FITPED | AI

Deep Learning

Jozef Kapusta
Robert Rouš
Ján Skalka
Małgorzata Przybyła-Kasperek
Júlia Tomanová

www.fitped.eu

2024

Funded by the European Union

Erasmus+ FITPED-AI
Future IT Professionals Education in Artificial Intelligence
(Project 2021-1-SK01-KA220-HED-000032095)

FITPED | AI

Deep Learning

Published on

November 2024

Authors

Jozef Kapusta | Teacher.sk, Slovakia

Robert Rouš | Mendel University in Brno, Czech Republic

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Júlia Tomanová | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Piet Kommers | Helix5, Netherland

Vladimiras Dolgopolovas | Vilnius University, Lithuania

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095

Funded by
the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Slovak Academic Association for International Cooperation. Neither the European Union nor the granting authority can be held responsible for them.

ISBN 978-80-558-2229-7

TABLE OF CONTENTS

Introduction to Neural Networks

Chapter 1

1.1 Introduction

🕮 1.1.1

A neural network is a system in computers that tries to work like the human brain. Just like our brain has neurons that help us think, a neural network has "artificial neurons" that help it process information. These artificial neurons are connected in layers, and each layer has a specific task. The first layer gets the input (like an image or a piece of text), and then the network processes this input through the different layers to make decisions or predictions.

In simple terms, neural networks are designed to learn from examples. For example, if you show a neural network many pictures of cats, it can learn to recognize what a cat looks like. Then, when you show it a new picture, it can tell whether it's a cat or not, even if it hasn't seen that exact picture before.

Neural networks are useful because they can handle complex tasks like recognizing images, understanding speech, and even driving cars! They are able to do things that regular computer programs cannot because they "learn" from data and improve over time.

📝 1.1.2

What is the main role of artificial neurons in a neural network?

- To process information
- To learn from data
- To make physical decisions
- To perform arithmetic calculations

🕮 1.1.3

Neural networks are needed because some problems are too complex for traditional computers. For example, understanding a photo, recognizing someone's voice, or predicting what you'll like to buy next can be really difficult for a regular computer program. Traditional programs need very clear instructions, but these tasks require understanding patterns in data, which is something neural networks are good at.

They are important in fields like medicine, where they can help doctors detect diseases from X-rays or other medical images. Neural networks can also be used in everyday technology, like your smartphone’s face recognition feature or the recommendation system on websites like YouTube or Netflix.

Another reason we need neural networks is that they can handle huge amounts of data. With the growth of the internet, there's so much data available, and neural networks can process all this information to find patterns that humans might miss.

1.1.4

In what ways can neural networks be used in everyday life?

- Recognizing faces on smartphones
- Recommending videos on YouTube
- Predicting the weather
- Writing essays for students

1.1.5

Neural networks can help in many areas by solving problems that require pattern recognition. For instance, in healthcare, they can help doctors by analyzing medical images and detecting early signs of diseases like cancer. In the business world, they help companies predict customer behavior, which can lead to better products or services.

In self-driving cars, neural networks help the car understand its surroundings. The car uses cameras and sensors to gather data, and the neural network processes this information to make decisions like when to turn, stop, or speed up.

Neural networks can also help in translating languages. For example, Google Translate uses neural networks to improve translations, making them more accurate and natural over time.

1.1.6

What is one way neural networks help in self-driving cars?

- They help the car understand its surroundings
- They predict the car's fuel efficiency
- They control the car's engine speed
- They translate languages spoken by passengers

📖 1.1.7

The principle behind neural networks

At the core of a neural network is the idea of learning from examples. When a neural network is trained, it gets a lot of data to look at. Initially, it doesn't know much and might make mistakes. However, it slowly improves by adjusting itself to make better decisions. This process is similar to how you learn to play a game or a sport by practicing over and over again.

Neural networks are made up of layers. The first layer receives input data, such as an image, and the final layer gives the output, such as whether the image contains a cat. The layers in between are where the neural network learns the important features that help make the decision. The more layers the network has, the better it can handle complicated tasks.

Training a neural network requires a lot of examples, and the network uses these to adjust its internal settings until it becomes better at solving the task.

📝 1.1.8

How does a neural network "learn"?

- By making predictions and adjusting based on feedback
- By memorizing all the data without analyzing it
- By reading books and articles
- By copying the results from another network

📖 1.1.9

Overfitting and underfitting

When training a neural network, it's important not to overtrain or undertrain the network. Overfitting happens when a network learns the training data too well, so well that it starts to focus on the tiny details or mistakes in the data. This can make the network very good at predicting the data it’s seen, but it might struggle with new data.

Underfitting is the opposite problem. This happens when the network doesn’t learn the data enough, meaning it doesn't make good predictions, even for the training data. A well-trained network finds the balance between these two problems and performs well on both seen and unseen data.

Finding this balance is essential to building a neural network that can generalize well and solve real-world problems.

1.1.10

What does "overfitting" mean in a neural network?

- The network learns the data too well, focusing on unnecessary details
- The network works perfectly with original data
- The network doesn't learn the data enough
- The network trains on too little data

1.1.11

As technology improves, so do neural networks. They are becoming better at tasks that were once thought to be impossible for machines. For example, AI can now generate realistic artwork, write essays, or even create music. These advancements are made possible because neural networks can process and learn from vast amounts of information.

In the future, neural networks could help with even more areas of our lives. They might help solve big global problems like climate change by predicting environmental changes or improving renewable energy use. As the networks continue to grow, their potential is almost limitless.

However, there are still challenges, such as making sure neural networks make fair and unbiased decisions. It's important to keep improving these systems and make them transparent and ethical.

1.1.12

What is a challenge that needs to be addressed in the future development of neural networks?

- Making sure they can handle more data
- Ensuring they make fair and unbiased decisions
- Limiting their use to only a few tasks
- Reducing their processing speed

1.2 Neural networks principle

📖 1.2.1

Machine learning algorithms are essential because they help solve problems that traditional programming approaches can't handle. In "classic" programming, humans write code that follows clear instructions. But some problems are too complex for this type of programming. For example, imagine trying to create a program that can recognize a 3D object from different angles, lighting conditions, or in a cluttered scene. It's almost impossible because we don’t even fully understand how our brains do it. Even if we did, writing such a program would be incredibly complicated.

That’s where machine learning comes in. Instead of writing every detail of the program by hand, machine learning algorithms learn from examples. They process many examples that show the right answer for each situation. This is much faster and more efficient than trying to code everything manually.

📝 1.2.2

What is a problem that traditional programming cannot solve easily?

- Recognizing objects from multiple angles and lighting
- Sorting numbers
- Calculating the cost of an item
- Solving simple math equations

📖 1.2.3

Neural networks are one of the best machine learning algorithms available today. They are designed to work by mimicking the human brain's structure. Just like our brains have neurons that work together to process information, neural networks have artificial neurons that work in layers to understand data. These networks can be used to solve complex problems that we can't easily code, like recognizing faces, identifying objects in images, or predicting behaviors.

Neural networks are particularly helpful because they can be trained using examples. For instance, if you show a neural network thousands of pictures of dogs, it can learn what a dog looks like. After this, the neural network can identify a dog in new pictures it has never seen before. The more examples you give it, the better it gets at solving the problem.

1.2.4

What do neural networks try to mimic?

- The human brain
- The internet
- A computer's processing speed
- A traditional computer program

1.2.5

Machine learning and neural networks are perfect for situations where it's difficult to write specific instructions. For example, detecting fraud in credit card transactions is a complex task. It’s not easy to create a program with simple rules that can accurately decide whether a transaction is fraudulent. Scammers are constantly changing their methods, so the program must change too. Instead of trying to write out every rule manually, machine learning algorithms look at many examples of transactions, including both legitimate and fraudulent ones. They learn patterns that help them recognize fraud, even when the scammer changes tactics.

This flexibility is one of the reasons machine learning is so valuable. Unlike traditional programs that need to be constantly updated by humans, machine learning systems can "learn" and adapt by themselves when new data is available.

1.2.6

Why is machine learning better for detecting fraud than traditional programming?

- It can adapt to new fraud tactics
- It can recognize patterns from past data
- It doesn’t need examples to learn
- It relies on static, predefined rules

1.2.7

The process of training a machine learning algorithm involves feeding it many examples. These examples show the algorithm what the correct output should be for a given input. For example, when training a neural network to recognize a cat, you would show it many images of cats. Over time, the neural network learns patterns in the images that help it identify a cat.

The goal is to create a "model" that can generalize, meaning it will work not just on the data it was trained on, but also on new, unseen data. If done correctly, the machine learning model can adapt to new situations and perform well even when the conditions change.

The more examples you give the system, the more accurate its predictions or decisions will be. So, the key to successful machine learning is having a large and diverse dataset to train the algorithm.

1.2.8

What is the goal of training a machine learning algorithm?

- To make the system generalize and work well on new data
- To make the system memorize the training data exactly
- To build a static program that can’t be updated
- To prevent any mistakes in the output

1.2.9

One of the key benefits of machine learning and neural networks is their ability to adapt to new data. This is particularly important when dealing with situations that change over time. For example, in the case of fraudulent credit card transactions, new fraud patterns emerge regularly. A traditional program would need to be manually updated every time a new scam is discovered, which takes time and effort.

However, a machine learning model can be trained on new data whenever it becomes available. This means the system can quickly learn to detect new types of fraud without the need for constant manual intervention. This ability to adapt makes machine learning more efficient and cost-effective compared to traditional programming.

1.2.10

How does machine learning adapt to new data?

- By adjusting its model with new examples
- By staying the same and not learning new patterns
- By memorizing old data only
- By automatically updating its rules

📖 1.2.11

Neural networks are often more efficient than writing traditional programs for complex tasks. The reason is simple: they can learn patterns from data without needing to define every step in a complicated program. For example, instead of writing specific rules for every possible situation a self-driving car might encounter, a neural network can be trained with lots of examples of driving scenarios. Over time, it learns to navigate different road conditions, traffic situations, and even unpredictable events, all by processing the examples.

This is not only faster but also cheaper, because the cost of collecting data and training the model is often much lower than hiring a team of programmers to write all the rules by hand.

📝 1.2.12

Why are neural networks more efficient than traditional programs for complex tasks?

- They learn patterns from data
- They require less data to function
- They can perform tasks faster than humans
- They don’t require examples to work

1.3 Neural networks theory

📖 1.3.1

Machine learning and neural networks are particularly useful for tasks that are too complex or dynamic to be solved using traditional programming. In these cases, instead of manually writing a program for each task, we rely on collecting numerous examples of input-output pairs.

A machine learning algorithm processes data and creates a program, often referred to as a "model," that can carry out the task. This model doesn't resemble a traditional program with explicit rules but instead relies on patterns and relationships derived from the data. If designed and trained properly, this program not only handles the data it was trained on but also performs well on new, unseen cases.

One of the key advantages of using machine learning is its adaptability. When circumstances or data patterns change, we don't need to rewrite the entire program. Instead, we can retrain the algorithm using updated data, allowing the system to evolve with new information. Additionally, although training machine learning models can require significant computational resources, it is often more cost-effective than

hiring teams of programmers to write and maintain traditional software for complex and evolving tasks.

1.3.2

Why is machine learning considered more adaptable than traditional programming?

- It can be retrained with new data to adapt to changes
- It automatically writes explicit rules for every task
- It relies on manually updated rules for changing data
- It does not require any data to function effectively

1.3.3

The reasons for studying neural networks are as follows:

1. **Understanding the real functioning of the brain**
2. **Understand the style of parallel computing inspired by neurons and their adaptive connections.** This calculation is a very different style from the sequential calculation. Such an approach should be good for tasks where the brain excels (such as vision). It should be unsuitable for tasks where the brain lags behind (for example, calculate 23 * 71). This new approach to information processing is represented by the theory of artificial neural networks. It is not only an effective IT tool for the creation and design of new parallel approaches to solving artificial intelligence problems, but it is also an integral part of modern neuroscience, which is used to access computer simulations of processes taking place in the brain.
3. **Solve practical problems using new brain-inspired machine learning algorithms.** Such algorithms are very useful, even if they are not a real (real) demonstration of how the brain works.

1.3.4

Which of the listed tasks is more suitable for solving using neural networks?

- Recognition of persons
- Calculating 1014 * 1024
- Finding persons with salary bigger than defined amount

📖 1.3.5

The NN theory is based on neurophysiological knowledge about the human brain. It tries to explain behavior based on the principle of information processing in nerve cells. The size of a human neuron is 20 μm. The human brain contains 20-100 billion neurons, with each neuron interconnected with 1,000-10,000 other neurons. The speed of propagation of impulses in the brain is approximately 400 km/h.

Thus, a neuron can receive signals from the surroundings from other neurons (dendrites), the neuron processes (integrates) the received signals, the neuron (axon) sends the processed input signals to other neurons from its surroundings.

We can even simulate one neuron (with complex processes). It is even much faster than the real thing. However, the power of the human brain is that:

- uses a large number of slow neurons
- they are grouped into a very complex network, the size, typology and geometry of which is inimitable
- they are very small and very "low-power"

Neurons are connected to each other in a complex network structure (called a neural network), while individual connections have either an **excitatory** (increase in activity) or an **inhibitory** (decrease in activity) character.

The system of connections and their excitation or the inhibitory character forms the **architecture of the neural network**, which alone determines the properties of the neural network.

📝 1.3.6

Which of the following is NOT a characteristic of the **human brain's neurons**?

- Neurons are large and consume a lot of power.
- Neurons are grouped into a complex network with unique geometry.
- Each neuron can connect to 1,000–10,000 other neurons.
- Neural connections can have excitatory or inhibitory effects.

📖 1.3.7

Neuron models are largely an abstraction of the mechanism of how neuron cells process information. It is impossible to create an exact analogy of the "computational" capabilities of a real neuron.

The simplest types of neural networks were proposed by **McCulloch** and **Pitts** in 1943. Their neuron model is an important landmark in the development of the theory of neural networks. The elementary unit of the McCulloch and Pitts neural network is the logical neuron (computational unit), and the state of the neuron is binary (ie, it has two possible states, 1 and 0).

The logic neuron system contains both **excitatory** inputs (described by binary variables x_1, x_2, ..., x_n, which amplify the response) and **inhibitory** inputs (described by binary variables x_{n+1}, x_{n+2}, ..., x_m, which weaken response).

Logical neurons and neural networks were first studied in the publication of Warren McCulloch and Walter Pitts "**A logical calculus of the ideas immanent to nervous activity**" from **1943**, which is a landmark in the development of the metaphor of connectionism in artificial intelligence and cognitive science. It has been shown that neural networks are an effective computational tool in the domain of Boolean functions.

It is interesting that the work of McCulloch and Pitts is very difficult to read, the mathematical-logical part of the work was probably written by Walter Pitts, who was self-taught both in logic and mathematics. Only thanks to American scientists, the logician S.C. Kleene and the computer scientist N. Minsky, this important work was "translated" in the second half of the 1950s into the standard language of contemporary logic and mathematics, thus making the ideas contained in it generally accessible and accepted.

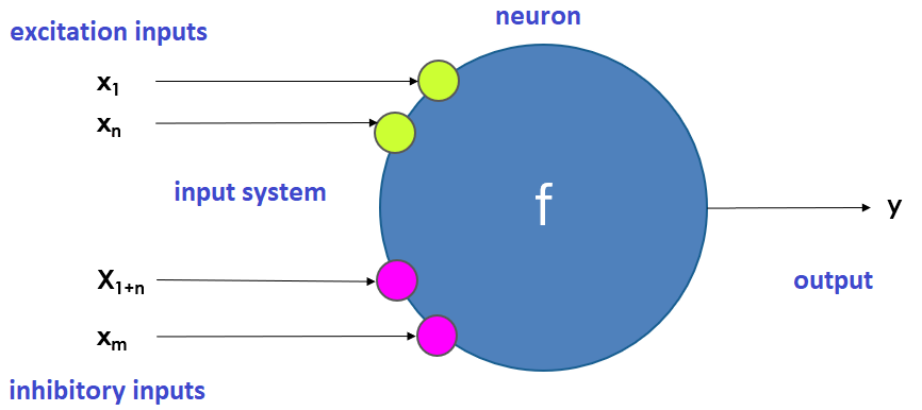
1.3.8

Which statements about the McCulloch and Pitts neuron model are true?

- The state of the logical neuron is binary, represented as 1 and 0.
- Inhibitory inputs in the model weaken the response.
- It models neurons with continuous, non-binary states.
- The neuron model was first introduced in the 1950s.

1.3.9

A logical neuron is an elementary unit of NN.



The logic neuron system contains excitatory inputs (binary variables x_1, x_2, ..., x_n, which **amplify the response**) and inhibitory inputs (binary variables x_{n+1}, x_{n+2}, ..., x_m, which **weaken the response**).

The state of a neuron is **binary** (ie it has two possible output states, 1 and 0).

The rule applies:

- the activity is **one** if the internal potential of the neuron defined as the difference between the sum of the excitatory input activities and the inhibitory input activities is greater than or equal to the **threshold b**,
- otherwise it is **zero**.

$$y = \begin{cases} 1 & (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m \geq b) \\ 0 & (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m < b) \end{cases}$$

1.3.10

What determines whether the state of a logical neuron is 1 or 0?

- The difference between excitatory and inhibitory inputs compared to the threshold.
- The total number of inputs, regardless of type.
- Whether the excitatory input activities alone exceed the threshold.
- The presence of more inhibitory inputs than excitatory inputs.

1.3.11

Let's take a closer look at the previous rule.

$$y = \begin{cases} 1 & (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m \geq b) \\ 0 & (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m < b) \end{cases}$$

Transferring **b** to the other side of the inequality, we get

$$y = \begin{cases} 1 & (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m - b \geq 0) \\ 0 & (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m - b < 0) \end{cases}$$

We can also rewrite the function **y** in the following form

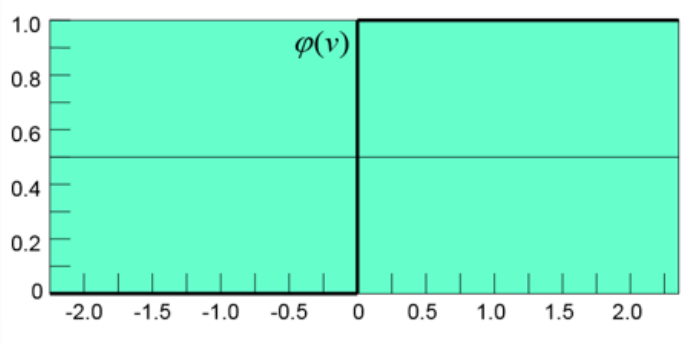
$$y = \varphi(v) = \varphi\ (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m - \boldsymbol{b})$$

when

$$\varphi(v) = \begin{cases} 1 & \text{if} \quad (v \geq 0) \\ 0 & \text{if} \quad (v < 0) \end{cases}$$

The function $\varphi(v)$ represents the well-known **signum** function in mathematics (the so-called "step function" or sign function).

The graph of this function is as follows:



In neural networks, the **signum function** is often used as an activation function, particularly in early models like the McCulloch-Pitts neuron. It helps determine whether a neuron "fires" (outputs 1) or remains inactive (outputs 0 or -1) based on its input.

The function is applied to the **net input** of the neuron, which is typically the weighted sum of inputs minus a threshold value (**b**). The signum function thus enables binary output decisions based on whether the neuron's inputs collectively meet or exceed the threshold.

1.3.12

What is the purpose of the signum function in a neural network?

- To determine whether the neuron "fires" based on the net input.
- To calculate the exact value of the neuron's output.
- To adjust the weights of the inputs dynamically.

- To prevent inhibitory inputs from affecting the output.

🕮 1.3.13

We already know that the output of an artificial neuron is defined as a signed (step) function.

$$\varphi(v) = \begin{cases} 1 & \text{if } (v \geq 0) \\ 0 & \text{if } (v < 0) \end{cases}$$

while ***v*** represents the sum of inputs and bias ***b***.

$$y = \varphi(v) = \varphi\ (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m - \boldsymbol{b})$$

Furthermore, we can implement simple modifications where each input xi is multiplied by +1 or -1 depending on whether it is an inhibitory or an excitatory input. Subsequently, we generally replace +1 or -1 with a weight $\boldsymbol{w_i}$

$$\begin{aligned} y = \varphi(v) &= \varphi\ (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m - \boldsymbol{b}) \\ &= \varphi\ (1x_1 + \ldots + 1x_n - 1x_{1+n} - \cdots - 1x_m - \boldsymbol{b}) \\ &= \varphi\ (w_1x_1 + \ldots + w_nx_n + w_{1+n}x_{1+n} + \cdots + w_mx_m - \boldsymbol{b}) \end{aligned}$$

whereas:

$$w_i = \begin{cases} 1 & (\text{connection } i \to j \text{ has an excitatory character}) \\ -1 & (\text{connection } i \to j \text{ has an inhibitory character}) \\ 0 & (\text{connection } i \to j \text{ does not exist}) \end{cases}$$

We can write the resulting activity of the neuron as

$$y = \varphi\ (\underbrace{w_1x_1 + w_mx_m}_{v} - b) = \varphi\left(\sum_{i=1}^{m} w_ix_i - b\right)$$

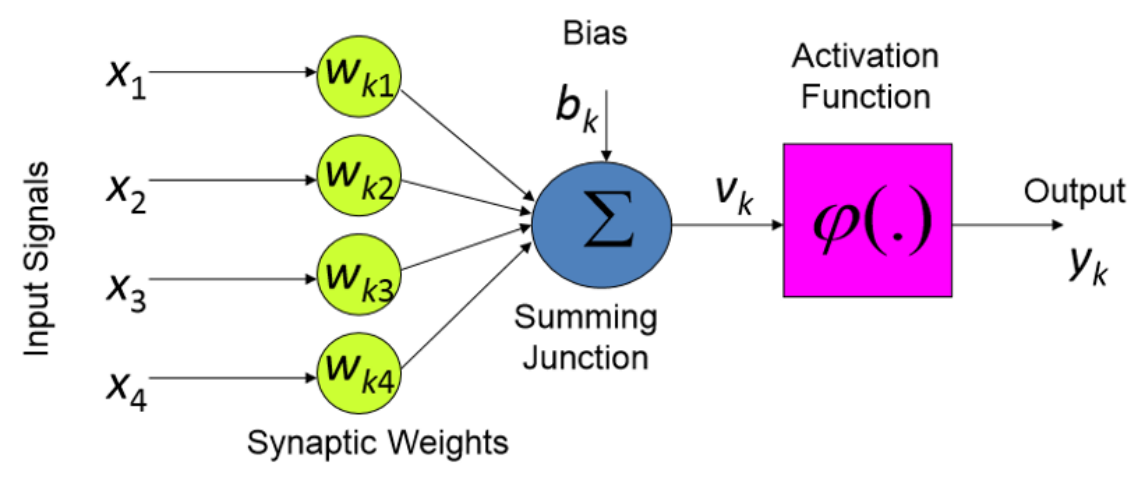
🕮 1.3.14

An artificial neuron model is defined as follows:

$$y = \varphi\left(\underbrace{\sum_{i=1}^{m} w_i x_i}_{y\_in} - b\right)$$

while **y_in** represents the so-called internal potential of the neuron. The internal potential of a neuron, often denoted as yiny_{\text{in}}yin, represents the cumulative input to the neuron before applying the activation function.

For example, for **four** inputs, we can imagine the neuron as follows



The neuron processes information through a set of **synapses**, which are connections characterized by their **weights** (thickness or strength). These weights determine the influence of each input on the neuron's output.

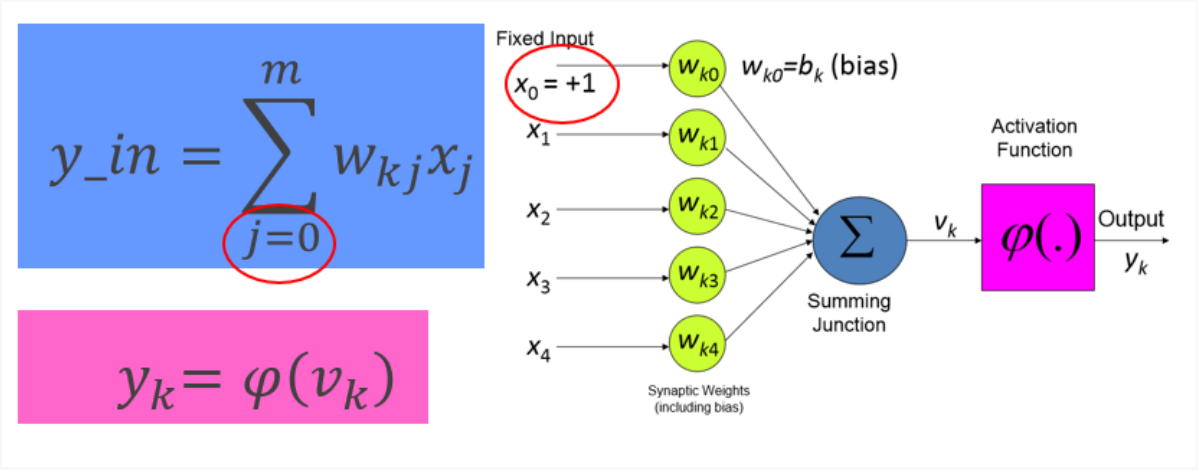
A neuron can also be expressed as

$$y_k = \varphi(y\_in + b_k)$$

while the internal potential is defined as follows:

$$y\_in = \sum_{j=1}^{m} w_{kj} x_j$$

In practice, bias is often not singled out separately (mainly due to simpler computer calculation). Bias is an external parameter of the artificial neuron and can be included directly in the summation.



1.3.15

Which statements about the internal potential (y_{in}) of a neuron are true?

- It represents the weighted sum of inputs, including the bias.
- Weights determine the influence of inputs on the neuron's output.
- The bias is always treated as a separate external parameter.
- The internal potential directly determines the weights of the inputs.

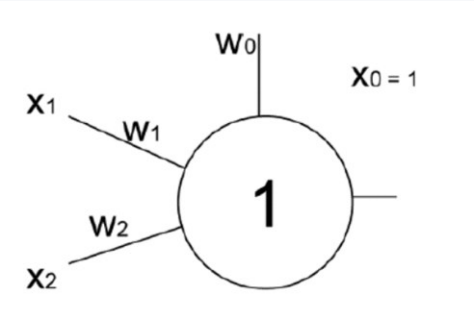
1.4 Implementation of Boolean binary functions

1.4.1

At the beginning of the era of neural networks, it was assumed that they would be able to simulate Boolean binary functions. Although of course it is not a priority of NN to simulate them, we can show several interesting properties of a logic neuron on this problem.

Let us therefore assume one logic neuron with two inputs, two weights and a bias. The activation function is a simple staircase function.

In our neuron, the weights are set as follows $w_1 = 1$; $w_2 = 1$ and bias $w_0 = -1.5$.



If we input the numbers $x_1 = 0$ and $x_2 = 1$, then we calculate the result of the neuron as follows:

$$y = \varphi\left(\sum_{i=0}^{m} w_i x_i\right)$$

$$y = \varphi(0 * 1 + 1 * 1 + 1 * -1{,}5) = \varphi(1 - 1{,}5) = \varphi(-0{,}5)$$

$$y = \varphi(-0{,}5) = 0$$

After inputting 0 and 1, we get the result 0. The artificial neuron **probably** implements the AND logical function.

The value tables of the logical AND function are known

x_1	x_2	$x_1 \wedge x_2$
0	0	0
1	0	0
0	1	0
1	1	1

For correctly set weights of a neuron implementing the AND function, it is therefore necessary that:

$$w_1 0 + w_2 0 + w_0 1 < 0$$

This inequality expresses the first row of the logic function table. The left side of the inequality must be less than zero, because only in this case the activation step function will give us a result equal to 0.

In this way, we create inequalities for each row of this table. To set the correct weights, it is necessary to solve a system of inequalities (inequalities for all 4 rows of the table)

$$w_1 0 + w_2 0 + w_0 1 < 0$$

$$w_1 0 + w_2 1 + w_0 1 < 0$$

$$w_1 1 + w_2 0 + w_0 1 < 0$$

$$w_1 1 + w_2 1 + w_0 1 > 0$$

The solution to this system of inequalities is, for example, the values:

$w_1 = 1$, $w_2 = 1$ and $w_0 = -1{,}5$.

$$1*0 + 1*0 - 1{,}5*1 < 0 \rightarrow y = 0;$$

$$1*0 + 1*1 - 1{,}5*1 < 0 \rightarrow y = 0;$$

$$1*1 + 1*0 - 1{,}5*1 < 0 \rightarrow y = 0;$$

$$1*1 + 1*1 - 1{,}5*1 > 0 \rightarrow y = 1;$$

In this way, we found the weights of the artificial neuron. ($w_1 = 1$, $w_2 = 1$ a $w_0 = -1{,}5$), for which it will implement a logical function **AND**.

1.4.2

What is required for an artificial neuron to implement the logical AND function?

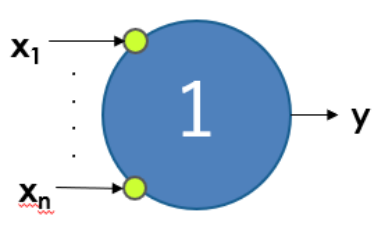
- A system of inequalities must be solved to determine appropriate weights and bias.
- The neuron must have at least three inputs and no bias.
- The activation function must always be linear.
- The weights must be negative to simulate logical functions.

1.4.3

Similarly, it is possible to find the weights of the neuron for the implementation of other Boolean functions. For example for the OR function it can be scales: $w_1 = 1$, $w_2 = 1$ and $w_0 = -0{,}5$),

#	x_1	x_2	$y_{OR}(x_1,x_2)$	$x_1 \vee x_2$
1	0	0	$\varphi(-1)$	0
2	0	1	$\varphi(0)$	1
3	1	0	$\varphi(0)$	1
4	1	1	$\varphi(1)$	1

$$y_{OR}(x_1,x_2) = \varphi\,(x_1 + x_2 - 0,5)$$



1.4.4

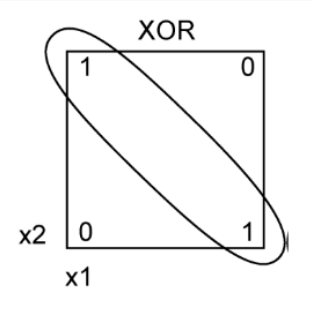
What weights and bias allow an artificial neuron to implement the logical OR function?

- w1=1,w2=1,w0=−0.5
- w1=2,w2=1,w0=−1
- w1=0,w2=0,w0=−0.5
- w1=1,w2=0,w0=−1.5

1.4.5

However, there are also Boolean functions that cannot be simulated by a logic neuron. An example of such a function is, for example, a function **XOR**.

x_1	x_2	$f_{xor}(x_1,x_2)$
0	0	0
0	1	1
1	0	1
1	1	0



In the case of this function, it is necessary to solve the following inequalities:

$$\begin{aligned} w_1 * 0 + w_2 * 0 + w_0 * 1 &< 0 \\ w_1 * 0 + w_2 * 1 + w_0 * 1 &> 0 \\ w_1 * 1 + w_2 * 0 + w_0 * 1 &> 0 \\ w_1 * 1 + w_2 * 1 + w_0 * 1 &< 0 \end{aligned}$$

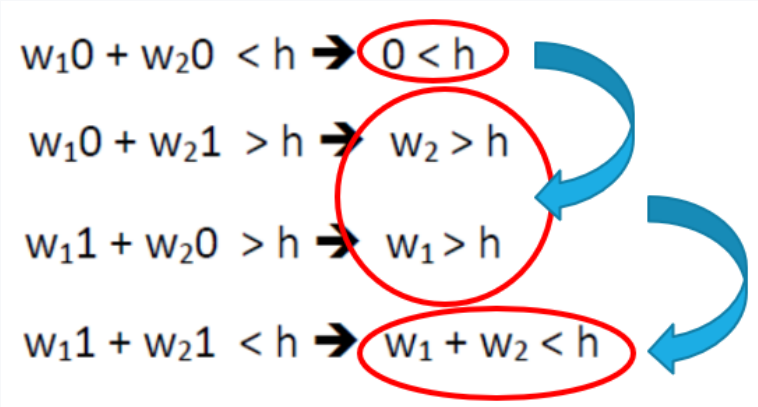
if we mark

$$-w_0 = h$$

we get:

$$\begin{aligned} w_1 0 + w_2 0 < h &\rightarrow 0 < h \\ w_1 0 + w_2 1 > h &\rightarrow w_2 > h \\ w_1 1 + w_2 0 > h &\rightarrow w_1 > h \\ w_1 1 + w_2 1 < h &\rightarrow w_1 + w_2 < h \end{aligned}$$

However, this system of equations has no solution. Since ***h*** is a positive number, w_2 and w_1 are greater than ***h***. Therefore, their sum cannot be less than ***h***.



It means that the **Logical XOR function can NOT be implemented by a single neuron**.

1.4.6

Why can't the logical XOR function be implemented by a single neuron?

- A single neuron cannot separate the XOR function's inputs using a linear decision boundary.
- XOR requires negative weights, which a single neuron cannot handle.
- XOR is not a Boolean function, so it cannot be represented by a neuron.
- The XOR function requires more than two inputs, which a single neuron cannot process.

1.4.7

The logical function **XOR** belongs to the so-called linear non-separable functions.

Definition:

The Boolean function $f(x_1, x_2,..., x_n)$ is linearly separable if there is such a plane $w_1x_1 + w_2x_2 + ...+ w_nx_n - J = 0$, that separates the space of input activities such that there are vertices in one part of the space rated ***0***, while in the other part of the space the vertices are rated ***1***.

Theorem: A logic neuron is able to simulate only those Boolean functions that are linearly separable.

1.4.8

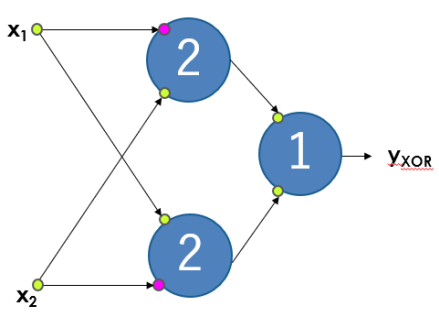
The question remains how to solve the XOR problem. In the case of Boolean functions, Boolean algebra tells us that a Boolean function can be rewritten in conjunctive clauses. Conjunctive clauses can be expressed by one logical neuron. We combine the outputs from these neurons into a disjunction using a neuron.

x_1	x_2	$f_{xor}(x_1,x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

We can attribute the XOR function as follows:

$$y = f(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

Only AND, OR and NOT functions are used in its transcription. All can be expressed by a single neuron. We can thus create the following network:



This is how any Boolean function can be accessed. The sentence applies:

Any Boolean function ***f*** is simulated using a 3-layer neural network.

3-layer neural networks containing logic neurons are universal computing devices for the domain of Boolean functions.

1.4.9

How can the XOR problem be solved using neural networks?

- By adding more layers to the neural network to create non-linear decision boundaries.
- By using multi-layer neural networks with non-linear activation functions.
- By using a single-layer perceptron with a linear activation function.
- By using a single neuron with the correct weights and bias.

Perceptron and Supervised Learning

Chapter 2

2.1 Perceptron

📖 2.1.1

The main objection to an artificial neuron (as defined by **McCulloch** and **Pitts**) is that it is not capable of learning, its parameters (weights and threshold coefficients) are fixed so that the neuron performs the required Boolean function (logical conjunction or conjunctive clause). Neural networks constructed from these neurons are designed to also perform a Boolean function of general form.

However, the neuron can also be taught. During active dynamics, the neuron performs the transformation of the input vectors to the output value. The parameters of the neuron are constant at this moment. On the other hand, adaptive dynamics is a process whose task is to set these parameters of the neuron so that the neuron performs the required transformation. The parameters that are adapted during the neuron's learning are usually only the weights of the input synapses of the neuron, including the synapse representing the threshold.

Frank Rosemblatt (1928 - 1969) included learning in the construction of the McCulloch and Pitts-type neuron. **Weight coefficients** and threshold coefficients **were considered variable** parameters of the "model", which are **set by** the **learning** process.

📝 2.1.2

What is the main limitation of the McCulloch and Pitts artificial neuron?

- Its parameters (weights and thresholds) are fixed and cannot be changed.
- It can only perform logical conjunctions and cannot handle other Boolean functions.
- It is incapable of performing Boolean functions.
- It cannot perform transformations without learning.

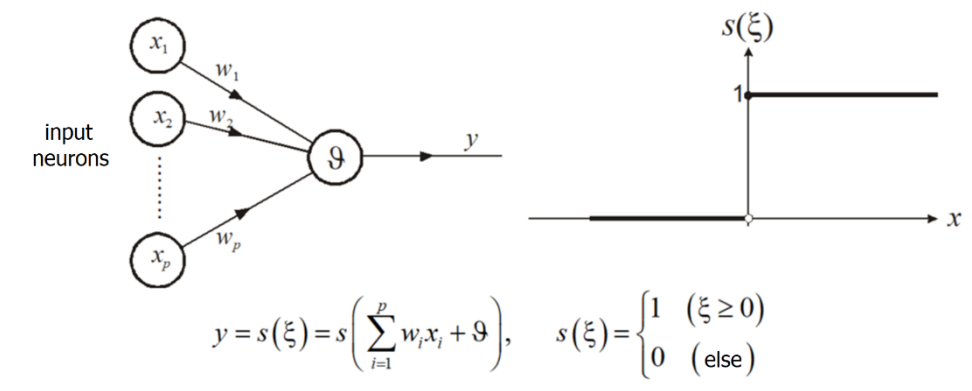
📖 2.1.3

Frank Rosemblatt's neuron was named **Perceptron**. It was inspired by the human eye. He modeled perception - perception, sensation, ability to perceive.

Its task was to recognize individual recorded characters using optical sensors arranged in a 20x20 array of elements. The basic goal of the adaptation process of the perceptron is to set the weighting coefficients of the connections so that the activities of neurons from the third layer (response area) correctly classify the image falling on the retina. Regardless of the original meaning, the term perceptron is used

for all feedforward neural networks, i.e. networks with a layered arrangement of neurons and one-way signal propagation from input to output.

Rosemblatt's Perceptron



Weight coefficients and threshold coefficient are real variable parameters

2.1.4

What is true about Frank Rosenblatt's perceptron?

- It was inspired by the human eye and modeled perception.
- The perceptron is now used to refer to all feedforward neural networks.
- The perceptron’s primary task was to classify images falling on the retina.
- It uses a multi-layered arrangement of neurons and bidirectional signal propagation.

2.1.5

The "scales" or "weights" of a neuron represent its memory. Learning happens when an adaptation algorithm adjusts these weights. This process typically occurs in steps, where the algorithm uses a set of input-output pairs (examples) to learn from. The algorithm tries to find the best way to transform inputs into outputs by analyzing known examples.

In a way, the adaptation algorithm works similarly to how humans solve problems by drawing on past experiences. It seeks the best transformation that maps input vectors to output vectors, hoping that this solution will work for new, unseen examples in the future.

Neural networks are powerful because they can find solutions to problems that are difficult or even impossible to solve analytically, as long as they have enough

examples. However, the main challenge is that the learned transformation is embedded within the network structure and cannot always be easily explained

2.1.6

What is the role of the weights in a neuron?

- They represent the memory of the neuron and store learned information.
- They represent the input values to the neuron.
- They are used to create the output values directly.
- They represent the input-output transformation of the entire network.

2.1.7

Adaptation algorithms in neural networks can be broadly classified into two categories: **supervised learning** and **unsupervised learning**. These two approaches handle learning in different ways:

Supervised learning

In supervised learning, we are given a finite, countable set of **input-output pairs**. These pairs consist of an **input vector** x and its corresponding **correct output** y_d. This means we already know the desired behavior of the system for each input. A countable finite set ***M*** of pairs ***x*** and $\boldsymbol{y_d}$ is available, which represent the inputs and the corresponding correct outputs of the solved task.

$$M = \{[x^1, y_d^1], [x^2, y_d^2], \dots, [x^{pmax}, y_d^{pmax}]\}$$

The goal is to teach the neural network how to transform input vectors into the correct output vectors by providing it with many examples of correct transformations. The set of all available values thus represents a known part of the system's behavior. This set is then used by the adaptive algorithm to train the network and also to verify its function. The set ***M*** of all available data is divided into two parts:

- **training set** is used to train the neural network by adjusting the weights based on the input-output pairs
- **test set** is used to evaluate the performance of the network after training to ensure it can correctly generalize to new, unseen examples
- The ratio between the size of the training and test sets can vary depending on the specific task or dataset.

Unsupervised learning

In unsupervised learning, the neural network is provided with only input data without any known correct outputs. The goal here is to allow the network to find patterns, groupings, or structures within the data on its own, without direct guidance from a labeled dataset.

2.1.8

Which statements are true about supervised learning in neural networks?

- Supervised learning requires both input data and corresponding correct outputs.
- The available data is divided into a training set and a test set.
- In supervised learning, the network tries to find patterns in data without labeled examples.
- The ratio of the training set to test set is fixed in supervised learning.

2.1.9

Training a neuron (or a feedforward neural network) generally happens in steps, which are repeated iteratively. Here's how it works:

- **Iterative training process** - the algorithm begins by presenting individual examples (input patterns) from the **training set** to the neuron. For each input, the neuron computes an output. Based on this output, the algorithm adjusts (or **corrects**) the weights of the neuron to make the output closer to the desired value. This process is repeated multiple times over all the patterns in the training set.
- One complete pass through all the examples in the training set is called a **learning epoch**. After each epoch, the weights have been adjusted based on the entire dataset, and the neuron has learned from all the training examples once.
- **Stopping criteria** - the training process doesn't go on forever. Stopping adaptation is most often achieved: achieving the desired small error of the transformation, by stopping the transformation error from falling, by reaching the maximum number of epochs.
- **Test set for evaluation** - to ensure that the neuron or network has generalized well (i.e., it can perform accurately on new, unseen data), we use a **test set**. This is a separate dataset that was not used during training. During the training process, the performance of the network is periodically tested on this set. If the network's performance on the test set starts to worsen as training continues, it may be a sign that the network is overfitting (becoming too specialized to the training set) and should stop training.

2.1.10

What is the main purpose of a test set during the training of a neural network?

- To evaluate the performance of the network during training and ensure generalization.
- To train the network by providing additional examples.
- To calculate the error after each epoch.
- To adjust the weights of the neurons during training.

2.1.11

Iterative learning of a neuron with a teacher follows these typical steps:

1. **Preprocessing of input data - t**he input data is prepared and transformed into a suitable form for the neural network.
2. **Defining the training and testing sets** - a set of examples (input-output pairs) is split into a training set (for learning) and a test set (for evaluation).
3. **Defining the network structure/neuron parameters** - the architecture of the network and the initial parameters (such as the number of neurons, layers, and other configurations) are decided.
4. **Initializing neuron weights** - the weights of the neuron are typically set to random values initially.
5. **Set learning epoch counter** - the counter for the number of learning epochs, n=0n = 0n=0, is initialized.

The learning epoch

Each learning epoch involves several steps:

- **Set epoch number** - increment the learning epoch counter n=n+1n = n + 1n=n+1 and check if the number of epochs has reached a maximum limit.
- **Select input vector** - a single input vector is selected from the training set. This can be done either deterministically or randomly.
- **Obtain neuron response** - the neuron produces an output based on the input.
- **Evaluate classification error** - the actual output is compared with the expected output to calculate the classification error.
- **Adjust weights** - based on the error, the weights of the neuron are adjusted to improve its response.
- **Repeat for all inputs** - if all inputs from the training set haven’t been tested yet, the process repeats by selecting the next input vector.
- **End of epoch evaluation** - at the end of the epoch, the total error across the training set is evaluated. If the error is below the desired threshold, the learning stops.

- If the error is insufficient or the performance is not satisfactory, the algorithm may return to earlier steps (like adjusting the network parameters or reinitializing weights) and continue learning.

2.1.12

Which steps are involved in a learning epoch of a neuron?

- Select one input vector from the training set.
- Obtain the response of the neuron and adjust weights.
- Evaluate the test set error.
- Preprocess the input data.

2.2 Hebbian learning

2.2.1

Hebbian learning is one of the most basic and intuitive learning rules for artificial neurons with binary inputs and outputs. It was proposed in 1949 by Canadian psychologist **Donald Hebb** while studying conditioned reflexes in the brain. Hebb's hypothesis was centered around how neural connections strengthen or weaken based on the timing of their activation, leading to the development of a fundamental learning principle used in artificial neural networks today.

Hebb's theory posits that:

- **Conditioned reflexes** - in the brain, conditioned reflexes form when the connections between individual neurons either strengthen or weaken based on their activation patterns.
- **Simultaneous activation** - when two neighboring neurons are active at the same time (i.e., both neurons are excited), the connection between them becomes **stronger**. This is the principle of "**cells that fire together, wire together.**"
- **Discordant activation** - if the neurons are activated at different times (i.e., not in sync), the connection between them weakens.

For an artificial neuron following Hebb's rule:

- If an input neuron is excited **and** the output neuron also responds appropriately (firing), the weight of the input is **increased** (strengthened).
- If the output neuron does not fire in response to the input neuron, the connection between them is **weakened**.

This means that the network adapts its weights based on **correlations** between the input and output activities. If an input consistently leads to the correct output, its associated weight increases, reinforcing the connection. Conversely, incorrect or unrelated activations weaken the connection.

2.2.2

Which of the following is a key idea of Hebbian learning?

- Neurons become stronger if they activate at the same time.
- Neurons that activate together strengthen their connection.
- Neural connections weaken when two neurons activate simultaneously.
- Learning is based on explicit feedback from the output.

2.2.3

If two neurons are active at the same time, they should have a greater degree of mutual interaction than neurons whose activity does not show correlation. In such a case, their interaction should be either zero or very small.

This means in practice that the synapses (weights) between neurons are strengthened if the activity of the input neuron leads to the activity of the neuron on the output side of the synapses.

For a neuron with binary input **x**, weights **w**, output **y** and predicted output $\mathbf{y_d}$,the Hebb rule can be written as:

- If the neuron is activated correctly (**y = 1**; $\mathbf{y_d = 1}$), then in the next step **n + 1** the $\mathbf{w_i}$ connections that caused this activation will be strengthened by the value **Δ**

$$(_{n+1}w_i = w_i + \Delta, \forall i: x_i = 1)$$

- If the neuron is not activated correctly (**y = 1; $\mathbf{y_d = 0}$**), the connections that caused this activation are weakened by the value **Δ**

$$(_{n+1}w_i = w_i - \Delta, \forall i: x_i = 1)$$

- If the neuron is not activated (**y = 0**), the weights s do not change (nothing happens)

2.2.4

According to Hebbian learning, what happens when a neuron is correctly activated?

- The synaptic weights between the neurons are strengthened.
- The synaptic weights between the neurons are weakened.
- The synaptic weights between the neurons remain unchanged.
- The output of the neuron can be ignored.

2.2.5

Another originally heuristic rule that is also applicable to general real inputs and outputs of a neuron is the **Delta rule**. The **Delta rule** is an important learning rule for adjusting the weights of neurons in a neural network. It is specifically used for **linear neurons** and is based on minimizing the difference between the actual output and the predicted output (the error).

The general form of the **Delta rule** is:

$\Delta w_i = \mu \cdot (y_d - y) \cdot x_i$

Where:

- Δw_i is the change in the weight,
- μ is the learning rate constant, which controls the speed of adaptation (its value lies between 0 and 1),
- y_d is the desired output,
- y is the actual output of the neuron,
- x_i is the input to the neuron.

In the Delta rule, the weight is updated based on the error ($y_d - y$). If the output is close to the desired output, the weight changes very little, and if the output is far from the desired value, the weight change is larger. This helps the neuron "learn" from its mistakes and gradually improve its performance.

The delta rule applies exactly to linear neurons, i.e. neurons with a linear activation transfer function, but after modification it is also applicable to neurons with a nonlinear activation transfer function.

The Delta rule can be slightly simplified by formula:

$$w_i^{n+1} = w^{in} + \mu \cdot (y_d - y)$$

Where:

- w_i^{n+1} is the updated weight after the learning step.
- w_i^n is the current weight before the learning step.
- μ is the learning rate, a constant that controls how much the weights are adjusted, it is a suitably chosen constant from the interval (0,1) affecting the adaptation speed.
- y_d is the desired output (target output).
- y is the actual output produced by the neuron.

Explanation:

- The current weight w_i^{n+1} is adjusted by an amount proportional to the error $(y_d - y)$. This error represents the difference between the desired output y_d and the actual output y produced by the neuron.
- The weight change is scaled by the learning rate μ, which controls how large the weight update will be.
- This update rule makes the weights move in the direction that reduces the error between the actual and desired outputs.
- If the neuron is **incorrectly activated**, the weight is adjusted to reduce the error in future predictions.
- If the neuron is **correctly activated**, the weight doesn't change significantly (but still adjusts by a small amount, if necessary).

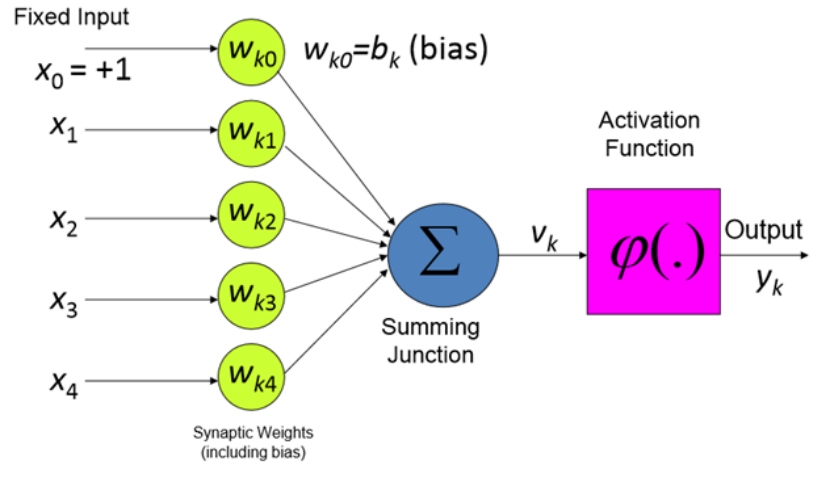
2.2.6

In the Delta rule, what does μ control?

- The speed of weight adjustment.
- The number of training epochs.
- The output of the neuron.
- The size of the training set.

2.2.7

We will use Hebbian learning in perceptron training. It is intended for dichotomous classification, i.e. splitting into two classes, where the classes are assumed to be linearly separable in the example space. There is a possibility to separate objects in the example space using a hyperplane, for example: a straight line in 2-dimensional or a plane in 3-dimensional space.



A neural network is a dynamic system, that is, a time-dependent system. We will talk about the state of the neuron in time ***t*** or in time ***t+1***.

$$u(t) = \sum_{j=1}^{m} w_j(t)x_j(t) + b$$

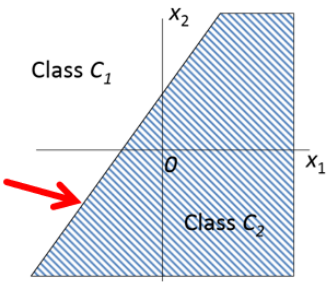
$$y(t) = \begin{cases} 1 & \text{if } u(t) \geq 0 \\ 0 & \text{if } u(t) < 0 \end{cases}$$

The separating hyperplane is given by the equation:

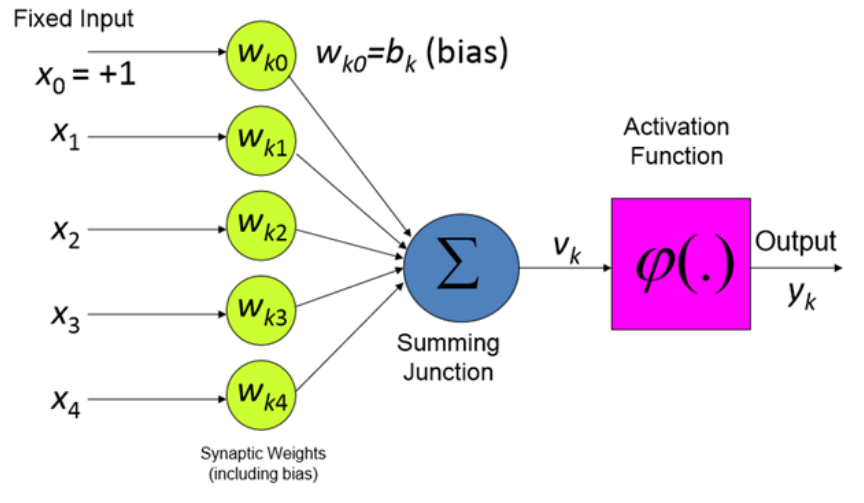
$$\sum_{j=1}^{m} w_j(t)x_j(t) + b = 0$$

or

$$w_1x_1 + w_2x_2 + b = 0$$



📖 2.2.8



The learning process is a search for appropriate synaptic weights. From a practical point of view, let's note:

$$\mathbf{w(t) = (w_0(t), w_1(t), w_2(t), ...,w_n(t))}$$

$$\mathbf{x(t) = (x_0(t), x_1(t), x_2(t), ...,x_n(t))}$$

where ***n*** is the number of neurons of the associative layer.

In the case of zero input, it holds that $\mathbf{w_0(t) = b}$ and $\mathbf{x_0(t) = 1}$

Assume a training sample of vectors:

$$\mathbf{(x(1),y(1)), (x(2),y(2)), ... (x(m),y(m)),}$$

where

$$\mathbf{y(t) = 1 \text{ if } x(t) \text{ is from class 1 (CL1)}}$$

$$\mathbf{y(t) = -1 \text{ if } x(t) \text{ is from class2 (CL2)}}$$

📝 2.2.9

In the context of a neural network, what does the bias weight $w_0(t)$ typically represent?

- A fixed value that adjusts the output of the neuron.
- The synaptic weight for the first input neuron.
- The output of the neuron.
- The number of neurons in the layer.

📖 2.2.10

Perceptron learning algorithm

- **Initialization of weights** - set initial weights w(t) = (w_0, w_1, w_2 ,..., w_n), typically to small random values, and set the bias w_0 to a small random value or zero.
- **If the input vector x(t) is correctly classified by w(t), then the weights do not change** - this means if the output of the perceptron is equal to the expected output (i.e., $y(t)=y_d(t)$, no weight adjustment is needed.

$$\begin{aligned} &\textit{if} \quad w(t)x(t) \geq 0 \text{ and } x(t) \in CL1, \quad y(t)=1 \\ &\textit{or} \quad w(t)x(t) < 0 \text{ and } x(t) \in CL2, \quad y(t)=-1 \\ &\textit{then } w(t+1) = w(t) \end{aligned}$$

- **If the input vector x(t) is misclassified** update the weights based on the error between the predicted output y(t) and the desired output $y_d(t)$. Weight update rule: $w(t+1) = w(t) + \eta(y_d(t)-y(t))\, x(t)$ where η is the learning rate, $y_d(t)$ is the desired output, y(t) is the predicted output, and x(t) is the input vector.

$$\begin{aligned} &\textit{if} \quad w(t)x(t) \geq 0 \text{ and } x(t) \in CL2 \\ &\textit{then} \quad w(t+1) = w(t) - \gamma x(t) \\ \\ &\textit{if} \quad w(t)x(t) < 0 \text{ and } x(t) \in CL1 \\ &\textit{then} \quad w(t+1) = w(t) + \gamma x(t) \end{aligned}$$

- **Repeat the process for all training examples** - after each input vector x(t)x(t)x(t) is processed, the weights are updated if necessary. After processing all the training examples (an epoch), check for convergence: If the perceptron correctly classifies all examples, stop training.
- **Terminate when convergence is achieved** - if the weights have been updated and the perceptron correctly classifies all training examples after an epoch, stop the learning process.If convergence is not reached, repeat steps 2–4 until all examples are correctly classified.

2.2.11

What happens when an input vector x(t) is correctly classified by the perceptron during the learning process?

- The weights do not change.
- The weights are updated.
- The training is stopped.
- The bias is set to zero.

📖 2.2.12

Perceptron convergence theorem

The Perceptron convergence theorem states that, for a given training set of vectors XXX that can be divided into two distinct classes, CL1 and CL2, which are linearly separable, the perceptron algorithm will always converge after a finite number of mistakes.

Key principles:

- Linearly separable data - there exists a hyperplane (a straight line in 2D, or a plane in 3D) that can separate the two classes (CL1 and CL2) perfectly.
- Mistakes during training - the perceptron learns by adjusting its weights every time it makes a mistake. The number of mistakes corresponds to the number of times the perceptron misclassifies an example.
- Convergence - after making a finite number of mistakes, the perceptron will reach a point where no further mistakes are made, and its weights will no longer change. This means the perceptron will have learned to reliably classify the vectors into their correct classes.

How the theorem works:

1. **Training process** - the perceptron receives input vectors from the training set and classifies them. If the classification is correct, the weights stay the same. If the classification is incorrect, the weights are adjusted according to the error.
2. **Convergence condition** - the perceptron will continue to adjust the weights until it no longer makes mistakes on the training set. This happens because the training data is linearly separable, and the perceptron is capable of finding a hyperplane that separates the classes without any further mistakes.

Practical implication

- **When the perceptron converges**, it means that the weights have adjusted in such a way that the perceptron can now reliably classify all examples in the training set correctly.
- The convergence is guaranteed only if the data is **linearly separable**. If the data cannot be separated by a straight line (or hyperplane), the perceptron will not converge.

2.2.13

What does the Perceptron convergence theorem guarantee?

- The perceptron will converge to a state where it no longer makes mistakes on a linearly separable dataset.
- The perceptron will always reach a state of zero error on any dataset.
- The perceptron will converge faster if the data is not linearly separable.
- The perceptron will eventually classify all data points as the same class.

2.3 Practical example

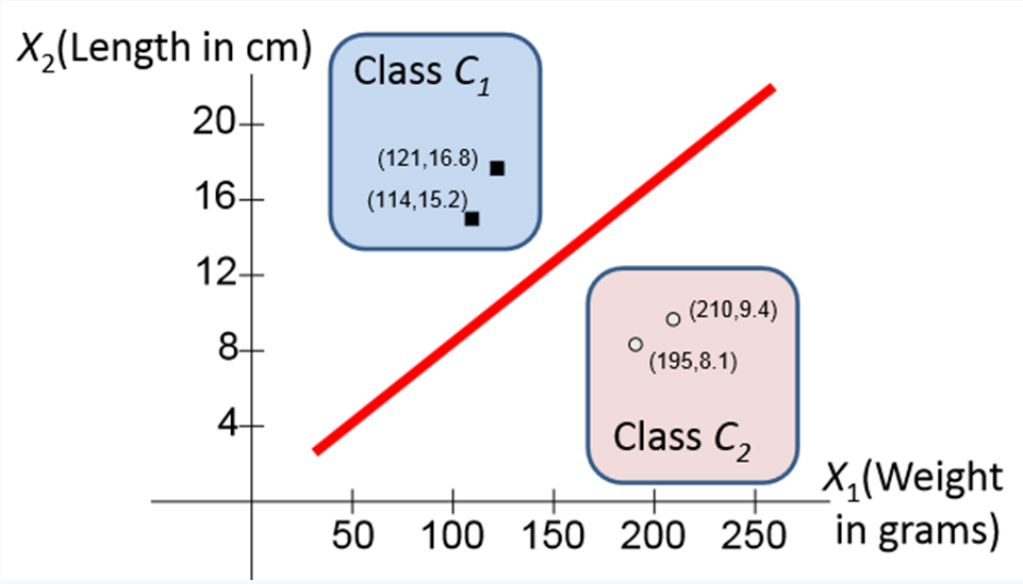
2.3.1

In a practical example, we will create a perceptron for fruit classification into two classes C1 and C2. We will adjust the weights of the perceptron using Hebb learning based on examples from the training set.

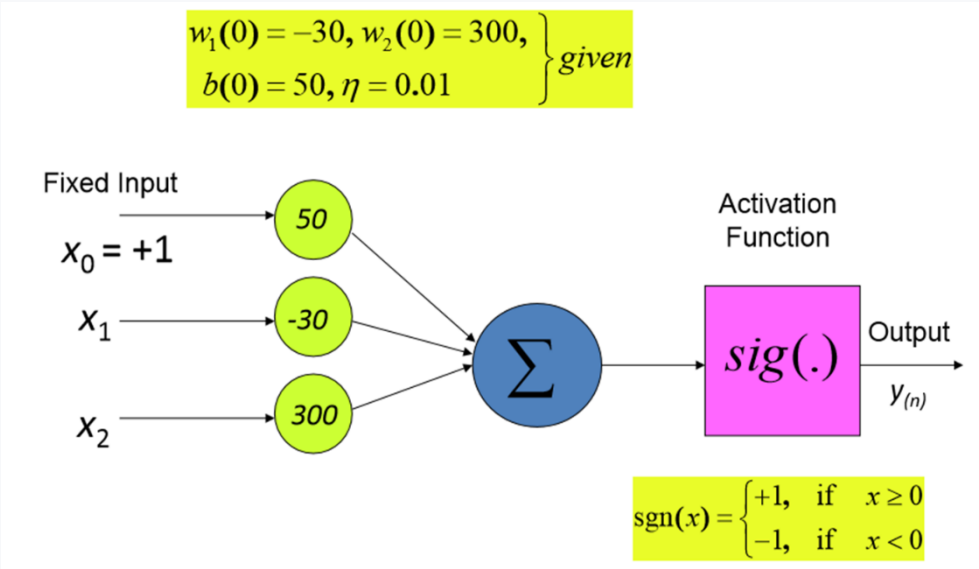
This contains two examples (121; 16.8), (114; 15.2) from the first class **C1** and two examples (210; 9.4), (195; 8.1) from the second class **C2**. The first value in each training example represents the weight of the fruit (in grams), the second its length (in cm).

	Weight (grams)	Length (cm)
Fruit 1 (Class C1)	121	16.8
	114	15.2
Fruit 2 (Class C2)	210	9.4
	195	8.1

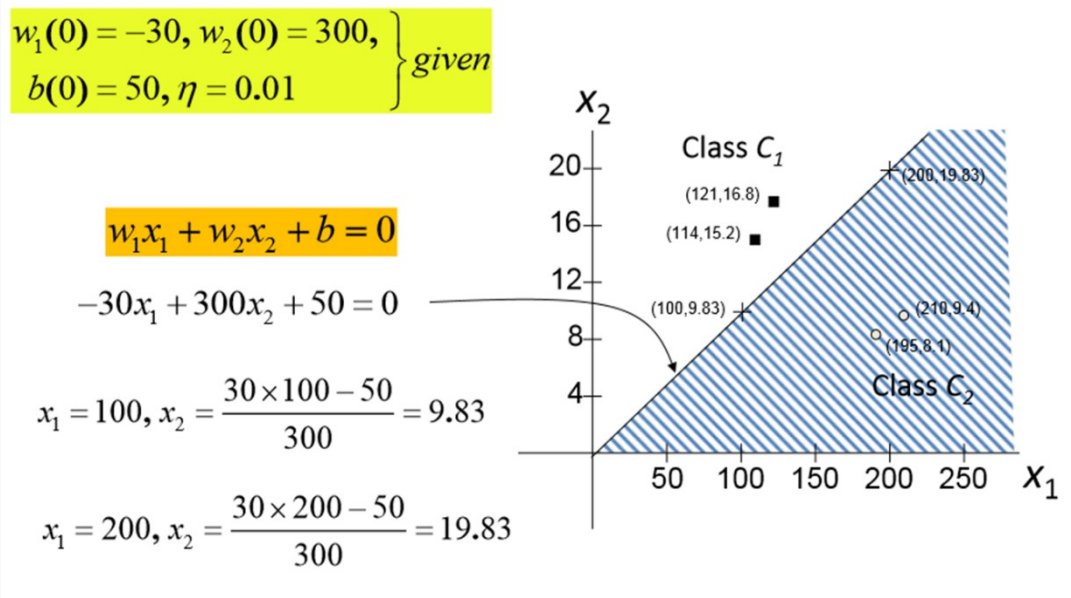
We can visualize the training set.



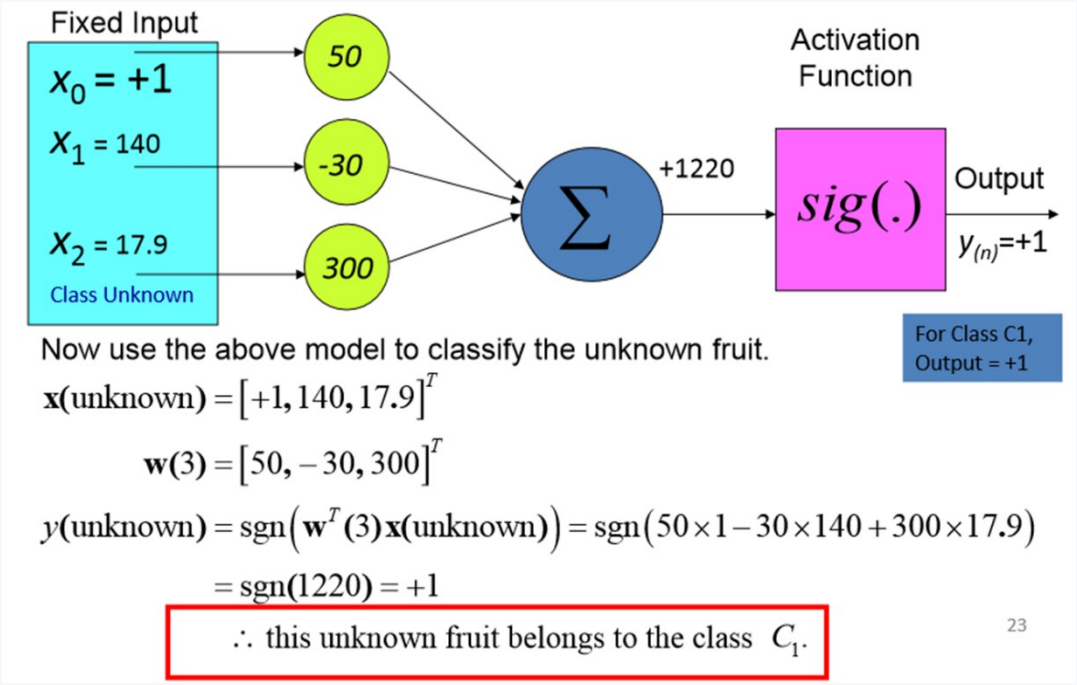
If we knew the correct setting of the weights, then it is obvious that I will also be able to correctly classify any fruit. Assume that we know the correct setting of the scales.



For this setting of weights, we can even determine a separating hyperplane that separates examples of one class from another.



With the correct setting of the scales, it is then easy to classify new unknown fruits. For example, I classify fruit with a weight of 140g and a length of 17.9 cm. By simply transferring the vector (140; 17.9) to the input of the perceptron, we can perform the calculation.



The perceptron result classified our input example (unknown fruit) into class **C1**.

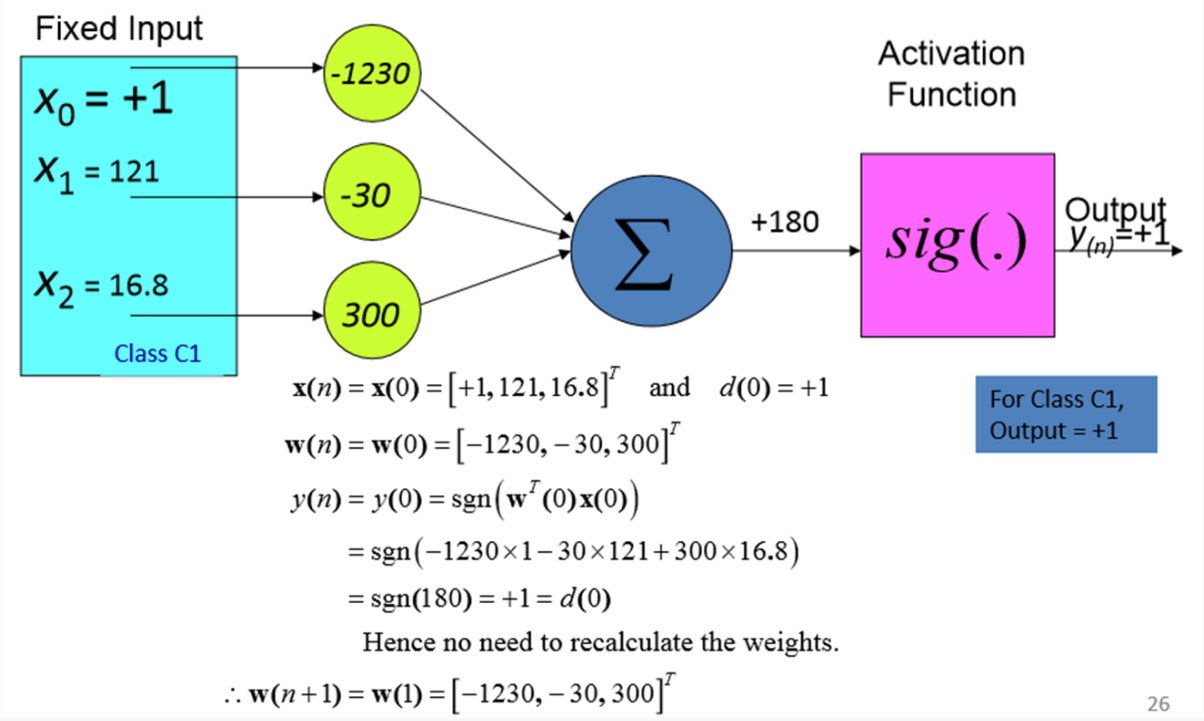
📖 2.3.2

The question remains how to correctly set the weights and bias value for the perceptron. In the step element, we set the weight values as follows ***w_1 = -30***; ***w_2 = 300*** and bias ***w_0 = -1230***

Subsequently, I will go through all examples of the training set and implement Hebbian learning.

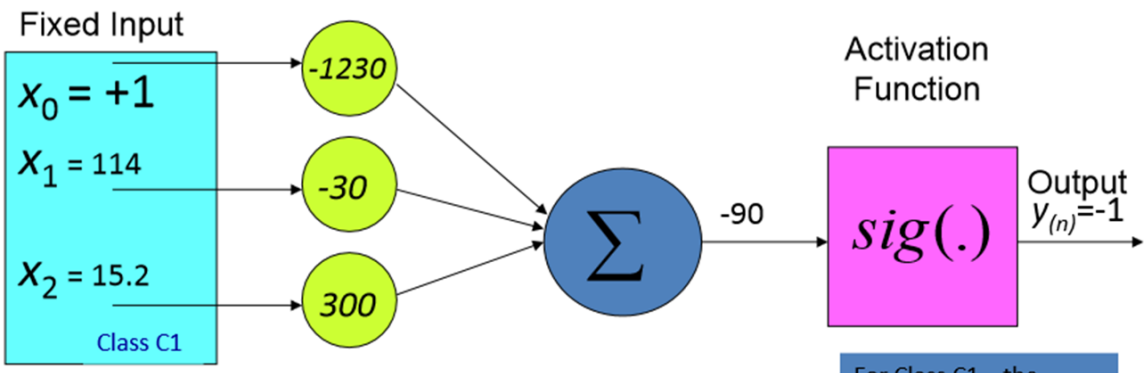
	Weight (grams)	Length (cm)
Fruit 1 (Class C1)	121	16.8
	114	15.2
Fruit 2 (Class C2)	210	9.4
	195	8.1

Let's take the first example of the training set **(121; 16.8)**, find out the response (result) of the neuron and compare the result with the value **+1** to find out if it is necessary to adjust the weights. Comparing the result with the value **+1** is important because the first example is to be classified in class **C1**, i.e. the result must be **+1**.



The result of the perceptron is **+1**, the example belongs to **C1**, i.e. the actual result should have been **+1** as well. For this reason, there is **no need** to adjust the weight.

In the case of the second example, however, we find that it is necessary to adjust the weights.



$$\mathbf{x}(1) = [+1, 114, 15.2]^T \quad \text{and} \quad d(1) = +1$$

$$\mathbf{w}(1) = [-1230, -30, 300]^T$$

$$y(1) = \text{sgn}\left(\mathbf{w}^T(1)\mathbf{x}(1)\right) = \text{sgn}\left(-1230 \times 1 - 30 \times 114 + 300 \times 15.2\right)$$

$$= \text{sgn}(-90) = -1 \neq d(1)$$

Hence **we have to** recalculate the weights.

We adjust the weights by applying the following formulas.

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta\left[d(n) - y(n)\right]\mathbf{x}(n)$$

$$\mathbf{w}(1) = [-1230, -30, 300]^T$$

$$\mathbf{x}(1) = [+1, 114, 15.2]^T$$

$$d(1) = +1, y(1) = -1, \eta = 0.01$$

$$\mathbf{w}(1+1) = \mathbf{w}(2) = [-1230, -30, 300]^T + 0.01\left[+1 - (-1)\right][+1, 114, 15.2]^T$$

$$= [-1230, -30, 300]^T + [+0.02, 2.28, 0.304]^T$$

$$\therefore \mathbf{w}(2) = [-1229.08, -27.72, 300.304]^T$$

2.3.3

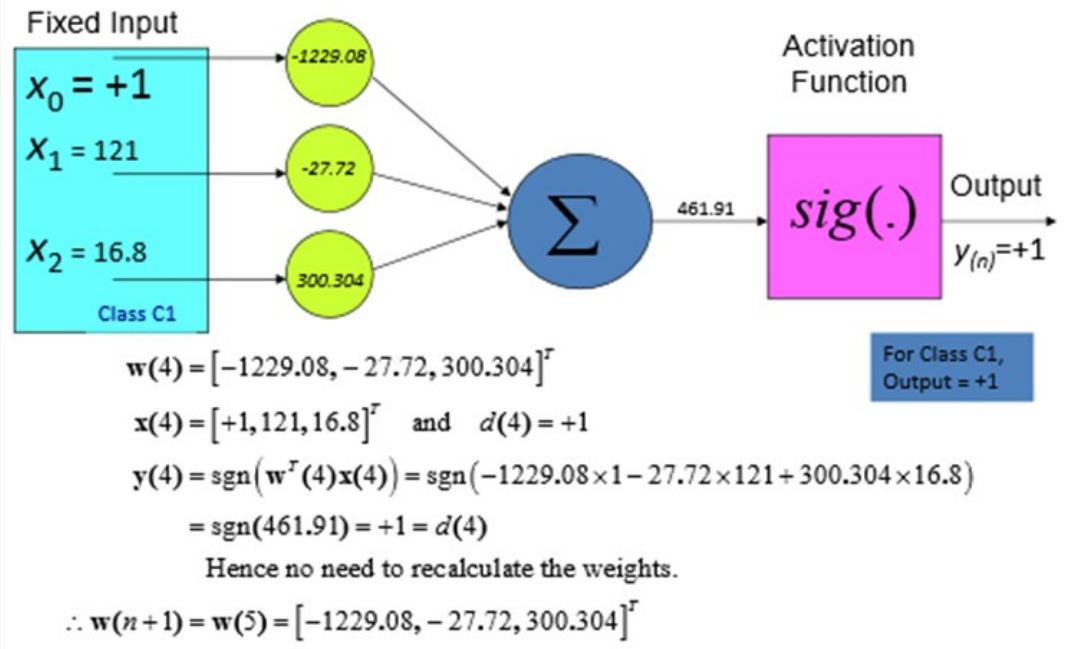
During perceptron learning, we used four examples of the training set to adjust the weights by successively feeding them to the input of the perceptron, and in case of a wrong result, we adjusted the weights using Hebbian learning. If we fed all the examples to the input of the perceptron and adjusted the weights if necessary, we realized one epoch of learning.

For the following four training examples

	Weight (grams)	Length (cm)
Fruit 1 (Class C1)	121	16.8
	114	15.2
Fruit 2 (Class C2)	210	9.4
	195	8.1

In the second epoch, we can implement the following steps:

1.



$$\mathbf{w}(4) = [-1229.08, -27.72, 300.304]^T$$

$$\mathbf{x}(4) = [+1, 121, 16.8]^T \quad \text{and} \quad d(4) = +1$$

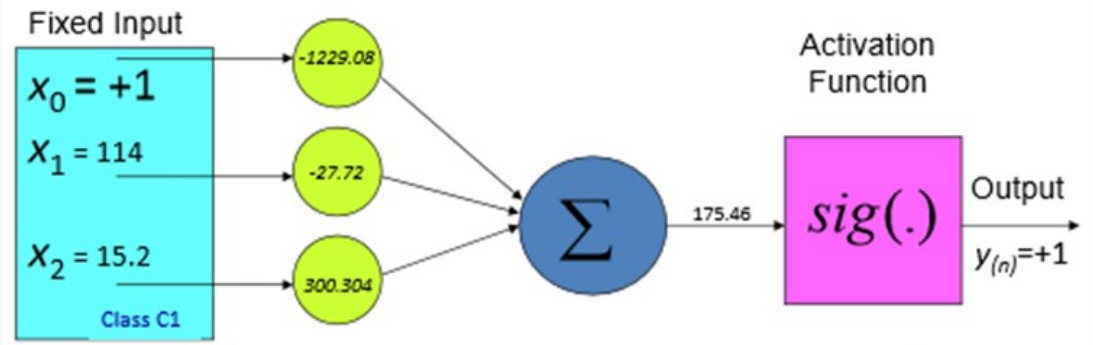
$$\mathbf{y}(4) = \text{sgn}\left(\mathbf{w}^T(4)\mathbf{x}(4)\right) = \text{sgn}\left(-1229.08 \times 1 - 27.72 \times 121 + 300.304 \times 16.8\right)$$

$$= \text{sgn}(461.91) = +1 = d(4)$$

Hence no need to recalculate the weights.

$$\therefore \mathbf{w}(n+1) = \mathbf{w}(5) = [-1229.08, -27.72, 300.304]^T$$

2.



$$\mathbf{w}(5) = [-1229.08, -27.72, 300.304]^T$$

$$\mathbf{x}(5) = [+1, 114, 15.2]^T \quad \text{and} \quad d(5) = +1$$

$$\mathbf{y}(5) = \text{sgn}\left(\mathbf{w}^T(5)\mathbf{x}(5)\right) = \text{sgn}(-1229.08 \times 1 - 27.72 \times 114 + 300.304 \times 15.2)$$

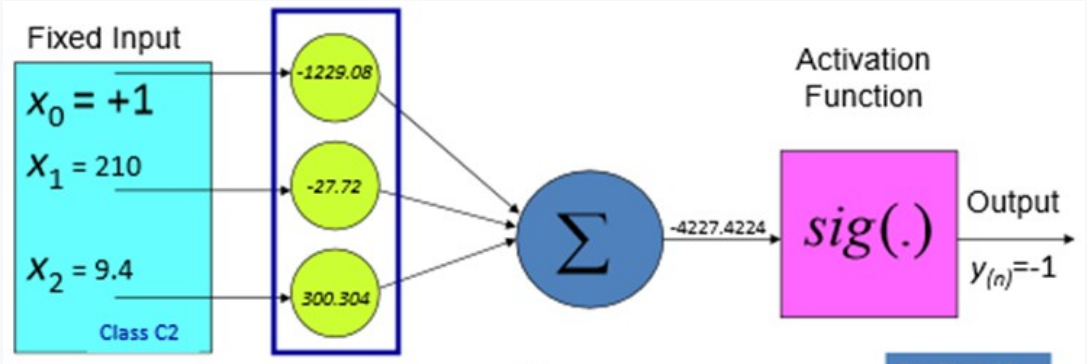
$$= \text{sgn}(175.46) = +1 = d(5)$$

Hence no need to recalculate the weights.

$$\therefore \mathbf{w}(n+1) = \mathbf{w}(6) = [-1229.08, -27.72, 300.304]^T$$

For Class C1, Output = +1

3.



$$\mathbf{w}(2) = [-1229.08, -27.72, 300.304]^T$$

$$\mathbf{x}(2) = [+1, 210, 9.4]^T \quad \text{and} \quad d(2) = -1$$

$$\mathbf{y}(2) = \text{sgn}\left(\mathbf{w}^T(2)\mathbf{x}(2)\right) = \text{sgn}(-1229.08 \times 1 - 27.72 \times 210 + 300.304 \times 9.4)$$

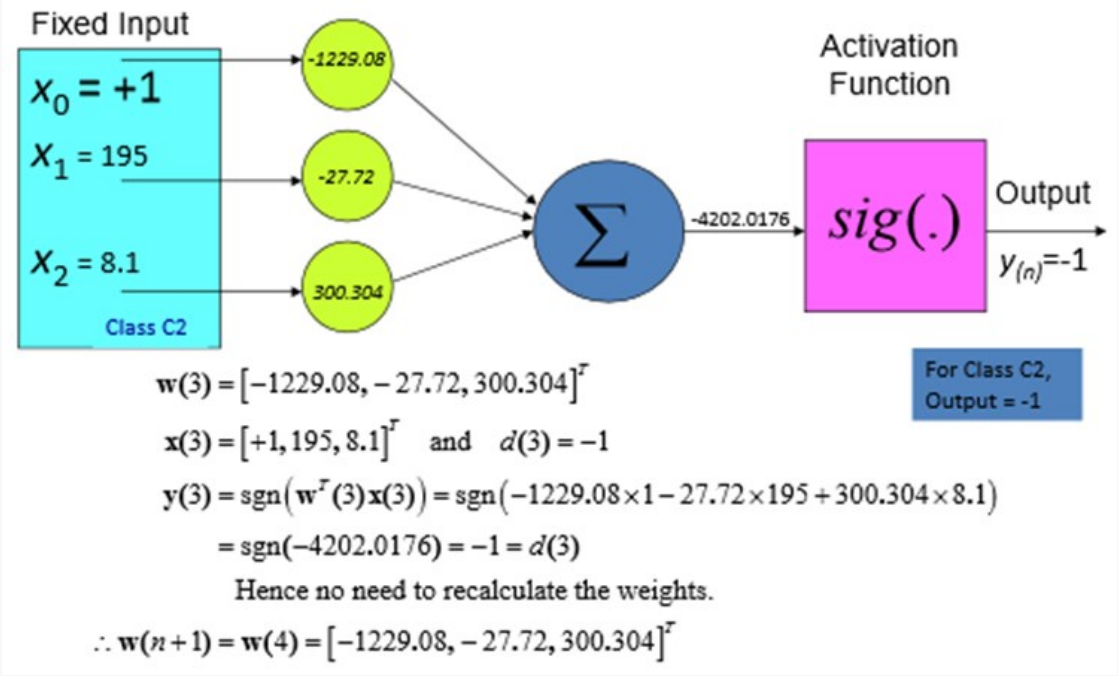
$$= \text{sgn}(-4227.4224) = -1 = d(2)$$

Hence no need to recalculate the weights.

$$\therefore \mathbf{w}(n+1) = \mathbf{w}(3) = [-1229.08, -27.72, 300.304]^T$$

For Class C2, Output = -1

4.



$$\mathbf{w}(3) = [-1229.08, -27.72, 300.304]^T$$

$$\mathbf{x}(3) = [+1, 195, 8.1]^T \quad \text{and} \quad d(3) = -1$$

$$\mathbf{y}(3) = \text{sgn}\left(\mathbf{w}^T(3)\mathbf{x}(3)\right) = \text{sgn}\left(-1229.08 \times 1 - 27.72 \times 195 + 300.304 \times 8.1\right)$$

$$= \text{sgn}(-4202.0176) = -1 = d(3)$$

Hence no need to recalculate the weights.

$$\therefore \mathbf{w}(n+1) = \mathbf{w}(4) = [-1229.08, -27.72, 300.304]^T$$

Note that we have not changed any weights in this epoch. It is obvious that if we were to implement other epochs, nothing would change. So we found the right balance setting.

We can define a separating hyperplane for this weight setting

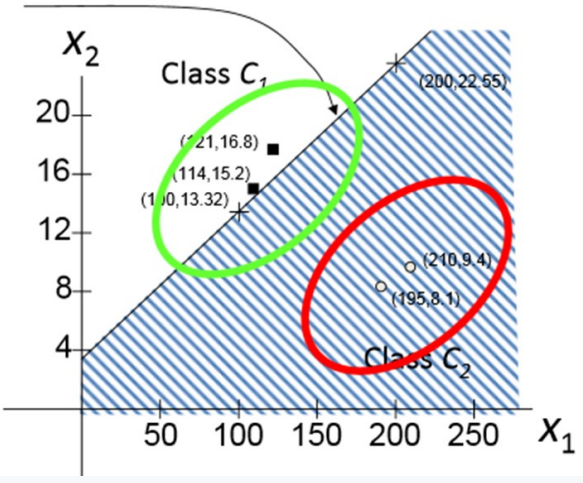
Therefore the decision boundary obtained after Neural Network training is:

$$-27.72x_1 + 300.304x_2 - 1229.08 = 0$$

$$x_1 = 100,\ x_2 = \frac{27.72 \times 100 + 1229.08}{300.30} = 13.32$$

$$x_1 = 200,\ x_2 = \frac{27.72 \times 200 + 1229.08}{300.30} = 22.55$$

The new decision boundary can now be used to classify any unknown item.



2.3.4

For the previous example, we will create a simple source code. The only library we will need is numpy.

```
from numpy import array
```

We will work with known training data.

	Weight (grams)	Length (cm)
Fruit 1 (Class C1)	121	16.8
	114	15.2
Fruit 2 (Class C2)	210	9.4
	195	8.1

We will copy these into the array ***training_data***

```
from numpy import array
training_data = [
    (array([121,16.8]), 1),
    (array([114,15.2]), 1),
    (array([210,9.4]), -1),
    (array([195,8.1]), -1),
]
print(training_data)
```

Program output:

```
[(array([121. ,  16.8]), 1), (array([114. ,  15.2]), 1),
(array([210. ,   9.4]), -1), (array([195. ,   8.1]), -1)]
```

We define a signed (step) function.

```
def activation_fn(x):
    if x>=0:
        return 1
    else:
        return -1
```

In the general solution, we set the initial values of weights and bias randomly. In our example, we will set these values directly, according to the previous settings.

```
# we set weights
w = array([-30,300])
b = -1230
eta = 0.01
```

In our example, for the sake of clarity, we will create only one epoch, i.e. we recalculate the training set only once.

```
print('current weights: ' , w)
print('bias: ', b)

for i in range(0, 4):
    print('---')
    x, y = training_data[i]
    print('training data: ' , x , ', result: ', y)
    internal_energy = ((x * w).sum()) + b
    print('internal energy: ',internal_energy)
    prediction = activation_fn(internal_energy)
    print('prediction: ',prediction)
    error = y - prediction
    if (error != 0):
        print('needed to change weights')
        w = w + (eta * error * x)
        b = b + (eta * error * 1)
    print('current weights: ' , w)
    print('bias: ', b)
```

Program output:

```
current weights:  [-30 300]
bias:  -1230
---
training data:  [121.   16.8] , result:  1
internal energy:  180.0
prediction:  1
current weights:  [-30 300]
bias:  -1230
---
training data:  [114.   15.2] , result:  1
internal energy:  -90.0
prediction:  -1
needed to change weights
current weights:  [-27.72  300.304]
bias:  -1229.98
---
```

```
training data:  [210.    9.4] , result:  -1
internal energy:  -4228.3224
prediction:  -1
current weights:  [-27.72  300.304]
bias:  -1229.98
---
training data:  [195.    8.1] , result:  -1
internal energy:  -4202.9176
prediction:  -1
current weights:  [-27.72  300.304]
bias:  -1229.98
```

For completeness, we also calculate a straight line as a separating hyperplane given by the correct setting of weights and bias

```
def line(x):
    y = (w[0]*x + b)/(w[1]*(-1))
    return y
```

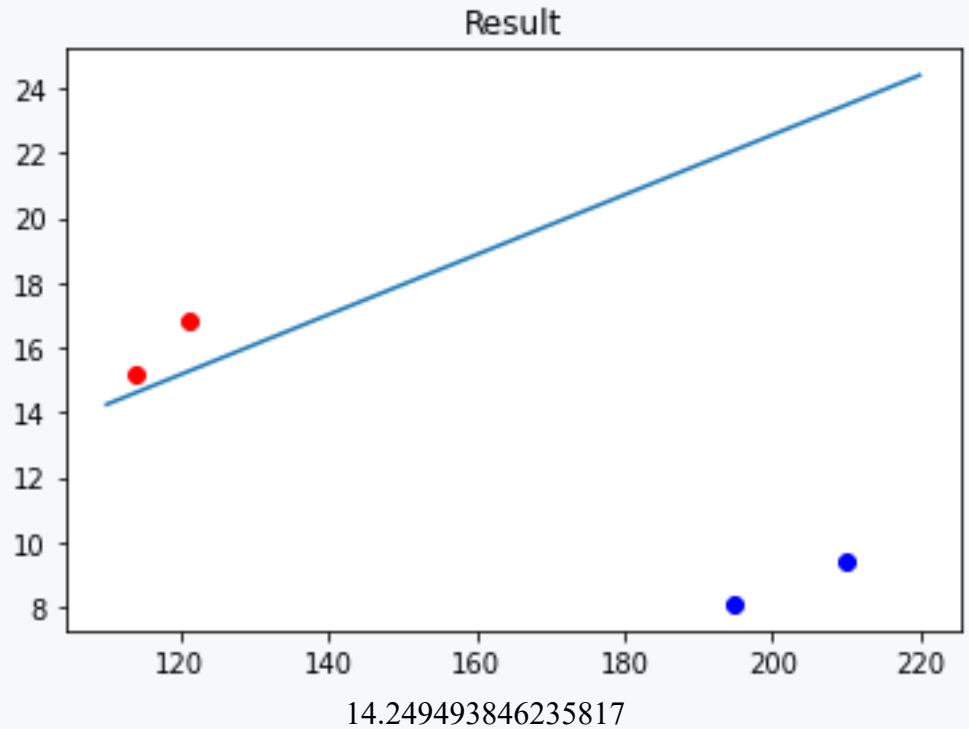
We draw the separating hyperplane graphically.

```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
ax = plt.subplot()
ax.set_title("Result")
# Plot the training points
#ax.scatter(x[:, 0], x[:, 1], c=q, cmap=cm_bright)


for x, expected in training_data:
    if expected==1:
        pattern='r'
    else:

        pattern='b'
    print(x[0])
    ax.scatter(x[0], x[1], color=pattern)

plt.plot([110,220],[line(110),line(220)])
```

```
#plt.plot([25,200],[50,200])

plt.show()
print(line(110))
print(line(220))
```

Program output:

```
/home/johny/.local/lib/python3.9/site-
packages/matplotlib/projections/__init__.py:63: UserWarning:
Unable to import Axes3D. This may be due to multiple versions
of Matplotlib being installed (e.g. as a system package and as
a pip package). As a result, the 3D projection is not
available.
  warnings.warn("Unable to import Axes3D. This may be due to
multiple versions of "
121.0
114.0
210.0
195.0
```



14.249493846235817

```
24.40320475251745
```

The last step will be to use the trained values to predict the unknown fruit

```
def estimate(vector):
    internal_energy = ((vector * w).sum()) + b
```

```
    prediction = activation_fn(internal_energy)
    return prediction
```

In the case of a fruit that is 180 g and 10 cm, we can find out that it belongs to the second class C2

```
vector = array([180,10])
print(estimate(vector))
```

Program output:

```
-1
```

In the case of a fruit that is 140 g and 20 cm, we can find that it belongs to the first class C1

```
vector = array([140,20])
print(estimate(vector))
```

Program output:

```
1
```

Feedforward Neural Network

Chapter 3

3.1 Single layer perceptron

🕮 3.1.1

The **activation function** of a neuron determines how the neuron processes input signals to produce an output. It is a mathematical function applied to the input sum (**ini(t)**) received by the neuron.

The state of a neuron **i** is defined by the variable $\mathbf{y_i}$, expressed as:

$$\mathbf{y_i = f(ini)}$$

where, f() represents the **activation function**.

The common activation functions is s**igned (step) function** defined as**:**

$$\mathbf{y_i = f\,(in_i) = 1 \text{ if } in_i >= 0 \text{ or } 0 \text{ if } in_i < 0}$$

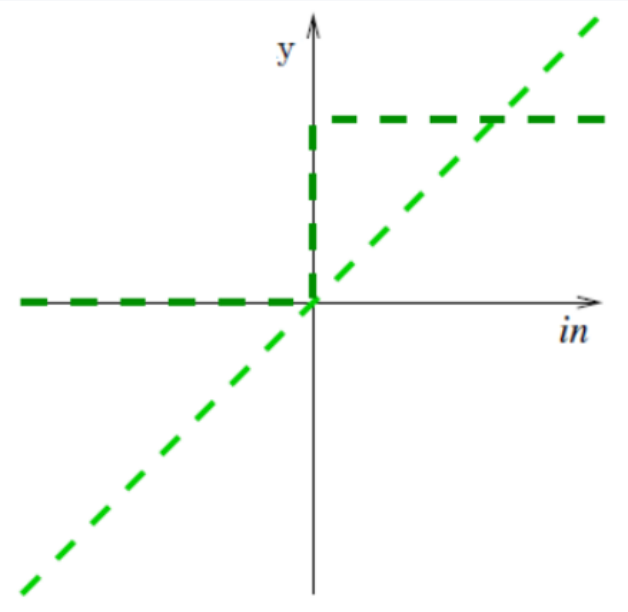
Function produces binary outputs (e.g., -1 or 1, or 0 and 1) and can be used for simple decision-making tasks.

However, as well as the signed function, we can also use other functions, e.g. **linear function**.

$$\mathbf{y_i = f\,(in_i) = in_i}$$

which produces continuous outputs. It is suitable for tasks where outputs need to vary proportionally with inputs.

We can visualize the graphs of both functions:



These activation functions play a key role in determining how a single-layer perceptron operates. While the step function is often used in classification tasks, the linear function can be helpful for regression or less complex relationships.

3.1.2

Which activation function is commonly used in perceptrons for binary classification tasks?

- Step Function
- Linear Function
- Exponential Function
- Sigmoid Function

3.1.3

In addition to the step function and linear function, other activation functions are often employed in neural networks for their unique properties.

A piecewise linear function

The piecewise linear function is defined by different linear equations over specific intervals of the input. We can define it for example as:

$$f(\text{ini}) = \begin{cases} 1 & \text{if ini} > \frac{1}{2}, \\ \text{ini} & \text{if } -\frac{1}{2} \leq \text{ini} \leq \frac{1}{2}, \\ -1 & \text{if ini} < -\frac{1}{2}. \end{cases}$$

where for inputs greater than 1/2, the output saturates at 1, for inputs smaller than −1/2, the output saturates at -1 and for inputs between −1/2 and 1/2, the output is linear and proportional to the input.

The graph of this function has three distinct regions:

- A flat line at y=1 for inputs above 1/2
- A linear slope between y=−1/2 and y=1/2
- A flat line at y=−1 for inputs below −1/2.

This piecewise function can be used in networks where we want bounded outputs while preserving some level of proportionality for intermediate input values. It offers

a balance between sharp decision boundaries (step functions) and smoothness (sigmoidal functions).

3.1.4

What is the output of the piecewise linear activation function if the input ini = 0?

- 0
- 1
- -1
- -1/2

3.1.5

Sigmodial function

The sigmoidal function is a smooth and differentiable function, commonly used in neural networks due to its useful properties in the learning process. The mathematical form of the sigmoidal function with your specified exponent is:

$$f(\text{ini}) = \frac{1}{1 + e^{-\alpha \cdot \text{ini}}}$$

where α is a positive constant that controls the slope or steepness of the curve. Higher values of α result in a sharper transition between the outputs.

Output values range is from 0 to 1, making it ideal for probabilistic interpretations. The function is smooth and differentiable, which is crucial for gradient-based learning algorithms.

Graph at ini = 0, the function's output is 0.5, as ini→∞ f(ini)→1 and as ini→−∞, f(ini)→0.

This function is very often used in feedforward neural networks. The function is "smooth", this feature is very important for setting the weights in the learning process. A smooth function is differentiable.

3.1.6

What is the effect of increasing the parameter α in the sigmoidal function?

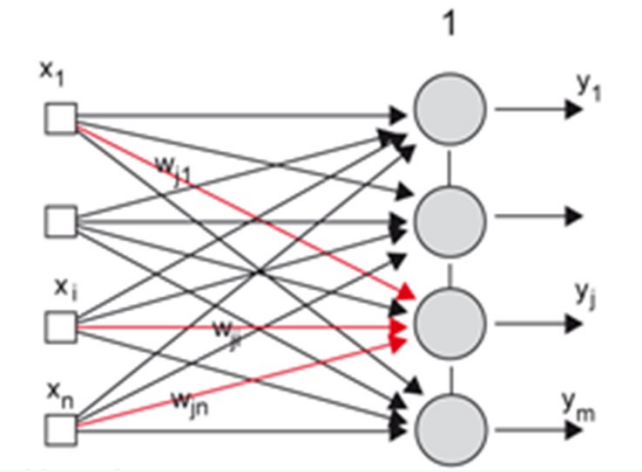
- It sharpens the transition between low and high output values.
- It increases the range of output values.
- It reduces the smoothness of the function.
- It decreases the maximum output value.

3.1.7

We already know that a learning neuron alone can solve linearly separable problems. We also know that for other types of problems, neurons can be connected into neural networks.

Conceptually, the simplest network is the Single Layer Perceptron. These are ***M*** independent, parallel working neurons. Thus, each of these neurons realizes the transformation of the input vector to the output value independently of the other neurons.

From the point of view of organizational dynamics, the network consists of ***N*** neurons of the input layer and a layer of ***M*** output neurons. Both layers are fully connected to each other when every ***j***-th output neuron is connected to all input neurons.



In the course of active dynamics, the network generally realizes the display from $R^n \rightarrow R^m$,which was set during the adaptation dynamics. The specific values of the output values are given by the activation transfer functions of the output neurons, that is, for example, in the case of sigmoidal activation functions approximating a sharp nonlinearity, it is the realization of the display from $R^n \rightarrow (0,1)^m$

3.1.8

What is the key characteristic of a single layer perceptron network?

- It is made up of multiple independent neurons, with each output neuron connected to all input neurons.
- It can solve any type of problem, whether linearly separable or not.
- It consists of a single neuron capable of solving complex problems.
- It does not require adaptation dynamics to perform transformations.

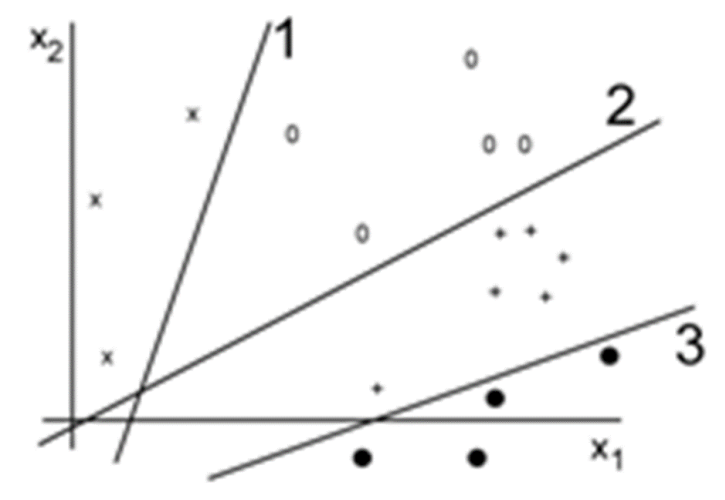
3.1.9

Let's consider the classification possibilities of a single-layer perceptron. If we consider the mapping $R^n \rightarrow \{0,1\}^m$, .e. the classification mapping into two classes, we can find ***m*** separating hyperplanes in space, one for each neuron of the output layer.

However, the mere multiplication of neurons in the output layer does not bring any change in classification possibilities compared to a simple perceptron, because the neural network lacks the possibility to further combine the outputs of individual neurons and thus enable classification into several classes.

For illustration, we present a graphic representation of the classification of the perceptron into the mentioned four classes.

	x	0	+	●
y_1	1	-1	-1	-1
y_2	1	1	-1	-1
y_3	1	1	1	-1



3.1.10

Why does a single-layer perceptron have limited classification capabilities?

- Its neurons work independently without combining their outputs.
- It can only use a step activation function.
- It cannot classify into more than two classes.
- Adding more output neurons exponentially increases complexity.

3.1.11

Non-linearity in real problems

Many real-world problems are inherently non-linear, meaning their solutions cannot be found by simple linear boundaries or transformations. Single-layer perceptrons are insufficient for such problems as they rely solely on linear classification. To address this limitation, the development of more complex neural network architectures became essential.

A major breakthrough occurred in 1986 when Rumelhart, Hinton, and Williams introduced the error backpropagation method. This training rule revolutionized neural networks by enabling the adjustment of weights in feedforward networks with hidden layers. These hidden neurons allow networks to learn complex patterns and solve non-linear problems, greatly expanding their application.

Multilayer feedforward artificial neural networks (ANNs) trained with backpropagation are particularly powerful. In these networks, neurons are organized in layers, with each layer fully connected to the next. Signals are passed only in one direction-forward-from the input layer to the output layer. There are no backward or intra-layer connections. This structure ensures efficient processing and is the

foundation of many modern neural network models used in complex problem-solving.

3.1.12

What key feature allows multilayer feedforward networks to solve non-linear problems?

- The presence of hidden layers with neurons.
- The use of step activation functions in neurons.
- Connections within the same layer.
- The ability to classify linearly separable problems.

3.1.13

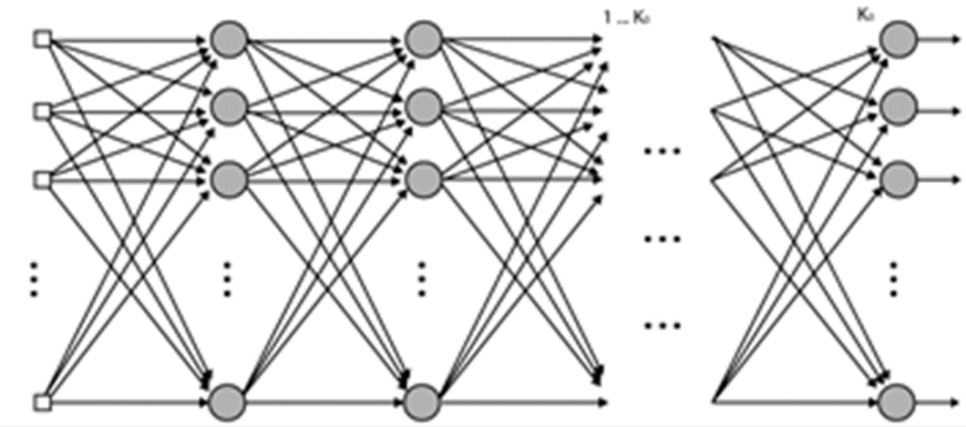
Multilayer perceptron

The first typical feedforward multilayer neural network is the **Multilayer perceptron (MLP)**. It is composed of multiple layers of neurons, where adjacent layers are fully connected. This means that each neuron in one layer connects to every neuron in the next layer, ensuring comprehensive communication between layers. Importantly, there are no connections within the same layer.

The structure of the MLP includes an input layer, one or more hidden layers, and an output layer. The number of neurons in the hidden layers is determined by the nature of the task and usually falls between the number of neurons in the input and output layers. Each layer processes the input it receives and passes the output to the next, realizing a mapping from R^n to R^m. This enables the MLP to handle complex transformations and tasks.

he activity in an MLP progresses in discrete steps. During each step, the outputs of the neurons in a given layer are calculated simultaneously. For a neuron in layer k, its output depends on the weighted inputs from all neurons in the previous layer, summed and passed through an activation function. The dimension n of the input vector corresponds to the number of neurons in the input layer and dictates the structure of subsequent layers.

The number of neurons in the hidden layers can be different, it is chosen according to the nature of the solved task, usually in the range between the number of input and output neurons.



A multi-layer perceptron realizes the mapping $R^n \rightarrow R^m$

The activity takes place in steps ***k***, while the outputs of the ***j***-th neurons in layer ***k*** are calculated in parallel according to the known relationship:

$$^{k}y_j = \sigma(\zeta) = \sigma(\sum_{i=0}^{n} {}^{k}w_{ji}x_i)$$

where ***n*** expresses the dimension of the input vector as well as the number of neurons in the ***k***-th layer

3.1.14

What determines the number of neurons in the hidden layers of a Multilayer Perceptron?

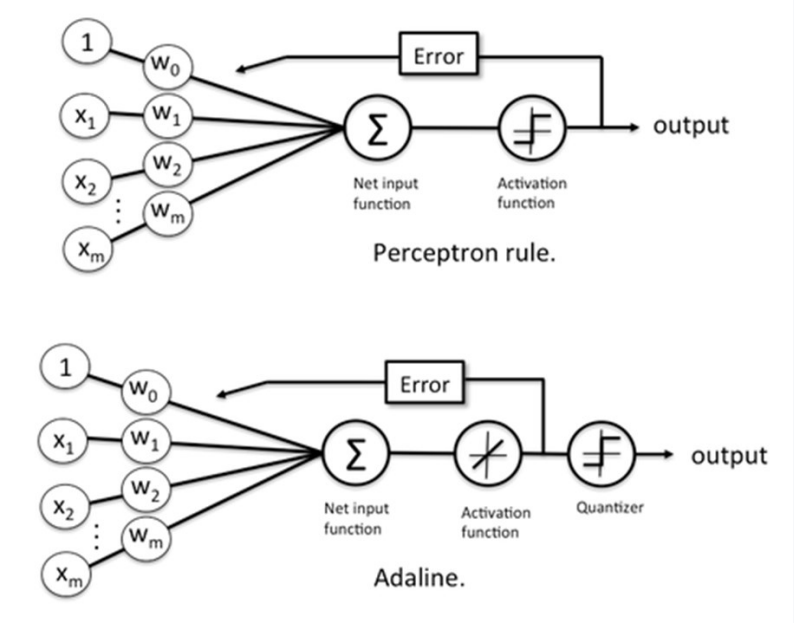
- The nature of the task being solved.
- The dimension of the input vector.
- The number of neurons in the output layer.
- The number of connections within a layer.

3.2 Adaline and Madaline

3.2.1

In practice, instead of a simple perceptron, a continuous perceptron is often used, the so-called **Adaline** (Adaptive Linear Neuron).

Adaline and Simple Perceptron have the same topology, but different learning method as well as the overall focus of the NN. It was described by Widrow and Hoff in 1960. Similar to the perceptron, it is used for linear classification into two classes.



Perceptron uses class labels to learn weight values. Adaline uses continuous values (based on the input) to figure out the weight values, which is "stronger" because it tells us "by how much" we classified correctly or incorrectly. Adaline's learning algorithm is different. It uses the so-called **gradient learning method**. The requirement is that the learning behavior is as similar as possible to the overall behavior of the teacher.

3.2.2

What is the primary difference between Adaline and the Simple Perceptron?

- Adaline uses continuous values, while the Simple Perceptron uses class labels for learning weights.
- Adaline has a different network topology than the Simple Perceptron.
- Adaline uses class labels, while the Simple Perceptron uses continuous values for learning weights.
- Adaline cannot perform linear classification into two classes.

🕮 3.2.3

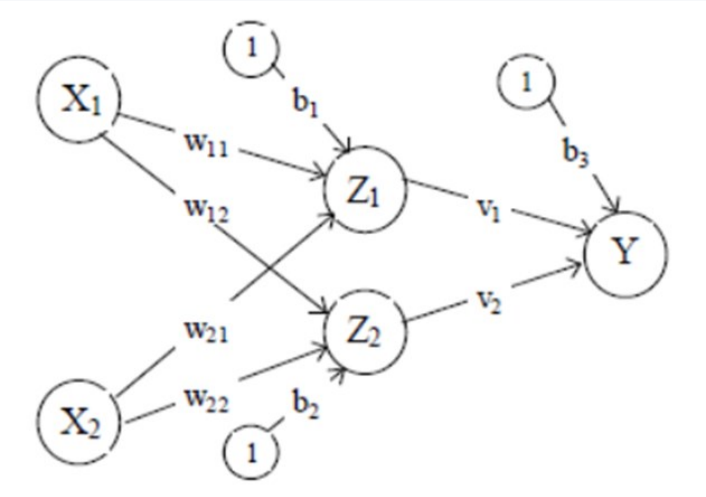
Madaline neural network

The Madaline neural network (Many Adaptive Linear Neurons) is one of the simplest feedforward neural networks. It consists of Adaline neurons as its basic building blocks. These Adaline neurons are responsible for the initial processing, and the network's primary function is to perform logical operations based on their outputs.

In Madaline, the output signal Y is determined by the logical "OR" function, which is activated when at least one of the hidden neurons (Z_1 or Z_2) generates an output signal. The weights v_1, v_2, and the bias b_3 for the output neuron are fixed with values of 1/2. The Adaline neurons in the hidden layer use a step activation function to output either 1 or -1. This network is adapted using the MRI (Multiple Regression Interpretation) adaptation algorithm, which updates only the input weights connecting the hidden layer neurons.

The training process involves initializing the weights and biases, then performing updates based on the error signal between the predicted output and the actual target output. The algorithm adjusts the weights of the connections leading to the hidden neurons and their corresponding biases when the network's output does not match the desired output. This allows the network to learn and improve its classification abilities.

The basic element is the Adaline neuron. The output signal ***Y*** is equal to ***1***, if at least one value of the signal coming from the hidden neurons (i.e.Z_1, Z_2 or both at the same time).



Madaline uses the MRI Adaptation Algorithm (1960) to adapt only the input weights to the hidden layer.

The weight values to the output neuron ***Y*** are fixed.

The output neuron ***Y*** performs the logical OR function. The weights v_1, v_2 and b_3 are fixed, that is:

```
v1 = 1/2, v2 = 1/2, b3 = 1/2
```

the activation function for Z_1, Z_2 and ***Y*** is a classic step function (sign function).

We will consider the training patterns ***[s,t]***, where s is the input vector and ***t*** is the output signal

The procedure:

1. Initialization of v_1, v_2 and b_3. Initialization of remaining weights - random. Setting the learning coefficient ***α***
2. For each training pair ***s:t***

- Activate input neurons $x_i=s_i$
- Calculate the input values of the hidden layers and the actual output value of Madaline:

```
z_in1 = b1 + x1*w11 + x2*w21
z_in2 = b2 + x1*w12 + x2*w22
z1 = f(z_in1)
z2 = f(z_in2)

y_in = b3 + z1*v1 + z2*v2
y = f(y_in)
```

3.Update weight coefficients - **network learning**:

- If ***y = t*** (i.e. the output of the network is equal to the output of the training pattern), then the weights and biases do not change
- For ***y ≠ t*** and ***t = 1***, so for weight values on connections to Z_J ***(J=1,2)***:

```
wiJ(new) = wiJ(old) + alfa * (1 - z_inJ)*xi
bJ(new) = bJ(old) + alfa * (1 - z_inJ)
```

- For ***y ≠ t*** and ***t = -1***, so for weight values on connections to ***ZJ (J=1,2)***:

```
wiK(new) = wiK(old) + alfa * (-1 - z_inJ)*xi
bK(new) = bK(old) + alfa * (-1 - z_inJ)
```

3.2.4

What is the main function of the output neuron Y in the Madaline network?

- It performs the logical "OR" function on the signals from Z1 and Z2.
- It calculates the sum of all inputs from the hidden neurons.
- It performs the logical "AND" function on the signals from Z1 and Z2.
- It combines the outputs from all layers using a weighted sum.

From Shallow Learning to Deep Learning

Chapter 4

4.1 Definition of deep learning

📖 4.1.1

Simple kinds of networks were discussed in previous sections. Structures such as multi-layer perceptron can be called **shallow neural networks** (SNNs). ANNs that have many hidden layers containing weights are called deep neural networks, and the process of training them is called deep learning. By increasing the number of layers and making the ANN deeper, the model becomes more flexible and will be able to model more complex functions. However, to gain this increase in flexibility, you need more training data and more computation power to train the model.

The term "deep" refers to the number of layers within a neural network. Traditional machine learning models typically process input data through a small number of layers, often fewer than a dozen. In contrast, deep learning models can contain hundreds or even thousands of layers. The use of deep neural networks allows for the automatic extraction of features at multiple levels of abstraction, which is critical for processing complex data such as images, audio, and text. By stacking multiple layers on top of each other, each layer can learn to transform the input data to make it easier for the next layer to learn a more abstract representation. This process can continue for many layers, allowing the network to learn highly complex representations of the input data.

Overall, the term "deep" in deep learning refers to the depth of the neural network, and the ability of deep neural networks to learn highly complex representations of the input data.

📝 4.1.2

What does the term *"deep"* in deep learning primarily refer to?

- The number of layers in the neural network
- The amount of data processed by the network
- The size of each layer in the network
- The computational power required for training

📖 4.1.3

Both deep neural networks (**DNNs**) and shallow neural networks (**SNNs**) are types of artificial neural networks (**ANNs**) used for machine learning tasks. They share several similarities, including:

1. Activation functions: Both DNNs and SNNs use activation functions to introduce nonlinearity into the network, allowing it to model complex relationships between the input and output.
2. Backpropagation: Both DNNs and SNNs use a backpropagation algorithm to update the network weights based on the error between the predicted output and the actual output during training.
3. Gradient descent: Both DNNs and SNNs use gradient descent optimization algorithms to minimize the error between the predicted output and the actual output during training.
4. Similar architecture: SNNs and DNNs can have similar architectures, such as a series of fully connected layers or convolutional layers, followed by a final output layer.

However, the main difference between DNNs and SNNs is the number of layers they have. While SNNs typically have only one or two layers, DNNs have many layers, allowing them to learn more complex and abstract representations of the input data. Additionally, DNNs require more computational resources and can be more difficult to train compared to SNNs, due to the larger number of parameters and potential issues such as vanishing gradients.

4.1.4

Deep neural networks have

- Long training times
- Small number of hidden layers
- Generally worse performance than MLP and are used for simple tasks

4.1.5

What is the primary difference between deep neural networks (DNNs) and shallow neural networks (SNNs)?

- DNNs have more layers, enabling them to learn more complex representations of input data.
- DNNs do not use activation functions, while SNNs do.
- SNNs cannot use backpropagation, while DNNs can.
- SNNs require more computational resources compared to DNNs.

📖 4.1.6

Misconceptions about deep learning

Deep learning has revolutionized many areas of artificial intelligence, demonstrating remarkable capabilities in fields such as image recognition, natural language processing, and more. However, misconceptions about its capabilities and applications often lead to unrealistic expectations or misuse. Addressing these misconceptions is crucial for effectively leveraging deep learning in real-world scenarios.

1. Deep learning can solve any problem: While deep learning has shown impressive results in many areas, it is not a panacea for all problems. It works well for problems with large amounts of labeled data, but it may not be suitable for smaller datasets or problems where data labeling is difficult.
2. Deep learning is a magic bullet: Deep learning requires significant expertise in data preparation, model architecture design, and hyperparameter tuning. It is not a magic bullet that can be easily applied to any problem without careful consideration and experimentation.
3. Deep learning models always outperform other methods: While deep learning models have shown state-of-the-art performance on many benchmarks, they are not always the best choice for a particular problem. In some cases, simpler models or other machine learning techniques may be more effective.
4. Deep learning requires massive amounts of data: While deep learning models generally perform better with more data, they can also be effective with smaller datasets or with techniques such as transfer learning or data augmentation.
5. Deep learning is only for computer science experts: While deep learning does require a certain level of technical expertise, there are many tools and libraries available that make it more accessible to researchers and practitioners without a background in computer science.

📝 4.1.7

Which of the following are common misconceptions about deep learning?

- Deep learning always requires massive amounts of data to work.
- Deep learning can solve any problem.
- Deep learning models always outperform simpler methods.
- Deep learning requires expertise in data preparation and model tuning.
- Tools like TensorFlow and PyTorch have made deep learning more accessible.

📖 4.1.8

Hyperparameters

In deep learning, hyperparameters are parameters that are set before training begins and are not learned during the training process. Unlike the weights and biases of the model, which are adjusted during training, hyperparameters remain fixed. They control the behavior and performance of the model, such as its architecture, optimization method, and training parameters. Examples of hyperparameters include learning rate, batch size, and number of epochs:

- Learning rate determines the step size for updating weights during training. A high learning rate might cause the model to overshoot the optimal point, while a low rate could slow down the learning process.
- Number of epochs refers to the number of complete passes through the entire training dataset. More epochs allow the model to learn better but may increase the risk of overfitting.
- Batch size specifies the number of samples processed together in a single iteration. Smaller batch sizes provide more updates but require more computational effort.

Hyperparameters are critical because they influence the training process's efficiency and accuracy. Incorrect settings can lead to suboptimal models or even training failures. For instance, setting an inappropriate learning rate can hinder convergence, while an unsuitable batch size might result in unstable training.

📝 4.1.9

Which of the following statements about hyperparameters in deep learning are true?

- Hyperparameters include learning rate, batch size, and number of epochs.
- Hyperparameters must be set before training begins.
- Hyperparameters are adjusted during training to improve model performance.
- Hyperparameters directly control the behavior of backpropagation.

📖 4.1.10

Hyperparameter tuning

Hyperparameter tuning involves finding the optimal values for hyperparameters to maximize the model's performance on a given task. This process is essential

because even slight variations in hyperparameters can significantly affect a model's behavior and outcomes. Common methods for tuning include **manual adjustments**, **grid search** (systematic exploration of all possible combinations), and **random search** (randomly sampling combinations).

Some automated approaches, such as **Bayesian optimization** and **genetic algorithms**, help optimize hyperparameters more efficiently by using intelligent sampling techniques. Additionally, techniques like **transfer learning** can reduce the need for extensive tuning by leveraging pre-trained models.

Once training starts, hyperparameters remain unchanged. The weights and biases of the model are updated during the training process, but hyperparameters such as learning rate and dropout rate stay constant. If a hyperparameter needs to be modified, the entire training process must be restarted, emphasizing the importance of careful selection.

Proper hyperparameter tuning improves training efficiency, reduces overfitting, and enhances the model's generalization to unseen data. For example, a well-tuned learning rate ensures quicker convergence, while the appropriate regularization parameters help prevent overfitting.

4.1.11

Which of the following are true about hyperparameter tuning?

- Transfer learning can reduce the need for extensive hyperparameter tuning.
- Grid search systematically explores possible hyperparameter combinations.
- Hyperparameter tuning is critical for optimizing model performance.
- Hyperparameter tuning adjusts weights and biases during training.

4.2 Tensors

4.2.1

Tensors are the building blocks of deep learning. They are multi-dimensional arrays that store numerical data in specific shapes, such as vectors, matrices, or higher-dimensional structures. These shapes enable tensors to represent complex data efficiently, making them a core component of artificial neural networks (ANNs). Tensors are used to represent inputs, outputs, intermediate computations, and learned parameters such as weights and biases during training.

Tensors play a crucial role in deep learning by enabling mathematical operations that transform input data into meaningful predictions. For instance, in image recognition,

tensors store pixel values, while in natural language processing, they represent word embeddings. Their flexibility allows them to handle diverse data types, such as images, audio, and text, which are often large and complex datasets.

Think of tensors as containers for numbers that can be manipulated mathematically. This manipulation enables deep learning models to perform tasks like recognizing patterns, classifying objects, and generating predictions. Tensors can hold numbers arranged in one or more dimensions, making them adaptable to a wide range of tasks and applications.

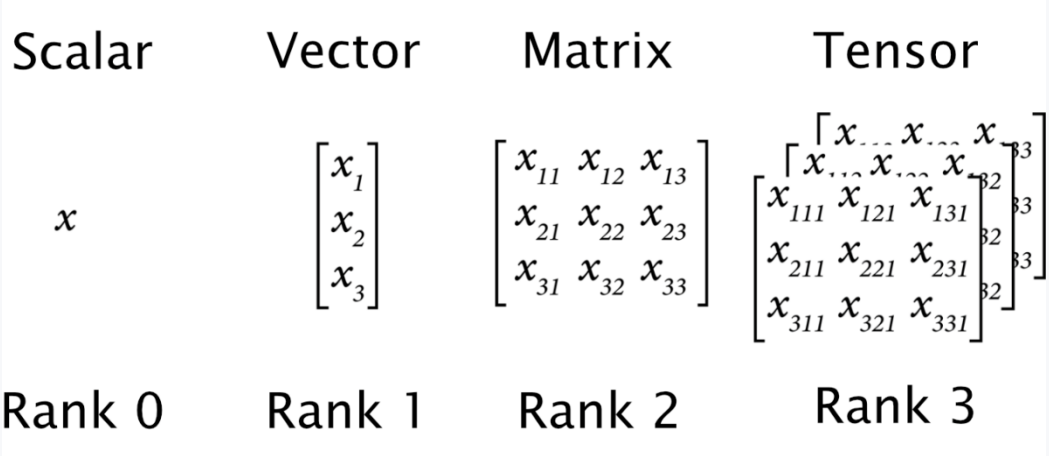
4.2.2

Which of the following statements about tensors are true?

- Tensors are multi-dimensional arrays used to store numerical data in deep learning.
- Tensors can handle data types such as images, audio, and text.
- Tensors are manipulated mathematically to perform operations in deep learning.
- Tensors represent only the input data in deep learning models.

4.2.3

Tensors are characterized by their shape, which determines their dimensions. The number of dimensions, also known as the tensor's rank, defines its type. Tensors can range from 0-dimensional scalars to higher-dimensional arrays used for complex tasks.



- **Scalar (0-D tensor)** consists of a single number, which makes it a zero-dimensional array. It is an example of a zero order tensor. Scalars have no axes. For example, the width of an object is scalar.

- **Vector (1-D tensor)** is one-dimensional arrays and are an example of the first order tensor. They can be considered lists of values. Vectors have an axis. The size of a given object by width, height and depth is an example of a vector field.
- **Matrix (2-D tensor)** is two-dimensional table with two axes. They are an example of second-order tensors. The matrix can be used to store the size of several objects. Each dimension of the matrix includes the size of each object (width, height, depth) and the other dimension of the matrix is used to distinguish between objects.
- **Tensor** are the general entities that contain scalars, vectors, and matrices, although the name is generally reserved for tensors of level 3 or higher. Tensors can be used to store the size and location of many objects over time. The first dimension of the matrix includes the size of each object (width, height, depth), the second dimension is used to distinguish the object, and the third dimension describes the position of these objects over time.

Although all these entities are considered tensors, the term "tensor" is often reserved for those with three or more dimensions.

4.2.4

Order types of tensors by dimensionality:

- Scalar
- Matrix
- Higher-dimensional tensor
- Vector

4.2.5

Tensors use

In deep learning, tensors serve as the universal data structure for inputs, outputs, and computations. For example, a tensor might hold pixel values for an image classification model or word embeddings in a natural language processing task. During training, tensors represent weights and biases that are updated iteratively to minimize error.

Tensors undergo a series of transformations in neural networks. For instance, input tensors pass through layers of neurons, where weights and biases adjust the data. These transformations are both linear and non-linear, enabling models to learn from data and make predictions.

Tensors are highly efficient for processing large and complex datasets. Their multi-dimensional nature allows for parallel computations, which are crucial for training large-scale models. Moreover, deep learning frameworks like **TensorFlow** and **PyTorch** provide powerful tools for tensor manipulation, making them accessible even for beginners.

4.2.6

Which of the following are examples of tensor use in deep learning?

- Representing pixel values in an image.
- Updating weights during backpropagation.
- Storing word embeddings in a language model.
- Manipulating strings in a database.

4.2.7

Project: Tensor definition

Tensors can be created using the **Variable** class present in the TensorFlow library and passing in a value representing the tensor.

```
import tensorflow as tf
tensor1 = tf.Variable([1,2,3], dtype=tf.int32,
name='my_tensor', trainable=True)
print(tensor1)
```

Program output:

```
```

- **dtype** - the datatype of the Variable object (for the tensor defined above, the datatype is tf.int32). The default value for this attribute is determined from the values passed.
- **shape** - the number of dimensions and length of each dimension of the Variable object (for the tensor defined above, the shape is [3]). The default value for this attribute is also determined from the values passed.
- **name** - the name of the Variable object (for the tensor defined above, the name of the tensor is defined as 'my_tensor'). The default for this attribute is Variable.

- **trainable** - this attribute indicates whether the Variable object can be updated during model training (for the tensor defined above, the trainable parameter is set to true). The default for this attribute is true.

```
# returns shape of the tensor
print(tensor1.shape)

# returns rank of the tensor
print(tf.rank(tensor1))
```

Program output:

```
(3,)
tf.Tensor(1, shape=(), dtype=int32)
```

This Python code snippet, leveraging the TensorFlow library, demonstrates the concept of tensor ranks and shapes in deep learning. Tensors are fundamental data structures in machine learning, representing multidimensional arrays of numbers.

The code begins by importing the TensorFlow library, a powerful tool for numerical computation, particularly in machine learning and deep learning. It then proceeds to create tensors of varying ranks: scalars, vectors, matrices, and higher-dimensional tensors.

For each tensor, the code calculates and prints its rank and shape. The rank of a tensor refers to the number of dimensions it possesses. The shape, on the other hand, specifies the size of each dimension.

```
import tensorflow as tf

# Scalar (rank 0)
int_variable = tf.Variable(4113, tf.int16)

# Vector (rank 1)
vector_variable = tf.Variable([0.23, 0.42, 0.35], tf.float32)

# Matrix (rank 2)
matrix_variable = tf.Variable([[4113, 7511, 6259], [3870,
6725, 6962]], tf.int32)

# Tensor (rank 3)
tensor_variable = tf.Variable([[[4113, 7511, 6259], [3870,
6725, 6962]],
```

```
                                [[5102, 7038, 6591], [3661,
5901, 6235]],
                                [[951, 1208, 1098], [870, 645,
948]]])

# Check rank and shape of each tensor
print("Rank of int_variable:", tf.rank(int_variable).numpy())
print("Shape of int_variable:", int_variable.shape.as_list())

print("Rank of vector_variable:",
tf.rank(vector_variable).numpy())
print("Shape of vector_variable:",
vector_variable.shape.as_list())

print("Rank of matrix_variable:",
tf.rank(matrix_variable).numpy())
print("Shape of matrix_variable:",
matrix_variable.shape.as_list())

print("Rank of tensor_variable:",
tf.rank(tensor_variable).numpy())
print("Shape of tensor_variable:",
tensor_variable.shape.as_list())
```

Program output:

```
Rank of int_variable: 0
Shape of int_variable: []
Rank of vector_variable: 1
Shape of vector_variable: [3]
Rank of matrix_variable: 2
Shape of matrix_variable: [2, 3]
Rank of tensor_variable: 3
Shape of tensor_variable: [3, 2, 3]
```

4.2.8

What is the rank of a scalar tensor?

- 0
- 1
- 2
- 3 or more

4.3 TensorFlow examples

4.3.1

Project: Iris flower classification with a simple neural network

This assignment involves building a simple neural network in TensorFlow to classify Iris flower species based on sepal and petal measurements. You will use the Iris dataset from scikit-learn and train a model to predict petal width given sepal length and sepal width.

This code demonstrates the construction and training of a basic neural network with TensorFlow to classify Iris flowers. The Iris dataset from scikit-learn provides data on sepal and petal dimensions of three flower species. The goal is to predict the petal width (represented by the fourth feature) based on the first three features (sepal length, sepal width, and petal length).

Our neural network will have five nodes in the hidden layer. We are feeding in three values: the sepal length (S.L.), the sepal width (S.W.), and the petal length (P.L.). The target will be the petal width. In total, there will be 26 total variables in the model.

```
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

Load Iris dataset. It contains data about different types of iris plant and their attributes.

```
import pandas as pd

iris = datasets.load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
print(df.head())

# Get Sepal length, Sepal width, Petal length
x_vals = np.array([x[0:3] for x in iris.data])
# Get Petal Width
y_vals = np.array([x[3] for x in iris.data])
```

Program output:

```
   sepal length (cm)  sepal width (cm)  petal length (cm)
petal width (cm)
0                5.1               3.5                1.4
0.2
1                4.9               3.0                1.4
0.2
2                4.7               3.2                1.3
0.2
3                4.6               3.1                1.5
0.2
4                5.0               3.6                1.4
0.2
```

Use predefined seed to make results reproducible - reproducibility is a crucial concept. It means that a given experiment or analysis can be repeated by another researcher and yield the same results. This is essential for validating findings, sharing knowledge, and building upon existing work.

```
# make results reproducible
seed = 3
np.random.seed(seed)
tf.random.set_seed(seed)
```

```
# Split data into training and testing sets (80/20 split)
train_indices = np.random.choice(len(x_vals),
round(len(x_vals) * 0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) -
set(train_indices)))
x_vals_train, x_vals_test = x_vals[train_indices],
x_vals[test_indices]
y_vals_train, y_vals_test = y_vals[train_indices],
y_vals[test_indices]
```

Min-max normalization is a technique used to scale numerical features to a specific range, typically between 0 and 1. This is essential in machine learning because features with different scales can have a significant impact on the model's performance.

The normalize_cols function implements min-max normalization for each column (feature) in the input matrix m. It calculates the minimum and maximum values for each column and then rescales the values using the formula above. The np.nan_to_num function is used to handle potential NaN values that might arise during the normalization process.

```
# Normalize features (min-max normalization)
def normalize_cols(m):
  col_max = m.max(axis=0)
  col_min = m.min(axis=0)
  return (m-col_min) / (col_max - col_min)

x_vals_train = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test))
```

Batch size refers to the number of training examples used in one iteration of the optimization process. It's a crucial hyperparameter that can significantly impact the training process and model performance.

The batch_size is set to 50. This means that during each training iteration, the model will be trained on a batch of 50 randomly selected training examples. The model's parameters will be updated based on the gradients computed from this batch. By using a batch size of 50, the model can balance the trade-off between computational efficiency and generalization performance.

```
# Declare batch size
batch_size = 50

# Initialize input data
x_data = tf.keras.Input(dtype=tf.float32, shape=(3,))
```

Declare the network

This code defines the architecture of a simple neural network with one hidden layer to predict the petal width of an Iris flower based on its sepal length, sepal width, and petal length.

Variable initialization:

- a1 and a2 - weight matrices for the hidden and output layers, respectively.
- b1 and b2 - bias vectors for the hidden and output layers.

The weights and biases are initialized randomly using a normal distribution with a specified seed for reproducibility.

Model architecture:

- Hidden Layer - takes the input features x_data as input, applies a linear transformation (tf.matmul) using the weights a1 and biases b1., applies the ReLU activation function to introduce non-linearity.
- Output layer - takes the output of the hidden layer as input and applies a linear transformation using the weights a2 and bias b2.

Model compilation

- tf.keras.Model class is used to define the model, specifying the input (x_data) and output (output).
- model.summary() method provides a concise overview of the model's architecture, including the number of parameters in each layer.

Optimizer:

- stochastic Gradient Descent (SGD) is chosen as the optimizer to update the model's weights and biases during training.
- a learning rate of 0.005 is set to control the step size of the gradient descent updates.

```
# Create variables for both NN layers
hidden_layer_nodes = 5
a1 = tf.Variable(tf.random.normal(shape=[3,
hidden_layer_nodes], seed=seed))  # Weights for hidden layer
b1 = tf.Variable(tf.random.normal(shape=[hidden_layer_nodes],
seed=seed))  # Biases for hidden layer
a2 = tf.Variable(tf.random.normal(shape=[hidden_layer_nodes,
1], seed=seed))  # Weights for output layer
b2 = tf.Variable(tf.random.normal(shape=[1], seed=seed))  #
Bias for output layer

# Define hidden layer operation with ReLU activation
hidden_output = tf.keras.layers.Lambda(lambda x:
tf.nn.relu(tf.matmul(x, a1) + b1))

# Define output layer operation with linear activation (for
regression)
output = tf.matmul(hidden_layer, a2) + b2

# Build the model
```

```
model = tf.keras.Model(inputs=x_data, outputs=output,
name="1layer_neural_network")

# Print model summary
model.summary()

# Declare optimizer
optimizer = tf.keras.optimizers.SGD(0.005)
```

Program output:

```
WARNING:tensorflow:The following Variables were used in a
Lambda layer's call (tf.linalg.matmul), but are not present in
its tracked objects:   . This is a strong indication that the
Lambda layer should be rewritten as a subclassed Layer.
WARNING:tensorflow:The following Variables were used in a
Lambda layer's call (tf.__operators__.add), but are not
present in its tracked objects:   . This is a strong
indication that the Lambda layer should be rewritten as a
subclassed Layer.
Model: "1layer_neural_network"
_________________________________________________________________
___
 Layer (type)                Output Shape              Param #
=================================================================
===
 input_1 (InputLayer)        [(None, 3)]               0

 lambda (Lambda)             (None, 5)                 0

 tf.linalg.matmul (TFOpLambd  (None, 1)                0
 a)

 tf.__operators__.add (TFOpL  (None, 1)                0
 ambda)
```

Following code implements a training loop for a simple neural network. The goal is to minimize the Mean Squared Error (MSE) between the network's predictions and the true values of the target variable (petal width in this case).

The training process involves iteratively updating the model's parameters (weights and biases) using gradient descent. This is done by calculating the gradients of the loss function with respect to the parameters and then adjusting the parameters in the direction that minimizes the loss.

- Batching - a random subset of training data is selected in each iteration (batch). This helps to introduce randomness and improve generalization.
- Forward Pass - the current batch of input data is fed into the neural network to obtain predictions.
- Loss Calculation - the Mean squared error between the predicted values and the true target values is calculated.
- Gradient Calculation tf.GradientTape context manager is used to record operations and compute gradients. The gradients of the loss with respect to the model's parameters are calculated.
- Parameter Update - the optimizer (SGD in this case) updates the model's parameters using the calculated gradients.
- Evaluation - the model is evaluated on the test set to monitor its performance during training. The test loss is calculated and stored for later analysis.
- Logging - the training loss is printed every 50 iterations to track progress.

```
# Training loop
loss_vec = []
test_loss = []
for i in range(500):
  rand_index = np.random.choice(len(x_vals_train),
size=batch_size)
  rand_x = x_vals_train[rand_index]
  rand_y = np.transpose([y_vals_train[rand_index]])

  # Open a GradientTape.
  with tf.GradientTape(persistent=True) as tape:
    # Forward pass.
    output = model(rand_x)

    # Apply loss function (MSE)
    loss = tf.reduce_mean(tf.square(rand_y - output))
    loss_vec.append(np.sqrt(loss))

    # Get gradients of loss with reference to the variables
"a1", "b1", "a2" and "b2" to adjust.
    gradients_a1 = tape.gradient(loss, a1)
    gradients_b1 = tape.gradient(loss, b1)
    gradients_a2 = tape.gradient(loss, a2)
    gradients_b2 = tape.gradient(loss, b2)

    # Update the variables "a1", "b1", "a2" and "b2" of the
model.
    optimizer.apply_gradients(zip([gradients_a1, gradients_b1,
gradients_a2, gradients_b2], [a1, b1, a2, b2]))
```

```
    # Forward pass.
    output_test = model(x_vals_test)
    # Apply loss function (MSE) on test
    loss_test =
tf.reduce_mean(tf.square(np.transpose([y_vals_test]) -
output_test))
    test_loss.append(np.sqrt(loss_test))

    if (i+1)%50==0:
      print('Generation: ' + str(i+1) + '. Loss = ' +
str(np.mean(loss)))
```

Program output:

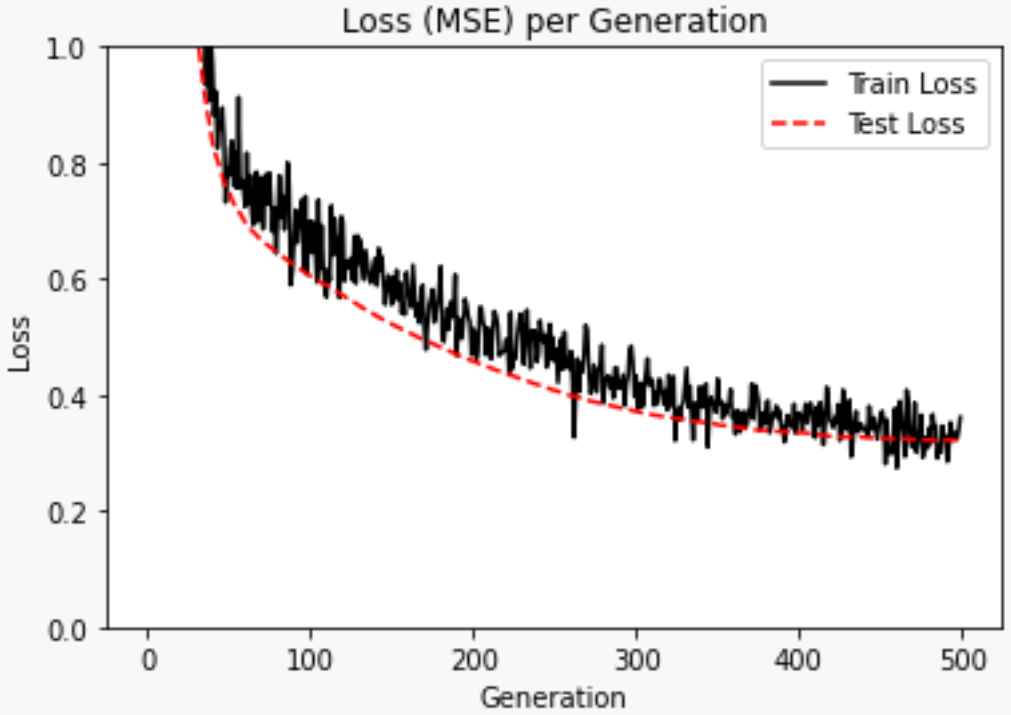
```
WARNING:tensorflow:Calling GradientTape.gradient on a
persistent tape inside its context is significantly less
efficient than calling it outside the context (it causes the
gradient ops to be recorded on the tape, leading to increased
CPU and memory usage). Only call GradientTape.gradient inside
the context if you actually want to trace the gradient in
order to compute higher order derivatives.
WARNING:tensorflow:Calling GradientTape.gradient on a
persistent tape inside its context is significantly less
efficient than calling it outside the context (it causes the
gradient ops to be recorded on the tape, leading to increased
CPU and memory usage). Only call GradientTape.gradient inside
the context if you actually want to trace the gradient in
order to compute higher order derivatives.
WARNING:tensorflow:Calling GradientTape.gradient on a
persistent tape inside its context is significantly less
efficient than calling it outside the context (it causes the
gradient ops to be recorded on the tape, leading to increased
CPU and memory usage). Only call GradientTape.gradient inside
the context if you actually want to trace the gradient in
order to compute higher order derivatives.
WARNING:tensorflow:Calling GradientTape.gradient on a
persistent tape inside its context is significantly less
efficient than calling it outside the context (it causes the
gradient ops to be recorded on the tape, leading to increased
CPU and memory usage). Only call GradientTape.gradient inside
the context if you actually want to trace the gradient in
order to compute higher order derivatives.
Generation: 50. Loss = 0.6266393
Generation: 100. Loss = 0.46858525
Generation: 150. Loss = 0.34825706
```

```
Generation: 200. Loss = 0.219747
Generation: 250. Loss = 0.24899033
Generation: 300. Loss = 0.15766421
Generation: 350. Loss = 0.13695493
Generation: 400. Loss = 0.13414612
Generation: 450. Loss = 0.11820017
Generation: 500. Loss = 0.13063623
```

Plot the result of training

```
# Plot loss (MSE) over time
plt.ylim([0, 1.0])
plt.plot(loss_vec, 'k-', label='Train Loss')
plt.plot(test_loss, 'r--', label='Test Loss')
plt.title('Loss (MSE) per Generation')
plt.legend(loc='upper right')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

Program output:



4.3.2

Project: More complex network with 3 hidden layers

(by https://github.com/nfmcclure/tensorflow_cookbook)

Implement a neural network model to predict birthweight based on various factors extracted from a dataset.

Dataset:

- original: https://github.com/nfmcclure/tensorflow_cookbook/raw/master/01_Introduction/07_Working_with_Data_Sources/birthweight_data/birthweight.dat
- local: https://priscilla.fitped.eu/data/deep_learning/birthweight.dat

The 'Low Birthrate Dataset' is a dataset from a famous study by Hosmer and Lemeshow in 1989 called, "Low Infant Birth Weight Risk Factor Study". This example is predicting birth weights in a low birth weight database. We will create a neural network with three hidden layers. The low birth weight data set includes actual birth weights and a variable indicating whether the given birth weight is over or below 2,500 grams. In this example, we will make the target the actual birth weight (regression) and then see what is the accuracy of the classification at the end. Finally, our model should be able to identify whether the birth weight is 2500 grams.

```
import warnings
warnings.filterwarnings("ignore")
import tensorflow as tf
import matplotlib.pyplot as plt
import csv
import random
import numpy as np
import requests
```

1. Data acquisition and preprocessing

- The code downloads the birth weight data from a URL and saves it locally.
- It reads the data into a list of lists, extracting features of interest (e.g., age, weight, smoking habits).
- The target variable (birth weight) is separated from the features.

```
# Data file
birth_weight_file = 'birth_weight.csv'
```

```
# download data and create data file
birthdata_url =
'https://github.com/nfmcclure/tensorflow_cookbook/raw/master/0
1_Introduction/07_Working_with_Data_Sources/birthweight_data/b
irthweight.dat'
birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\r\n')
birth_header = birth_data[0].split('\t')
birth_data = [[float(x) for x in y.split('\t') if len(x)>=1]
for y in birth_data[1:] if len(y)>=1]
with open(birth_weight_file, "w") as f:
  writer = csv.writer(f)
  writer.writerows([birth_header])
  writer.writerows(birth_data)
  f.close()

# read birth weight data into memory
birth_data = []
with open(birth_weight_file, newline='') as csvfile:
  csv_reader = csv.reader(csvfile)
  birth_header = next(csv_reader)
  for row in csv_reader:
    birth_data.append(row)

birth_data = [[float(x) for x in row] for row in birth_data]

# Extract y-target (birth weight)
y_vals = np.array([x[8] for x in birth_data])

# Filter for features of interest
cols_of_interest = ['AGE', 'LWT', 'RACE', 'SMOKE', 'PTL',
'HT', 'UI']
x_vals = np.array([[x[ix] for ix, feature in
enumerate(birth_header) if feature in cols_of_interest] for x
in birth_data])
```

- Data is split into training and testing sets using an 80/20 split.

```
# set batch size for training
batch_size = 150

# make results reproducible
seed = 3
```

```
np.random.seed(seed)
tf.random.set_seed(seed)

# Split data into train/test = 80%/20%
train_indices = np.random.choice(len(x_vals),
round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) -
set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

- Features are normalized using min-max scaling to ensure values are between 0 and 1.

```
# Record training column max and min for scaling of non-
training data
train_max = np.max(x_vals_train, axis=0)
train_min = np.min(x_vals_train, axis=0)

# Normalize by column (min-max norm to be between 0 and 1)
def normalize_cols(mat, max_vals, min_vals):
  return (mat - min_vals) / (max_vals - min_vals)

x_vals_train = np.nan_to_num(normalize_cols(x_vals_train,
train_max, train_min))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test,
train_max, train_min))
```

2. Model definition

- Functions **init_weight** and **init_bias** are used to initialize weight and bias variables with appropriate shapes and standard deviations.
- **tf.keras.Input** layer is defined to specify the input shape (number of features).

```
# Define Variable Functions (weights and bias)
def init_weight(shape, st_dev):
  weight = tf.Variable(tf.random.normal(shape, stddev=st_dev))
  return(weight)

def init_bias(shape, st_dev):
  bias = tf.Variable(tf.random.normal(shape, stddev=st_dev))
```

```
  return(bias)

# Initialize input data
x_data = tf.keras.Input(dtype=tf.float32, shape=(7,))
```

- Three fully connected (dense) hidden layers are created with ReLU activation functions. Each layer uses custom-defined **fully_connected** functions with weight and bias variables. The number of hidden nodes in each layer can be adjusted for experimentation.
- A final fully connected layer with one output neuron predicts the birthweight.

```
# Create a fully connected layer:
def fully_connected(input_layer, weights, biases):
  return tf.keras.layers.Lambda(lambda x:
tf.nn.relu(tf.add(tf.matmul(x, weights),
biases)))(input_layer)

#--------Create the first layer (25 hidden nodes)--------
weight_1 = init_weight(shape=[7,25], st_dev=5.0)
bias_1 = init_bias(shape=[25], st_dev=10.0)
layer_1 = fully_connected(x_data, weight_1, bias_1)

#--------Create second layer (10 hidden nodes)--------
weight_2 = init_weight(shape=[25, 10], st_dev=5.0)
bias_2 = init_bias(shape=[10], st_dev=10.0)
layer_2 = fully_connected(layer_1, weight_2, bias_2)

#--------Create third layer (3 hidden nodes)--------
weight_3 = init_weight(shape=[10, 3], st_dev=5.0)
bias_3 = init_bias(shape=[3], st_dev=10.0)
layer_3 = fully_connected(layer_2, weight_3, bias_3)

#--------Create output layer (1 output value)--------
weight_4 = init_weight(shape=[3, 1], st_dev=5.0)
bias_4 = init_bias(shape=[1], st_dev=10.0)
final_output = fully_connected(layer_3, weight_4, bias_4)

# Build the model
model = tf.keras.Model(inputs=x_data, outputs=final_output,
name="multiple_layers_neural_network")

# Print model summary
model.summary()
```

Program output:

```
WARNING:tensorflow:
The following Variables were used a Lambda layer's call
(lambda), but
are not present in its tracked objects:


It is possible that this is intended behavior, but it is more
likely
an omission. This is a strong indication that this layer
should be
formulated as a subclassed Layer rather than a Lambda layer.
WARNING:tensorflow:
The following Variables were used a Lambda layer's call
(lambda_1), but
are not present in its tracked objects:


It is possible that this is intended behavior, but it is more
likely
an omission. This is a strong indication that this layer
should be
formulated as a subclassed Layer rather than a Lambda layer.
WARNING:tensorflow:
The following Variables were used a Lambda layer's call
(lambda_2), but
are not present in its tracked objects:


It is possible that this is intended behavior, but it is more
likely
an omission. This is a strong indication that this layer
should be
formulated as a subclassed Layer rather than a Lambda layer.
WARNING:tensorflow:
The following Variables were used a Lambda layer's call
(lambda_3), but
are not present in its tracked objects:


It is possible that this is intended behavior, but it is more
likely
an omission. This is a strong indication that this layer
should be
```

```
formulated as a subclassed Layer rather than a Lambda layer.
Model: "multiple_layers_neural_network"
_________________________________________________________________
___
 Layer (type)                Output Shape              Param #
=================================================================
===
 input_1 (InputLayer)        [(None, 7)]               0

 lambda (Lambda)             (None, 25)                0

 lambda_1 (Lambda)           (None, 10)                0

 lambda_2 (Lambda)           (None, 3)                 0

 lambda_3 (Lambda)           (None, 1)                 0

=================================================================
===
Total params: 0
```

3. Model training

- The Adam optimizer is used with a learning rate of 0.025 to adjust the model's weights and biases during training.
- A training loop iterates for 200 epochs:

1. A mini-batch of data is randomly selected from the training set.
2. The model's forward pass is performed to obtain predictions for the mini-batch.
3. Mean absolute error is used as the loss function to measure the difference between predictions and actual birth weights.
4. The GradientTape context is used to efficiently calculate gradients of the loss function with respect to the model's weights and biases.
5. The gradients are applied to update the weights and biases using the Adam optimizer, improving the model's ability to predict birthweight.

- After each epoch, the model's performance is evaluated on the testing set by calculating the MAE loss and printing it every 25 epochs.

```
# Declare Adam optimizer
optimizer = tf.keras.optimizers.Adam(0.025)

# Training loop
```

```
loss_vec = []
test_loss = []
for i in range(200):
  rand_index = np.random.choice(len(x_vals_train),
size=batch_size)
  rand_x = x_vals_train[rand_index]
  rand_y = np.transpose([y_vals_train[rand_index]])

  # Open a GradientTape.
  with tf.GradientTape(persistent=True) as tape:
    # Forward pass.
    output = model(rand_x)

    # Apply loss function (MSE)
    loss = tf.reduce_mean(tf.abs(rand_y - output))
    loss_vec.append(loss)

  # Get gradients of loss with reference to the weights and
bias variables to adjust.
  gradients_w1 = tape.gradient(loss, weight_1)
  gradients_b1 = tape.gradient(loss, bias_1)
  gradients_w2 = tape.gradient(loss, weight_2)
  gradients_b2 = tape.gradient(loss, bias_2)
  gradients_w3 = tape.gradient(loss, weight_3)
  gradients_b3 = tape.gradient(loss, bias_3)
  gradients_w4 = tape.gradient(loss, weight_4)
  gradients_b4 = tape.gradient(loss, bias_4)

  # Update the weights and bias variables of the model.
  optimizer.apply_gradients(zip([gradients_w1, gradients_b1,
gradients_w2, gradients_b2, gradients_w3, gradients_b3,
gradients_w4, gradients_b4], [weight_1, bias_1, weight_2,
bias_2, weight_3, bias_3, weight_4, bias_4]))

  # Forward pass.
  output_test = model(x_vals_test)
  # Apply loss function (MSE) on test
  temp_loss =
tf.reduce_mean(tf.abs(np.transpose([y_vals_test]) -
output_test))
  test_loss.append(temp_loss)

  if (i+1) % 25 == 0:
```

```
    print('Generation: ' + str(i+1) + '. Loss = ' +
str(loss.numpy()))
```

Program output:

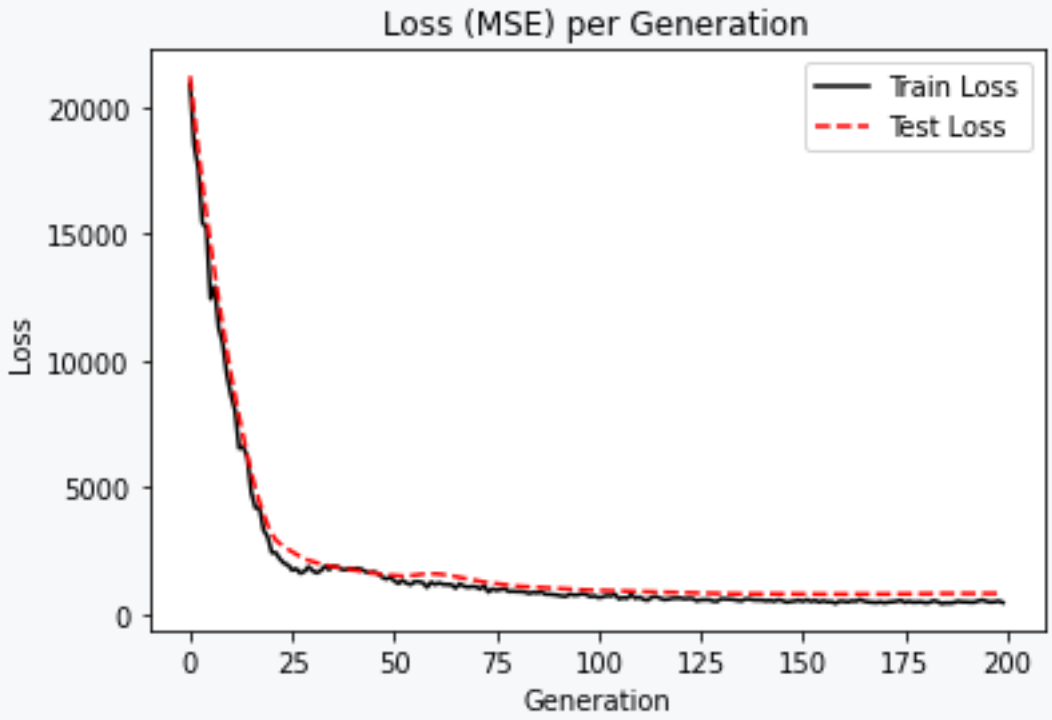
```
Generation: 25. Loss = 1921.8654
Generation: 50. Loss = 1452.7341
Generation: 75. Loss = 987.58563
Generation: 100. Loss = 709.25836
Generation: 125. Loss = 509.8613
Generation: 150. Loss = 540.57904
Generation: 175. Loss = 535.96893
Generation: 200. Loss = 439.15442
```

4. Evaluation and prediction

- After training, the code calculates the model's accuracy on the training and testing sets. Accuracy here refers to correctly classifying whether a birthweight is below a certain threshold (e.g., 2500 grams in this case).
- Finally, the code demonstrates how to make predictions for new data points by performing normalization and feeding them through the trained model.

```
# Plot loss (MSE) over time
plt.plot(loss_vec, 'k-', label='Train Loss')
plt.plot(test_loss, 'r--', label='Test Loss')
plt.title('Loss (MSE) per Generation')
plt.legend(loc='upper right')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

Program output:




```
# Model Accuracy
actuals = np.array([x[0] for x in birth_data])
test_actuals = actuals[test_indices]
train_actuals = actuals[train_indices]
test_preds = model(x_vals_test)
train_preds = model(x_vals_train)
test_preds = np.array([1.0 if x < 2500.0 else 0.0 for x in
test_preds])
train_preds = np.array([1.0 if x < 2500.0 else 0.0 for x in
train_preds])
# Print out accuracies
test_acc = np.mean([x == y for x, y in zip(test_preds,
test_actuals)])
train_acc = np.mean([x == y for x, y in zip(train_preds,
train_actuals)])
print('On predicting the category of low birthweight from
regression output (<2500g):')
print('Test Accuracy: {}'.format(test_acc))
print('Train Accuracy: {}'.format(train_acc))
```

Program output:

```
On predicting the category of low birthweight from regression
output (<2500g):
Test Accuracy: 0.7631578947368421
Train Accuracy: 0.7748344370860927
```

- Example of prediction for new data

```
# Need new vectors of 'AGE', 'LWT', 'RACE', 'SMOKE', 'PTL',
'HT', 'UI'
new_data = np.array([[35, 185, 1., 0., 0., 0., 1.],
                     [18, 160, 0., 1., 0., 0., 1.]])
new_data_scaled = np.nan_to_num(normalize_cols(new_data,
train_max, train_min))
new_logits = model(new_data_scaled)
new_preds = np.array([1.0 if x < 2500.0 else 0.0 for x in
new_logits])

print('New Data Predictions: {}'.format(new_preds))
```

Program output:

```
New Data Predictions: [1. 0.]
```

Convolutional Neural Networks - CNNs

Chapter 5

5.1 CNN description

📖 5.1.1

Basic description of CNN

A Convolutional Neural Network (**CNN**) is a type of deep neural network commonly used in image and video recognition tasks.

The key feature of a CNN is its ability to learn **hierarchical representations of input data** through a series of **convolutional layers**. These layers apply a set of learnable filters to the input data, extracting local features such as edges and textures. The output of each convolutional layer is then passed through a non-linear activation function to introduce non-linearity and create more complex features.

After several convolutional layers, the output is passed through a pooling layer which reduces the spatial resolution of the feature maps while retaining the most important features. Finally, the output of the last pooling layer is passed through one or more fully connected layers to produce a final output, typically a probability distribution over the possible classes.

CNNs have been shown to be highly effective in a wide range of image recognition tasks, including object detection, image segmentation, and facial recognition. They have also been applied to other types of data such as audio and natural language processing.

📖 5.1.2

What is convolution

In a Convolutional Neural Network (CNN), convolution refers to the process of applying a set of filters to the input data in order to **extract local features**.

In the context of image processing, the input data is typically a 3D array representing an image, with dimensions for **width**, **height**, and **color channels**. The filters, also known as kernels or feature detectors, are smaller 3D arrays that slide over the input data, computing a dot product between the filter and the input at each location, and producing an output in the form of a 2D activation map.

The filters are learned through backpropagation during training, and each filter is optimized to detect a particular feature of the input data, such as edges or corners. Multiple filters are used in each convolutional layer, and the output of each filter is

combined to produce a set of activation maps, which are then passed through a non-linear activation function such as ReLU.

Convolutional layers are typically followed by pooling layers, which reduce the spatial resolution of the activation maps while retaining the most important features, and then by additional convolutional layers to extract higher-level features.

The use of convolutional layers in CNNs has been shown to be highly effective in image recognition tasks, and has also been applied to other types of data such as audio and natural language processing.

5.2 Layers and architectures I.

🕮 5.2.1

Convolutional Neural Networks (CNNs) are a type of deep learning architecture widely used for tasks involving image data, such as object recognition, image classification, and segmentation. The power of CNNs comes from their ability to automatically learn and extract hierarchical features from input data. This is achieved through a combination of different types of layers, each performing a specific role in the network.

Layers in a CNN work together to process and transform the input data step by step. For example, an image of a cat is passed through a CNN, where initial layers might detect edges and textures, and later layers identify more complex patterns like eyes, whiskers, or the overall shape of a cat.

The choice of layers in a CNN depends on the task and the structure of the input data. Each layer type contributes uniquely to how the data is processed, helping the model to extract meaningful patterns and make accurate predictions. Let's explore the commonly used layers in CNNs.

Layer types

There are several types of layers commonly used in Convolutional Neural Networks (CNNs), including:

- **Convolutional layers** are the core building blocks of CNNs. They use filters (or kernels) to scan the input data and extract local features such as edges, textures, and patterns. Each filter focuses on a specific aspect of the data, enabling the network to learn spatial relationships.
- **Pooling layers** reduce the spatial dimensions of the feature maps, making the network more efficient and reducing the risk of overfitting. Common pooling methods include max pooling, which selects the maximum value from a region, and average pooling, which computes the average value.

- **Fully connected layers** found in the final stages of the network, these layers connect every neuron to all neurons in the preceding layer. They are typically used for high-level tasks like classification or regression.
- **Activation layers** introduce non-linearity to the model, allowing it to learn complex patterns. Activation functions like ReLU, sigmoid, and tanh are commonly used to transform the output of previous layers.
- **Normalization layers** - by normalizing the outputs of the previous layer, normalization layers improve the stability and performance of the network. Batch normalization is a popular technique that reduces internal covariate shifts.
- **Dropout layers** prevent overfitting by randomly "dropping out" neurons during training. This forces the network to learn robust features that generalize well to new data.
- **Upsampling layers** increase the spatial resolution of feature maps, often used in tasks like image generation or segmentation. They achieve this by repeating values or learning new values through transposed convolutions.

The specific architecture of a CNN will depend on the particular task and the structure of the input data, but most CNNs will include some combination of these layers.

5.2.2

Which of the following statements about CNN layers are correct?

- Convolutional layers are used to extract features like edges and textures.
- Fully connected layers are used for classification tasks in CNNs.
- Pooling layers increase the spatial resolution of feature maps.
- Dropout layers help reduce overfitting during training.

5.2.3

Convolution layer

The convolutional layer is a fundamental building block of a Convolutional Neural Network (CNN). It applies a set of filters to the input image to extract features, by performing a convolution operation between the input image and a set of learnable filters. Each filter slides over the entire input image, computing a dot product between the filter weights and the pixel values at each position. The result of this operation is a feature map that highlights the presence of certain features in the input image.

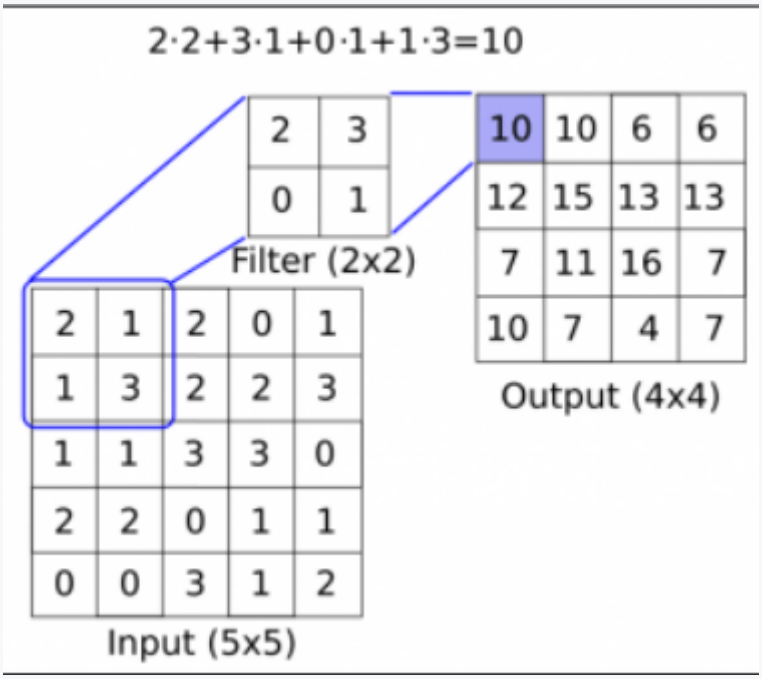
The learnable filters in the convolutional layer represent different characteristics of the input image, such as edges, textures, or colors. By stacking multiple

convolutional layers on top of each other, the CNN can learn increasingly complex and abstract features from the input image.

Each convolutional layer typically has a number of hyperparameters, such as the number of filters, the size of the filters (kernel size), and the stride (the amount the filter shifts between each computation). The size of the output feature map is determined by the size of the input image, the size of the filter, and the stride, with smaller strides resulting in larger output feature maps.

Convolutional layers are important in CNNs because they enable the network to extract useful features from the input image in a hierarchical manner, allowing it to identify complex patterns and structures that are relevant to the task at hand. They are widely used in computer vision tasks such as image classification, object detection, and semantic segmentation.

Application of a 2x2 convolutional filter across a 5x5 input matrix producing a new 4x4 feature layer



5.2.4

What is the primary purpose of the convolutional layer in a Convolutional Neural Network?

- To apply a set of filters for feature extraction from the input image
- To reduce the spatial resolution of the input image
- To randomly drop neurons during training

- To connect every neuron in one layer to every neuron in the previous layer

📖 5.2.5

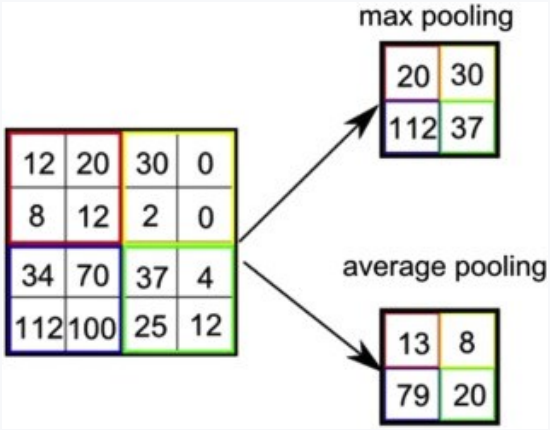
Pooling layer

The pooling layer is a type of layer in a Convolutional Neural Network (CNN) that performs a downsampling operation on the input feature map. The pooling layer reduces the spatial dimensions (width and height) of the input feature map while preserving the depth dimension (number of channels) by combining the outputs of adjacent neurons in the feature map.

The most commonly used type of pooling layer is max pooling, where the maximum value in each local region of the feature map is taken as the output. For example, a max pooling layer with a 2x2 kernel and stride of 2 would divide the input feature map into non-overlapping 2x2 regions and take the maximum value in each region as the output. This operation reduces the spatial dimensions of the feature map by a factor of two.

Another type of pooling layer is average pooling, where the average value in each local region of the feature map is taken as the output. Average pooling can also be used to reduce the spatial dimensions of the feature map.

The pooling layer is used in CNNs to reduce the spatial dimensions of the feature map, which can help to reduce the computational cost of the network and prevent overfitting by reducing the number of parameters in the model. Additionally, the pooling layer can help to extract invariant features from the input by taking the maximum or average value in each local region, which can improve the robustness of the model to variations in the input data.



5.2.6

What is the primary purpose of the pooling layer in a Convolutional Neural Network

- To reduce the spatial dimensions of the feature map while preserving the depth dimension
- To increase the spatial resolution of the feature map
- To apply filters for feature extraction from the input image
- To introduce non-linearity to the output of the previous layer

5.2.7

Fully connected layer

A fully connected layer, also called a dense layer, is a type of layer in a neural network where every neuron in the layer is connected to every neuron in the previous layer.

In a fully connected layer, the input is a vector and the output is another vector of a specified size, which represents the activations of the layer. Each neuron in the fully connected layer applies a weighted sum of the activations from the previous layer, followed by a non-linear activation function, to produce its output.

Fully connected layers are commonly used in the final layers of a neural network, where they can be used for **classification** or **regression** tasks. They are also used in certain types of networks such as Multi-Layer Perceptrons (MLPs), where all layers are fully connected. However, in some types of networks such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs), fully connected layers are used only in the final layers of the network, after convolutional or recurrent layers have been used to extract features from the input data.

5.2.8

In a fully connected layer, what is the relationship between neurons in the current layer and the previous layer?

- Every neuron in the layer is connected to every neuron in the previous layer
- Each neuron is connected to a few neurons in the previous layer
- Neurons are not connected to neurons in the previous layer
- Each neuron connects to neurons in the next layer only

📖 5.2.9

Activation layer

An activation layer, also known as an activation function or nonlinearity, is a type of layer in a neural network that introduces nonlinearity into the network's output.

The purpose of an activation layer is to apply a mathematical function to the output of the previous layer in order to introduce nonlinearity. Without an activation layer, the neural network would simply be a linear function, which would not be able to model complex relationships between the input and output data.

There are several different types of activation functions used in deep learning, including:

- Sigmoid function: maps the input to a value between 0 and 1, and is commonly used in binary classification problems.

$$g(z) = \frac{1}{1 + e^{-z}}$$

- Rectified Linear Unit (ReLU): sets all negative values in the input to zero, and is commonly used in image classification problems.

$$g(z) = max(0, z)$$

- Leaky ReLU

$$g(z) = max(\epsilon z, z); z \ll 1$$

- Hyperbolic tangent (tanh): maps the input to a value between -1 and 1, and is commonly used in recurrent neural networks.

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Activation layers are typically placed after each convolutional or dense layer in a neural network, except for the output layer, which often uses a different activation function depending on the problem being solved.

5.2.10

What is the primary role of an activation layer in a neural network?

- To introduce nonlinearity to the network's output
- To perform a linear transformation of the input
- To reduce the number of layers in the network
- To increase the number of neurons in the network

5.3 Layers and architectures II.

5.3.1

Normalization layer

A normalization layer, also known as a **batch normalization layer**, is a type of layer in a neural network that is used to normalize the input data before passing it to the next layer. The purpose of normalization is to ensure that the input data has a mean of 0 and a standard deviation of 1, which can improve the performance and stability of the network.

The normalization process involves subtracting the mean of the input data from each data point, and then dividing the result by the standard deviation of the input data. This makes the input data have a zero mean and a standard deviation of 1, which can help prevent the input from causing the activation functions to saturate, which can cause the network to stop learning.

Normalization layers are commonly used in deep learning architectures, particularly in convolutional neural networks (CNNs) and recurrent neural networks (RNNs). They are typically placed after the convolutional or recurrent layers, but before the activation function.

5.3.2

What is the main purpose of a normalization layer in a neural network?

- To ensure the input data has a mean of 0 and a standard deviation of 1
- To improve the learning rate by increasing the variance of the input data
- To add additional layers to the network
- To reduce the number of training epochs

📖 5.3.3

Dropout layer

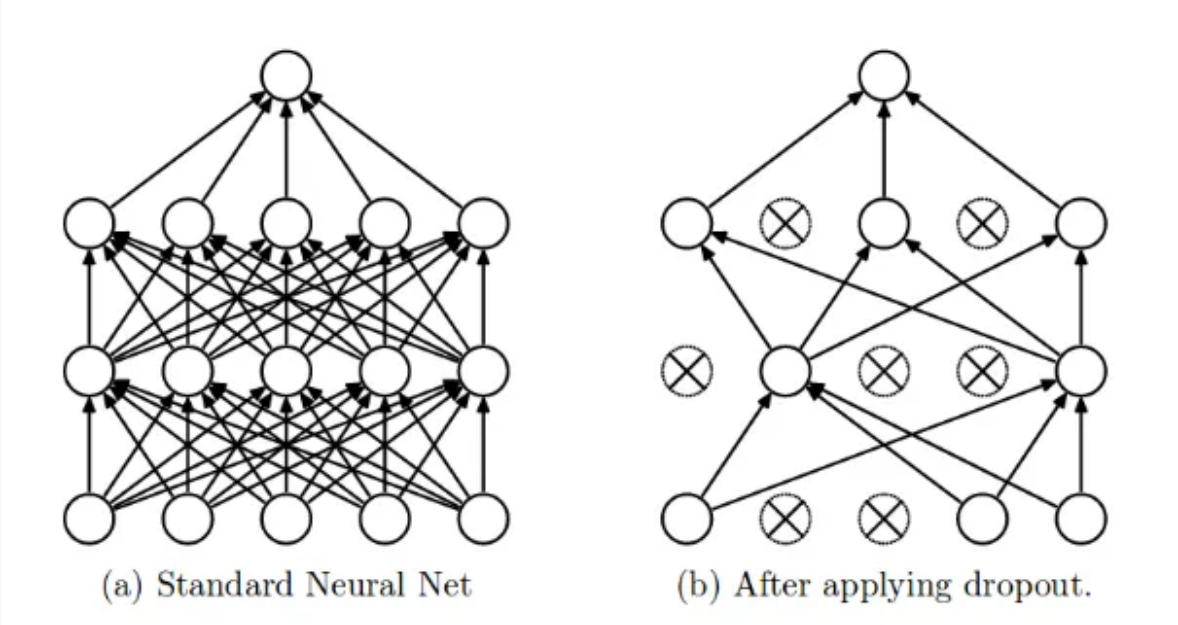
Dropout is a regularization technique used in deep neural networks to prevent overfitting. A dropout layer is a type of layer in a neural network that randomly drops out, or "turns off," a certain percentage of the neurons during training. The neurons that are dropped out change with each training iteration, which makes the network more robust and less likely to overfit to the training data.

The purpose of the dropout layer is to prevent the network from relying too heavily on any one feature or neuron, and to encourage the network to learn more robust features that are useful across multiple inputs. Dropout can also help prevent the network from memorizing noise or outliers in the training data.

During training, a dropout layer randomly selects a percentage of the neurons to drop out, based on a specified dropout rate. The remaining neurons are then scaled by a factor equal to 1 / (1 - dropout rate), in order to compensate for the dropped out neurons. During testing, all of the neurons are used, and their output is scaled by the same factor as during training.

Dropout layers are commonly used in deep learning architectures, particularly in convolutional neural networks (CNNs) and fully connected neural networks. They are typically placed after each dense layer in the network.

Example of dropout



5.3.4

What is the purpose of a dropout layer in a neural network?

- To prevent overfitting by randomly turning off certain neurons during training
- To increase the depth of the network
- To scale the output of neurons during training
- To decrease the learning rate of the optimizer

5.3.5

Upsampling layer

An upsampling layer, also known as a deconvolutional layer or a transposed convolutional layer, is a type of layer in a neural network that is used for upsampling or increasing the spatial resolution of the input.

The purpose of an upsampling layer is to increase the resolution of feature maps while preserving their spatial information. This is useful in tasks such as image segmentation, where the goal is to classify each pixel in an image into different classes.

An upsampling layer works by reversing the process of a convolutional layer. In a convolutional layer, a filter is applied to the input feature map to produce an output feature map with reduced spatial resolution. In an upsampling layer, a filter is applied to the output feature map to produce an upsampled feature map with increased spatial resolution.

There are several types of upsampling layers used in deep learning, including:

- Nearest neighbor upsampling - simply duplicates the values in the input feature map to create a larger output feature map.
- Bilinear upsampling - uses a weighted average of the four nearest pixels in the input feature map to generate each pixel in the output feature map.
- Transposed convolutional upsampling uses a learnable filter to map each pixel in the output feature map to a patch of pixels in the input feature map, and then applies a convolution to generate the output feature map.

Upsampling layers are commonly used in architectures such as fully convolutional networks (FCNs) and U-Net architectures for tasks such as semantic segmentation, image super-resolution, and generative modeling.

5.3.6

What is the main purpose of an upsampling layer in a neural network?

- To increase the spatial resolution of feature maps while preserving their spatial information
- To decrease the computational complexity of the network
- To reduce the number of channels in the feature maps
- To normalize the input feature map before passing it to the next layer

5.3.7

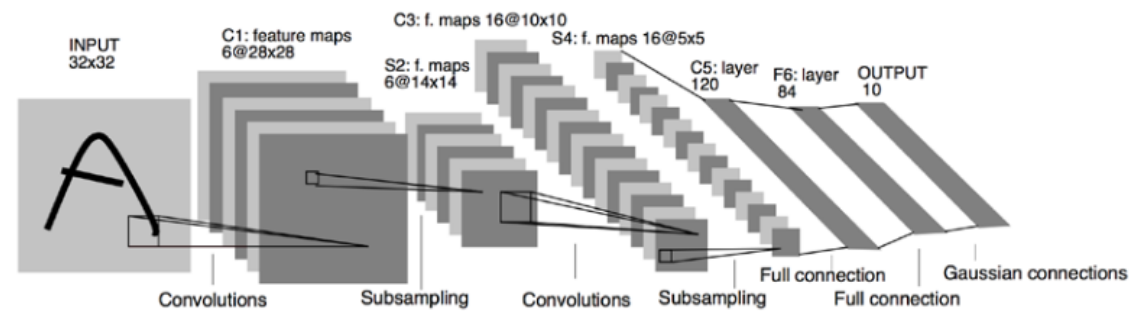
Architectures

CNNs have evolved over the years, with numerous architectures developed for various tasks. Here, we will explore some of the most well-known CNN architectures and their contributions to the field.

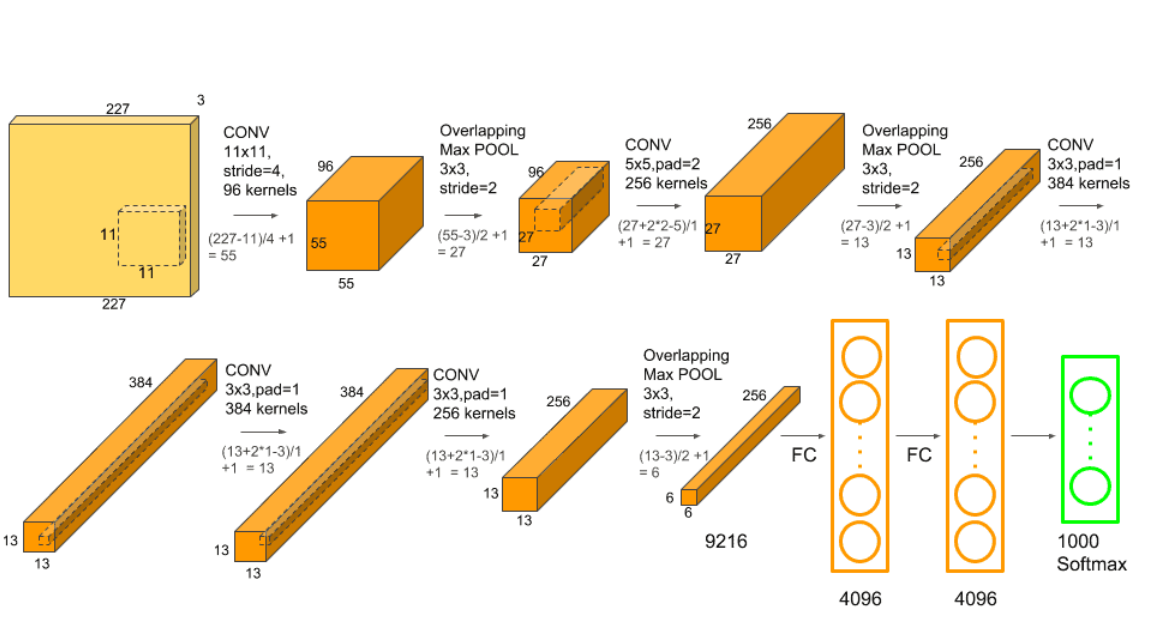
1. **LeNet -** developed by Yann LeCun in the 1990s, LeNet is one of the earliest CNNs. It was specifically designed for handwritten digit recognition and played a pivotal role in the development of deep learning.
2. **AlexNet** developed in 2012 by Alex Krizhevsky and others, achieved breakthrough performance on the ImageNet dataset, significantly improving the accuracy of image classification tasks. It popularized the use of CNNs for large-scale image classification.
3. The Visual Geometry Group at Oxford introduced **VGG** in 2014. This architecture is known for its simplicity and depth, consisting of multiple convolutional layers followed by fully connected layers.
4. **GoogLeNet**, introduced by Google researchers in 2014, is known for its Inception module, which allows the network to perform multiple convolution operations of different sizes in parallel, improving performance.
5. **ResNet**, developed by Microsoft in 2015, introduced the concept of residual connections, enabling the training of very deep networks by allowing the gradient to flow more easily through the network.
6. **DenseNet**, introduced in 2017 by Facebook AI Research, promotes dense connections between layers, which helps reuse features more efficiently and reduces the number of parameters.
7. Developed by Google in 2017, **MobileNet** is a lightweight architecture designed for mobile and embedded devices, focusing on efficiency and minimal computational resource usage.

These CNN architectures have paved the way for more advanced networks and have been applied in a wide range of tasks, from image classification to object detection.

LeNet-5 architecture visualization



AlexNet architecture visulazation



5.3.8

Which of the following are CNN architectures developed to improve performance on large-scale image classification tasks?

- AlexNet
- ResNet
- DenseNet
- LeNet

5.4 Practical applications

📖 5.4.1

Practical applications

CNNs have revolutionized many fields due to their ability to process and analyze image, video, and even non-visual data. Here are some key practical applications of CNNs:

1. CNNs are widely used for **image classification**, where they assign labels to input images from a predefined set. This is applicable to tasks like object recognition, facial recognition, and scene classification.
2. **Object detection** - CNNs are also used for locating and classifying objects within images or videos. This is important for applications like self-driving cars, security surveillance, and robotics.
3. **Semantic segmentation** - in this task, CNNs assign a label to every pixel in an image, enabling more detailed object detection and analysis. It's used for tasks like medical image analysis and autonomous driving.
4. **Image generation** - CNNs can generate new images based on a set of input parameters or styles, which is useful in creative applications like art and design.
5. **Style transfer** - CNNs can take the style of one image and apply it to another, combining the content of one image with the style of another. This technique is popular in digital art creation.
6. **Medical imaging** - CNNs are used in healthcare for diagnosing diseases from medical images, such as identifying tumors, analyzing X-rays, and MRI scans.
7. CNNs can **analyze video data** for tasks like action recognition, tracking objects across frames, and generating captions for video content.
8. **Natural language processing** - although typically used for image-related tasks, CNNs are also applied in NLP tasks like sentiment analysis, text classification, and language translation.
9. **Recommendation systems** - CNNs help in creating systems that recommend products, movies, or services based on user preferences and behavior.
10. **Autonomous vehicles** - in self-driving cars, CNNs are used to detect and classify objects on the road, such as pedestrians, other vehicles, and traffic signs.
11. **Agriculture** - CNNs can analyze satellite imagery to monitor crops, predict yields, and detect issues like pests or diseases.
12. **Robotics** - CNNs are used in robotics for object recognition, manipulation, and navigation, making robots smarter and more autonomous.
13. **Art and music** - CNNs can generate new pieces of art or music based on given styles or classify existing art/music according to genre.
14. **Gaming** - CNNs aid in video game development for tasks such as character recognition, animation, and interactive game environments.

15. **Finance** - CNNs are applied in finance for tasks like fraud detection, stock market prediction, and risk assessment.

These diverse applications show how CNNs are transforming various industries by automating complex tasks and providing insights from large amounts of data.

5.4.2

Which of the following is a common application of CNNs in the healthcare industry?

- Disease diagnosis from medical images
- Fraud detection
- Stock market prediction
- Object recognition in self-driving cars

5.4.3

In which application are CNNs used to detect and classify objects on the road?

- Autonomous Vehicles
- Gaming
- Agriculture
- Natural Language Processing

5.4.4

Image augmentation

Image augmentation is a technique used in deep learning to increase the size and diversity of the training set by applying transformations to the original images. This technique helps to reduce overfitting and improve the performance of the model.

There are various image augmentation techniques used in deep learning, such as:

- Rotation - rotates the image by a certain angle to create new training examples.
- Flipping - flipps the image horizontally or vertically to create a mirrored version of the original image.
- Translation - shifts the image horizontally or vertically to create a new training example.

- Scaling - rescales the image to create a smaller or larger version of the original image.
- Shearing - sheares the image to create a slanted version of the original image.
- Zooming - zooms into or out of the image to create a new training example.

These techniques can be used individually or in combination to generate a large and diverse set of training data. Image augmentation is particularly useful when the size of the original dataset is small, as it allows the model to generalize better and avoid overfitting to the training data.

A bad example of image augmentation in CNN would be applying random transformations that are not relevant to the problem being solved. For instance, if the task is to recognize handwritten digits, applying random rotations or flipping the images horizontally or vertically may not help improve the performance of the model, and may even introduce noise and confuse the model. Another bad example would be applying excessive transformations that distort the original image beyond recognition. For example, scaling an image to a very small size or shearing it to a very high degree may create a new training example, but the resulting image may be so distorted that it does not resemble the original image, making it difficult for the model to learn from it.

In general, image augmentation techniques should be carefully chosen based on the problem being solved, and should aim to create new training examples that are relevant and diverse, without introducing noise or distorting the original images too much.

5.4.5

Which of the following is an example of a transformation used in image augmentation?

- Scaling
- Random noise generation
- Image compression
- Image encoding

5.4.6

What is a potential issue with applying excessive image transformations, like extreme shearing or scaling, in image augmentation?

- It could distort the image so much that it no longer resembles the original content, making it difficult for the model to learn.
- It may generate irrelevant training data.
- It reduces the number of training examples.
- It improves model overfitting.

5.4.7

Project: Character recognition (LeNET-5 example)

This example downloads the MNIST handwritten digits and creates a simple CNN network based on LeNet-5 to predict the digit category (0-9).

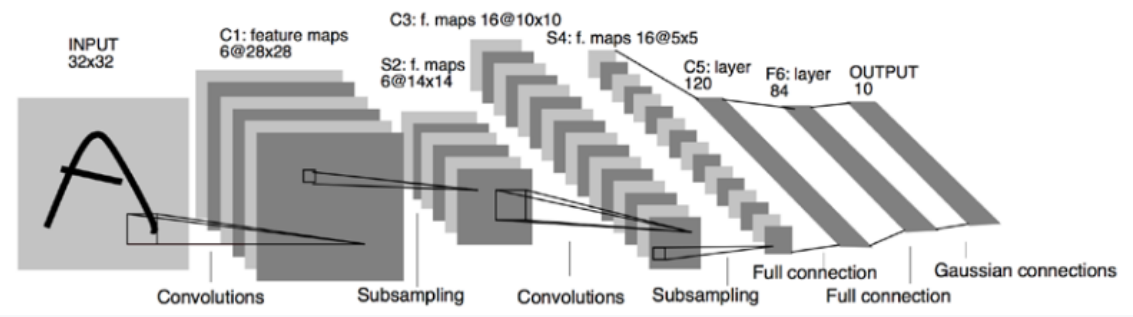
LeNet-5 is a classic convolutional neural network architecture designed for handwritten digit recognition, and was introduced by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner in 1998. It was one of the earliest successful attempts to apply deep learning techniques to image recognition tasks.

The LeNet-5 architecture consists of seven layers, including three convolutional layers and two fully connected layers. It takes as input a grayscale image of size 32x32 pixels, and outputs a probability distribution over the ten possible digit classes.

The first layer is a convolutional layer with six 5x5 filters, followed by a max-pooling layer with a 2x2 window. The second convolutional layer has 16 5x5 filters, again followed by a max-pooling layer with a 2x2 window. The third convolutional layer has 120 5x5 filters, and is followed by two fully connected layers, with 84 and 10 neurons respectively. The final layer uses a softmax activation function to produce the probability distribution over the ten digit classes.

LeNet-5 was a groundbreaking model in the field of deep learning, and its architecture has been used as a starting point for many subsequent models in image recognition and other fields.

Network visualization



Dataset

MNIST (Modified National Institute of Standards and Technology) is a widely-used dataset in the field of machine learning, specifically in the area of computer vision. It consists of 70,000 grayscale images of handwritten digits, with a resolution of 28x28 pixels. The dataset is split into a training set of 60,000 images and a test set of 10,000 images. MNIST is often used as a benchmark dataset for image classification tasks, particularly for testing and comparing different machine learning algorithms, including convolutional neural networks (CNNs). The task is to correctly classify the images into their corresponding digit class, from 0 to 9. MNIST has been used extensively for teaching purposes in machine learning and computer vision, as it is relatively small and easy to work with compared to many other datasets in the field. It has also been used as a baseline for evaluating the performance of more complex datasets and models.

```
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

```
# Load data from dataset
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()
# Reshape
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
#Padding the images by 2 pixels
x_train = np.pad(x_train, ((0,0),(2,2),(2,2),(0,0)),
'constant')
x_test = np.pad(x_test, ((0,0),(2,2),(2,2),(0,0)), 'constant')
```

Depth of the image (number of channels) is 1 because these images are grayscale. We'll also set up a seed to have reproducible results

```
image_width = x_train[0].shape[0]
image_height = x_train[0].shape[1]
num_channels = 1 # grayscale = 1 channel

seed = 98
np.random.seed(seed)
tf.random.set_seed(seed)
```

Parameters used for model training

```
batch_size = 100
evaluation_size = 500
epochs = 300
eval_every = 5
```

Normalize our images to change the values of all pixels to a common scale

```
x_train = x_train / 255
x_test = x_test/ 255
```

Declare model layers

```
input_data = tf.keras.Input(dtype=tf.float32,
shape=(image_width,image_height, num_channels), name="INPUT")

# First Conv-ReLU-MaxPool Layer
conv1 = tf.keras.layers.Conv2D(filters=6, kernel_size=5,
padding='VALID', activation="relu", name="C1")(input_data)
max_pool1 = tf.keras.layers.MaxPool2D(pool_size=2, strides=2,
padding='SAME', name="S1")(conv1)
# Second Conv-ReLU-MaxPool Layer
conv2 = tf.keras.layers.Conv2D(filters=16, kernel_size=5,
padding='VALID', strides=1, activation="relu",
name="C3")(max_pool1)
max_pool2 = tf.keras.layers.MaxPool2D(pool_size=2, strides=2,
padding='SAME', name="S4")(conv2)
# Flatten Layer
flatten = tf.keras.layers.Flatten(name="FLATTEN")(max_pool2)
# First Fully Connected Layer
fully_connected1 = tf.keras.layers.Dense(units=120,
activation="relu", name="F5")(flatten)
# Second Fully Connected Layer
```

```
fully_connected2 = tf.keras.layers.Dense(units=84,
activation="relu", name="F6")(fully_connected1)
# Final Fully Connected Layer
final_model_output = tf.keras.layers.Dense(units=10,
activation="softmax", name="OUTPUT")(fully_connected2)
model = tf.keras.Model(inputs= input_data,
outputs=final_model_output)
```

Compile the model with the sparse categorical cross-entropy loss and the ADAM optimizer.

```
model.compile(optimizer="adam",
loss="sparse_categorical_crossentropy", metrics=["accuracy"] )
```

Show model summary

```
model.summary()
```

Program output:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 INPUT (InputLayer)          [(None, 32, 32, 1)]       0

 C1 (Conv2D)                 (None, 28, 28, 6)         156

 S1 (MaxPooling2D)           (None, 14, 14, 6)         0

 C3 (Conv2D)                 (None, 10, 10, 16)        2416

 S4 (MaxPooling2D)           (None, 5, 5, 16)          0

 FLATTEN (Flatten)           (None, 400)               0

 F5 (Dense)                  (None, 120)               48120
```

```
train_loss = []
train_acc = []
test_acc = []
```

```
for i in range(epochs):
  rand_index = np.random.choice(len(x_train), size=batch_size)
  rand_x = x_train[rand_index]
  rand_y = y_train[rand_index]
  history_train = model.train_on_batch(rand_x, rand_y)

  if (i+1) % eval_every == 0:
    eval_index = np.random.choice(len(x_test),
size=evaluation_size)
    eval_x = x_test[eval_index]
    eval_y = y_test[eval_index]
    history_eval = model.evaluate(eval_x,eval_y)
    # Record and print results
    train_loss.append(history_train[0])
    train_acc.append(history_train[1])
    test_acc.append(history_eval[1])
    acc_and_loss = [(i+1), history_train[0], history_train[1],
history_eval[1]]
    acc_and_loss = [np.round(x,2) for x in acc_and_loss]
    print('Epoch # {}. Train Loss: {:.2f}. Train Acc (Test
Acc): {:.2f} ({:.2f})'.format(*acc_and_loss))
```

```
print(history_train[0])
```

Program output:

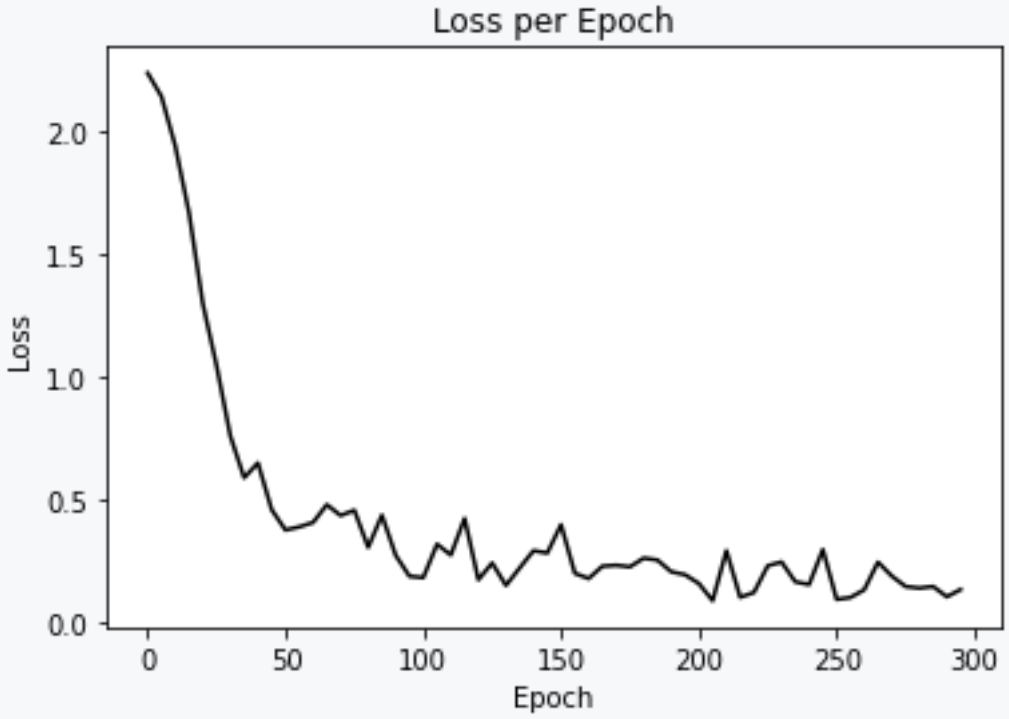
```
0.13152286410331726
```

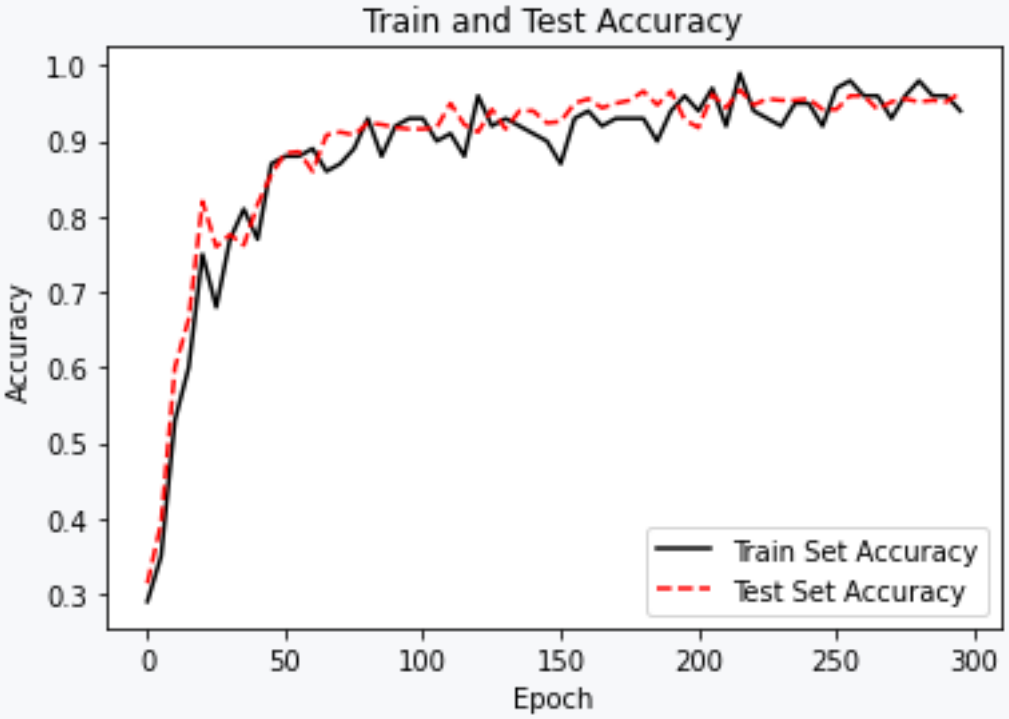
Plot the loss and accuracy.

```
# Matlotlib code to plot the loss and accuracy
eval_indices = range(0, epochs, eval_every)
# Plot loss over time
plt.plot(eval_indices, train_loss, 'k-')
plt.title('Loss per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

# Plot train and test accuracy
plt.plot(eval_indices, train_acc, 'k-', label='Train Set
Accuracy')
plt.plot(eval_indices, test_acc, 'r--', label='Test Set
Accuracy')
```

```
plt.title('Train and Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```

Program output:



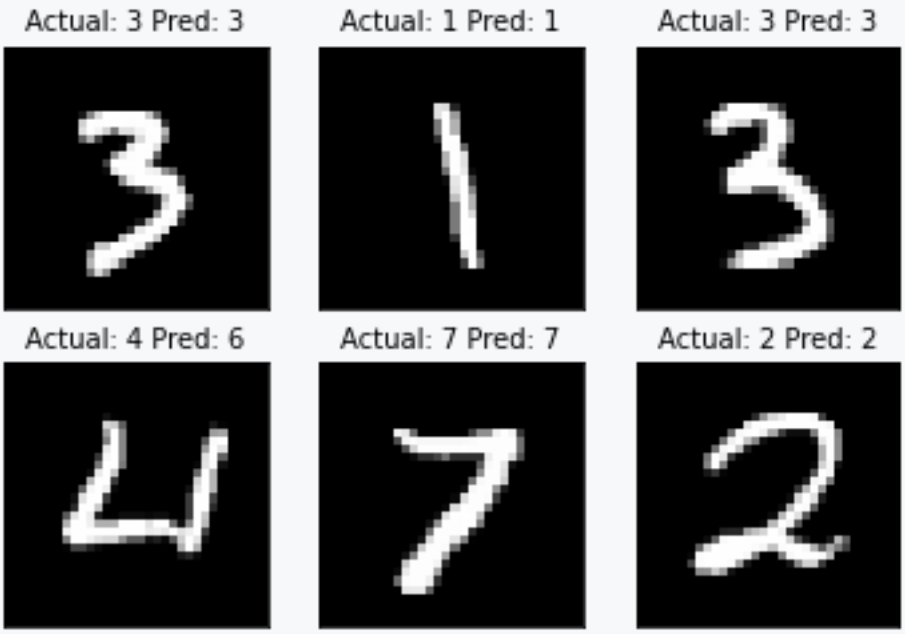


Results for six examples

```
# Plot some samples and their predictions
actuals = y_test[30:36]
preds = model.predict(x_test[30:36])
predictions = np.argmax(preds,axis=1)
images = np.squeeze(x_test[30:36])
Nrows = 2
Ncols = 3
for i in range(6):
  plt.subplot(Nrows, Ncols, i+1)
  plt.imshow(np.reshape(images[i], [32,32]), cmap='Greys_r')
  plt.title('Actual: ' + str(actuals[i]) + ' Pred: ' +
str(predictions[i]), fontsize=10)
  frame = plt.gca()
  frame.axes.get_xaxis().set_visible(False)
  frame.axes.get_yaxis().set_visible(False)

plt.show()
```

Program output:

```
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 166ms/step
```



5.4.8

Project: More complex CNN

Create CNN to identify 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

This example shows more complex CNN model with dropout Extending the depth of CNN networks is done in a standard fashion: we just repeat the convolution, max pooling, and ReLU in series until we are satisfied with the depth. Many of the more accurate image recognition networks operate in this fashion.

Dataset

CIFAR-10 is a popular image classification dataset used in machine learning and computer vision research. It consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

The dataset is divided into 50,000 training images and 10,000 test images, and is often used as a benchmark for image classification models. The small size of the images and the diversity of the classes make it a challenging dataset for machine learning models to accurately classify. It has been widely used to evaluate the performance of deep learning models such as convolutional neural networks (CNNs).

```
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Parameters. Using 20 epochs takes a lot of time in training. It can be lowered but at a cost of accuracy.

```
# Set dataset and model parameters
batch_size = 128
buffer_size= 128
epochs=4 #20

#Set transformation parameters
crop_height = 24
crop_width = 24
```

```
cifar_classes = ['airplane', 'automobile', 'bird', 'cat',
'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

Load CIFAR dataset

```
# Get data
print('Getting/Transforming Data.')
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar10.load_data()
```

Program output:

```
Getting/Transforming Data.
```

```
print(x_train.shape)
```

Program output:

```
(50000, 32, 32, 3)
```

Define a reading function that will load and distort the images slightly for training

```
# Define CIFAR reader
def read_cifar_files(image, label):
  final_image = tf.image.resize_with_crop_or_pad(image,
crop_width, crop_height)
  final_image = image / 255

  # Randomly flip the image horizontally, change the
brightness and contrast
  final_image = tf.image.random_flip_left_right(final_image)
  final_image =
tf.image.random_brightness(final_image,max_delta=0.1)
  final_image =
tf.image.random_contrast(final_image,lower=0.5, upper=0.8)

  return final_image, label
```

```
dataset_train = tf.data.Dataset.from_tensor_slices((x_train,
y_train))
dataset_test = tf.data.Dataset.from_tensor_slices((x_test,
y_test))
```

```
def show(image, label):
```

```
  plt.figure()
  plt.imshow(image)
  plt.title(cifar_classes[label.numpy()[0]])
  plt.axis('off')

for image, label in dataset_train.take(2):
  show(image, label)
  image, label = read_cifar_files(image, label)
  show(image, label)
```

Program output:


```
dataset_train_processed =
dataset_train.shuffle(buffer_size).batch(batch_size).map(read_
cifar_files)
dataset_test_processed =
dataset_test.batch(batch_size).map(read_cifar_files)
```

Model definition

```
model = keras.Sequential(
    [# First Conv-ReLU-Conv-ReLU-MaxPool Layer
     tf.keras.layers.Conv2D(input_shape=[32,32,3],
```

```
                            filters=32,
                            kernel_size=3,
                            padding='SAME',
                            activation="relu",
                            kernel_initializer='he_uniform',
                            name="C1"),
    tf.keras.layers.Conv2D(filters=32,
                           kernel_size=3,
                           padding='SAME',
                           activation="relu",
                           kernel_initializer='he_uniform',
                           name="C2"),
     tf.keras.layers.MaxPool2D((2,2),
                               name="P1"),
     tf.keras.layers.Dropout(0.2),
    # Second Conv-ReLU-Conv-ReLU-MaxPool Layer
     tf.keras.layers.Conv2D(filters=64,
                            kernel_size=3,
                            padding='SAME',
                            activation="relu",
                            kernel_initializer='he_uniform',
                            name="C3"),
    tf.keras.layers.Conv2D(filters=64,
                           kernel_size=3,
                           padding='SAME',
                           activation="relu",
                           kernel_initializer='he_uniform',
                           name="C4"),
     tf.keras.layers.MaxPool2D((2,2),
                               name="P2"),
     tf.keras.layers.Dropout(0.2),
    # Third Conv-ReLU-Conv-ReLU-MaxPool Layer
     tf.keras.layers.Conv2D(filters=128,
                            kernel_size=3,
                            padding='SAME',
                            activation="relu",
                            kernel_initializer='he_uniform',
                            name="C5"),
    tf.keras.layers.Conv2D(filters=128,
                           kernel_size=3,
                           padding='SAME',
                           activation="relu",
                           kernel_initializer='he_uniform',
                           name="C6"),
```

```
    tf.keras.layers.MaxPool2D((2,2),
                              name="P3"),
    tf.keras.layers.Dropout(0.2),
    # Flatten Layer
    tf.keras.layers.Flatten(name="FLATTEN"),
    # Fully Connected Layer
    tf.keras.layers.Dense(units=128,
                          activation="relu",
                          name="D1"),
   tf.keras.layers.Dropout(0.2),
   # Final Fully Connected Layer
   tf.keras.layers.Dense(units=10,
                         activation="softmax",
                         name="OUTPUT")
   ])
```

Model compilation

```
from keras.optimizers import SGD
model.compile(
   # optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
model.summary()
```

Program output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 C1 (Conv2D)                 (None, 32, 32, 32)        896

 C2 (Conv2D)                 (None, 32, 32, 32)        9248

 P1 (MaxPooling2D)           (None, 16, 16, 32)        0

 dropout (Dropout)           (None, 16, 16, 32)        0

 C3 (Conv2D)                 (None, 16, 16, 64)        18496

 C4 (Conv2D)                 (None, 16, 16, 64)        36928
```

```
P2 (MaxPooling2D)            (None, 8, 8, 64)          0

dropout_1 (Dropout)          (None, 8, 8, 64)          0

C5 (Conv2D)                  (None, 8, 8, 128)         73856

C6 (Conv2D)                  (None, 8, 8, 128)         147584

P3 (MaxPooling2D)            (None, 4, 4, 128)         0
```

Start training

```
history = model.fit(dataset_train_processed,
                    validation_data=dataset_test_processed,
                    epochs=epochs)
```

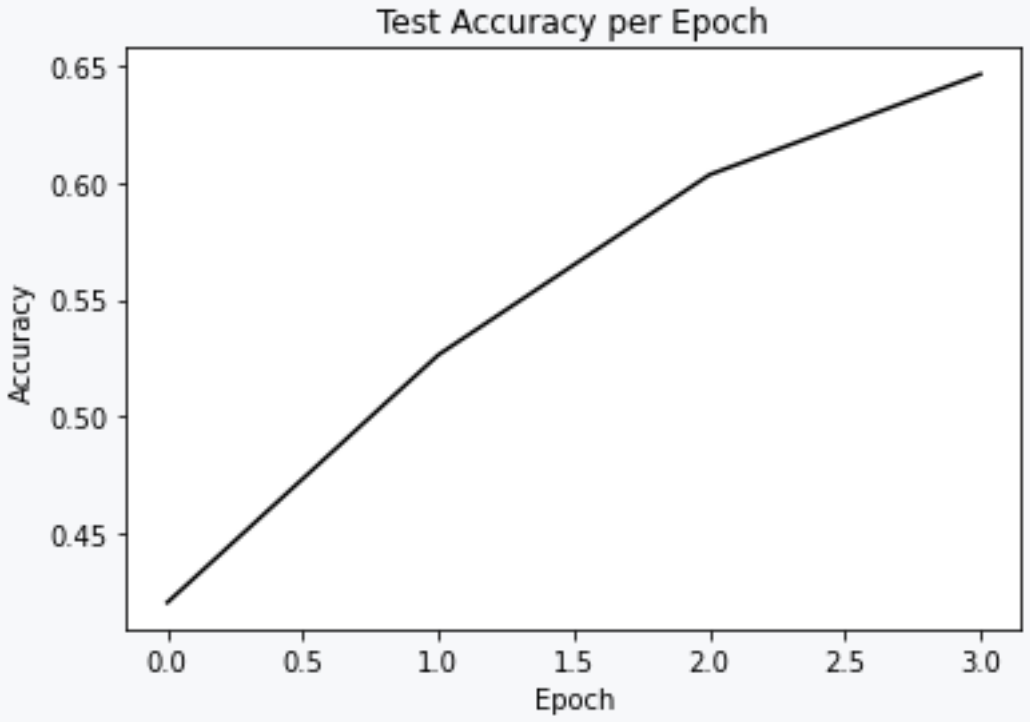
Program output:

```
385/391 [============================>.] - ETA: 1s - loss:
1.1362 - accuracy: 0.5991
386/391 [============================>.] - ETA: 1s - loss:
1.1365 - accuracy: 0.5991
387/391 [============================>.] - ETA: 1s - loss:
1.1362 - accuracy: 0.5991
388/391 [============================>.] - ETA: 0s - loss:
1.1362 - accuracy: 0.5989
389/391 [============================>.] - ETA: 0s - loss:
1.1360 - accuracy: 0.5991
390/391 [============================>.] - ETA: 0s - loss:
1.1353 - accuracy: 0.5993
391/391 [==============================] - ETA: 0s - loss:
1.1353 - accuracy: 0.5993
391/391 [==============================] - 112s 285ms/step -
loss: 1.1353 - accuracy: 0.5993 - val_loss: 1.0027 -
val_accuracy: 0.6467
```

```
# Print loss and accuracy
# Matlotlib code to plot the loss and accuracy
epochs_indices = range(0, epochs, 1)

# Plot loss over time
```

```
plt.plot(epochs_indices, history.history["loss"], 'k-')
plt.title('Softmax Loss per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Softmax Loss')
plt.show()

# Plot accuracy over time
plt.plot(epochs_indices, history.history["val_accuracy"], 'k-
')
plt.title('Test Accuracy per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```

Program output:



5.5 Pre-trained networks

🕮 5.5.1

Retraining existing CNN models

Retraining existing CNN models is a technique widely used in transfer learning, where a pre-trained convolutional neural network (CNN) is adapted to solve a new task. Instead of building a CNN from scratch, this approach leverages the knowledge and

features learned by a pre-trained model on a large dataset. Retraining is especially valuable when the new dataset is small or when computational resources are limited.

The retraining process typically involves modifying the pre-trained CNN by removing its output layers, which were designed for its original task, and replacing them with new layers suited to the new task. For example, the original output layer designed for classifying 1,000 ImageNet categories can be replaced with a layer for classifying just a few categories in a new dataset. The early layers of the pre-trained model are often frozen, meaning their weights remain unchanged during training. This ensures that the fundamental features already learned (e.g., edges and textures) are preserved. The new layers are then trained using the new dataset, while optional fine-tuning of the frozen layers can enhance the model's performance.

Retraining saves time and computational resources compared to training a CNN from scratch, as the pre-trained model has already captured generic features useful across a variety of tasks. It also helps achieve better performance when the new dataset is small, as the model can transfer its understanding of similar data. This approach is commonly used in fields like medical imaging, where datasets are limited, and in applications like facial recognition or object detection, where high accuracy is essential.

5.5.2

What is the main advantage of retraining a pre-trained CNN model instead of training one from scratch?

- It reduces computational resources and time.
- It avoids the need for fine-tuning.
- It always results in higher accuracy.
- It requires a larger dataset.

5.5.3

In retraining a CNN, what happens to the early layers of the pre-trained model?

- They are frozen, meaning their weights are not updated.
- They are replaced by new layers.
- They are completely removed from the network.
- They are updated with random weights.

5.5.4

Project: Retraining example

We will use transfer learning from a pre-trained network for CIFAR-10. The idea is to reuse the weights and structure of the prior model from the convolutional layers and retrain the fully connected layers at the top of the network. This method is called **fine-tuning**.

Inception model

Inception-v3 is a convolutional neural network architecture designed for image recognition and classification, and was introduced by Google researchers in 2015. It is an improvement over the earlier Inception-v1 and Inception-v2 models, and features a number of innovations to improve both accuracy and efficiency. The Inception-v3 architecture consists of many layers, including multiple convolutional and pooling layers, as well as a number of "inception" modules. These modules use a combination of 1x1, 3x3, and 5x5 convolutions to extract features from the input image at different scales and resolutions. In addition to these standard layers, Inception-v3 also includes a number of specialized layers, such as batch normalization layers, which help to improve the training process, and a global average pooling layer, which helps to reduce the number of parameters in the model. Inception-v3 has achieved state-of-the-art results on a number of image recognition benchmarks, and its architecture has been used as a starting point for many subsequent models in the field of computer vision.

Dataset

We know: CIFAR-10 is a popular image classification dataset used in machine learning and computer vision research. It consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

The dataset is divided into 50,000 training images and 10,000 test images, and is often used as a benchmark for image classification models. The small size of the images and the diversity of the classes make it a challenging dataset for machine learning models to accurately classify. It has been widely used to evaluate the performance of deep learning models such as convolutional neural networks (CNNs).

```
import warnings
warnings.filterwarnings("ignore")
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications.inception_v3 import
InceptionV3
```

```
from tensorflow.keras.applications.inception_v3 import
preprocess_input, decode_predictions
```

1. Prepare data

```
# Set dataset parameters
batch_size = 32
buffer_size= 1000
```

Download the dataset and declare the 10 categories to reference when saving the images later on

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar10.load_data()

objects = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
```

2. Initialize the data pipeline

Inception v3 is pretrained on the ImageNet dataset, so our CIFAR-10 images must match the format of these images. The width and height expected should be no smaller than 75, so we will resize our images to 75x75 spatial size. Then, the images should be normalized, so we will apply the inception preprocessing task (the preprocess_input method) on each image.

```
# Create training and testing datasets from tensor slices of
input features and labels
dataset_train = tf.data.Dataset.from_tensor_slices((x_train,
y_train))
dataset_test = tf.data.Dataset.from_tensor_slices((x_test,
y_test))

def preprocess_cifar10(img, label):
    # Cast image to float32 for compatibility with
preprocessing functions
    img = tf.cast(img, tf.float32)
    # Resize image to 75x75 pixels for compatibility with
InceptionV3 input size
    img = tf.image.resize(img, (75, 75))
    # Apply InceptionV3-specific preprocessing to the image
```

```
    return
tf.keras.applications.inception_v3.preprocess_input(img),
label

# Shuffle and batch the training dataset, then apply the
preprocessing function
dataset_train_processed =
dataset_train.shuffle(buffer_size).batch(batch_size).map(prepr
ocess_cifar10)
# Batch the testing dataset and apply the preprocessing
function
dataset_test_processed =
dataset_test.batch(batch_size).map(preprocess_cifar10)
```

We want to load the weights without the classification head.

```
# Load the InceptionV3 model with the following
configurations:
# - include_top=False: Excludes the fully connected top
layers, making it suitable for feature extraction
# - weights="imagenet": Loads pretrained weights from the
ImageNet dataset
# - input_shape=(75,75,3): Specifies the input size as 75x75
pixels with 3 color channels (RGB)
inception_model = InceptionV3(
    include_top=False,
    weights="imagenet",
    input_shape=(75, 75, 3)
)
```

We build our own model on top of the InceptionV3 model by adding a classifier with three fully connected layers.

```
# Extract the output of the InceptionV3 model as the base
feature map
x = inception_model.output

# Apply Global Average Pooling to reduce each feature map to a
single value
x = keras.layers.GlobalAveragePooling2D()(x)

# Add a dense layer with 1024 units and ReLU activation for
further feature learning
```

```
x = keras.layers.Dense(1024, activation="relu")(x)

# Add another dense layer with 128 units and ReLU activation
for additional feature abstraction
x = keras.layers.Dense(128, activation="relu")(x)

# Add the final dense layer with 10 units (corresponding to 10
classes) and softmax activation for classification
output = keras.layers.Dense(10, activation="softmax")(x)

# Define the full model, using the InceptionV3 model's input
and the newly added output layers
model = keras.Model(inputs=inception_model.input,
outputs=output)
```

We'll set the base layers in Inception as not trainable. Only the classifier weights will be updated during the back-propagation phase (not the Inception weights):

```
 Freeze all layers in the InceptionV3 base model to retain
pre-trained weights during training
for inception_layer in inception_model.layers:
    inception_layer.trainable = False

# Compile the model with the Adam optimizer, sparse
categorical cross-entropy loss for integer labels,
# and accuracy as the evaluation metric
model.compile(optimizer="adam",
loss="sparse_categorical_crossentropy", metrics=["accuracy"])
```

Show model architecture

```
model.summary()
```

```
# Start training
model.fit(x=dataset_train_processed ,
          validation_data=dataset_test_processed)
```

Program output:

```
1556/1563 [============================>.] - ETA: 0s - loss:
1.1405 - accuracy: 0.6028
1557/1563 [============================>.] - ETA: 0s - loss:
1.1405 - accuracy: 0.6029
```

```
1558/1563 [============================>.] - ETA: 0s - loss:
1.1406 - accuracy: 0.6028
1559/1563 [============================>.] - ETA: 0s - loss:
1.1405 - accuracy: 0.6029
1560/1563 [============================>.] - ETA: 0s - loss:
1.1403 - accuracy: 0.6030
1561/1563 [============================>.] - ETA: 0s - loss:
1.1402 - accuracy: 0.6030
1562/1563 [============================>.] - ETA: 0s - loss:
1.1401 - accuracy: 0.6030
1563/1563 [==============================] - 152s 94ms/step -
loss: 1.1400 - accuracy: 0.6030 - val_loss: 1.0638 -
val_accuracy: 0.6242
```

Accuracy at the end is over 60%.

Remember that we are fine-tuning the model and retraining the fully connected layers at the top to fit our 10-category data.

5.6 Object detection

🕮 5.6.1

Binary classification

Binary classification is the simplest approach for image classification tasks. It involves categorizing images into just two distinct classes, such as "cat" or "not cat." This simplicity makes it a common starting point for understanding classification models. In convolutional neural networks (CNNs), the process begins with a convolutional operation to extract features from images. Pooling layers, like max pooling and average pooling, are then used to reduce the spatial dimensions while retaining important features. The pooling output is further processed by a flattening layer, converting it into a single column of data.

To enhance model performance, image augmentation is applied to create diverse training datasets, and batch normalization is used to stabilize and accelerate learning. These components distinguish CNNs from other artificial neural networks (ANNs). The binary classifier concludes with a dense output layer having a single unit activated by a sigmoid function, producing probabilities for the two classes. While effective for simple tasks, binary classifiers can be extended to classify more than two objects, transitioning into the realm of object classification.

5.6.2

What activation function is typically used in the output layer of a binary image classifier?

- Sigmoid
- ReLU
- Softmax
- Tanh

5.6.3

Object classification in image classification

Object classification extends beyond binary tasks by categorizing images into multiple classes or identifying specific objects within images. The simplest form is **image classification**, where the goal is to assign a single label to the entire image, identifying its most probable category. Traditional CNNs are commonly used for this purpose.

In **classification with localization**, the task becomes more complex, as the model must identify the category of an object and also locate it within the image using bounding boxes. Simplified models like You Only Look Once (YOLO) or R-CNN (Region-based Convolutional Neural Network) are often employed for this.

Detection takes this a step further, aiming to detect, localize, and classify multiple objects within the same image. The output includes multiple bounding boxes and their associated class labels. Advanced models like YOLO and R-CNN excel in this domain, offering solutions to challenges like overlapping objects and varying scales. Thus, object classification tasks vary in complexity, with detection requiring both high accuracy and precise localization.

5.6.4

Which models are commonly used for object detection tasks?

- YOLO
- R-CNN
- VGG
- LeNet

📖 5.6.5

Region-based convolutional neural network

Region-based convolutional neural network (R-CNN) is a popular object detection algorithm that uses a combination of region proposals and convolutional neural networks to localize and classify objects in an image. Its process involves three key steps:

1. **Region proposal generation** - the algorithm uses a selective search to generate regions of interest in an image.
2. **Feature extraction** - each region is passed through a CNN to extract features.
3. **Classification and localization** - a classifier categorizes objects, while bounding box regression refines object location.

R-CNN was introduced in 2014 by Ross Girshick, et al. as an improvement over previous object detection algorithms that used hand-crafted features and sliding windows to classify objects. R-CNN was one of the first object detection algorithms to use deep learning and has since been improved upon with faster variants, such as Fast R-CNN and Faster R-CNN, which use a single network for region proposal and classification, leading to faster and more accurate object detection. There are improved version Fast R-CNN and Faster R-CNN.

📝 5.6.6

What is the first step in the R-CNN process?

- Region proposal generation
- Bounding box regression
- Feature extraction
- Classification

📖 5.6.7

YOLO

YOLO (You Only Look Once) is a deep learning object detection model that can detect objects in real-time images and videos with high accuracy. It was developed by Joseph Redmon, and it stands out from other object detection models because of its speed and efficiency.

YOLO uses a single neural network that can directly predict the bounding boxes and class probabilities for multiple objects in an image in one shot. This means that the network only needs to look at the image once to detect objects, as opposed to the traditional two-stage methods where the image is first segmented into regions of interest, and then those regions are classified. YOLO's single-stage approach makes it significantly faster than other object detection models while maintaining high accuracy.

YOLO has been updated with several versions, including YOLOv2, YOLOv3, and YOLOv4, each with its own improvements and optimizations to increase speed and accuracy. YOLO is widely used in various applications, including self-driving cars, surveillance systems, and object recognition in social media.

5.6.8

What are the advantages of YOLO compared to other object detection algorithms?

- Real-time detection
- Single-pass processing
- Requires hand-crafted features
- Uses sliding windows for object detection

5.6.9

Single shot detector

Single shot detector (SSD) is an object detection algorithm that belongs to the family of one-stage detectors, meaning that it performs object detection in a single forward pass of the neural network. SSD is based on a fully convolutional neural network that predicts the class scores and the bounding box coordinates of multiple objects in an image.

The key idea behind SSD is to use a set of default bounding boxes with different aspect ratios and scales at each spatial location in the feature map of the last convolutional layer. These default bounding boxes act as templates to detect objects of different sizes and shapes. The network predicts the offsets and scales of these default bounding boxes to obtain the final predicted bounding boxes.

Compared to other object detection algorithms, SSD has the advantage of being faster and more accurate, especially for detecting small objects. It has been used in various applications, such as autonomous driving, robotics, and surveillance systems.

In autonomous drones, SSD is used to detect small objects like birds or obstacles during flight. Its ability to handle multiple object sizes and its speed make it ideal for time-sensitive applications.

5.6.10

What is a key feature of SSD that makes it efficient?

- Employs default bounding boxes
- Uses selective search for region proposals
- Divides the image into a grid for detection
- Combines hand-crafted features with deep learning

5.7 Accuracy measurement

5.7.1

Object detection performance evaluation

Object detection performance evaluations typically involve measuring the accuracy of a model in detecting and localizing objects within an image. Some common metrics used for evaluation include:

- Precision - the proportion of true positive detections (correctly identified objects) over the total number of detections made by the model.
- Recall - the proportion of true positive detections over the total number of objects present in the image.
- Intersection over union (IoU)- a measure of the overlap between the ground truth bounding box and the predicted bounding box. IoU is typically used to determine whether a detection is a true positive or a false positive.
- Average precision - a metric that combines both precision and recall, by computing the area under the precision-recall curve.
- Mean average precision - the average AP across all object categories in the dataset.
- F1 score - the harmonic mean of precision and recall, which provides a balanced measure of the model's accuracy.

These metrics are used to evaluate the performance of different object detection models and to compare them against each other on various datasets.

5.7.2

Which metrics are commonly used to evaluate object detection models?

- Precision
- Mean average precision
- Learning rate
- Epoch count

5.7.3

What does Intersection over Union measure?

- The overlap between the ground truth and predicted bounding boxes
- The balance between precision and recall
- The total number of detections made by a model
- The area under the precision-recall curve

5.7.4

Which metrics combine precision and recall for object detection?

- F1 score
- Average precision
- IoU
- Precision

Recurrent Neural Networks - RNNs

Chapter 6

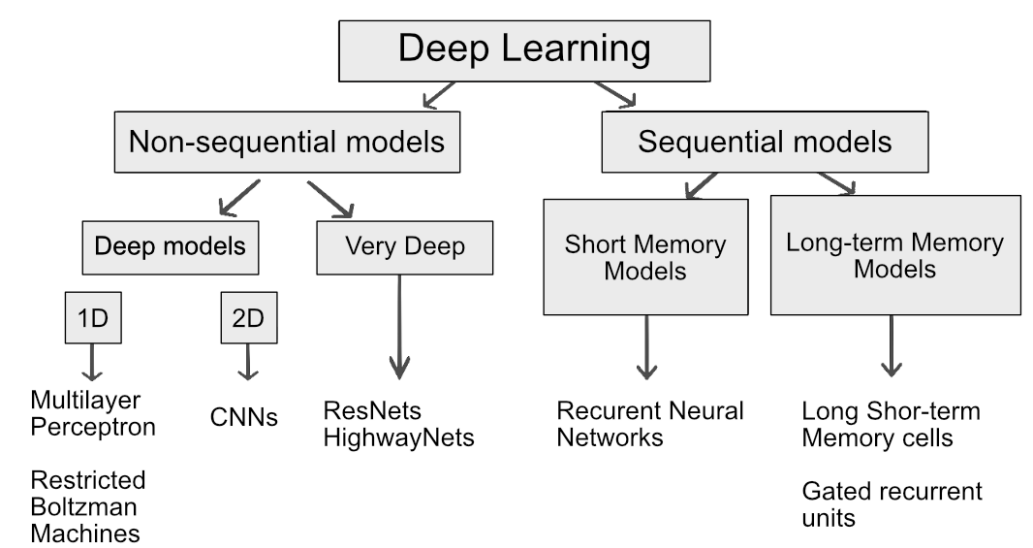
6.1 RNN overview

📖 6.1.1

Recurrent neural networks (RNNs) are a type of neural network that are commonly used for processing sequential data. Unlike traditional neural networks that process fixed-length inputs, RNNs can handle inputs of variable lengths by maintaining a "memory" of the previous inputs that they have processed. RNNs use this memory to make predictions based on the current input and the context provided by the previous inputs.

RNNs consist of a series of repeating units that take an input and produce an output while maintaining an internal state that captures the "memory" of previous inputs. This internal state is passed on to the next unit in the sequence, allowing the network to maintain a context across multiple inputs. The output of the final unit in the sequence is typically fed into a fully connected layer to produce the final output of the network.

RNNs are particularly well-suited for tasks such as language modeling, speech recognition, and natural language processing, where the input data is inherently sequential and the context of previous inputs is important for making accurate predictions.



6.1.2

What is the key feature that enables RNNs to handle sequential data?

- Maintaining a "memory" of previous inputs

- The use of convolutional layers
- Processing fixed-length inputs only
- The absence of an internal state

6.1.3

For which tasks are RNNs particularly well-suited?

- Language modeling
- Object detection
- Speech recognition
- Image segmentation

6.1.4

Sequential data and deep learning models

Sequential data refers to data sets in which each data point depends on previous data. Consider it a sentence, which consists of a series of words that are related to each other. A verb is linked to a subject and an adverb is linked to a verb. Another example is a stock price, where the price on a particular day is related to the price of the previous days. Traditional neural networks are not suitable for processing this type of data. There is a specific type of architecture that can ingest data sequences. A RNN model is a specific type of deep learning architecture in which the output of the model is returned to the input. This type of model has its own challenges (known as disappearing and exploding gradients).In many ways, a RNN is a representation of how the brain can work. RNN uses memory to help them learn. But how can they do this if the information flows only in one direction? To understand this, you first need to examine sequential data. This is a type of data that requires work memory to process data effectively. Until now, you have only investigated non-sequence models, such as perceptron or CNN.

Typical examples of sequential data:

1. **Time series data** includes data that is collected over time, such as stock prices, weather data, or sensor data.
2. **Natural language processing data** includes text data, such as words or sentences, that have a specific sequence.
3. **Music data** includes audio data that has a temporal order, such as music notes or beats.
4. **Video data** includes data that is captured from a sequence of images, such as videos or motion capture data.

RNNs mimic certain aspects of how the human brain processes sequences, leveraging memory to make predictions or generate outputs that depend on historical inputs.

6.1.5

Why are traditional neural networks unsuitable for processing sequential data?

- They lack the ability to capture dependencies over time.
- They do not use convolutional layers.
- They cannot process non-image data.
- They require labeled data for training.

6.1.6

Which of the following are typical examples of sequential data?

- Stock prices
- Music notes
- Static images
- Object boundaries

6.1.7

Difference between RNN and CNN

Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) are both types of deep learning models, but they are designed for different types of input data and tasks.

RNNs are typically used for sequential data, where the order of the data matters, such as time series or natural language processing. They use feedback connections between neurons to maintain a memory of previous inputs, allowing them to model temporal dependencies in the data.

CNNs, on the other hand, are typically used for data that has a grid-like structure, such as images, audio spectrograms, or even text in the form of 2D word embeddings. They use convolutional layers to extract local features from the data, and pooling layers to reduce the spatial resolution while retaining the most important features.

In terms of architecture, RNNs typically have a single recurrent layer or multiple stacked recurrent layers, while CNNs can have multiple convolutional layers, followed by pooling layers and then fully connected layers for classification. RNNs are trained using backpropagation through time (BPTT), while CNNs are trained using backpropagation through the convolutional layers.

In summary, the main difference between RNNs and CNNs is that RNNs are designed for sequential data, while CNNs are designed for grid-like data such as images.

6.1.8

What is the main difference between RNNs and CNNs?

- RNNs are designed for sequential data, while CNNs are designed for grid-like data such as images.
- RNNs use pooling layers, while CNNs use recurrent layers.
- RNNs are used for grid-like data, while CNNs are used for sequential data.
- CNNs are designed for temporal data, while RNNs are used for spatial data.

6.1.9

RNNs can be typically used for

- Sequential data
- Image data
- Grid structured data

6.1.10

Typical applications of RNNs

RNNs are powerful models primarily used for tasks that involve sequential data, where the order of data points and the context within the sequence matter. Some common applications of RNNs are:

1. **Natural language processing** - RNNs excel in NLP tasks such as language modeling, machine translation, sentiment analysis, and speech recognition. They are capable of capturing the temporal structure of language, which helps in understanding the relationship between words over time.
2. **Time series analysis** - RNNs are frequently used for analyzing time-dependent data, such as stock prices, weather data, or sensor data. By

analyzing patterns in time series data, RNNs can make predictions about future events or detect anomalies that might indicate a problem.
3. **Image and video captioning** - In this application, RNNs are used to generate captions for images or descriptions of video sequences. The RNN processes visual information sequentially, capturing features and generating human-readable descriptions of the content.
4. **Music generation** - RNNs are also used to generate new music by learning patterns and structures from existing pieces. The model can then predict and generate a sequence of notes, creating compositions that follow musical structures.
5. **Handwriting recognition** - RNNs are employed in handwriting recognition to process the sequence of strokes and convert them into text. The temporal aspect of the strokes is captured by the RNN, making it effective for recognizing handwritten words.
6. **Speech recognition** - RNNs are integral to speech recognition systems, which convert spoken words into text. The sequential nature of speech data makes RNNs well-suited for this task, as they can recognize patterns in sound waves over time.

Overall, RNNs are most effective for tasks where the relationships between consecutive data points are important, such as time series forecasting, language processing, and sequential pattern recognition.

6.1.11

Which of the following tasks can RNNs be used for?

- Natural language processing
- Time series analysis
- Image classification
- Object detection

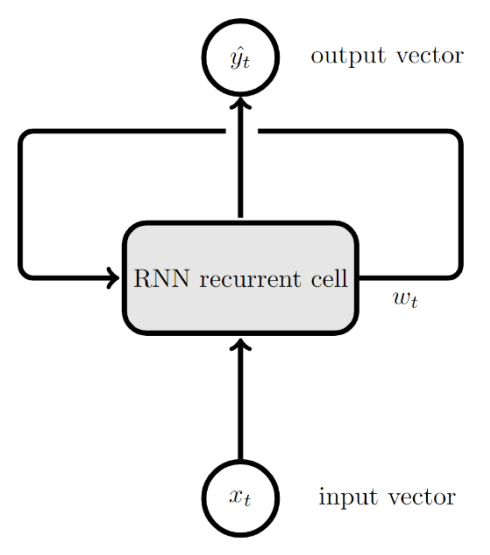
6.2 Layers and architectures

6.2.1

RNN building blocks

The first formulation of a recurrent-like neural network was created by John Hopfield in 1982.

The information is transformed into a vector that can be processed by a machine. The RNN then processes the vector sequence one at a time. When processing each vector, it passes through the previous hidden state. The hidden state stores information from the previous step, acting as a memory type. This is done by combining the input and the previous hidden state with a tanh function that compresses values between -1 and 1.



6.2.2

Which of the following statements are true about the building blocks of an RNN?

- The RNN processes a sequence of vectors one at a time.
- The hidden state stores information from previous steps.
- RNNs use a ReLU activation function to compress values.
- The hidden state only stores the current input vector.

6.2.3

In feed-forward neural networks, data propagates in one direction only, that is, from input to output. This is good approach for single input you need to process (such as image data seen in CNNs previously) but it does not work well for a sequence of data. RNNs are particularly suitable to handle cases where you have an input sequence instead of a single input. These are important for problems in which data sequences are transmitted to give a single output.

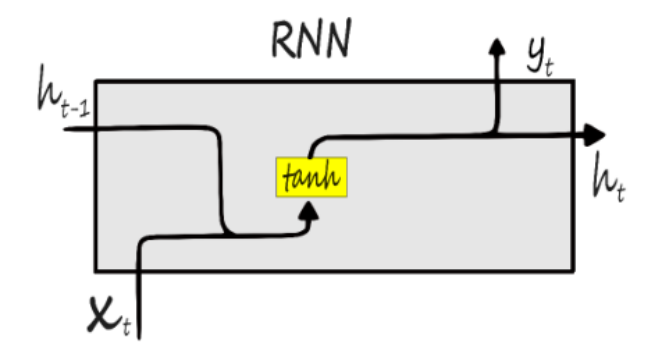
Simply put, RNNs are networks that offer a mechanism to persist previously processed data over time and use it to make future predictions. It provides information about the previous step to the next one. This mechanism is called

recurrent because information is being passed from one time step to the next within the network.

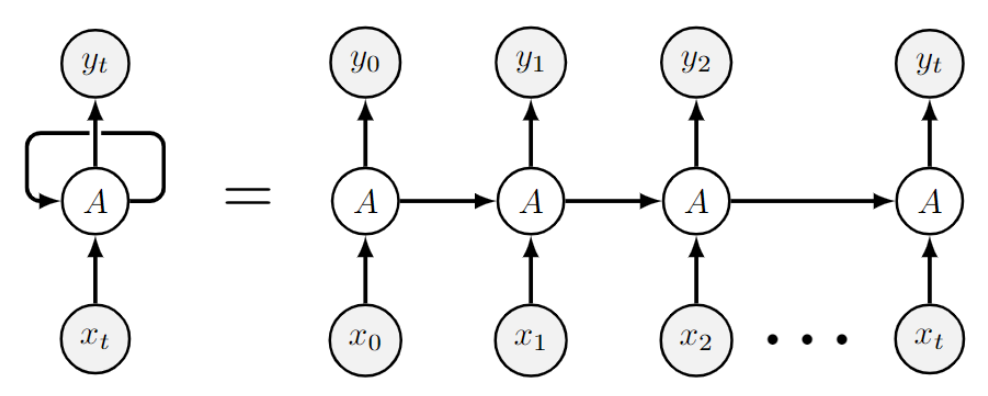
RNN maintains the inner state H_t, combine it with the next input data X_{t+1}, make a prediction, Y_{t+1}, and store the new inner state H_{t+1}.The key idea is that state update is a combination of the previous state time step and the current input received by the network.

Given an example:

1. At the start RNN is initialized altogether with the hidden state of that network. You can indicate a sentence in which you are interested in predicting the next word. The RNN calculation consists simply of them moving through the words in this sentence.
2. At each time step, you include both the current word you're considering, and the previous hidden state of your RNN in the network. This can then generate a prediction for the next word in the sequence and use this information to update its hidden state.
3. Finally, after you have passed through all the words of the sentence, your prediction for this missing word is simply the output of the RNN at this last step of time.



As can be seen in the previous image the non-linear activation function is applied to get new state h_t and the output y_t.



6.2.4

What is the key mechanism that allows RNNs to handle sequential data?

- Information is passed from one time step to the next within the network.
- Information is passed from input to output only.
- Each time step processes data independently.
- Data is stored in a fixed layer for all inputs.

6.2.5

The vanishing gradient problem

The vanishing gradient problem in RNNs is a common issue that arises during the training process. When backpropagating the error through multiple layers of the network, the gradients, which are used to update the weights, tend to become very small. This is especially problematic for long sequences of data, as the gradients shrink exponentially with each time step. The underlying cause of this issue is the repeated multiplication of the weight matrix during backpropagation. If the weight matrix has eigenvalues less than 1, the gradients diminish rapidly as they are propagated backward through time, making it challenging for the network to learn from long-term dependencies.

The vanishing gradient problem can severely hinder the performance of an RNN, particularly in tasks where context from distant time steps is essential for making accurate predictions. In natural language processing, for instance, understanding the meaning of a sentence often requires knowledge of earlier words or phrases. Similarly, in speech recognition, the context of earlier sounds is critical for correctly identifying later parts of the speech. If the gradients vanish during training, the model may fail to capture such long-term dependencies, leading to poor performance on tasks that require temporal understanding.

To mitigate the vanishing gradient problem, several modifications to the standard RNN architecture have been proposed. One of the most successful solutions is the Long Short-Term Memory (LSTM) network. LSTMs use a specialized gating mechanism to regulate the flow of information through the network, allowing them to retain important information for longer periods and avoid the problem of vanishing gradients. Another approach is the Gated Recurrent Unit (GRU), which is similar to LSTMs but with a simpler structure. Both LSTMs and GRUs are widely used in practice and have demonstrated significant improvements in handling long-term dependencies in sequential data.

Despite the advancements with LSTMs and GRUs, the vanishing gradient problem remains a fundamental challenge in training RNNs. Understanding and addressing this issue is crucial for effectively applying RNNs to a wide range of sequential tasks, such as language modeling, machine translation, and time series forecasting.

Researchers continue to explore alternative architectures and training techniques that can further alleviate the vanishing gradient problem, improving the efficiency and accuracy of RNN-based models.

6.2.6

What is the primary cause of the vanishing gradient problem in RNNs?

- Gradients shrink exponentially as they are backpropagated through time.
- The weight matrix has eigenvalues greater than 1.
- The network uses a fixed learning rate.
- The network lacks sufficient layers.

6.2.7

Long short-term memory

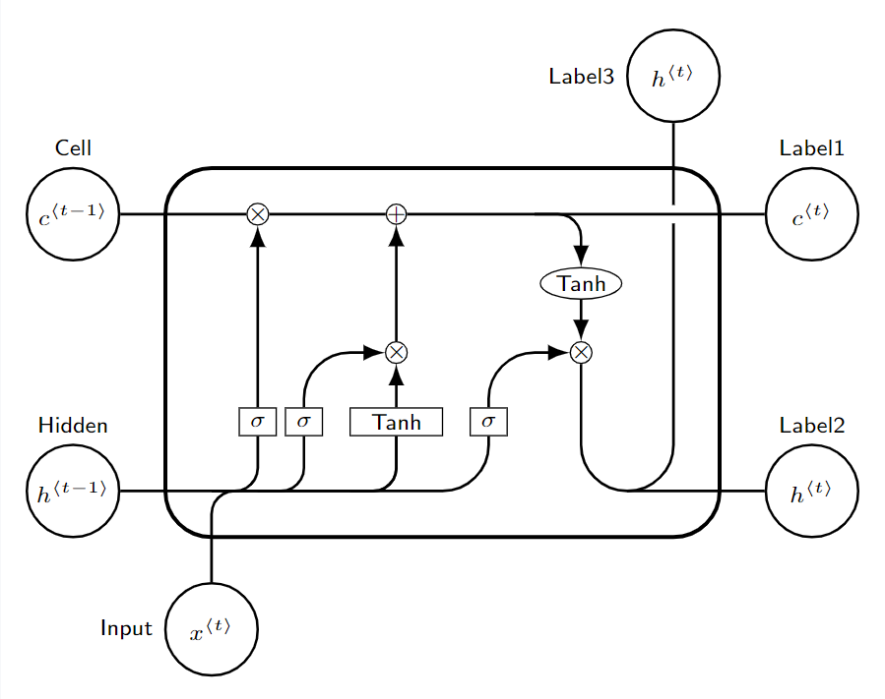
Long short-term memory (LSTM) networks are a type of RNN designed to address the vanishing gradient problem, a common issue in traditional RNNs. In standard RNNs, the gradients used for weight updates can become very small as they are backpropagated through many time steps, making it difficult for the model to learn long-term dependencies. LSTM networks overcome this challenge by introducing an internal memory state that can store information for long periods, allowing them to retain critical data over long sequences. This ability to maintain long-term memory makes LSTMs particularly effective for tasks like natural language processing, speech recognition, and time series analysis.

One of the main differences between LSTM cells and traditional RNN cells is the presence of memory states and gates that control the flow of information. At each time step, an LSTM cell takes in three inputs: the current input, the previous hidden state, and the previous memory state. It then processes these inputs using three key gates: the forget gate, the input gate, and the output gate. The forget gate determines how much of the previous memory state should be discarded, ensuring that the model does not retain unnecessary or outdated information. The input gate controls how much of the current input should be used to update the memory state, allowing the model to incorporate new information. Finally, the output gate controls how much of the memory state should be passed to the next time step, enabling the model to make predictions based on the current context.

These gates and the memory state make LSTM cells highly effective at learning and retaining long-term dependencies in sequential data. For example, in natural language processing, LSTM networks can remember the meaning of words from earlier in a sentence, which is crucial for understanding the sentence as a whole.

Similarly, in speech recognition, LSTMs can retain information about previous sounds, which is important for recognizing the current word or phoneme accurately.

LSTMs have become a standard tool in deep learning for handling sequential data and have significantly improved the performance of models in various fields. They are widely used in applications like machine translation, sentiment analysis, and even video analysis, where understanding temporal dependencies is essential for making accurate predictions. Their ability to maintain and manipulate memory over long sequences makes LSTMs an essential component in modern deep learning architectures.



6.2.8

What is the primary function of the gates in an LSTM cell?

- They control the flow of information into and out of the memory state.
- They determine how much of the previous hidden state to keep.
- They define the sequence length for training.
- They prevent overfitting in the model.

📖 6.2.9

Steps in LSTM

The processing steps in an LSTM cell are crucial for its ability to manage long-term dependencies in sequential data. These steps control how information flows through the cell, allowing it to remember important details over time and forget irrelevant ones. Here's a breakdown of the key steps in the LSTM cell:

1. **Forget** - in this step, the LSTM cell decides which information from the previous memory state should be discarded. This is done using the *forget gate*. The forget gate looks at the previous hidden state and the current input to produce a value between 0 and 1, indicating how much of the previous memory should be retained. A value of 0 means "forget everything," while a value of 1 means "keep everything."
2. **Store** - the LSTM cell then decides what new information should be stored in the memory. This is done using the *input gate*. The input gate takes the current input and the previous hidden state to decide which part of the new information should be added to the memory state. This step helps the cell learn and remember new information over time, contributing to the model's ability to handle long-term dependencies.
3. **Update** - after storing new information, the memory state is updated. The previous memory state is combined with the new information that was stored in the previous step. The forget gate's output determines how much of the previous memory is kept, while the input gate decides how much of the new input is added. The result is a new memory state that includes both retained and updated information.
4. **Generate** - finally, the LSTM cell generates an output based on the current memory state. This is done using the *output gate*. The output gate controls how much of the memory state should be passed on to the next time step, which could be the next LSTM cell or the final output of the model. The output is typically passed through an activation function (like the *tanh* function) to produce a value between -1 and 1.

These processing steps - forget, store, update, and generate - work together to allow the LSTM to learn and maintain important information over time, making it highly effective for tasks like language modeling, time series prediction, and speech recognition.

6.2.10

What is the primary function of the "forget" step in an LSTM cell?

- To decide which information from the previous memory state should be discarded.
- To store new information in the memory.
- To generate the output for the next time step.
- To combine the previous hidden state with the current input.

6.2.11

Architectures

RNNs have several known architectures that are commonly used for various tasks. Here are some of the most well-known architectures:

1. **Simple RNN** is the simplest form of RNN and consists of a single layer of recurrent neurons. It is used for simple sequential tasks, such as language modeling and stock price prediction.
2. **LSTM (Long Short-Term Memory)** was developed to address the vanishing gradient problem in simple RNNs. It has an internal memory cell and three gates (input, forget, and output) that control the flow of information through the network. LSTMs are commonly used for tasks such as speech recognition and text classification.
3. **GRU (Gated Recurrent Unit)** is similar to the LSTM but has fewer parameters. It has two gates (reset and update) that control the flow of information through the network. GRUs are commonly used for tasks such as language modeling and machine translation.
4. **Bidirectional RNN** processes the input sequence in both forward and backward directions and combines the outputs to produce a final output. It is commonly used for tasks such as speech recognition and sentiment analysis.
5. **Encoder-Decoder** consists of two RNNs: an encoder network that processes the input sequence and a decoder network that generates the output sequence. It is commonly used for tasks such as machine translation and image captioning.
6. **Attention-based RNN** uses an attention mechanism to selectively focus on parts of the input sequence that are relevant to the current output. It is commonly used for tasks such as machine translation and text summarization.

Overall, the choice of architecture depends on the specific task and the properties of the input and output sequences.

6.2.12

Which of the following RNN architectures is designed to address the vanishing gradient problem and includes an internal memory cell with three gates (input, forget, and output)?

- LSTM (Long Short-Term Memory)
- Simple RNN
- GRU (Gated Recurrent Unit)
- Bidirectional RNN

6.3 Practical examples with RNNs

6.3.1

Project: ANN on sequential data - Nvidia stock price prediction

Apply an artificial neural network (ANN) to predict the stock price of Nvidia using historical stock data. You will preprocess the data, design a neural network, and evaluate the model's performance.

This is example of regular ANN used for sequential data

Dataset:

- original: https://raw.githubusercontent.com/PacktWorkshops/The-TensorFlow-Workshop/master/Chapter09/Exercise9.01/NVDA.csv
- local: https://priscilla.fitped.eu/data/deep_learning/NVDA.csv

```
import warnings
warnings.filterwarnings("ignore")
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

Load data from csv file

```
import io
import requests
url="https://priscilla.fitped.eu/data/deep_learning/NVDA.csv"
data = pd.read_csv(url)
```

Show data head and tail

```
print(data.head())
```

Program output:

```
         Date        Open        High     Low       Close  Adj
Close    Volume
0  2015-07-22  19.650000  19.650000  19.17  19.410000
18.851749  8911800
1  2015-07-23  19.450001  19.940001  19.41  19.650000
19.084845  4247900
2  2015-07-24  19.790001  19.809999  19.34  19.420000
18.861464  4721100
3  2015-07-27  19.250000  19.530001  19.09  19.309999
18.754622  4810500
4  2015-07-28  19.360001  19.860001  19.16  19.730000
19.162542  4957700
```

```
print(data.tail())
```

Program output:

```
            Date        Open        High         Low
Close   Adj Close  \
1254  2020-07-15  416.570007  417.320007  402.230011
409.089996  409.089996
1255  2020-07-16  400.600006  408.269989  395.820007
405.390015  405.390015
1256  2020-07-17  409.019989  409.940002  403.510010
408.059998  408.059998
1257  2020-07-20  410.970001  421.250000  406.269989
420.429993  420.429993
1258  2020-07-21  420.519989  422.399994  411.470001
413.140015  413.140015

        Volume
1254  10099600
1255   8624100
1256   6657100
1257   7121300
1258   6925900
```

```
# Split Training data
data_training = data[data['Date']<'2019-01-01'].copy()
```

```
# Split Testing data
data_test = data[data['Date']>='2019-01-01'].copy()
```

```
training_data = data_training.drop\
                (['Date', 'Adj Close'], axis = 1)
print(training_data.head())
```

Program output:

```
        Open       High    Low      Close   Volume
0  19.650000  19.650000  19.17  19.410000  8911800
1  19.450001  19.940001  19.41  19.650000  4247900
2  19.790001  19.809999  19.34  19.420000  4721100
3  19.250000  19.530001  19.09  19.309999  4810500
4  19.360001  19.860001  19.16  19.730000  4957700
```

Normalisation process

```
scaler = MinMaxScaler()
training_data = scaler.fit_transform(training_data)
```

```
X_train = []
y_train = []
```

```
print(training_data.shape[0])
```

Program output:

```
868
```

```
for i in range(60, training_data.shape[0]):  # Loop through
the data starting from index 60
    X_train.append(training_data[i-60:i])  # Append a sequence
of 60 previous data points to X_train
    y_train.append(training_data[i, 0])    # Append the
current data point (first feature) to y_train
```

```
X_train, y_train = np.array(X_train), np.array(y_train)  #
Convert X_train and y_train lists into NumPy arrays
X_train.shape, y_train.shape  # Return the shapes of the
training data arrays
print(X_train.shape)  # Print the shape of X_train (features)
print(y_train.shape)  # Print the shape of y_train (target
labels)
```

Program output:

```
(808, 60, 5)
(808,)
```

```
X_old_shape = X_train.shape  # Store the original shape of
X_train
X_train = X_train.reshape(X_old_shape[0],
X_old_shape[1]*X_old_shape[2])  # Flatten the second and third
dimensions into one
print(X_train.shape)  # Print the new shape of X_train after
reshaping
```

Program output:

```
(808, 300)
```

Following code imports the essential components to build a neural network model:

1. **Sequential** is used to create a linear stack of layers for the model.
2. **Input** layer, which can define the shape of the input data (although it is typically inferred in the Sequential model).
3. **Dense** is a fully connected layer, which connects every neuron to every other neuron in the layer.
4. **Dropout** - a regularization technique that helps prevent overfitting by randomly setting a fraction of input units to 0 at each update during training time.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout
```

```
# Model definition
regressor_ann = Sequential()  # Create a sequential model

# Input layer with shape (300,)
regressor_ann.add(Input(shape = (300,)))

# First dense layer with 512 units and ReLU activation,
followed by dropout with rate 0.2
regressor_ann.add(Dense(units = 512, activation = 'relu'))
regressor_ann.add(Dropout(0.2))

# Second dense layer with 128 units and ReLU activation,
followed by dropout with rate 0.3
```

```
regressor_ann.add(Dense(units = 128, activation = 'relu'))
regressor_ann.add(Dropout(0.3))

# Third dense layer with 64 units and ReLU activation,
followed by dropout with rate 0.4
regressor_ann.add(Dense(units = 64, activation = 'relu'))
regressor_ann.add(Dropout(0.4))

# Fourth dense layer with 16 units and ReLU activation,
followed by dropout with rate 0.5
regressor_ann.add(Dense(units = 16, activation = 'relu'))
regressor_ann.add(Dropout(0.5))

# Output layer with 1 unit (regression output)
regressor_ann.add(Dense(units = 1))
```

```
regressor_ann.summary()
```

Program output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               154112

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 128)               65664

 dropout_1 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 64)                8256

 dropout_2 (Dropout)         (None, 64)                0
```

Following line of code compiles the **regressor_ann** model, specifying the following:

- Optimizer: 'adam' is used for training. It's a popular optimization algorithm because it adapts the learning rate during training, often leading to faster convergence.

- Loss function 'mean_squared_error' - is commonly used in regression tasks. It calculates the average of the squared differences between the predicted and actual values. The model aims to minimize this loss during training, improving its predictions.

```
regressor_ann.compile(optimizer='adam', \
                      loss = 'mean_squared_error')
```

Start training

```
regressor_ann.fit(X_train, y_train, epochs=10, batch_size=32)
```

Program output:

```
21/26 [=======================>......] - ETA: 0s - loss:
0.0309
26/26 [==============================] - 0s 6ms/step - loss:
0.0306
Epoch 10/10

 1/26 [>.............................] - ETA: 0s - loss:
0.0214
11/26 [===========>..................] - ETA: 0s - loss:
0.0251
21/26 [=======================>......] - ETA: 0s - loss:
0.0262
26/26 [==============================] - 0s 5ms/step - loss:
0.0270
```

```
## Test and predict stock price
## Prepare test dataset
print(data_test.head())
```

Program output:

```
           Date        Open        High         Low
Close   Adj Close  \
868  2019-01-02  130.639999  138.479996  130.050003
136.220001  135.547104
869  2019-01-03  133.789993  135.160004  127.690002
127.989998  127.357750
870  2019-01-04  130.940002  137.729996  129.699997
136.190002  135.517258
```

```
871  2019-01-07  138.500000  144.889999  136.429993
143.399994  142.691620
872  2019-01-08  146.690002  146.779999  136.899994
139.830002  139.139282

       Volume
868  12718800
869  17638800
870  14640500
871  17729000
872  19650400
```

```
print(data_training.tail(60))
```

Program output:

```
           Date        Open        High         Low
Close   Adj Close  \
808  2018-10-04  285.269989  286.250000  276.179993
279.290009  277.632599
809  2018-10-05  278.290009  280.799988  267.540009
269.859985  268.258514
810  2018-10-08  266.500000  271.160004  260.079987
265.769989  264.192780
811  2018-10-09  264.940002  268.760010  262.799988
265.540009  263.964203
812  2018-10-10  261.260010  263.109985  245.600006
245.690002  244.231964
813  2018-10-11  242.169998  247.559998  234.259995
235.130005  233.734634
814  2018-10-12  245.509995  249.539993  239.649994
246.539993  245.076920
815  2018-10-15  246.000000  246.000000  235.339996
235.380005  233.983154
816  2018-10-16  239.929993  246.279999  237.940002
245.830002  244.371155
817  2018-10-17  248.339996  249.880005  241.080002
243.059998  241.617569
818  2018-10-18  245.860001  247.410004  237.089996
239.529999  238.108536
819  2018-10-19  241.759995  242.550003  227.699997
229.169998  227.810013
820  2018-10-22  231.279999  235.320007  227.070007
231.220001  229.847855
```

```
821  2018-10-23  220.429993  224.190002  216.710007
221.059998  219.748138
822  2018-10-24  219.509995  221.389999  198.850006
199.410004  198.226608
823  2018-10-25  195.470001  209.750000  193.679993
207.839996  206.606567
824  2018-10-26  198.309998  204.839996  193.119995
198.289993  197.113251
825  2018-10-29  203.990005  204.130005  176.009995
185.619995  184.518448
826  2018-10-30  186.550003  203.399994  185.619995
203.000000  201.795303
827  2018-10-31  209.649994  212.589996  204.009995
210.830002  209.578857
828  2018-11-01  212.300003  218.490005  207.190002
218.110001  216.815628
829  2018-11-02  217.729996  222.000000  210.210007
214.919998  213.644562
830  2018-11-05  214.389999  215.330002  205.279999
211.770004  210.513275
831  2018-11-06  211.449997  214.850006  209.559998
211.059998  209.807495
832  2018-11-07  213.750000  217.410004  211.179993
213.789993  212.521271
833  2018-11-08  211.399994  211.429993  203.830002
205.990005  204.767578
834  2018-11-09  202.399994  209.320007  201.039993
205.669998  204.449463
835  2018-11-12  201.979996  202.869995  188.660004
189.539993  188.415192
836  2018-11-13  193.490005  204.210007  193.240005
199.309998  198.127213
837  2018-11-14  206.300003  206.880005  192.830002
197.190002  196.019791
838  2018-11-15  196.949997  205.300003  195.500000
202.389999  201.188919
839  2018-11-16  163.320007  170.660004  161.610001
164.429993  163.454178
840  2018-11-19  161.789993  161.820007  144.630005
144.699997  143.841278
841  2018-11-20  134.059998  154.259995  133.309998
149.080002  148.195297
842  2018-11-21  154.619995  155.300003  143.610001
144.710007  143.851242
```

```
843  2018-11-23  143.309998  149.589996  142.789993
145.000000  144.139511
844  2018-11-26  149.889999  153.470001  146.559998
153.050003  152.141739
845  2018-11-27  152.000000  157.009995  150.550003
153.729996  152.817703
846  2018-11-28  158.479996  160.279999  153.130005
160.070007  159.120102
847  2018-11-29  160.000000  161.500000  156.139999
157.360001  156.582672
848  2018-11-30  157.750000  163.860001  155.720001
163.429993  162.622681
849  2018-12-03  172.600006  174.679993  167.339996
170.039993  169.200043
850  2018-12-04  168.240005  168.440002  156.500000
157.110001  156.333908
851  2018-12-06  151.440002  158.490005  150.809998
158.289993  157.508072
852  2018-12-07  158.460007  158.869995  145.619995
147.610001  146.880844
853  2018-12-10  145.800003  152.860001  145.649994
151.860001  151.109848
854  2018-12-11  155.559998  155.889999  145.000000
148.190002  147.457977
855  2018-12-12  148.419998  152.779999  144.820007
148.899994  148.164474
856  2018-12-13  150.789993  153.380005  147.440002
148.889999  148.154495
857  2018-12-14  147.210007  150.589996  145.500000
146.449997  145.726563
858  2018-12-17  145.240005  148.149994  141.240005
143.580002  142.870728
859  2018-12-18  145.350006  150.330002  144.250000
146.940002  146.214142
860  2018-12-19  145.580002  147.740005  136.429993
138.509995  137.825806
861  2018-12-20  138.169998  141.800003  132.690002
135.100006  134.432632
862  2018-12-21  136.169998  137.500000  128.460007
129.570007  128.929977
863  2018-12-24  126.489998  129.979996  124.500000
127.080002  126.452255
864  2018-12-26  128.940002  133.139999  124.459999
133.100006  132.442535
```

```
865  2018-12-27  130.990005  132.380005  125.180000
131.169998  130.522049
866  2018-12-28  132.000000  137.389999  130.309998
133.649994  132.989807
867  2018-12-31  135.399994  136.710007  132.259995
133.500000  132.840530

       Volume
808   9780500
809  10665900
810  10215300
811   6837500
812  17123500
813  18135900
814  15205900
815  11244000
816  10217800
817   8241700
818  13100500
819  15340200
820   9221100
821  15660900
822  22107200
823  23793000
824  16619600
825  18950400
826  20179800
827  18644300
828  14163200
829  11324000
830   9483300
831   7475300
832  12095300
833  12783800
834  10331000
835  15427900
836  16117800
837  13164500
838  21017700
839  49088000
840  42445500
841  42300800
842  25637400
843  10299200
```

```
844  20370800
845  18451500
846  20113100
847  13729300
848  18239100
849  22270100
850  20302800
851  17307700
852  17041900
853  15736800
854  16797800
855  16353400
856  11784600
857  11795500
858  16571700
859  14109300
860  18634100
861  18739700
862  21593500
863  11596000
864  17377500
865  15926100
866  15718200
867  11628500
```

```
past_60_days = data_training.tail(60)
```

```
# Get the last 60 days of data from training
past_60_days = data_training.tail(60)

# Concatenate past 60 days with the test data
df = pd.concat([past_60_days, data_test], ignore_index=True)

# Drop 'Date' and 'Adj Close' columns if they exist
df = df.drop(['Date', 'Adj Close'], axis=1, errors='ignore')

# Scale the features using the already fitted scaler
inputs = scaler.transform(df)
```

```
# Initialize X_test and y_test
X_test = []
y_test = []

# Create the sequences for X_test and corresponding y_test
```

```
for i in range(60, inputs.shape[0]):
    X_test.append(inputs[i-60:i])  # 60 previous days as
features
    y_test.append(inputs[i, 0])    # Target is the first
feature (e.g., 'Open' or the first column)

# Convert X_test and y_test to numpy arrays
X_test, y_test = np.array(X_test), np.array(y_test)

# Check the original shape of X_test before reshaping
print("Original shape of X_test:", X_test.shape)

# Ensure that X_test has 3 dimensions (samples, time_steps,
features)
if len(X_test.shape) == 3:
    # Reshape X_test from (samples, time_steps, features) to
(samples, time_steps * features)
    X_old_shape = X_test.shape
    X_test = X_test.reshape(X_old_shape[0], X_old_shape[1] *
X_old_shape[2])

    # Print the new shape of X_test and y_test
    print("Reshaped shape of X_test:", X_test.shape)
else:
    print("X_test is not in the expected 3D format.")

print("Shape of y_test:", y_test.shape)
```

Program output:

```
Original shape of X_test: (391, 60, 5)
Reshaped shape of X_test: (391, 300)
Shape of y_test: (391,)
```

- **regressor_ann.predict(X_test)** makes predictions using the trained model (regressor_ann) on the test set (X_test).

```
y_pred = regressor_ann.predict(X_test)
```

Program output:

```
 1/13 [=>............................] - ETA: 1s
13/13 [==============================] - 0s 2ms/step
```

```
print(scaler.scale_)
```

Program output:

```
[3.70274364e-03 3.65992009e-03 3.75248621e-03 3.70301815e-03
 1.09875621e-08]
```

- Scale with the maximum value:

```
scale = 1/3.70274364e-03
print(scale)
y_pred = y_pred*scale
y_test = y_test*scale
```

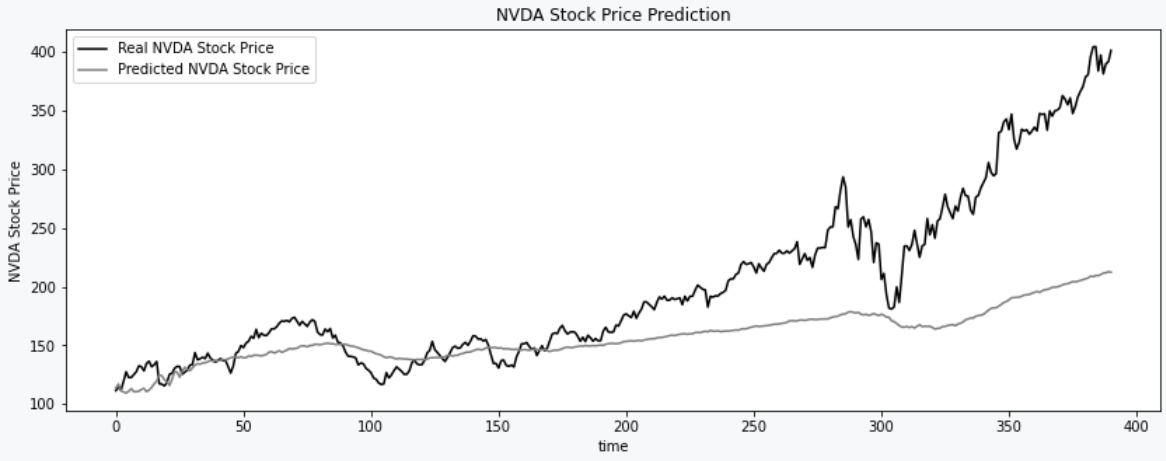
Program output:

```
270.0700067909643
```

Show the result of the predicted stock price. This result is not as good as using the RNN network in the next example.

```
plt.figure(figsize=(14,5))
plt.plot(y_test, color = 'black', label = "Real NVDA Stock
Price")
plt.plot(y_pred, color = 'gray', label = 'Predicted NVDA Stock
Price')
plt.title('NVDA Stock Price Prediction')
plt.xlabel('time')
plt.ylabel('NVDA Stock Price')
plt.legend()
plt.show()
```

Program output:



6.3.2

Project: RNN with LSTM layer - Nvidia stock price prediction

Apply an artificial neural network (RNN with LSTM) to predict the stock price of Nvidia using historical stock data. You will preprocess the data, design a neural network, and evaluate the model's performance.

This example demonstrates using of RNN with LSTM layer for prediction of Nvidia Stock value.

Dataset:

- original: https://raw.githubusercontent.com/PacktWorkshops/The-TensorFlow-Workshop/master/Chapter09/Exercise9.01/NVDA.csv
- local: https://priscilla.fitped.eu/data/deep_learning/NVDA.csv

```
import warnings
warnings.filterwarnings("ignore")
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

Read data and show from the source

```
import io
import requests
url="https://priscilla.fitped.eu/data/deep_learning/NVDA.csv"
data = pd.read_csv(url)
print(data.head())
```

Program output:

```
         Date       Open       High    Low      Close  Adj
Close   Volume
0  2015-07-22  19.650000  19.650000  19.17  19.410000
18.851749  8911800
1  2015-07-23  19.450001  19.940001  19.41  19.650000
19.084845  4247900
2  2015-07-24  19.790001  19.809999  19.34  19.420000
18.861464  4721100
3  2015-07-27  19.250000  19.530001  19.09  19.309999
18.754622  4810500
```

```
4  2015-07-28  19.360001  19.860001  19.16  19.730000
19.162542  4957700
```

```
print(data.tail())
```

Program output:

```
           Date        Open        High         Low
Close   Adj Close  \
1254  2020-07-15  416.570007  417.320007  402.230011
409.089996  409.089996
1255  2020-07-16  400.600006  408.269989  395.820007
405.390015  405.390015
1256  2020-07-17  409.019989  409.940002  403.510010
408.059998  408.059998
1257  2020-07-20  410.970001  421.250000  406.269989
420.429993  420.429993
1258  2020-07-21  420.519989  422.399994  411.470001
413.140015  413.140015

        Volume
1254  10099600
1255   8624100
1256   6657100
1257   7121300
1258   6925900
```

```
# Split Training data
data_training = data[data['Date']<'2019-01-01'].copy()
print(data_training)
```

Program output:

```
           Date        Open        High         Low
Close   Adj Close  \
0    2015-07-22   19.650000   19.650000   19.170000
19.410000   18.851749
1    2015-07-23   19.450001   19.940001   19.410000
19.650000   19.084845
2    2015-07-24   19.790001   19.809999   19.340000
19.420000   18.861464
3    2015-07-27   19.250000   19.530001   19.090000
19.309999   18.754622
4    2015-07-28   19.360001   19.860001   19.160000
19.730000   19.162542
```

```
..          ...         ...         ...         ...
...         ...
863  2018-12-24  126.489998  129.979996  124.500000
127.080002  126.452255
864  2018-12-26  128.940002  133.139999  124.459999
133.100006  132.442535
865  2018-12-27  130.990005  132.380005  125.180000
131.169998  130.522049
866  2018-12-28  132.000000  137.389999  130.309998
133.649994  132.989807
867  2018-12-31  135.399994  136.710007  132.259995
133.500000  132.840530

       Volume
0     8911800
1     4247900
2     4721100
3     4810500
4     4957700
..        ...
863  11596000
864  17377500
865  15926100
866  15718200
867  11628500

[868 rows x 7 columns]
```

```
# Split Testing data
data_test = data[data['Date']>='2019-01-01'].copy()
print(data_test)
```

Program output:

```
            Date        Open        High         Low
Close   Adj Close  \
868   2019-01-02  130.639999  138.479996  130.050003
136.220001  135.547104
869   2019-01-03  133.789993  135.160004  127.690002
127.989998  127.357750
870   2019-01-04  130.940002  137.729996  129.699997
136.190002  135.517258
871   2019-01-07  138.500000  144.889999  136.429993
143.399994  142.691620
```

```
872   2019-01-08  146.690002  146.779999  136.899994
139.830002  139.139282
...          ...         ...         ...         ...
...         ...
1254  2020-07-15  416.570007  417.320007  402.230011
409.089996  409.089996
1255  2020-07-16  400.600006  408.269989  395.820007
405.390015  405.390015
1256  2020-07-17  409.019989  409.940002  403.510010
408.059998  408.059998
1257  2020-07-20  410.970001  421.250000  406.269989
420.429993  420.429993
1258  2020-07-21  420.519989  422.399994  411.470001
413.140015  413.140015

        Volume
868   12718800
869   17638800
870   14640500
871   17729000
872   19650400
...        ...
1254  10099600
1255   8624100
1256   6657100
1257   7121300
1258   6925900

[391 rows x 7 columns]
```

```
training_data = data_training.drop(['Date', 'Adj Close'], axis
= 1)
print(training_data.head())
```

Program output:

```
         Open        High    Low       Close   Volume
0  19.650000  19.650000  19.17  19.410000  8911800
1  19.450001  19.940001  19.41  19.650000  4247900
2  19.790001  19.809999  19.34  19.420000  4721100
3  19.250000  19.530001  19.09  19.309999  4810500
4  19.360001  19.860001  19.16  19.730000  4957700
```

```
scaler = MinMaxScaler()
```

```
training_data = scaler.fit_transform(training_data)
print(training_data)
```

Program output:

```
[[1.48109745e-03 4.39186751e-04 3.00198896e-04 3.70305518e-04
  8.35120643e-02]
 [7.40552430e-04 1.50056724e-03 1.20079559e-03 1.25902987e-03
  3.22671736e-02]
 [1.99948527e-03 1.02477031e-03 9.38121551e-04 4.07335700e-04
  3.74664879e-02]
 ...
 [4.13744593e-01 4.13021997e-01 3.98101261e-01 4.14219607e-01
  1.60582121e-01]
 [4.17484345e-01 4.31358175e-01 4.17351508e-01 4.23403077e-01
  1.58297807e-01]
 [4.30073651e-01 4.28869458e-01 4.24668845e-01 4.22847646e-01
  1.13361974e-01]]
```

```
X_train = []
y_train = []
print(training_data.shape[0])
```

Program output:

```
868
```

```
for i in range(60, training_data.shape[0]):
  X_train.append(training_data[i-60:i])
  y_train.append(training_data[i, 0])
```

```
X_train, y_train = np.array(X_train), np.array(y_train)
print(X_train.shape)
print(y_train.shape)
```

Program output:

```
(808, 60, 5)
(808,)
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout
```

Definition of a model

```
# Define the model as a Sequential model
regressor = Sequential()

# Add the first LSTM layer with 50 units, relu activation, and
return sequences
regressor.add(LSTM(units=50, activation='relu',
return_sequences=True, input_shape=(X_train.shape[1], 5)))
# Add Dropout layer with 20% rate to prevent overfitting
regressor.add(Dropout(0.2))

# Add the second LSTM layer with 60 units, relu activation,
and return sequences
regressor.add(LSTM(units=60, activation='relu',
return_sequences=True))
# Add Dropout layer with 30% rate
regressor.add(Dropout(0.3))

# Add the third LSTM layer with 80 units, relu activation, and
return sequences
regressor.add(LSTM(units=80, activation='relu',
return_sequences=True))
# Add Dropout layer with 40% rate
regressor.add(Dropout(0.4))

# Add the fourth LSTM layer with 120 units, relu activation
regressor.add(LSTM(units=120, activation='relu'))
# Add Dropout layer with 50% rate
regressor.add(Dropout(0.5))

# Add the output Dense layer with 1 unit (for regression task)
regressor.add(Dense(units=1))
```

Print model layers

```
regressor.summary()
```

Program output:

```
 lstm (LSTM)                 (None, 60, 50)            11200

 dropout (Dropout)           (None, 60, 50)            0
```

```
 lstm_1 (LSTM)                (None, 60, 60)            26640

 dropout_1 (Dropout)          (None, 60, 60)            0

 lstm_2 (LSTM)                (None, 60, 80)            45120
```

```
regressor.compile(optimizer='adam', loss =
'mean_squared_error')
```

Start training

```
regressor.fit(X_train, y_train, epochs=10, batch_size=32)
```

Program output:

```
24/26 [==========================>...] - ETA: 0s - loss:
0.0092
25/26 [==========================>..] - ETA: 0s - loss:
0.0096
26/26 [==============================] - ETA: 0s - loss:
0.0096
26/26 [==============================] - 3s 115ms/step - loss:
0.0096
Epoch 10/10

 1/26 [>.............................] - ETA: 2s - loss:
0.0051
 2/26 [=>............................] - ETA: 2s - loss:
0.0109
 3/26 [==>...........................] - ETA: 2s - loss:
0.0127
 4/26 [===>..........................] - ETA: 2s - loss:
0.0114
 5/26 [====>.........................] - ETA: 2s - loss:
0.0101
 6/26 [=====>........................] - ETA: 2s - loss:
0.0091
 7/26 [=======>......................] - ETA: 2s - loss:
0.0093
 8/26 [========>.....................] - ETA: 2s - loss:
0.0096
```

```
 9/26 [=========>....................] - ETA: 1s - loss:
0.0100
10/26 [==========>....................] - ETA: 1s - loss:
0.0100
11/26 [===========>....................] - ETA: 1s - loss:
0.0101
12/26 [============>....................] - ETA: 1s - loss:
0.0099
13/26 [==============>.............] - ETA: 1s - loss:
0.0100
14/26 [===============>.............] - ETA: 1s - loss:
0.0104
15/26 [================>.............] - ETA: 1s - loss:
0.0100
16/26 [=================>............] - ETA: 1s - loss:
0.0097
17/26 [==================>...........] - ETA: 1s - loss:
0.0094
18/26 [====================>.........] - ETA: 0s - loss:
0.0092
19/26 [=====================>........] - ETA: 0s - loss:
0.0095
20/26 [======================>.......] - ETA: 0s - loss:
0.0097
21/26 [========================>.....] - ETA: 0s - loss:
0.0096
22/26 [=========================>....] - ETA: 0s - loss:
0.0094
23/26 [==========================>...] - ETA: 0s - loss:
0.0096
24/26 [===========================>..] - ETA: 0s - loss:
0.0097
25/26 [============================>.] - ETA: 0s - loss:
0.0098
26/26 [==============================] - ETA: 0s - loss:
0.0098
26/26 [==============================] - 3s 115ms/step - loss:
0.0098
```

```
# Get the last 60 days of data from training
past_60_days = data_training.tail(60)

# Concatenate past 60 days with the test data
df = pd.concat([past_60_days, data_test], ignore_index=True)
```

```
# Drop 'Date' and 'Adj Close' columns if they exist
df = df.drop(['Date', 'Adj Close'], axis=1, errors='ignore')

# Scale the features using the already fitted scaler
inputs = scaler.transform(df)
print(inputs)
```

Program output:

```
[[0.98500382 0.97617388 0.96472665 0.9627107  0.09305696]
 [0.95915875 0.95622728 0.93230523 0.92779115 0.10278535]
 [0.91550336 0.9209457  0.9043116  0.91264582 0.09783435]
 ...
 [1.44321835 1.42886941 1.44253078 1.4395483  0.05873841]
 [1.45043874 1.4702631  1.45288757 1.48535462 0.06383883]
 [1.4857999  1.47447198 1.47240054 1.4583597  0.06169186]]
```

```
X_test = []
y_test = []

for i in range(60, inputs.shape[0]):
  X_test.append(inputs[i-60:i])
  y_test.append(inputs[i, 0])

X_test, y_test = np.array(X_test), np.array(y_test)
print(X_test.shape, y_test.shape)
```

Program output:

```
(391, 60, 5) (391,)
```

```
y_pred = regressor.predict(X_test)
```

Program output:

```
 1/13 [=>............................] - ETA: 10s
 2/13 [===>..........................] - ETA: 0s
 3/13 [=====>........................] - ETA: 0s
 4/13 [========>.....................] - ETA: 0s
 5/13 [==========>...................] - ETA: 0s
 7/13 [==============>.............] - ETA: 0s
 8/13 [================>...........] - ETA: 0s
 9/13 [==================>.........] - ETA: 0s
10/13 [====================>.......] - ETA: 0s
11/13 [======================>.....] - ETA: 0s
12/13 [========================>...] - ETA: 0s
```

```
13/13 [==============================] - 1s 51ms/step
```

```
print(scaler.scale_)
```

Program output:

```
[3.70274364e-03 3.65992009e-03 3.75248621e-03 3.70301815e-03
 1.09875621e-08]
```

Get the scaling factor

```
scale = 1/3.70274364e-03
print(scale)
```

Program output:

```
270.0700067909643
```

```
# scale the data
y_pred = y_pred*scale
y_test = y_test*scale

print(y_pred[:10])
```

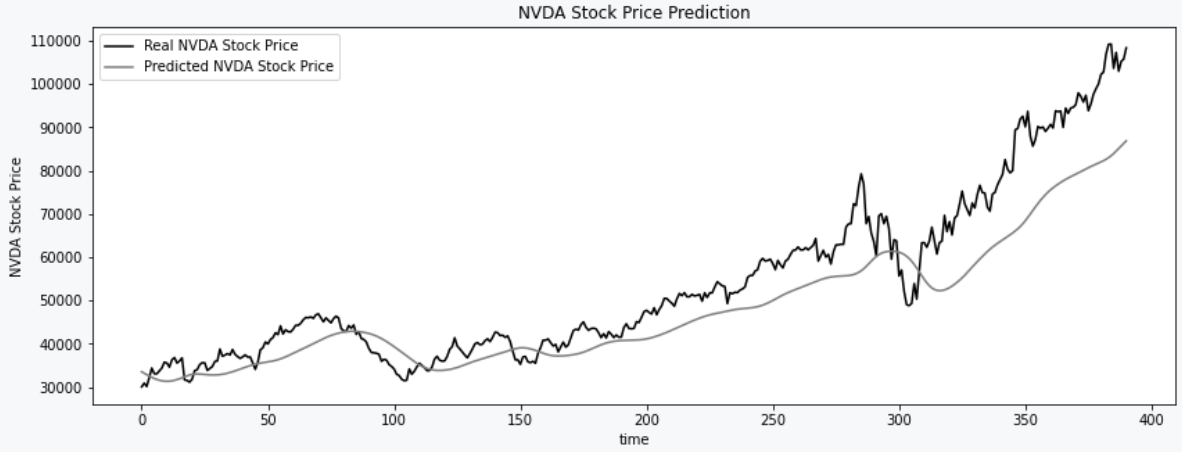
Program output:

```
[[33555.266]
 [33222.266]
 [32894.14 ]
 [32577.812]
 [32283.715]
 [32020.54 ]
 [31795.977]
 [31615.73 ]
 [31484.248]
 [31404.578]]
```

```
plt.figure(figsize=(14,5))
plt.plot(y_test, color = 'black', label = "Real NVDA Stock
Price")
plt.plot(y_pred, color = 'gray', label = 'Predicted NVDA Stock
Price')
plt.title('NVDA Stock Price Prediction')
plt.xlabel('time')
```

```
plt.ylabel('NVDA Stock Price')
plt.legend()
plt.show()
```

Program output:



Generative Models

Chapter 7

7.1 Overview

🕮 7.1.1

Generative models are a fundamental category of machine learning models that focus on creating new data samples that resemble the patterns found in the training data. Unlike predictive models, which forecast outcomes based on input data, generative models are designed to capture the underlying distribution of the data. This allows them to generate entirely new examples that are indistinguishable from real data.

Generative models rely on many of the techniques used in other areas of deep learning, such as preprocessing data, fine-tuning hyperparameters, and leveraging neural network architectures like CNNs and RNNs. These shared methods ensure a smooth transition familiar with predictive models. However, the objective of generative models is broader, aiming to model complex distributions rather than making specific predictions.

The ability of generative models to create new, realistic samples has opened up a world of possibilities. From generating lifelike images to creating entirely new pieces of music, these models play a pivotal role in advancing artificial intelligence. They also demonstrate the powerful interplay between data science and creativity, making them an essential topic for anyone studying deep learning.

By understanding generative models, you can explore innovative applications across various fields. The concepts learned from predictive models provide a strong foundation for tackling this exciting area of machine learning.

📝 7.1.2

Which of the following best describes the primary goal of generative models?

- To generate new data samples similar to training data
- To make accurate predictions based on input data
- To identify anomalies in real-time datasets
- To optimize hyperparameters for neural networks

📝 7.1.3

Generative models differ from predictive models because they:

- Aim to model and recreate the data distribution
- Focus on understanding the relationships within input data

- Are exclusively used for creative industries
- Do not rely on neural network architectures like CNNs or RNNs

📖 7.1.4

Generative models are applied in numerous domains, showcasing their versatility in handling complex tasks. These applications span creative industries, technical fields, and practical solutions, proving their relevance in real-world scenarios.

One of the most visible uses of generative models is in **media creation**. They are used to generate realistic images of people, animals, and objects, often blurring the line between real and artificial. In text generation, these models can produce coherent narratives for chatbots, text summarization, or dialogue systems. Music generation further exemplifies their creativity by composing original pieces or mimicking specific musical styles.

Generative models enhance machine learning by generating synthetic data for training, particularly when real data is limited. This process, known as data augmentation, helps improve the accuracy and robustness of predictive models. Additionally, anomaly detection benefits from generative models by identifying unusual patterns that deviate from the expected data distribution.

Beyond creation and augmentation, generative models have applications in style transfer, where they transform images or videos into different artistic styles while preserving their content. They are also used in simulations, enabling the modeling of complex phenomena like weather patterns, traffic flow, or biological systems such as protein folding.

📝 7.1.5

Which application of generative models involves creating new training data to improve a model's accuracy?

- Data augmentation
- Anomaly detection
- Style transfer
- Simulation

📖 7.1.6

Typical applications of generative models in deep learning

1. Image generation - generative models can be used to generate realistic images of faces, animals, objects, and scenes.
2. Text generation - generative models can generate natural language text for applications such as chatbots, text summarization, and dialogue systems.
3. Music generation - generative models can create new music pieces based on existing songs or styles.
4. Video generation - generative models can create video sequences with realistic motions and actions.
5. Data augmentation - generative models can be used to generate synthetic data for training deep learning models and improving their performance.
6. Anomaly detection - generative models can detect anomalies in datasets by learning the distribution of normal data and identifying samples that do not conform to it.
7. Style transfer - generative models can transform images or videos to different styles while preserving their content.
8. Simulation - generative models can simulate complex physical or biological systems, such as weather patterns, traffic flow, or protein folding.

📝 7.1.7

A generative model is used to transform an image into a different artistic style while retaining its content. This application is known as:

- Style transfer
- Text generation
- Simulation
- Data augmentation

📝 7.1.8

Which of the following is NOT an application of generative models in deep learning?

- Predicting stock prices
- Image generation
- Data augmentation
- Music generation

7.1.9

Select the correct applications of generative models in deep learning:

- Text summarization
- Weather simulation
- Anomaly detection
- Object detection

7.2 Generative models applications

7.2.1

Text generation

Natural Language Processing (NLP) is a subfield of computer science and artificial intelligence that deals with the interactions between computers and human languages. It involves the ability of machines to read, understand, and interpret human language in the form of text or speech, and to generate natural language responses in turn. NLP technologies are used in a variety of applications, such as language translation, sentiment analysis, chatbots, speech recognition, and text summarization or text generation

Some common steps of pre-processing data for training model include data cleaning, transformation, and data reduction.

- Dataset cleaning encompasses the conversion of the case to lowercase, removing punctuation.
- Tokenization in NLP is the process of breaking down text into smaller units called tokens. The tokens are essentially words or phrases, which can be further used for analysis, processing, or generating new text. Tokenization can be performed at different levels such as word level, subword level, or character level, depending on the requirements of the NLP task. In practice, tokenization involves various steps such as splitting text into sentences, removing punctuation, converting text to lowercase, and splitting words into individual tokens. Tokenization is a fundamental step in many NLP tasks such as text classification, named entity recognition, and machine translation.
- In NLP, padding is a technique used to make all the text sequences of the same length. It is done by adding a special token (usually a zero or a PAD token) at the end of the shorter sentences, so that all the sequences have the same length. Padding is necessary for training neural networks on text data because the networks require fixed-size inputs, and if the inputs are of different lengths, it can cause issues during training. Once the padding is

done, the padded sequences can be fed to the neural network for further processing.
- Stemming is a technique used in NLP to reduce a word to its root form, called a stem. This is done by removing the suffixes (endings) of words, which may be different forms of the same root word. For example, "running," "ran," and "runner" all have the same root word "run," and stemming would reduce all of these words to the same stem "run." The goal of stemming is to reduce the complexity of text data and to group together similar words so that they can be treated as a single entity during text analysis. Stemming is often used as a pre-processing step before other NLP tasks, such as text classification or sentiment analysis.

7.2.2

What is the purpose of tokenization in Natural language processing?

- To split text into smaller units like words or phrases
- To remove padding from sequences
- To reduce words to their root forms
- To transform all text to uppercase

7.2.3

Which of the following are common pre-processing steps in NLP?

- Dataset cleaning
- Tokenization
- Stemming
- Adding punctuation

7.2.4

Generative adversarial networks

Generative Adversarial Networks (GANs) are a fascinating class of deep learning models that have revolutionized the field of generative modeling. A GAN consists of two main components: a generator and a discriminator, both of which are neural networks. The generator's role is to create data samples, such as images, audio, or text, starting from random noise. Meanwhile, the discriminator acts as a classifier, distinguishing between real data (from a dataset) and fake data (produced by the generator). This unique setup pits the two networks against each other in a dynamic and iterative training process.

The training of GANs involves an adversarial game between the generator and the discriminator. Initially, the generator creates data that is clearly unrealistic, as it is only beginning to learn the distribution of the training data. The discriminator, on the other hand, is trained to identify these generated samples as fake. Over successive iterations, the generator learns to produce data that increasingly resembles real samples, while the discriminator becomes more adept at identifying subtle differences between real and fake data. The interplay between these two networks drives the learning process.

One of the critical challenges in training GANs is achieving a balance between the generator and discriminator. If the discriminator becomes too strong, it can easily detect fake data, causing the generator to struggle to improve. Conversely, if the generator outpaces the discriminator, the discriminator may fail to learn effectively. Techniques such as tweaking the loss functions, adjusting learning rates, and using advanced architectures can help address these challenges and ensure stable training.

GANs have found applications across numerous domains due to their ability to generate high-quality synthetic data. In computer vision, GANs are used to create photorealistic images, enhance image resolution, and generate artworks. In audio processing, GANs have been applied to synthesize realistic speech or music. In text generation, GANs contribute to tasks such as creating natural language content or augmenting datasets for improved model training. These versatile applications demonstrate the power and potential of GANs in solving real-world problems.

Despite their success, GANs are not without limitations. Training GANs can be computationally expensive and prone to instability, requiring careful tuning of hyperparameters. Additionally, GANs sometimes generate outputs that are visually appealing but lack diversity, a problem known as mode collapse. Addressing these limitations is an active area of research, with new variants of GANs being proposed to overcome these challenges and expand their applications further.

7.2.5

What is the primary role of the discriminator in a GAN?

- To classify data as real or fake
- To generate data samples
- To adjust the generator's learning rate
- To increase the diversity of generated data

7.2.6

Which of the following are typical applications of GANs?

- Image generation
- Speech synthesis
- Data compression
- Hyperparameter tuning

7.2.7

Deep convolutional generative adversarial networks

Deep Convolutional Generative Adversarial Networks (DCGANs) are an extension of Generative Adversarial Networks (GANs) that incorporate deep convolutional neural networks (CNNs) in their architecture. Introduced in 2015, DCGANs were designed to improve the generation of high-quality images, addressing some of the limitations of traditional GANs in capturing fine details and spatial dependencies in visual data.

In a DCGAN, both the generator and discriminator networks are built using convolutional layers. The generator takes a random noise vector as input and transforms it into an image through a series of up-sampling operations using transpose convolutional layers. This process allows the generator to learn and replicate the spatial features that define real images, resulting in outputs that are visually coherent and realistic. Meanwhile, the discriminator, also a deep CNN, classifies whether an input image is real (from the dataset) or fake (produced by the generator) using a series of down-sampling operations.

The key advantage of using convolutional layers in DCGANs lies in their ability to model spatial hierarchies. Convolutional operations allow the generator to build images with consistent textures, patterns, and structures, while the discriminator learns to identify nuanced differences between real and synthetic images. This architecture makes DCGANs particularly effective for tasks where visual quality is critical.

DCGANs have been successfully applied in various image generation tasks. For example, they have been used in image synthesis to generate entirely new images that resemble real-world objects. In image inpainting, DCGANs can fill in missing parts of an image in a visually consistent way, which is useful for repairing damaged photos. Additionally, they are employed in style transfer, where they enable the transformation of an image's visual style while preserving its content.

Despite their strengths, DCGANs face challenges similar to traditional GANs, including mode collapse and instability during training. However, their use of convolutional architectures has significantly advanced the field of generative

modeling, making them a foundational approach for tasks that require high-quality image generation.

7.2.8

What distinguishes DCGANs from traditional GANs?

- Use of deep convolutional layers in both networks
- Use of random noise as input
- Focus on text generation
- Lack of a discriminator network

7.2.9

Which tasks are DCGANs particularly suited for?

- Image synthesis
- Image inpainting
- Style transfer
- Text summarization

7.2.10

Deepfake

Deepfake technology leverages AI and deep learning techniques to create synthetic media, such as videos, images, or audio recordings. These can either be entirely fabricated or manipulated versions of authentic content, making them appear genuine to the human eye or ear. The term "deepfake" originates from the use of deep learning algorithms in generating these falsified outputs. While deepfakes can be incredibly convincing, they also raise significant ethical and security concerns.

One of the primary challenges posed by deepfake technology is its potential for misuse. Deepfakes have been employed in the creation of fake news, leading to misinformation and public distrust. They have also been used for impersonation, allowing malicious actors to convincingly mimic someone’s appearance or voice. Non-consensual uses, such as creating deepfake pornography, represent another alarming misuse of this technology. These applications highlight the urgent need for regulation, detection technologies, and public awareness about deepfakes.

On the other hand, deepfake technology also offers positive applications. In the film and entertainment industry, deepfakes can be used to bring historical figures to life,

de-age actors, or create realistic special effects. Similarly, deepfake-based simulations are valuable in training and education, where realistic scenarios help improve engagement and learning outcomes. For instance, they could simulate complex conversations or historical reenactments in a classroom setting.

DCGANs can play a role in generating deepfakes, especially in creating synthetic images or video frames. Although DCGANs are not explicitly designed for this purpose, their ability to generate high-quality and realistic images makes them a useful component in a broader pipeline. For example, DCGANs might generate realistic facial features, which can then be blended into video content to create a deepfake.

While DCGANs contribute to the technical foundations of deepfake generation, their use emphasizes the dual-edged nature of AI technologies. They exemplify how advanced AI can simultaneously drive innovation and raise ethical questions about its applications.

7.2.11

What is major concern associated with deepfake technology?

- Potential misuse for creating fake news or impersonations
- Lack of realistic simulations
- Limited use in entertainment
- Inability to use AI in the generation process

7.2.12

Which applications of deepfake technology can be considered positive?

- Generating realistic simulations for training
- De-aging actors in films
- Creating non-consensual content
- Misinforming the public through fake news

7.3 Examples of generative models

7.3.1

Project: Generating a text using RNN

Prepare the code uses a generative model (LSTM) to generate text based on a dataset of article headlines. The steps shoul be included data preprocessing, tokenization, padding, building the neural network model, training it, and finally using it for text generation.

Dataset:

- original: https://raw.githubusercontent.com/PacktWorkshops/The-TensorFlow-Workshop/master/Chapter09/Datasets/Articles.csv
- local: https://priscilla.fitped.eu/data/deep_learning/articles.csv

```
import warnings
warnings.filterwarnings("ignore")
from keras.utils import pad_sequences
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
import tensorflow.keras.utils as ku
from keras.preprocessing.text import Tokenizer
import pandas as pd
import numpy as np
from keras.callbacks import EarlyStopping
import string, os
```

2. Fetching and cleaning the dataset

```
url="https://priscilla.fitped.eu/data/deep_learning/articles.csv"
data = pd.read_csv(url)
print(data.columns)

our_headlines = []
our_headlines.extend(list(data.headline.values))

our_headlines = [h for h in our_headlines if h != "Unknown"]
print(len(our_headlines))
```

Program output:

```
Index(['abstract', 'articleID', 'articleWordCount', 'byline',
'documentType',
       'headline', 'keywords', 'multimedia', 'newDesk',
'printPage', 'pubDate',
       'sectionName', 'snippet', 'source', 'typeOfMaterial',
'webURL'],
      dtype='object')
831
```

- Remove punctuation.
- Convert text to lowercase.
- Ensure that only ASCII characters remain.

```
def clean_text(txt):
    txt = "".join(v for v in txt if v not in
string.punctuation).lower()
    txt = txt.encode("utf8").decode("ascii",'ignore')
    return txt

corpus = [clean_text(x) for x in our_headlines]
print(corpus[60:80])
```

Program output:

```
['lets go for a win on opioids', 'floridas vengeful governor',
'how to end the politicization of the courts', 'when dr king
came out against vietnam', 'britains trains dont run on time
blame capitalism', 'questions for no license plates here using
art to transcend prison walls', 'dry spell', 'are there
subjects that should be offlimits to artists or to certain
artists in particular', 'that is great television', 'thinking
in code', 'how gorsuchs influence could be greater than his
vote', 'new york today how to ease a hangover', 'trumps gifts
to china', 'at penn station rail mishap spurs large and
lasting headache', 'chemical attack on syrians ignites worlds
outrage', 'adventure is still on babbos menu', 'swimming in
the fast lane', 'a national civics exam', 'obama adviser is
back in the political cross hairs', 'the hippies have won']
```

3. Tokenization

- The Tokenizer is fit on the corpus to assign each unique word a corresponding integer.
- The function **get_seq_of_tokens** creates sequences of tokens (word representations) from the corpus.
- It splits the text into n-grams (sequences of words), which are used for training the model.

```
tokenizer = Tokenizer()

def get_seq_of_tokens(corpus):
    ## tokenization
    tokenizer.fit_on_texts(corpus)
    all_words = len(tokenizer.word_index) + 1

    ## convert data to sequence of tokens
    input_seq = []
    for line in corpus:
        token_list = tokenizer.texts_to_sequences([line])[0]
        for i in range(1, len(token_list)):
            n_gram_sequence = token_list[:i+1]
            input_seq.append(n_gram_sequence)
    return input_seq, all_words

our_sequences, all_words = get_seq_of_tokens(corpus)
print(our_sequences[:20])
```

Program output:

```
[[169, 17], [169, 17, 665], [169, 17, 665, 367], [169, 17,
665, 367, 4], [169, 17, 665, 367, 4, 2], [169, 17, 665, 367,
4, 2, 666], [169, 17, 665, 367, 4, 2, 666, 170], [169, 17,
665, 367, 4, 2, 666, 170, 5], [169, 17, 665, 367, 4, 2, 666,
170, 5, 667], [6, 80], [6, 80, 1], [6, 80, 1, 668], [6, 80, 1,
668, 10], [6, 80, 1, 668, 10, 669], [670, 671], [670, 671,
129], [670, 671, 129, 672], [673, 674], [673, 674, 368], [673,
674, 368, 675]]
```

4. Padding sequences and preparing data

- This step ensures all input sequences are of the same length by padding shorter sequences with zeros (using pad_sequences).
- The function also separates the last token as the label (the next word to predict) and the rest as predictors.

- One-hot encoding - labels are converted into one-hot encoded vectors using to_categorical, making them suitable for classification tasks.

```
def generate_padded_sequences(input_seq):
    max_sequence_len = max([len(x) for x in input_seq])
    input_seq = np.array(pad_sequences\
                         (input_seq, maxlen=max_sequence_len,
\
                          padding='pre'))

    predictors, label = input_seq[:,:-1],input_seq[:,-1]
    label = ku.to_categorical(label, num_classes=all_words)
    return predictors, label, max_sequence_len

predictors, label, max_sequence_len =
generate_padded_sequences(our_sequences)
```

5. Model creation

- The input layer is an embedding layer, which learns a dense representation for words in a lower-dimensional space (10 dimensions here).
- A LSTM layer is added to capture long-term dependencies in the text data. A dropout layer is also included to prevent overfitting.
- The final dense layer uses softmax activation to output a probability distribution over the vocabulary, predicting the next word.
- The model is compiled with the categorical cross-entropy loss function and the Adam optimizer.

```
def create_model(max_sequence_len, all_words):
    input_len = max_sequence_len - 1
    model = Sequential()

    # Add Input Embedding Layer
    model.add(Embedding(all_words, 10,
input_length=input_len))

    # Add Hidden Layer 1 - LSTM Layer
    model.add(LSTM(100))
    model.add(Dropout(0.1))

    # Add Output Layer
    model.add(Dense(all_words, activation='softmax'))
```

```
    model.compile(loss='categorical_crossentropy',
optimizer='adam')

    return model

model = create_model(max_sequence_len, all_words)
model.summary()
```

Program output:

```
Model: "sequential"
_________________________________________________________________
___
 Layer (type)                 Output Shape              Param #
=================================================================
===
 embedding (Embedding)        (None, 18, 10)            24220

 lstm (LSTM)                  (None, 100)               44400

 dropout (Dropout)            (None, 100)               0

 dense (Dense)                (None, 2422)              244622

=================================================================
===
```

6. Training the model

- The model is trained on the predictors and label using 200 epochs. During training, the model adjusts its parameters to minimize the loss function.

```
model.fit(predictors, label, epochs=200, verbose=5)
```

Program output:

```
Epoch 194/200
Epoch 195/200
Epoch 196/200
Epoch 197/200
Epoch 198/200
Epoch 199/200
Epoch 200/200
```

7. Text generation

- The **generate_text function** generates a sequence of words by predicting one word at a time based on the seed text.
- It uses the trained model to predict the next word, appends it to the seed text, and repeats the process until the desired number of words (**next_words**) is generated.

```
def generate_text(seed_text, next_words, model,
max_sequence_len):
    for _ in range(next_words):
        token_list =
tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], \
                                   maxlen=max_sequence_len-1,
\
                                   padding='pre')
        predicted = model.predict(token_list, verbose=0)

        output_word = ""
        for word,index in tokenizer.word_index.items():
            if index == predicted.any():
                output_word = word
                break
        seed_text += " "+output_word
    return seed_text.title()
```

8. Output

- The function is called with different seed texts to generate unique sentences.

```
print(generate_text("10 Ways", 11, model, max_sequence_len))
print(generate_text("europe looks to", 8, model,
max_sequence_len))
print(generate_text("best way", 10, model, max_sequence_len))
print(generate_text("homeless in", 10, model,
max_sequence_len))
print(generate_text("Unexpected results", 10, model, \
                    max_sequence_len))
print(generate_text("critics warn", 10, model,
max_sequence_len))
```

Program output:

```
10 Ways The The The The The The The The The The The
```

```
Europe Looks To The The The The The The The The
Best Way The The The The The The The The The The
Homeless In The The The The The The The The The
Unexpected Results The The The The The The The The The
Critics Warn The The The The The The The The The
```

The result you are seeing, where the model generates text like "10 Ways The The The The The The The The The," is a common issue in text generation tasks, particularly when working with models that generate text one word at a time. The repeated "The" is a sign of over-reliance on the most frequent word in the vocabulary. By introducing diversity in the sampling method and improving the training data, you can encourage the model to generate more varied and coherent text.

Resources

Chapter 8

8.1 Bibliography

🕮 8.1.1

Literature

1. Artificial Neural Networks (Part 1) - Classification Using Single Layer Perceptron Model - http://scholastictutors.webs.com/
2. Becker, D.: Deep Learning in Python, https://www.datacamp.com/courses/deep-learning-in-python
3. Blaha, M.: Umělá inteligence - http://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence
4. Hinton, G.: Neural Networks for Machine Learning, University of Toronto. https://www.coursera.org/learn/neural-networks/home/welcome
5. Kvasnička, V.: Neurónové siete http://www2.fiit.stuba.sk/~kvasnicka/NeuralNetworks/index.html
6. Sinčák, P., Andrejková, G.: Neurónové siete - Inžiniersky prístup (1. diel) http://neuron-ai.tuke.sk/cig/source/publications/books/NS1/html/index.html
7. Volná, E.: Neuronové sítě 1 - https://www1.osu.cz/~volna/Neuronove_site_skripta.pdf

Notable sources

1. https://github.com/khoih-prog/deeplearning-models
2. https://raw.githubusercontent.com/PacktWorkshops/The-TensorFlow-Workshop/master/

🕮 8.1.2

Statement regarding the use of Artificial Intelligence in content creation

This content has been developed with the assistance of artificial intelligence tools, specifically ChatGPT, Gemini, and Notebook LM. These AI technologies were utilized to enhance the text by providing suggestions for rephrasing, improving clarity, and ensuring coherence throughout the material. The integration of these AI tools has enabled a more efficient content creation process while maintaining high standards of quality and accuracy.

The use of AI in this context adheres to all relevant guidelines and ethical considerations associated with the deployment of such technologies. We acknowledge the importance of transparency in the content creation process and

aim to provide a clear understanding of how artificial intelligence has contributed to the final product.

PRISCILLA

priscilla.fitped.eu