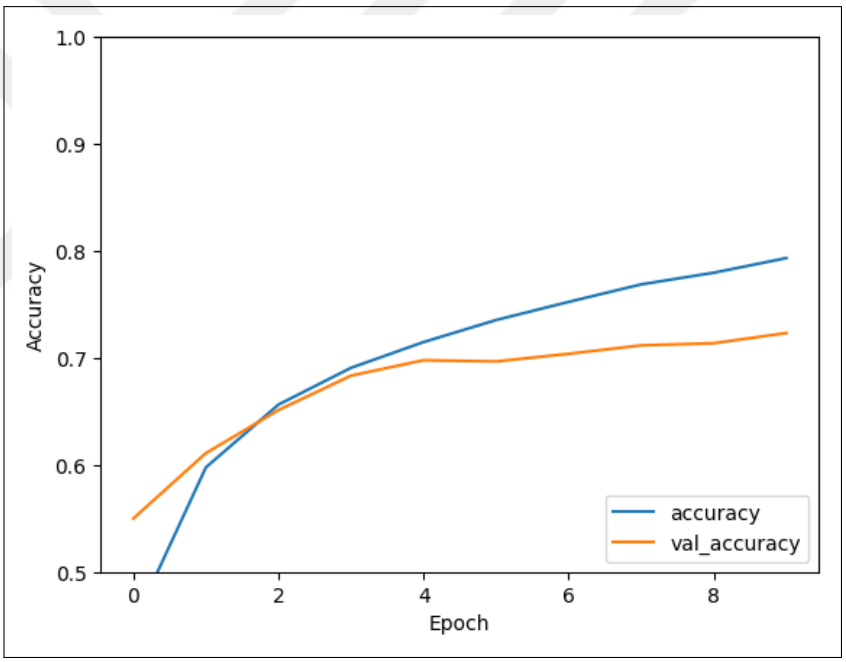


Figure 4.9. Training and validation loss

Figure 4.9 reveals network avoided overfitting, shown by close alignment between training and validation set accuracies.

Post review of five layers to better understand CNN structure during model training, we used confusion matrix to assess classification model performance. For confusion matrix, appended code end of provided script for this model to illustrate.

You can find model's confusion matrix and constructed network in appendix A chapter.

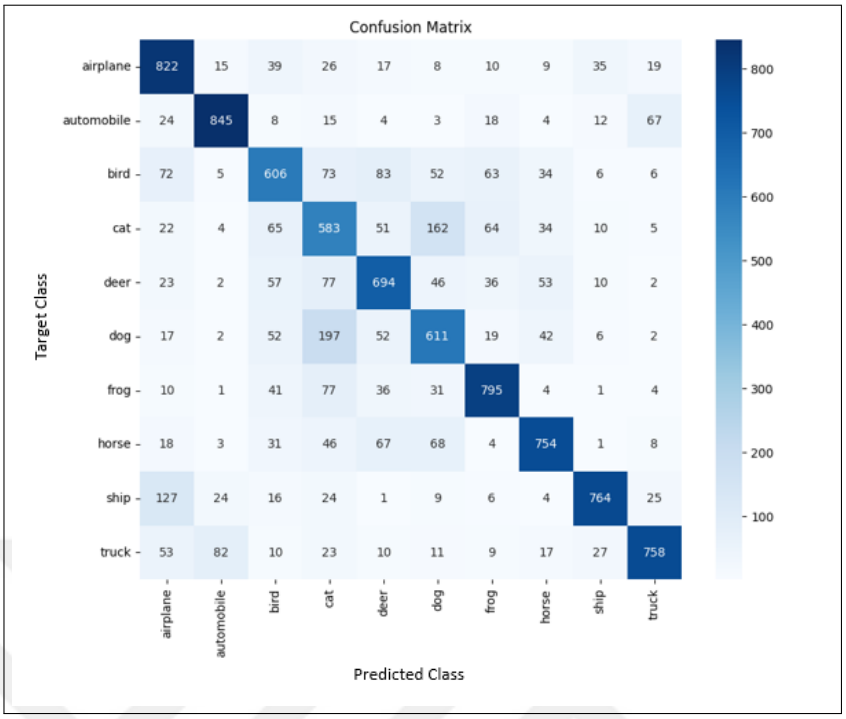


Figure 4.10. Predicted classes by CNN for image classification

Additionally, as shown in Figure 4.10, network accurately predicts image classes. However, few images (e.g., bird vs. airplane, deer vs. horse) were misclassified, yet error rate remains significantly lower than correct classification rate.

Acknowledgments: Codes in chapter 4 adapted from [2], with modifications to inputs. Figures 1.1, 1.2, 1.3, 3.1, 3.2, 3.3, 3.6, 3.7, 4.1, 4.3, 4.4, 4.5, 4.6 drawn by me; other figures (4.8, 4.7) sources cited in references. Unspecified figures from specific codes. Links to codes in appendix A.

5. CONCLUSION

In this thesis, we examined various mathematical techniques of deep learning. We analyzed the mathematical tools used in deep learning and explored challenges such as minimizing convex cost functions and overcoming overfitting.

Initially, we investigated the history of deep learning algorithms and the process of their popularization. By emphasizing the impact of technological advancements on deep learning, we highlighted how rapidly this field is evolving. While attempting to understand the working principles of deep learning and neural networks, we explored their applications in various domains such as law, finance, and health services. These applications illustrate the versatility of deep learning and its significance across various industries. We explained the fundamental differences between deep learning and neural networks using diagrams. Additionally, we explored different types of activation functions used to enhance the learning capability of neural networks.

Our primary aim was to optimize the cost function. To achieve this, we learned gradient computation, utilized the stochastic gradient descent (SGD) method, and grasped the basic principles of backpropagation to train artificial neural networks. Practical exercises conducted in Matlab provided us with hands-on experience in applying these algorithms and evaluating their performance.

In the subsequent stages, we constructed and trained a basic neural network model to classify images of different objects. Through this process, by investigating the roles of layers and the deeper structure of the model, we gained a better understanding of neural network training and structure.

In conclusion, this thesis addresses various mathematical techniques of deep learning, covering both theoretical foundations and practical applications. It demonstrates the power and potential of deep learning across diverse fields. Our work has enabled us to better understand how neural networks function, how to minimize functions during training using mathematical methods, and what is crucial for successful neural network training. This thesis serves as a solid foundation for future research.

REFERENCES

1. CNN ile CIFAR-10 veri setindeki görüntülerin tanınması [cited 2024 10 May]. Available from: http://buyukveri.firat.edu.tr/2018/04/11/derin-ogrenme-ile-cifar-10-goruntulerini-tanima/.

2. Higham F, Higham J. Example of an artificial neural network,the general setup, stochastic gradient,back propagation. *Deep learning: An introduction for applied mathematicians*. 2019.

3. Derin öğrenme çalışmaları [cited 2024 7 May]. Available from: https://alitunacanonar.medium.com/derin-%C3%B6%C4%9Frenme-%C3%A7al%C4%B1%C5%9Fmalar%C4%B1-ge%C3%A7mi%C5%9Ften-g%C3%BCn%C3%BCm%C3%BCze-fa8d2188dcfb.

4. Reis N. History,structure,deep Learning,layer network. *Deep learning*. 2016.

5. Doğan F, Türkoğlu I. Derin öğrenme modelleri ve uygulama alanlarına ilişkin bir derleme. *DergiPark*. 2018.

6. Pullen J. But what is a Neural Network? *Neural Networks*. 2017.

7. Heaton J. *Artifical intelligence for humans volume 3: Deep learning and neural networks*. United States: Manning publications; 2015.

8. Irwin Sobel [cited 2024 7 May]. Available from: https://en.wikipedia.org/wiki/Irwin_Sobel.

APPENDIX A: APPENDIX ABOUT OTHER CODES

In this paper, you will have access to all the code used within the thesis but not shared in code format. Figures drawn on top of the code are also available at the following links.

- **The code's link for obtaining the images in Figure 3.5, Figure 2.1 and Figure 3.4;**

  https://colab.research.google.com/drive/1B6g-LfABvmLbQu1XIOB2iP8xWl1sub5D#scrollTo=L-TjEw-8ZQbg

- **For the appropriate code to obtain the model's confusion matrix and the network that we build at this link.**

  https://colab.research.google.com/drive/131YzipDe9za33FBBpxCXHmon-90R1x-I#scrollTo=umwZL7znDKng

  https://www.tensorflow.org/tutorials/images/cnn