Deep Learning - introduction

Published on

Work in progress version

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

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

Funded by
the European Union

The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

TABLE OF CONTENTS

Introduction to Neural Networks

Chapter 1

1.1 Introduction to neural networks

🕮 1.1.1

Neural networks are currently considered by many to be one of the best machine learning algorithms. Why do we need machine learning algorithms?

It is necessary to realize that there are areas where the "classic" programming approach is insufficient.

For example, it is very difficult to write programs that solve problems such as recognizing a three-dimensional object from multiple perspectives in different lighting and in a “crowded scene.” Why? Because we don't know how to write such a program at all. Because we don't know how it is "done" even in our brain. Even if we had a good idea of how to do it, the program would be very complicated.

Also, for example, it is very difficult to write a program to calculate the probability that a certain credit card transaction is fraudulent. There are probably no rules that are simple and reliable. It is necessary to combine a very large number of weak rules.

Scammers are changing "tactics". The program must be constantly changing.

These are the tasks for machine learning and neural networks. Instead of writing a program by hand for each specific task, we collect many examples that determine the correct output for a given input. A machine learning algorithm then takes these examples and creates a program that does the work.

A program created by a learning algorithm can look very different from a typical handwritten program. If we do it right, the program works correctly for new cases as well as for those cases we haven't trained. If the situation changes, the program can also be changed by training to new data. The cost of "huge calculations" is often cheaper than the cost of a team of programmers to create a "classic" program for a given task.

🕮 1.1.2

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.

1.1.3

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

- Recognition of persons
- Calculate 1014 * 1024

1.1.4

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.

1.1.5

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).

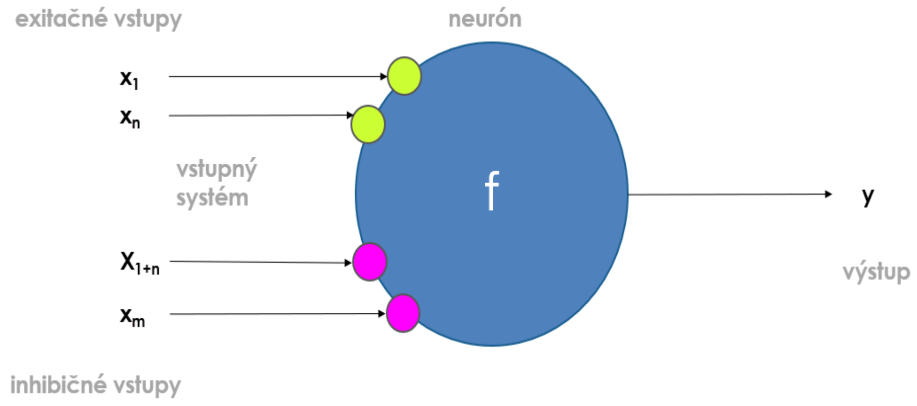
The logic neuron system contains both **excitatory** inputs (described by binary variables x_1, x_2, ..., x_n, which amplify the response) and **inhibitory** inputs (described by binary variables x_{n+1}, x_{n+2}, ..., x_m, which weaken response).

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.

1.1.6

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** (i. e. 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**.

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

1.1.7

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:

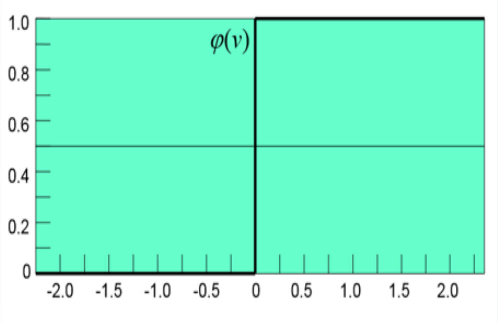
$$y = \varphi(v) = \ \varphi\ (x_1 + \ldots + x_n - x_{1+n} - \cdots - x_m \boldsymbol{- b})$$

when:

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

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:



1.1.8

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

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

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

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

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 $\boldsymbol{w_i}$

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

whereas:

$$w_i = \begin{cases} 1 & (spoj\ i \rightarrow j\ má\ excitačný\ charakter) \\ -1 & (spoj\ i \rightarrow j\ má\ inhibičný\ charakter) \\ 0 & (spoj\ i \rightarrow j\ neexistuje) \end{cases}$$

We can write the resulting activity of the neuron as:

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

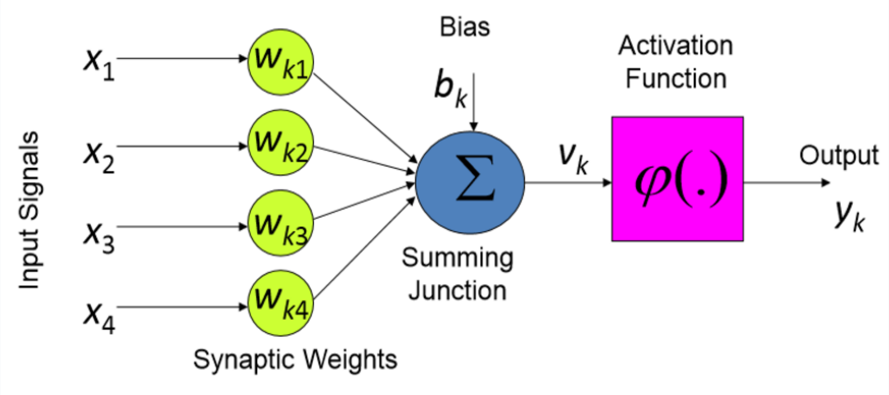
1.1.9

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.

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



The neuron model consists of a set of synapses characterized by their thickness or **weight**.

A neuron can also be expressed as:

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

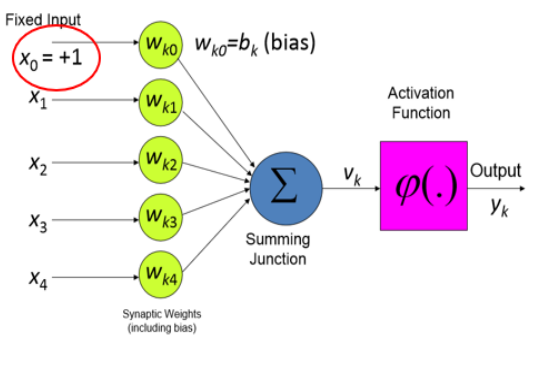
while the internal potential is defined as follows:

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

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.

$$y\_in = \sum_{j=0}^{m} w_{kj} x_j$$

$$y_k = \varphi(v_k)$$



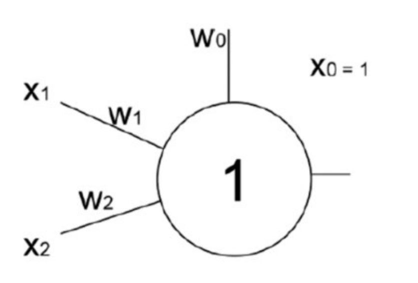
1.2 Implementation of Boolean binary functions

1.2.1

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:

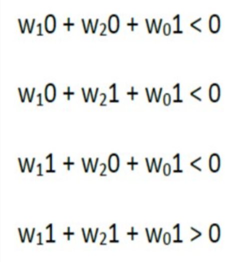
x_1	x_2	$x_1 \wedge x_2$
0	0	0
1	0	0
0	1	0
1	1	1

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)



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

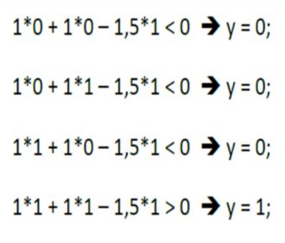
$w_1 0 + w_2 1 + w_0 1 < 0$

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

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

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

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



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

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

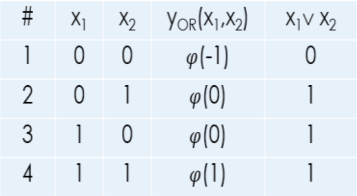
$1^*1 + 1^*0 - 1{,}5^*1 < 0 \rightarrow y = 0;$

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

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**.

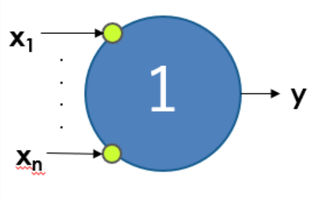
1.2.2

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

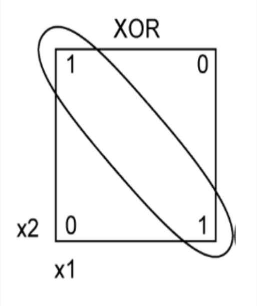
$$y_{OR}(x_1,x_2) = \varphi\ (x_1 + x_2 - 0{,}5)$$



1.2.3

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:

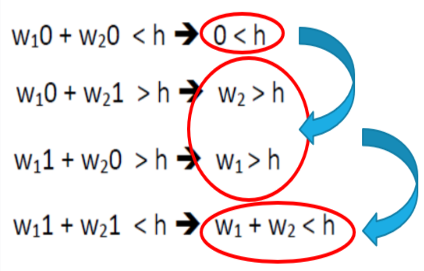
$w_1 0 + w_2 0 < h \rightarrow 0 < h$

$w_1 0 + w_2 1 > h \rightarrow w_2 > h$

$w_1 1 + w_2 0 > h \rightarrow w_1 > h$

$w_1 1 + w_2 1 < h \rightarrow w_1 + w_2 < h$

However, this system of equations has no solution. Since ***h*** is a positive number, ***w₂*** and ***w₁*** are greater than ***h***. Therefore, their sum cannot be less than ***h***.



$w_1 0 + w_2 0 < h \rightarrow 0 < h$

$w_1 0 + w_2 1 > h \rightarrow w_2 > h$

$w_1 1 + w_2 0 > h \rightarrow w_1 > h$

$w_1 1 + w_2 1 < h \rightarrow w_1 + w_2 < h$

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

1.2.4

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

Definition:

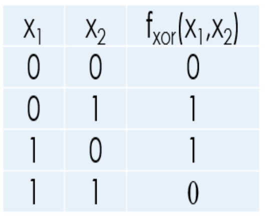
The Boolean function $\boldsymbol{f(x_1, x_2,..., x_n)}$ is linearly separable if there is such a plane $\boldsymbol{w_1x_1 + w_2x_2 + ...+ w_nx_n - J = 0}$, that separates the space of input activities such that there are vertices in one part of the space rated ***0***, while in the other part of the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

Theorem:

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

1.2.5

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.

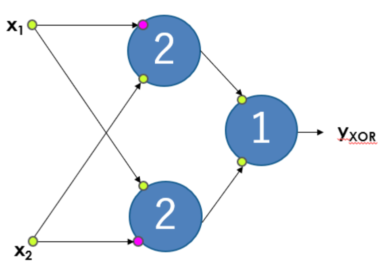


x_1	x_2	$f_{xor}(x_1,x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

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.

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.

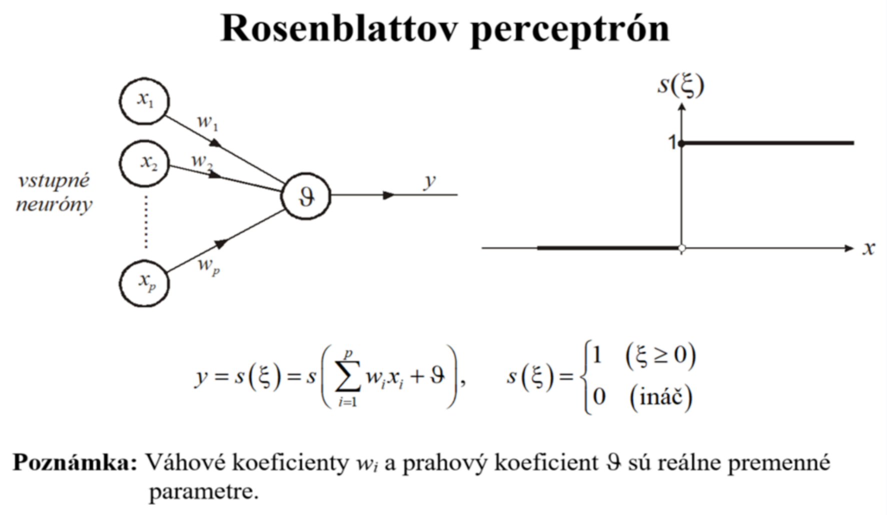
However, the neuron can also be taught. During active dynamics, the neuron performs the transformation of the input vectors to the output value. The parameters of the neuron are constant at this moment. On the other hand, adaptive dynamics is a process whose task is to set these parameters of the neuron 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

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.



Rosenblattov perceptrón

$$y = s(\xi) = s\left(\sum_{i=1}^{p} w_i x_i + \vartheta\right), \qquad s(\xi) = \begin{cases} 1 & (\xi \geq 0) \\ 0 & (\text{ináč}) \end{cases}$$

Poznámka: Váhové koeficienty w_i a prahový koeficient ϑ sú reálne premenné parametre.

2.1.3

The scales therefore represent the memory of the neuron. The learning of the neuron is provided by an adaptation algorithm that sets the value of the neuron's weights. This process usually takes place iteratively based on the presented examples, when the adaptation algorithm has at its disposal a set of pairs of input values and their corresponding outputs.

Thus, adaptation algorithms proceed similarly to a human when they search for solutions based on analogy with known examples. Setting the neuron's weights thus corresponds to finding the most accurate transformation of input vectors into output vectors based on known input values, with the assumption that the found transformation will be sufficiently general for other unknown examples of the given area.

NN can find transformations even in those tasks that are analytically difficult to solve or unsolvable. They need enough examples. The disadvantage is that the resulting transformation is hidden in the network structure, it cannot be used to explain the solution of the task.

2.1.4

We can divide adaptation algorithms into two basic areas:

- Unsupervised Learning,
- Supervised learning.

In supervised learning, 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.

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

So we have at our disposal examples of correct behavior, correct transformation of input vectors into output vectors. Direct transformation of input values to output values is unknown.

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
- test set.

The ratio of the number of elements in the training and testing set is not fixed.

2.1.5

The training of a neuron (and, after generalization, also of feedforward networks) usually takes place iteratively. The algorithm presents individual elements of the training set successively to the neuron, determines its response to the presented input and, based on the output, performs the correction of the neuron's weights. The **interval** in which **all patterns** of the training set **are presented at least onc**e is called the learning **epoch**.

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.

In order to **evaluate the moment** when it is appropriate to end the adaptation (the transformation found is sufficiently accurate, but at the same time does not lose its generality), we set aside a test set. During the adaptation, the performance of the found transformation is periodically tested on it.

2.1.6

Iterative learning of a neuron with a teacher takes place in the following typical steps:

1. Preprocessing of input data
2. Definition of training and testing set
3. Definition of network structure / neuron parameters
4. Initialize neuron weights, usually random numbers
5. ***n=0***; learning epoch counter
6. THE LEARNING EPOCH
7. Evaluation of the success of the network on the test set, if it is insufficient, go to point 4, ev. 3. Alternatively, terminate the algorithm with failure.

While the following steps are implemented in the LEARNING EPOCH:

- set the learning epoch to ***n = n+1*** and verify that the number of epochs is not greater than max
- ***n***-th learning epoch:
    - select one input vector from the training set (deterministically, randomly),
    - obtaining the response of the neuron, evaluating the classification error based on the comparison of the actual and planned output,
    - correction of neuron weights based on error,
    - if the learning epoch is not complete (not all inputs from the training set have been tested), go to select the next input vector,
    - t the end of the epoch, it evaluates the classification error over the entire training set. If the error is less than the desired error, stop learning.

2.2 Hebbian Learning

2.2.1

Hebbian learning represents the oldest intuitive rule for a neuron with binary inputs and outputs. It was defined in 1949 by Canadian psychologist Donald Hebb while studying conditioned reflexes. Hebb hypothesized that:

- conditioned reflexes are established on the basis of strengthening or weakening of connections between individual neurons,
- if two connected neighboring neurons are active at the same time, the coupling between them is strengthened, while discordant activations weaken it.

Therefore, if the neuron is excited by its inputs correctly, the inputs (their weights) are strengthened, otherwise, incorrect excitations are weakened. Thus, Hebb's rule is based on a neurophysiological analogy.

2.2.2

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 $\boldsymbol{x}$, weights $\boldsymbol{w}$, output $\boldsymbol{y}$ and predicted output $\boldsymbol{y_d}$,the Hebb rule can be written as:

- If the neuron is activated correctly (***y = 1***; ***y_d = 1***), then in the next step ***n + 1*** the ***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; yd = 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)

2.2.3

Another originally heuristic rule that is also applicable to general real inputs and outputs of a neuron is the **Delta rule**:

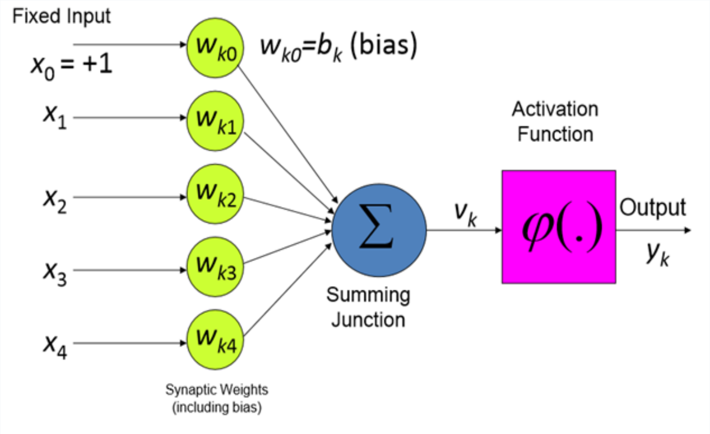
$$_{n+1}w_i = {}_nw_i + \mu * (y_d - y)$$

where ***μ*** is a suitably chosen constant from the interval ***(0,1)*** affecting the adaptation speed.

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.

2.2.4

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***.

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

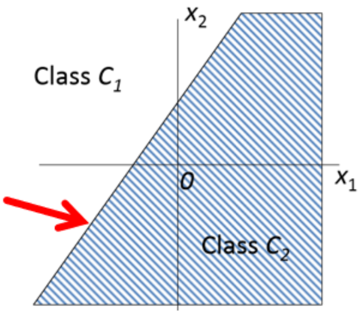
$$\text{y}(t) = \begin{cases} 1 & ak\ u(t) \geq 0 \\ 0 & ak\ u(t) < 0 \end{cases}$$

The separating hyperplane is given by the equation:

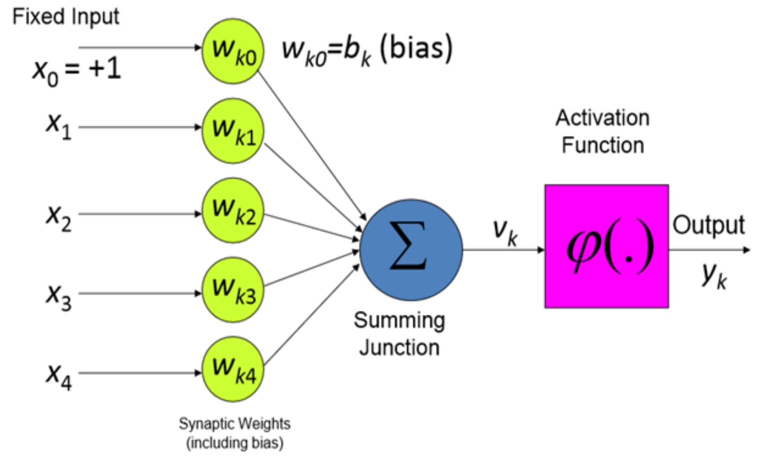
$$\sum_{j=1}^{m} w_j(t)x_j(t) + b = 0$$

alebo

$$w_1x_1 + w_2x_2 + b = 0$$



2.2.5



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

$$w(t) = (w_0(t), w_1(t), w_2(t), \dots, w_n(t)),$$

$$\text{x}(t) = (x_0(t), x_1(t), x_2(t), \dots, x_n(t)),$$

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

In the case of zero input, it holds that:

$$w_0(t) = b$$
$$\text{a}$$
$$x_0(t) = 1$$

Assume a training sample of vectors:

$$\big(x(1), y(1)\big), \big(x(2), y(2)\big), \dots, \big(x(m), y(m)\big),$$
$$\text{kde}$$
$$y(t) = 1 \; ak \; x(t) \in CL1,$$
$$y(t) = -1 \; ak \; x(t) \in CL1$$

2.2.6

Perceptron learning algorithm:

1. initialzation of weights,
2. if the input vector ***x(t)*** is correctly classified by ***w(t)***, then the weights do not change.

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

If the input vector ***x(t)*** is not correctly classified by ***w(t)***, then the weights change.

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

Where γ is the learning parameter and can represent any positive variable that is constant during the learning period or can also change $\gamma \to \gamma(t)$.

Alternatively, perceptron learning can also be expressed by the formula:

$$w(t+1) = w(t) + \gamma[d(t) - y(t)]x(t),$$

whereas:

$$d(t) = \begin{cases} +1 & if \; x(t) \in C1 \\ -1 & if \; x(t) \in C2 \end{cases}$$

2.2.7

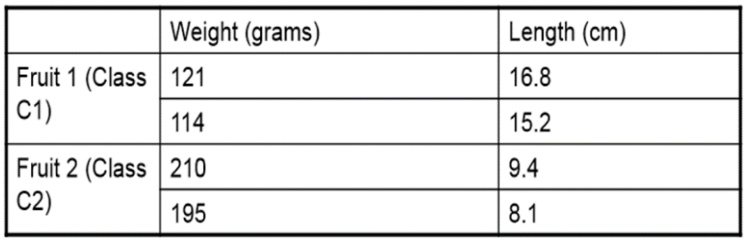
In neural networks, the perceptron convergence theorem holds:

Let us have a training set of vectors ***X*** that can only belong to two different classes **CL1** and **CL2**, which are linearly separable. After realizing "**k**" mistakes, the perceptron will definitely reach a state where it will not change its synaptic weights, when it converges. This means that it will reliably classify the vectors into the appropriate classes.

2.3 Example

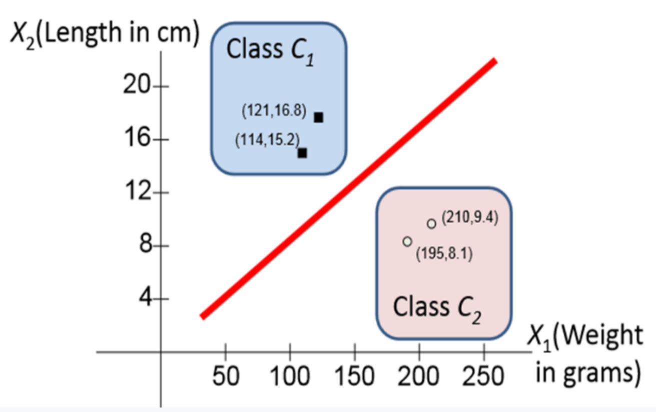
2.3.1

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).

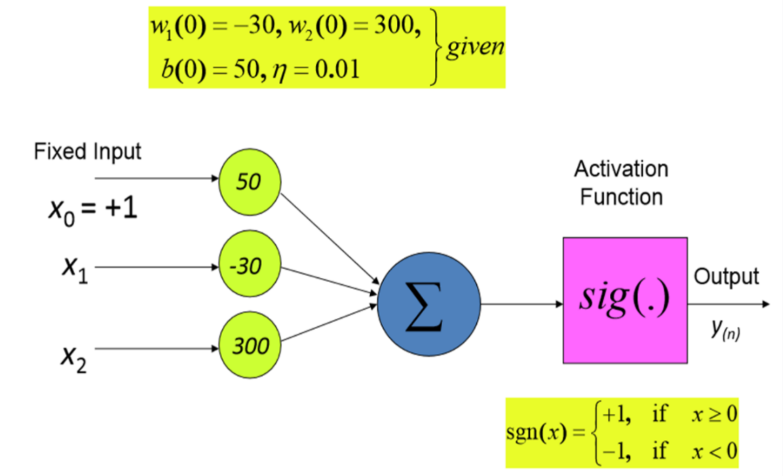


	Weight (grams)	Length (cm)
Fruit 1 (Class C1)	121	16.8
	114	15.2
Fruit 2 (Class C2)	210	9.4
	195	8.1

We can visualize the training set.



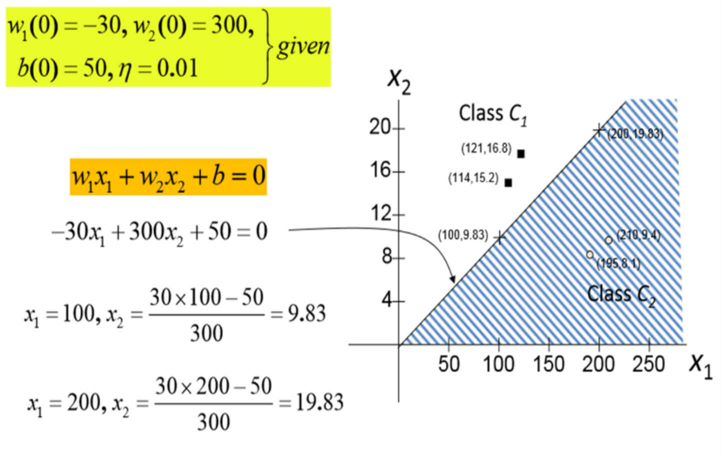
If we knew the correct setting of the weights, then it is obvious that I will also be able to correctly classify any fruit. Assume that we know the correct setting of the scales.



$$\left.\begin{array}{l} w_1(0) = -30,\ w_2(0) = 300, \\ b(0) = 50,\ \eta = 0.01 \end{array}\right\} given$$

$$\operatorname{sgn}(x) = \begin{cases} +1, & \text{if} \quad x \ge 0 \\ -1, & \text{if} \quad x < 0 \end{cases}$$

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



$$\left.\begin{array}{l} w_1(0) = -30,\ w_2(0) = 300, \\ b(0) = 50,\ \eta = 0.01 \end{array}\right\} given$$

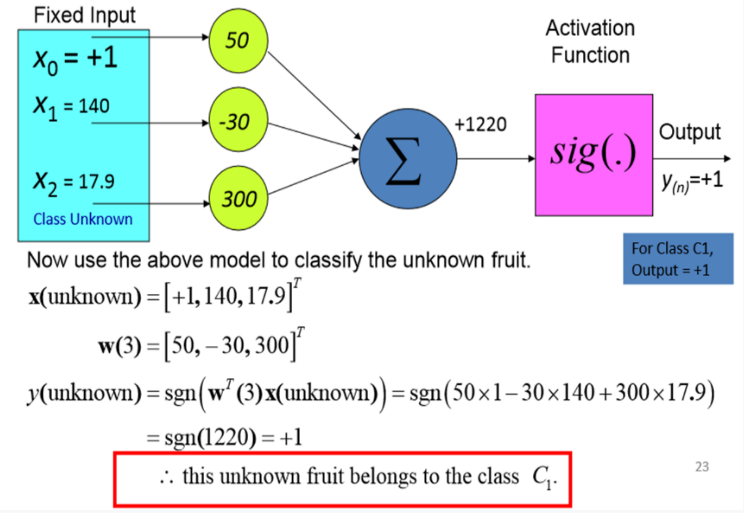
$$w_1 x_1 + w_2 x_2 + b = 0$$

$$-30x_1 + 300x_2 + 50 = 0$$

$$x_1 = 100,\ x_2 = \frac{30 \times 100 - 50}{300} = 9.83$$

$$x_1 = 200,\ x_2 = \frac{30 \times 200 - 50}{300} = 19.83$$

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.



Now use the above model to classify the unknown fruit.

$$\mathbf{x}(\text{unknown}) = [+1, 140, 17.9]^T$$

$$\mathbf{w}(3) = [50, -30, 300]^T$$

$$y(\text{unknown}) = \operatorname{sgn}\left(\mathbf{w}^T(3)\mathbf{x}(\text{unknown})\right) = \operatorname{sgn}\left(50 \times 1 - 30 \times 140 + 300 \times 17.9\right)$$

$$= \operatorname{sgn}(1220) = +1$$

$$\boxed{\therefore \text{ this unknown fruit belongs to the class } C_1.}$$

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

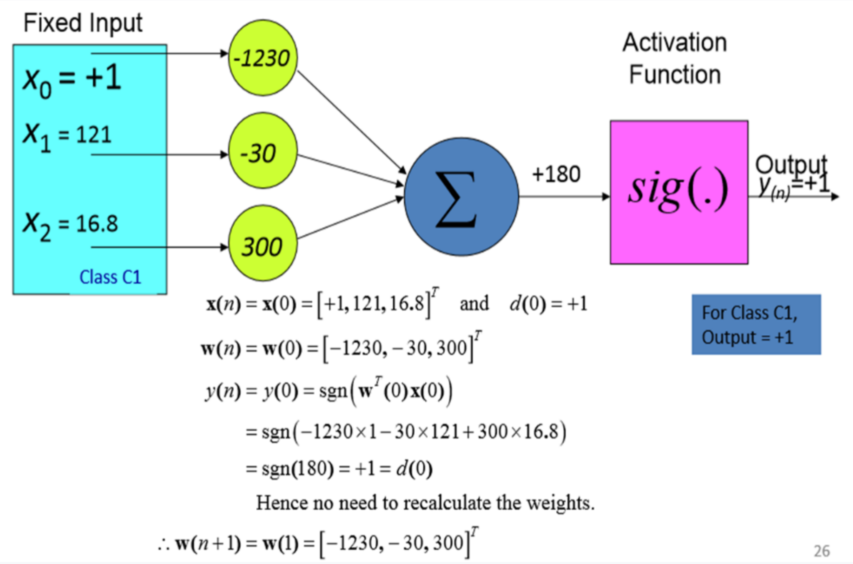
📖 2.3.2

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**.



$$\mathbf{x}(n) = \mathbf{x}(0) = [+1, 121, 16.8]^T \quad \text{and} \quad d(0) = +1$$

$$\mathbf{w}(n) = \mathbf{w}(0) = [-1230, -30, 300]^T$$

$$y(n) = y(0) = \operatorname{sgn}\left(\mathbf{w}^T(0)\mathbf{x}(0)\right)$$

$$= \operatorname{sgn}(-1230\times1 - 30\times121 + 300\times16.8)$$

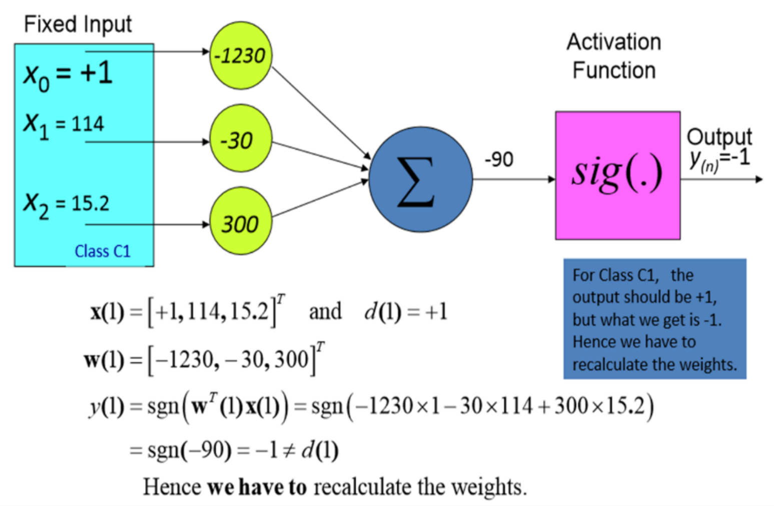
$$= \operatorname{sgn}(180) = +1 = d(0)$$

Hence no need to recalculate the weights.

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

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.



For Class C1, the output should be +1, but what we get is -1. Hence we have to recalculate the weights.

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

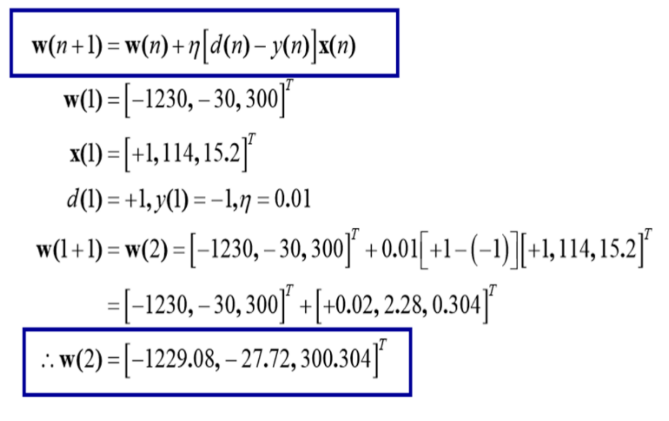
$$\mathbf{w}(1) = [-1230, -30, 300]^T$$

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

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

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

We adjust the weights by applying the following formulas.



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

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

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

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

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

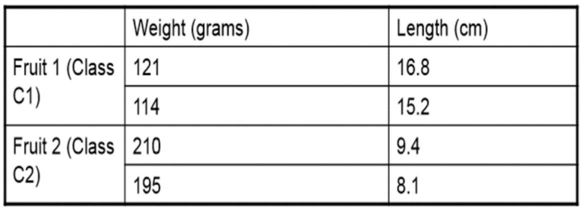
$$= [-1230, -30, 300]^T + [+0.02, 2.28, 0.304]^T$$

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

📖 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.

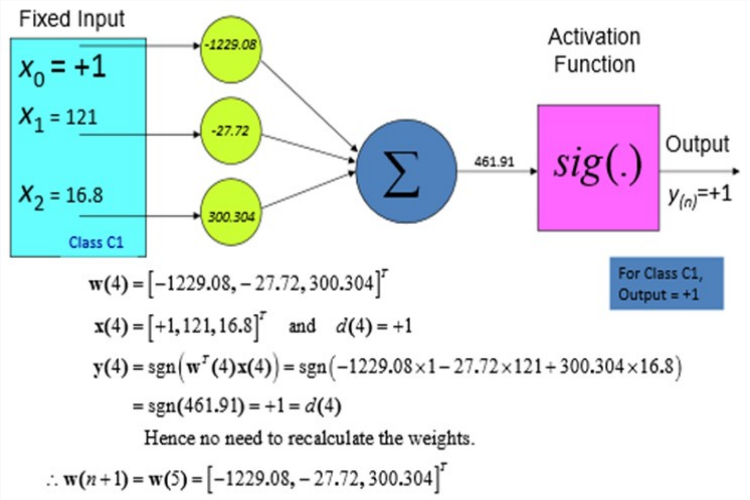
For the following four training examples:



	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.



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

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

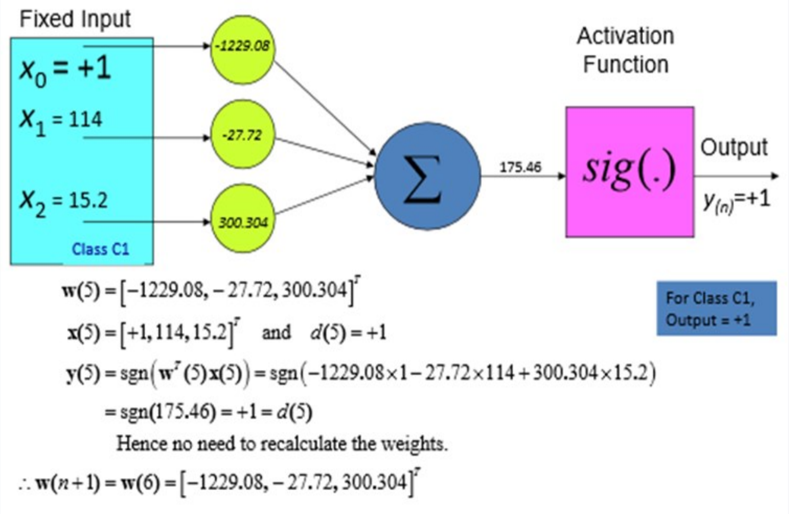
$$\mathbf{y}(4) = \text{sgn}\left(\mathbf{w}^T(4)\mathbf{x}(4)\right) = \text{sgn}\left(-1229.08\times 1 - 27.72\times 121 + 300.304\times 16.8\right)$$

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

Hence no need to recalculate the weights.

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

2.



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

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

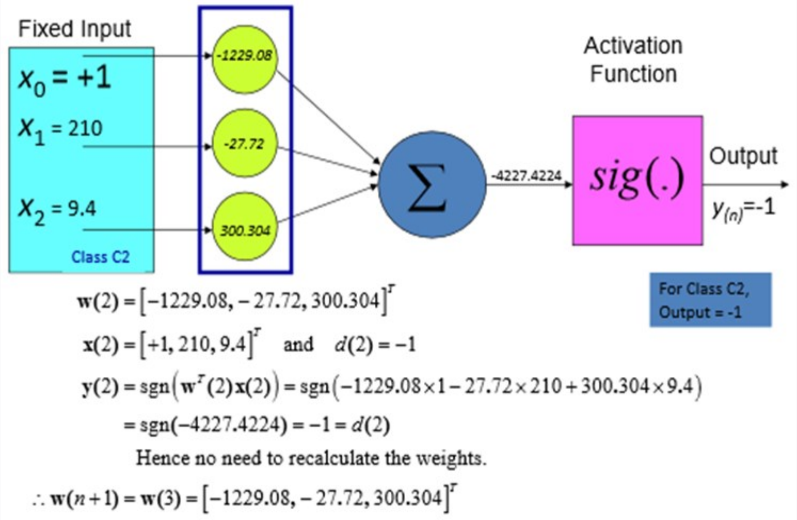
$$\mathbf{y}(5) = \text{sgn}\left(\mathbf{w}^T(5)\mathbf{x}(5)\right) = \text{sgn}\left(-1229.08\times 1 - 27.72\times 114 + 300.304\times 15.2\right)$$

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

Hence no need to recalculate the weights.

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

3.



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

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

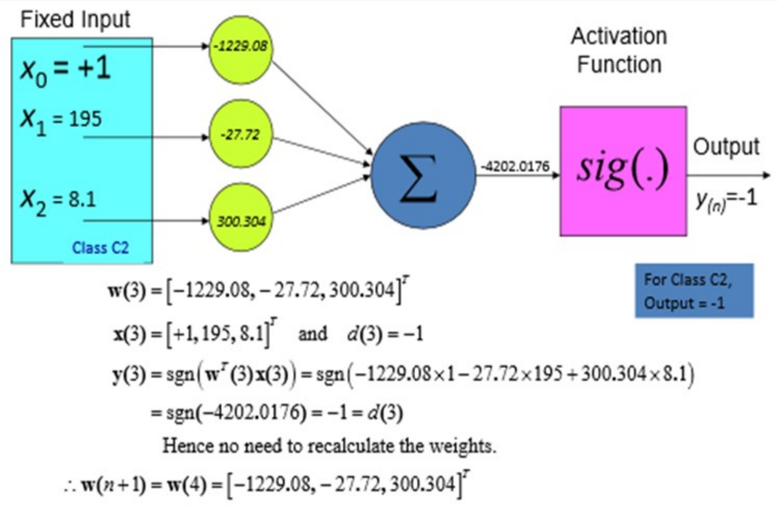
$$\mathbf{y}(2) = \text{sgn}\left(\mathbf{w}^T(2)\mathbf{x}(2)\right) = \text{sgn}\left(-1229.08\times 1 - 27.72\times 210 + 300.304\times 9.4\right)$$

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

Hence no need to recalculate the weights.

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

4.



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

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

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

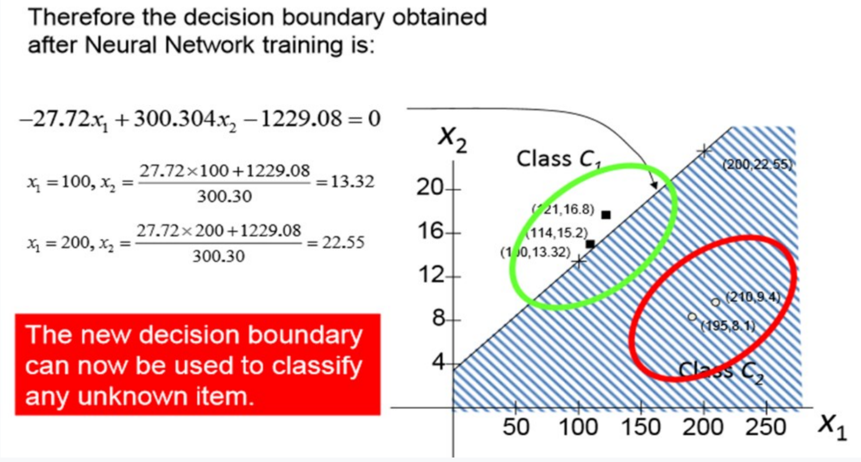
$$= \operatorname{sgn}(-4202.0176) = -1 = d(3)$$

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:

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

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

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

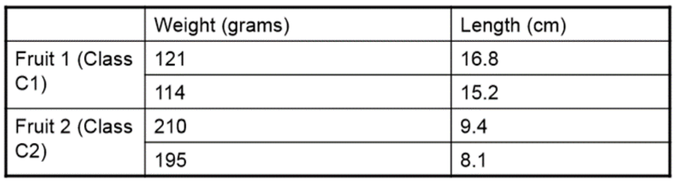
The new decision boundary can now be used to classify any unknown item.

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 aktivacna_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.

```
#nastavíme váhy
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('aktualne vahy: ' , w)
print('bias: ', b)

for i in range(0, 4):
    print('---')
    x, y = training_data[i]
    print('trenovacie data: ' , x , ', vysledok: ', y)
    vnutorna_energia = ((x * w).sum()) + b
    print('vnutorna energia: ',vnutorna_energia)
    predikcia = aktivacna_fn(vnutorna_energia)
    print('predikcia: ',predikcia)
    chyba = y - predikcia
    if (chyba != 0):
        print('potrebne je upravit vahy')
        w = w + (eta * chyba * x)
        b = b + (eta * chyba * 1)
    print('aktualne vahy: ' , w)
    print('bias: ', b)
```

Program output:

```
aktualne vahy:  [-30 300]
bias:  -1230
---
trenovacie data:  [121.   16.8] , vysledok:  1
vnutorna energia:  180.0
predikcia:  1
aktualne vahy:  [-30 300]
bias:  -1230
---
trenovacie data:  [114.   15.2] , vysledok:  1
vnutorna energia:  -90.0
predikcia:  -1
potrebne je upravit vahy
aktualne vahy:  [-27.72  300.304]
bias:  -1229.98
---
trenovacie data:  [210.    9.4] , vysledok:  -1
vnutorna energia:  -4228.3224
predikcia:  -1
aktualne vahy:  [-27.72  300.304]
bias:  -1229.98
---
trenovacie data:  [195.    8.1] , vysledok:  -1
vnutorna energia:  -4202.9176
predikcia:  -1
aktualne vahy:  [-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.

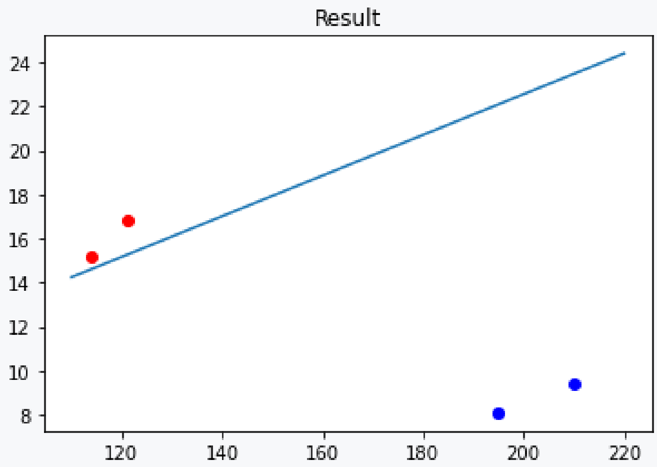
```
def priamka(x):
    y = (w[0]*x + b)/(w[1]*(-1))
    return y
```

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:
        vzor='r'
    else:

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

plt.plot([110,220],[priamka(110),priamka(220)])
#plt.plot([25,200],[50,200])

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

Program output:

```
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 odhad(vektor):
    vnutorna_energia = ((vektor * w).sum()) + b
    predikcia = aktivacna_fn(vnutorna_energia)
    return predikcia
```

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:

```
vektor = array([180,10])
print(odhad(vektor))
```

Program output:

```
-1
```

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

```
vektor = array([140,20])
print(odhad(vektor))
```

Program output:

```
1
```

📖 2.3.5

Literature:

- Hinton, G.: Neural Networks for Machine Learning, University of Toronto. https://www.coursera.org/learn/neural-networks/home/welcome
- Becker, D.: Deep Learning in Python, https://www.datacamp.com/courses/deep-learning-in-python
- Artificial Neural Networks (Part 1) - Classification Using Single Layer Perceptron Model - http://scholastictutors.webs.com/
- Blaha, M.: Umělá inteligence. http://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence
- Sinčák, P., Andrejková, G.: Neurónové siete - Inžiniersky prístup (1. diel) http://neuron-ai.tuke.sk/cig/source/publications/books/NS1/html/index.html
- Kvasnička, V.: Neurónové siete http://www2.fiit.stuba.sk/~kvasnicka/NeuralNetworks/index.html

Feedforward neural network

Chapter 3

3.1 Single layer perceptron

3.1.1

The activation function of a neuron is a function of the input to the neuron ***ini(t)***. Thus, the state of neuron ***i*** is defined by the variable ***yi*** in the form ***yi = f(ini).***

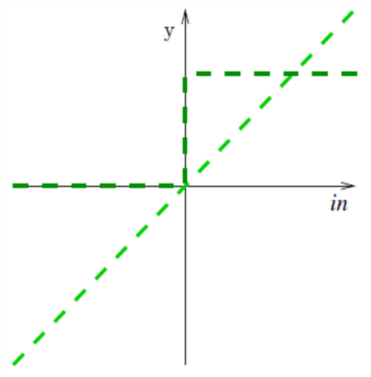
We call the function **f( )** the activation function of the neuron. So far, we have only used a signed (step) function for the activation function.

$$y_i = f(in_i) = \begin{cases} 1 & ak\ in_i \geq 0 \\ 0 & ak\ \ in_i < 0 \end{cases}$$

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

$$y_i = f(in_i) = in_i$$

We can visualize the graphs of both functions.



3.1.2

Among other activation functions, the following functions are also often used:

A piecewise linear function

$$y_i = f(in_i) = \begin{cases} 1 & ak\ in_i \geq \frac{1}{2} \\ in_i & ak\ in_i \in \left(\frac{-1}{2}, \frac{1}{2}\right) \\ -1 & ak\ in_i \leq \frac{-1}{2} \end{cases}$$

Sigmodial function

$$y_i = f(in_i) = \frac{1}{1+e^{-\alpha\, in_i}}$$

where α is the sigmoid steepness parameter

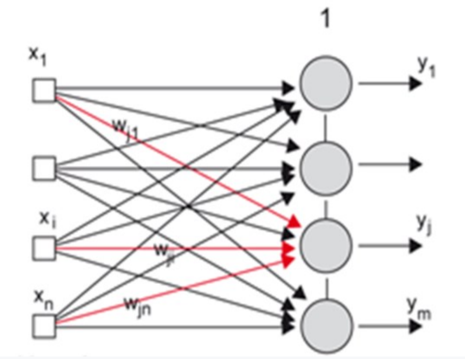
This function is very often used in feedforward neural networks. The function is "smooth", this feature is very important for setting the weights in the learning process. A smooth function is differentiable.

3.1.3

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.



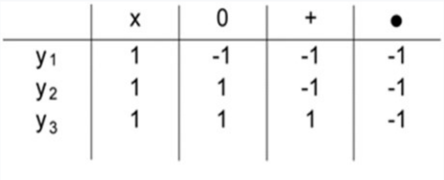
In the course of active dynamics, the network generally realizes the display from $\boldsymbol{R^n} \to \boldsymbol{R^m}$,which was set during the adaptation dynamics. The specific values of the output values are given by the activation transfer functions of the output neurons, that is, for example,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

3.1.4

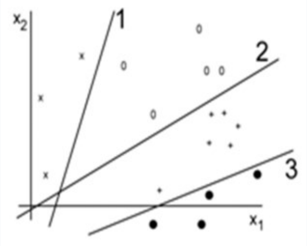
Let's consider the classification possibilities of a single-layer perceptron. If we consider the mapping $R^n \to \{0,1\}^m$, .e. the classification mapping into two classes, we can find ***m*** separating hyperplanes in space, one for each neuron of the output layer.

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



3.1.5

Most real problems are non-linear in nature. This means that they cannot be solved by adding linear bounds.

The problem with the limited computing power of perceptrons was solved only in 1986, when Rumelhart, Hinton and Williams (1986) introduced a training rule called the error backpropagation method for feedforward neural networks with hidden neurons.

Multilayer feed-forward artificial neural networks (multilayer feed-forward ANN), which are trained by the rule of back propagation of errors, are also capable of solving non-linear problems. Feedforward neural networks are characterized by the fact that there are only feedforward connections between neurons.

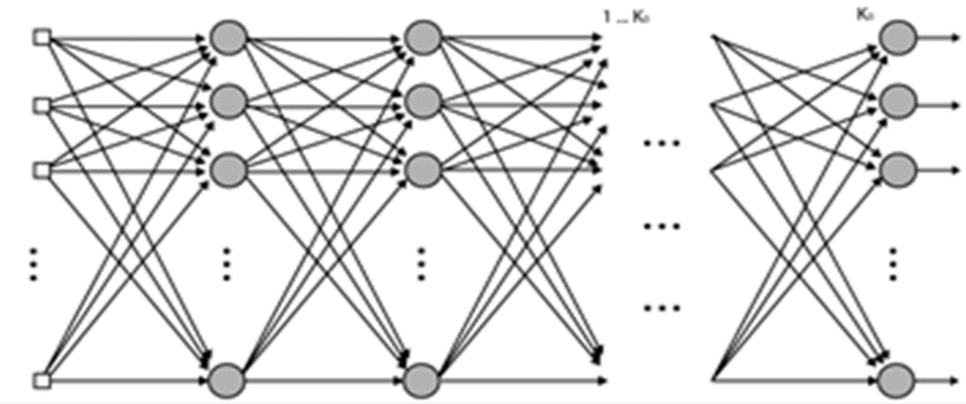
Each neuron of one layer sends signals to each neuron of the next layer. There are no connections to the previous layer or within one layer.

3.1.6

The first typical feedforward multilayer neural network is the **Multilayer Perceptron**.

For this type of neural network, adjacent layers are fully connected. It is also true that there are no links within one layer.

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 $\boldsymbol{R}^n \rightarrow \boldsymbol{R}^m$

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:

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

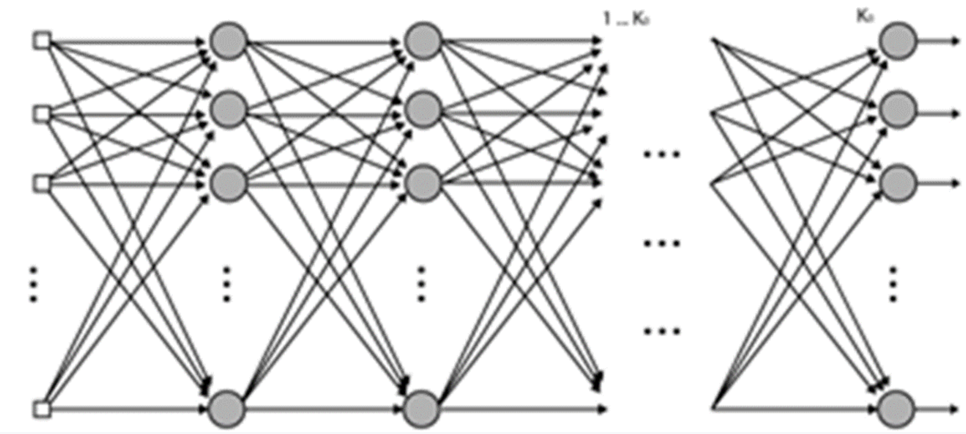
where ***n*** expresses the dimension of the input vector as well as the number of neurons in the ***k***-th layer

📖 3.1.7

The first typical feedforward multilayer neural network is the **Multilayer Perceptron**.

For this type of neural network, adjacent layers are fully connected. It is also true that there are no links within one layer.

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 $\boldsymbol{R}^n \rightarrow \boldsymbol{R}^m$

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:

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

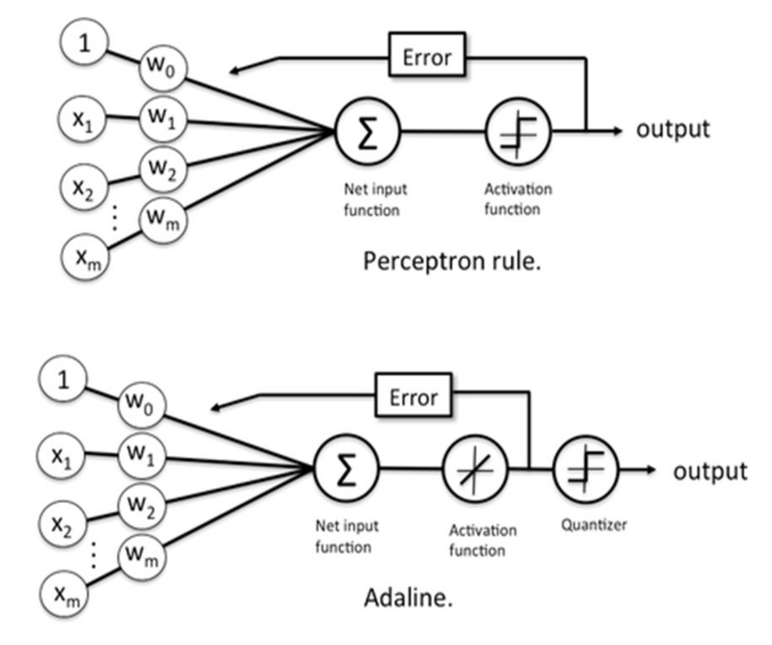
where ***n*** expresses the dimension of the input vector as well as the number of neurons in the ***k***-th layer

3.2 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.



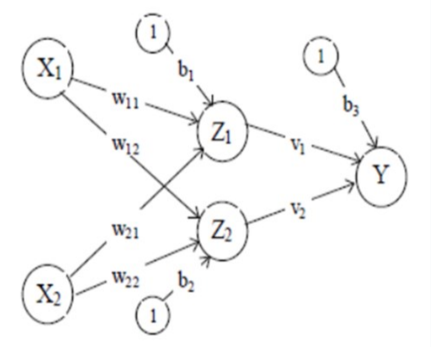
Perceptron uses class labels to learn weight values. Adaline uses continuous values (based on the input) to figure out the weight values, which is "stronger" because it tells us "by how much" we classified correctly or incorrectly. Adaline's learning algorithm is different. It uses the so-called **gradient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

3.2.2

Among the simplest feedforward neural networks is the **Madaline neural network** (Many Adaptive Linear Neurons).

Its 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)

The output neuron ***Y*** actually performs the logic function "OR" on the signals from neurons Z_1 and Z_2



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:

$$v_1 = \frac{1}{2}, v_2 = \frac{1}{2}, b_3 = \frac{1}{2},$$

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 program:

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:

$$z\_in_1 = b_1 + x_1 w_{11} + x_2 w_{21},$$
$$z\_in_2 = b_2 + x_1 w_{12} + x_2 w_{22}.$$

$$z_1 = f(z\_in_1),$$
$$z_2 = f(z\_in_2).$$

$$y\_in = b_3 + z_1 v_1 + z_2 v_2;$$
$$y = f(y\_in).$$

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)***:

$$w_{iJ}(new) = w_{iJ}(old) + \alpha(1 - z\_in_J)\, x_i.$$
$$b_J(new) = b_J(old) + \alpha(1 - z\_in_J).$$

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

$$w_{iK}(new) = w_{iK}(old) + \alpha(-1 - z\_in_K)\, x_i.$$
$$b_K(new) = b_K(old) + \alpha(-1 - z\_in_K).$$

3.2.3

Literature:

- *Kvasnička, V.: Neurónové siete - http://www2.fiit.stuba.sk/~kvasnicka/NeuralNetworks/index.html*
- Volná, E.: Neuronové sítě 1 - https://www1.osu.cz/~volna/Neuronove_site_skripta.pdf
- Hinton, G.: Neural Networks for Machine Learning, University of Toronto - https://www.coursera.org/learn/neural-networks/home/welcome
- Blaha, M.: Umělá inteligence - http://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence
- *Sinčák, P., Andrejková, G.: Neurónové siete - Inžiniersky prístup (1. diel) - http://neuron-ai.tuke.sk/cig/source/publications/books/NS1/html/index.html*

Moving 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 depth of the network, which is the number of layers it contains.

In traditional machine learning models, the input data is processed through a small number of layers, typically no more than a few dozen. However, deep learning models can have hundreds or even thousands of layers, which allows them to learn much more complex representations of the input data.

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

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.

4.1.3

Deep neural networks have:

- long training times,
- small number of hidden layers,
- generally worse performance than MLP and are used for simple tasks.

4.1.4

Misconceptions about deep learning

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.

It's important to have a clear understanding of the capabilities and limitations of deep learning to avoid unrealistic expectations and to use it effectively in solving real-world problems.

📖 4.1.5

Hyperparameters

In deep learning, a hyperparameter is a parameter that is set before the training process begins and is not learned by the model during training. Hyperparameters control the behavior and performance of the model and are usually set by the user based on prior knowledge or trial and error. Unlike the weights and biases of the model, which are learned during training, hyperparameters are fixed values that determine the architecture, optimization method, and training parameters of the model. Examples of hyperparameters in deep learning include:

- Learning rate: The learning rate determines the step size at which the optimizer updates the weights of the model during training.
- Number of epochs: The number of epochs determines the number of times the entire training dataset is passed through the model during training.
- Batch size: The batch size determines the number of samples that are processed by the model in each iteration during training.
- Network architecture: The network architecture determines the structure and depth of the neural network used for the task.
- Regularization parameters: Regularization parameters such as L1 and L2 regularization control the degree of regularization applied to the weights of the model during training.
- Dropout rate: The dropout rate determines the percentage of neurons that are randomly dropped out during training to prevent overfitting.

Hyperparameter tuning is the process of finding the optimal values of hyperparameters for a given task and dataset, usually through a combination of manual tuning and automated methods such as grid search and random search. They are usually determined through trial and error, experimentation, or using heuristics.

During the training process, the hyperparameters remain fixed, and only the weights of the neural network are updated through backpropagation. The hyperparameters define the structure of the neural network and the learning process, and changing them would require retraining the network from scratch.

4.2 Tensors

📖 4.2.1

In deep learning, a **tensor** is a multi-dimensional array of numerical data that can be represented as a sequence of numbers arranged in a specific shape, such as a vector, matrix, or higher-dimensional array.

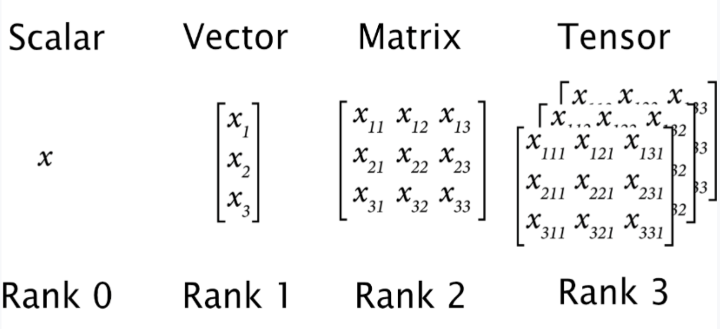
Tensors are used as the fundamental data structure for representing inputs, outputs, and **intermediate activations** in deep learning models. They allow for efficient computation and manipulation of large volumes of data, such as images, audio, and text.

In other words, you can think of a tensor as a container that can hold a lot of numbers arranged in a specific way, and that can be manipulated mathematically to perform various operations in a deep learning model.

Tensors can be considered as the basic components of ANNs – input data, output predictions, and weights that are learned during the training process are all tensors. The information is transmitted through a series of linear and non-linear transformations to transform input data into predictions.

4.2.2

Tensors can be represented as multidimensional arrays. The number of dimensions of the tensor spans is known as the tensor range. Tensors with 0, 1, and 2 ranks are often used and have their own names, scalars, vectors, and matrices, but the term tensors can be used to describe each of them.


Scalar
Vector
Matrix
Tensor
x
x_1 x_2 x_3
x_{11} x_{12} x_{13} x_{21} x_{22} x_{23} x_{31} x_{32} x_{33}
x_{111} x_{121} x_{131} x_{211} x_{221} x_{231} x_{311} x_{321} x_{331}
Rank 0
Rank 1
Rank 2
Rank 3

- **Scalar**: A scalar 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**: Vectors are 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**: The matrices are two-dimensional tables 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**: Tensors 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.

4.2.3

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)
```

- **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))
```

text

```
import tensorflow as tf
int_variable = tf.Variable(4113, tf.int16)
int_variable
tf.rank(int_variable)
tf.rank(int_variable).numpy()
int_variable.shape
int_variable.shape.as_list()
vector_variable = tf.Variable([0.23, 0.42, 0.35], tf.float32)
tf.rank(vector_variable).numpy()
vector_variable.shape.as_list()
matrix_variable = tf.Variable([[4113, 7511, 6259], [3870,
6725, 6962]], tf.int32)
```

```
tf.rank(matrix_variable).numpy()

matrix_variable.shape.as_list()
tensor_variable = tf.Variable([[[4113, 7511, 6259], \
                                [3870, 6725, 6962]], \
                               [[5102, 7038, 6591], \
                                [3661, 5901, 6235]], \
                               [[951, 1208, 1098], \
                                [870, 645, 948]]])
tf.rank(tensor_variable).numpy()
print(tensor_variable.shape.as_list())
```

4.3 Simple and more complex network example in TensorFlow

4.3.1

Example of simple neural network with one hidden layer

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 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:

```
iris = datasets.load_iris()
# 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])
```

Use predefined seed to make results reproducible:

```
# 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]

# Normalize by column (min-max norm)
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))

# Declare batch size
batch_size = 50

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

Declare the network:

```
# Create variables for both NN layers
hidden_layer_nodes = 5
a1 =
tf.Variable(tf.random.normal(shape=[3,hidden_layer_nodes],
seed=seed)) # inputs -> hidden nodes
b1 = tf.Variable(tf.random.normal(shape=[hidden_layer_nodes],
seed=seed))   # one biases for each hidden node
a2 =
tf.Variable(tf.random.normal(shape=[hidden_layer_nodes,1],
seed=seed)) # hidden inputs -> 1 output
b2 = tf.Variable(tf.random.normal(shape=[1], seed=seed))   # 1
bias for the output

# Declare model operations
hidden_output = tf.keras.layers.Lambda(lambda x:
tf.nn.relu(tf.add(tf.matmul(x, a1), b1)))
final_output = tf.keras.layers.Lambda(lambda x:
tf.nn.relu(tf.add(tf.matmul(x, a2), b2)))
```

```
hidden_layer = hidden_output(x_data)
output = final_output(hidden_layer)

# 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)
```

Since this is a regression problem, we will use mean squared error (MSE) as the loss function:

```
# 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)))
```

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()
```

4.3.2

More complex network with 3 hidden layers

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 tensorflow as tf
import matplotlib.pyplot as plt
import csv
import random
import numpy as np
import requests
```

```
# 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])

# 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]

# 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))

# 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,))

# 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()
```

```
# 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()))

# 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()

# 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))
```

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))
```

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 in CNNs and architectures

5.2.1

Layer types

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

1. **Convolutional Layers**: These layers apply a set of learnable filters to the input data, extracting local features such as edges and textures.
2. **Pooling Layers**: These layers reduce the spatial resolution of the feature maps while retaining the most important features, typically by taking the maximum or average value within a small region.
3. **Fully Connected Layers**: These layers connect every neuron in the layer to every neuron in the previous layer, and are typically used in the final layers of the network for classification or regression tasks.
4. **Activation Layers**: These layers introduce non-linearity to the output of the previous layer, typically through an activation function such as ReLU, sigmoid, or tanh.
5. **Normalization Layers**: These layers normalize the output of the previous layer to improve performance and reduce overfitting, typically through techniques such as batch normalization.
6. **Dropout Layers**: These layers randomly drop out some of the neurons in the previous layer during training to reduce overfitting and improve generalization.
7. **Upsampling Layers**: These layers increase the spatial resolution of the feature maps, typically by repeating the values within a small region.

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

Convolution layer

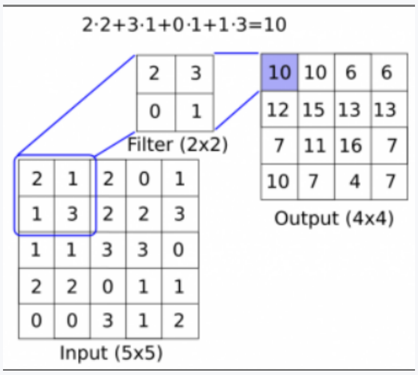
The convolutional layer is a fundamental building block of a Convolutional Neural Network (CNN). It applies a set of filters to the input image to extract features, by performing a convolution operation between the input image and a set of learnable filters. Each filter slides over the entire input image, 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



5.2.3

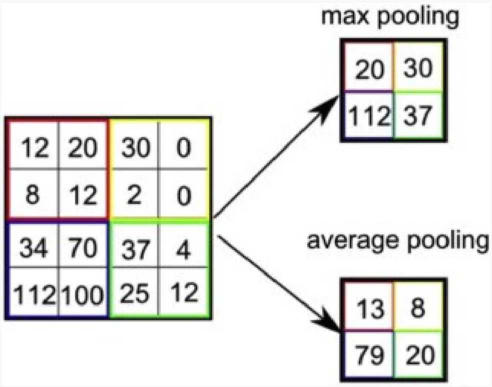
Pooling layer

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.



12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

max pooling

20	30
112	37

average pooling

13	8
79	20

5.2.4

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.

5.2.5

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.

5.2.6

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.

🕮 5.2.7

Dropout layer

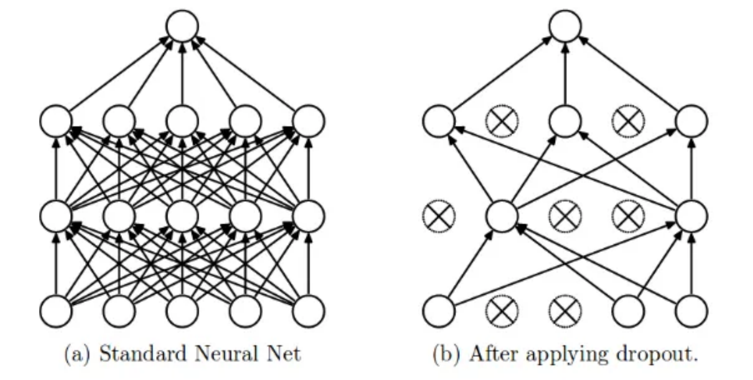
Dropout is a regularization technique used in deep neural networks to prevent overfitting. A dropout layer is a type of layer in a neural network that randomly drops out, or "turns off," a certain percentage of the neurons during training. The neurons that are dropped out change with each training iteration,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



🕮 5.2.8

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:

1. Nearest neighbor upsampling: This method simply duplicates the values in the input feature map to create a larger output feature map.
2. Bilinear upsampling: This method uses a weighted average of the four nearest pixels in the input feature map to generate each pixel in the output feature map.
3. Transposed convolutional upsampling: This method 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.

🕮 5.2.9

Architectures

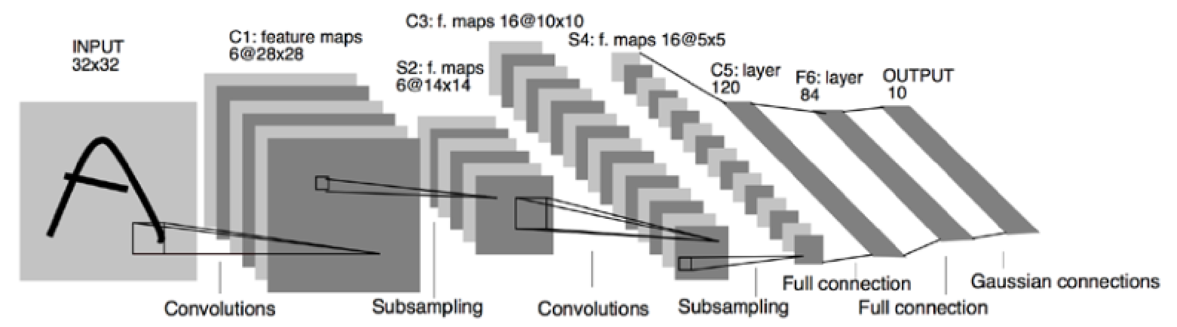
There are several popular architectures of Convolutional Neural Networks (CNNs), each with its own unique structure and purpose. Here are some of the most well-known ones:

1. LeNet: One of the earliest CNNs developed by Yann LeCun in the 1990s for handwritten digit recognition.
2. AlexNet: Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012, this was the first CNN to achieve state-of-the-art performance on the ImageNet dataset.

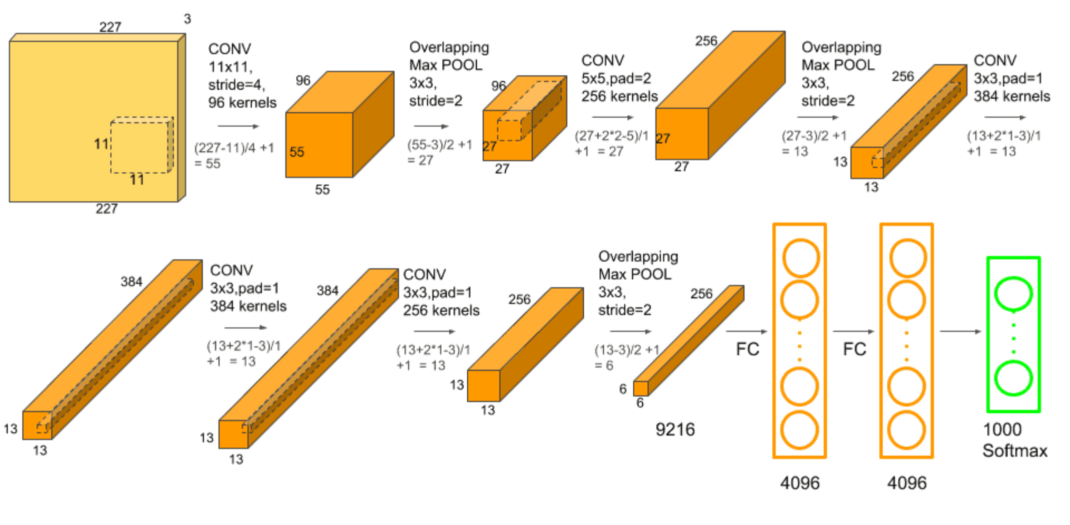
3. VGG: A CNN developed by the Visual Geometry Group at Oxford in 2014, which consists of a series of convolutional layers followed by fully connected layers.
4. GoogLeNet/Inception: Developed by researchers at Google in 2014, this architecture uses a novel "Inception module" that combines multiple different convolutional operations in parallel.
5. ResNet: Developed by Microsoft researchers in 2015, this architecture introduces the concept of residual connections, which allow information to flow more easily through the network and enable the training of much deeper networks.
6. DenseNet: Developed by researchers at Facebook AI Research in 2017, this architecture uses dense connections between layers, allowing for better feature reuse and reducing the number of parameters.
7. MobileNet: A family of lightweight CNN architectures developed by Google in 2017, designed for mobile and embedded devices with limited computational resources.

These are just a few examples of the many different CNN architectures that have been developed over the years, each with its own strengths and weaknesses depending on the specific task and dataset.

LeNet-5 architecture visualization



AlexNet architecture visulazation



5.3 Practical applications

5.3.1

Types of practical applications of CNNs

Convolutional Neural Networks (CNNs) have become a popular tool for a variety of image and video-related tasks, including:

1. Image Classification: CNNs are widely used for image classification tasks, where the goal is to assign a label to an input image from a set of predefined categories. This can be applied to tasks such as object recognition, facial recognition, and scene classification.
2. Object Detection: CNNs can also be used for object detection tasks, where the goal is to locate and classify objects within an image or video. This is useful for tasks such as self-driving cars, security cameras, and robotics.
3. Semantic Segmentation: CNNs can be used for semantic segmentation tasks, where the goal is to assign a label to every pixel in an image, allowing for more precise object detection and analysis.
4. Image Generation: CNNs can also be used for image generation tasks, such as generating realistic images from a given set of parameters or styles.
5. Style Transfer: CNNs can be used for style transfer tasks, where the goal is to apply the style of one image to another image, creating a new image that combines the content of one image with the style of another.
6. Medical Imaging: CNNs can be used in medical imaging applications, such as diagnosing diseases from medical images, identifying tumors, and analyzing scans.
7. Video Analysis: CNNs can be used for video analysis tasks, such as action recognition, video tracking, and video captioning.
8. Natural Language Processing: While CNNs are primarily used for image and video-related tasks, they can also be applied to natural language processing tasks such as sentiment analysis, text classification, and language translation.
9. Recommendation Systems: CNNs can be used to build recommendation systems that suggest products, movies, or other items based on user behavior and preferences.
10. Autonomous Vehicles: CNNs are being used in the development of self-driving cars, where they are used to detect and classify objects on the road such as pedestrians, other vehicles, and traffic signs.
11. Agriculture: CNNs can be used to analyze satellite imagery and other data to help farmers monitor crops, predict yields, and detect pests and diseases.
12. Robotics: CNNs can be used in robotics applications such as object recognition, grasping and manipulation, and navigation.
13. Art and Music: CNNs can be used to generate or classify art and music, such as creating original pieces based on certain styles or identifying the genre of a particular song.

14. Gaming: CNNs can be used in the development of video games, such as character recognition and animation.
15. Finance: CNNs can be used in finance for fraud detection, stock market prediction, and risk assessment.

5.4 Image augmentation

📖 5.4.1

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:

1. Rotation: Rotating the image by a certain angle to create new training examples.
2. Flipping: Flipping the image horizontally or vertically to create a mirrored version of the original image.
3. Translation: Shifting the image horizontally or vertically to create a new training example.
4. Scaling: Rescaling the image to create a smaller or larger version of the original image.
5. Shearing: Shearing the image to create a slanted version of the original image.
6. Zooming: Zooming 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.

5.5 CNN examples

5.5.1

LeNET-5 example - character recognition

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

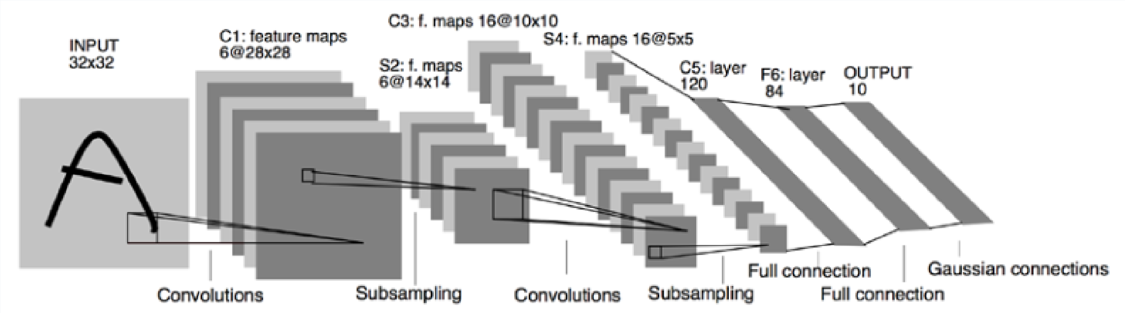
LeNet-5 is a classic convolutional neural network architecture designed for handwritten digit recognition, and was introduced by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner in 1998. It was one of the earliest successful attempts to apply deep learning techniques to image recognition tasks.

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 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()

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])
```

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()
```

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()
```

5.5.2

Advanced - more complex CNN

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 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=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)

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")
    ])
```

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

Start training:

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

# 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()
```

5.6 Pre-trained networks

📖 5.6.1

Retraining existing CNN models

Retraining existing CNN models refers to the process of taking a pre-trained convolutional neural network (CNN) and fine-tuning it on a new task or dataset. This approach is often used in transfer learning, where a pre-trained CNN is used as a starting point for a new task, instead of training a CNN from scratch.

In retraining, the pre-trained CNN is typically modified by removing the last few layers that were designed for the original task, and replacing them with new layers that are suitable for the new task. The remaining layers of the pre-trained CNN are frozen, meaning that their weights are not updated during training. The new layers added to the CNN are then trained using the new dataset, and the weights of the frozen layers are fine-tuned to improve the performance on the new task.

Retraining existing CNN models can save time and resources compared to training a CNN from scratch, as the pre-trained CNN has already learned useful features that can be leveraged for the new task. It also allows for transfer of knowledge from one task to another, which can improve the performance on the new task, especially when the new dataset is small or similar to the original dataset used to train the pre-trained CNN.

5.6.2

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

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 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

# 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']
```

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.

```
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):
    img = tf.cast(img, tf.float32)
    img = tf.image.resize(img, (75, 75))
    return
tf.keras.applications.inception_v3.preprocess_input(img) ,
label
```

```
dataset_train_processed =
dataset_train.shuffle(buffer_size).batch(batch_size).map(prepr
ocess_cifar10)
dataset_test_processed =
dataset_test.batch(batch_size).map(preprocess_cifar10)
```

We want to load the weights without the classification head.

```
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.

```
x = inception_model.output
x= keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(1024, activation="relu")(x)
x = keras.layers.Dense(128, activation="relu")(x)
output = keras.layers.Dense(10, activation="softmax")(x)

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):

```
for inception_layer in inception_model.layers:
    inception_layer.trainable= False

# Compile the model
 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)
```

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.

5.7 Object detection

🕮 5.7.1

Image classification - Binary classification

Binary classification is the simplest approach for classification models, since it classifies images into only two categories. In this chapter, we began with convolutional operation and discussed how to use it as an image transformer. Then you learned what a pool layer does and the differences between maximum and average pooling. Then we also studied how a flattening layer turns a pooled feature map into a single column. Then you learned how and why to use image augmentation and how to use batch normalization. These are the main components that distinguish CNN from other ANNs. Like other binary classifiers, binary image classifiers end with a dense layer with a unit and a sigmoid activation function. To provide more convenience, image classifiers can be equipped to classify more than two objects. These classifications are generally known as object classifications.

Object classification

The three different types of models for object classification are:

- Image classification: the goal is to classify the entire image into one of several predefined categories. The output is a single label that represents the most likely class for the entire image. For these problems, you'll use a traditional CNN.
- Classification with localization: the goal is to classify the entire image into one of several predefined categories, but also to locate the object within the image. The output is a bounding box that represents the location of the object within the image, as well as a label that represents the most likely class for the object. For this, you can use model such as simplified You Only Look Once (YOLO) or R-CNN.
- Detection: In object detection, the goal is to detect and localize all instances of objects of interest within the image, as well as classify them. The output is a set of **bounding boxes** that represent the locations of all the objects of interest in the image, as well as a label for each bounding box that represents the most likely class for the object. For this, you use YOLO model or an R-CNN.

Therefore, image classification is the simplest and most coarse-grained task, while object detection is the most complex and fine-grained task, requiring the localization and classification of multiple objects in an image.

📖 5.7.2

Well known object detection algorithms

R-CNN

R-CNN stands for Region-based Convolutional Neural Network. It 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.

The R-CNN algorithm works in three steps. First, it generates a set of region proposals using a selective search algorithm. These regions are then passed through a convolutional neural network to extract features that are used to classify the objects within each region. Finally, a bounding box regression algorithm is used to refine the location of the object within the region.

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.

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.

SSD - 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.

📖 5.7.3

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:

1. Precision: the proportion of true positive detections (correctly identified objects) over the total number of detections made by the model.
2. Recall: the proportion of true positive detections over the total number of objects present in the image.
3. 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.
4. Average Precision (AP): a metric that combines both precision and recall, by computing the area under the precision-recall curve.
5. Mean Average Precision (mAP): the average AP across all object categories in the dataset.
6. 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.

Recurrent Neural Networks – RNNs

Chapter 6

6.1 RNN overview

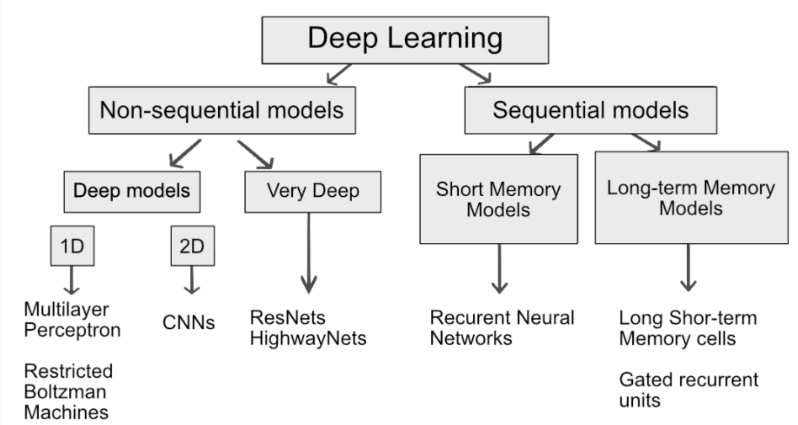
📖 6.1.1

Description of RNNs

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

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: This includes data that is collected over time, such as stock prices, weather data, or sensor data.
2. Natural Language Processing (NLP) data: This includes text data, such as words or sentences, that have a specific sequence.
3. Music data: This includes audio data that has a temporal order, such as music notes or beats.
4. Video data: This includes data that is captured from a sequence of images, such as videos or motion capture data.

6.1.3

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.

6.1.4

RNNs can be typically used for:

- Sequential data
- Image data
- Grid structured data

6.1.5

Typical applications of RNNs

1. Natural Language Processing: RNNs can be used to model the temporal structure of language, making them well-suited for tasks such as language modeling, machine translation, sentiment analysis, and speech recognition.
2. Time Series Analysis: RNNs can be used to analyze time series data, such as stock prices, weather data, or sensor data, to make predictions or detect anomalies.
3. Image and Video Captioning: RNNs can be used to generate captions or descriptions of images or videos by processing the visual information in a sequential manner.
4. Music Generation: RNNs can be used to generate new music by learning patterns and structures from existing music and then generating new sequences of notes.
5. Handwriting Recognition: RNNs can be used to recognize and classify handwritten text by processing the temporal sequence of strokes.
6. Speech Recognition: RNNs can be used for speech recognition tasks, such as converting spoken words to text.

Overall, RNNs are useful for tasks that involve sequential data, where the context and temporal dependencies between elements are important for the task.

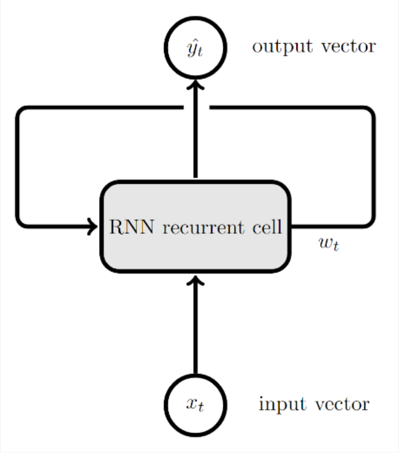
6.2 Layers and architectures in RNNs

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

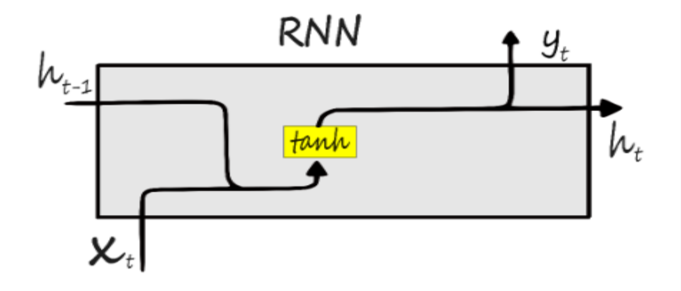
In feed-forward neural networks, data propagates in one direction only, that is, from input to output. This is good approach for single input you need to process (such as image data seen in CNNs previously) but it does not work well for a sequence of data. RNNs are particularly suitable to handle cases 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.

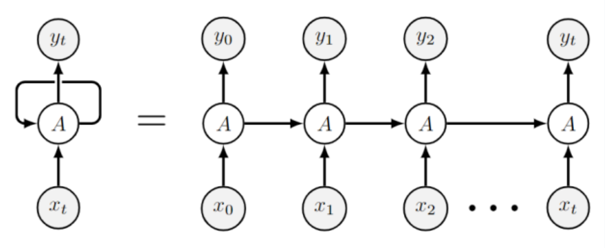
RNN maintains the inner state H_t, combine it with the next input data X_{t+1}, make a prediction, Y_{t+1}, and store the new inner state H_{t+1}.The key idea is that state update is a combination of the previous state time step and the current input received by the network.

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.



📖 6.2.3

The vanishing gradient problem

The vanishing gradient problem in RNNs refers to the issue where the gradient used for updating the weights of the network becomes extremely small during backpropagation, making it difficult for the network to learn and optimize long-term dependencies. This happens because the gradients of the loss function with respect to the weights in the earlier time steps of the RNN are multiplied by the weight matrix in each time step during backpropagation, and if this matrix has eigenvalues less than 1, the gradients can quickly vanish as they are propagated backward in time. As a result, the network may have difficulty learning long-term dependencies, which can be problematic for tasks such as speech recognition or natural language processing, where the meaning of a word or sentence may depend on information from several earlier time steps.

📖 6.2.4

LSTM

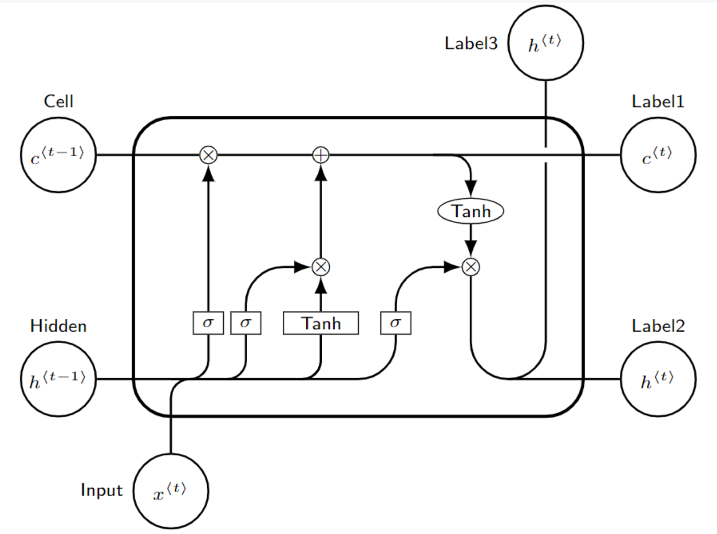
LSTM stands for Long Short-Term Memory, and it is a type of recurrent neural network (RNN) cell that is designed to better handle the vanishing gradient problem in traditional RNNs.

The key difference between LSTM cells and traditional RNN cells is that LSTM cells have an internal memory state that can selectively store or discard information at each time step. This allows LSTM cells to more effectively remember long-term dependencies in sequential data.

At each time step, an LSTM cell takes three inputs: the current input to the cell, the previous hidden state, and the previous memory state. It then uses a series of gates to control the flow of information into and out of the cell.

The forget gate determines how much of the previous memory state to forget, the input gate determines how much of the current input to use to update the memory state, and the output gate determines how much of the memory state to output to the next time step.

LSTM cells have become a popular building block in deep learning models for sequential data, including natural language processing, speech recognition, and time series analysis.



LSTM processing steps

1. Forget
2. Store
3. Update
4. Generate

📖 6.2.5

Architectures

Recurrent Neural Networks (RNNs) have several known architectures that are commonly used for various tasks. Here are some of the most well-known architectures:

1. Simple RNN: This architecture 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): This architecture 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): This architecture 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: This architecture 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: This architecture 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: This architecture 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.

6.3 Examples with RNNs

6.3.1

ANN on sequential data - Nvidia stock price prediction

This is example of regular ANN used for sequential data:

```
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://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter09/Exercise9.01/NVDA.csv"
data = pd.read_csv(url)
```

Show data head and tail:

```
print(data.head())

print(data.tail())

# 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())

scaler = MinMaxScaler()
training_data = scaler.fit_transform(training_data)

X_train = []
y_train = []

print(training_data.shape[0])

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)

X_train.shape, y_train.shape

print(X_train.shape)
print(y_train.shape)

X_old_shape = X_train.shape
X_train = X_train.reshape(X_old_shape[0],
X_old_shape[1]*X_old_shape[2])
print(X_train.shape)

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout

# Model definition
regressor_ann = Sequential()
```

```
regressor_ann.add(Input(shape = (300,)))
regressor_ann.add(Dense(units = 512, activation = 'relu'))
regressor_ann.add(Dropout(0.2))

regressor_ann.add(Dense(units = 128, activation = 'relu'))
regressor_ann.add(Dropout(0.3))

regressor_ann.add(Dense(units = 64, activation = 'relu'))
regressor_ann.add(Dropout(0.4))

regressor_ann.add(Dense(units = 16, activation = 'relu'))
regressor_ann.add(Dropout(0.5))

regressor_ann.add(Dense(units = 1))

regressor_ann.summary()

regressor_ann.compile(optimizer='adam', \
                      loss = 'mean_squared_error')
```

Start training:

```
regressor_ann.fit(X_train, y_train, epochs=10, batch_size=32)

## Test and predict stock price
## Prepare test dataset
print(data_test.head())

print(data_training.tail(60))

past_60_days = data_training.tail(60)

df = past_60_days.append(data_test, ignore_index = True)
df = df.drop(['Date', 'Adj Close'], axis = 1)
df.head()

inputs = scaler.transform(df)
```

```
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)

X_old_shape = X_test.shape
X_test = X_test.reshape(X_old_shape[0], \
                        X_old_shape[1] * X_old_shape[2])

X_test.shape, y_test.shape

y_pred = regressor_ann.predict(X_test)

print(scaler.scale_)
```

Scale with the maximum value:

```
scale = 1/3.70274364e-03
print(scale)

y_pred = y_pred*scale
y_test = y_test*scale
```

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()
```

6.3.2

RNN with LSTM Layer Nvidia Stock Prediction

This example demonstrates using of RNN with LSTM layer for prediction of Nvidia Stock value.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

Read data from the source:

```
import io
import requests
url="https://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter09/Exercise9.01/NVDA.csv"
data = pd.read_csv(url)
```

Show the head of the data:

```
print(data.head())
```

Show the tail of the data:

```
print(data.tail())

# Split Training data
data_training = data[data['Date']<'2019-01-01'].copy()
print(data_training)

# Split Testing data
data_test = data[data['Date']>='2019-01-01'].copy()
print(data_test)

training_data = data_training.drop(['Date', 'Adj Close'], axis
= 1)
print(training_data.head())

scaler = MinMaxScaler()
training_data = scaler.fit_transform(training_data)
print(training_data)

X_train = []
y_train = []
training_data.shape[0]

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:

```
regressor = Sequential()

regressor.add(LSTM(units= 50, activation = 'relu', \
                   return_sequences = True, \
                   input_shape = (X_train.shape[1], 5)))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units= 60, activation = 'relu', \
                   return_sequences = True))
regressor.add(Dropout(0.3))

regressor.add(LSTM(units= 80, activation = 'relu', \
                   return_sequences = True))
regressor.add(Dropout(0.4))

regressor.add(LSTM(units= 120, activation = 'relu'))
regressor.add(Dropout(0.5))

regressor.add(Dense(units = 1))
```

Print model layers:

```
regressor.summary()

regressor.compile(optimizer='adam', loss =
'mean_squared_error')
```

Start training:

```
regressor.fit(X_train, y_train, epochs=10, batch_size=32)

past_60_days = data_training.tail(60)

df = past_60_days.append(data_test, ignore_index = True)
df = df.drop(['Date', 'Adj Close'], axis = 1)
```

```
print(df.head())

inputs = scaler.transform(df)
print(inputs)

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)
X_test.shape, y_test.shape
```

Predict prices using the trained network:

```
y_pred = regressor.predict(X_test)

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 predicted data values:

```
print(y_pred)
```

Plot the prediction together with actual data:

```
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()
```

Generative models

Chapter 7

7.1 Generative models overview

7.1.1

Introduction

Generative models differ from predictive models because they aim to generate new samples from the same distribution of training data. Although the purpose of these models may differ greatly from that of previous sections, many concepts learned in previously can and will be used, including loading and preprocessing various data files, hyperparameter adjustment, and the usage of CNNs and RNNs.

7.1.2

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.

7.1.3

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 natural language processing (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 Natural Language Processing (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.

7.1.4

Generative Adversarial Networks (GANs)

GANs, or Generative Adversarial Networks, are a type of deep learning model that involves two neural networks: a **generator** and a **discriminator**. The generator takes in **random noise** as input and generates fake data, such as images or text. The discriminator is trained to distinguish between the generated fake data and real data.

During training, the generator tries to generate data that is realistic enough to fool the discriminator, while the discriminator tries to correctly classify the real and fake data. Over time, both networks improve, with the generator getting better at generating realistic data and the discriminator getting better at distinguishing between real and fake data.

GANs have been used for a variety of tasks, such as generating realistic images, synthesizing audio, and creating new text.

7.1.5

Deep Convolutional Generative Adversarial Networks (DCGANs)

Deep Convolutional Generative Adversarial Networks (DCGANs) are a type of Generative Adversarial Networks (GANs) that use deep convolutional neural networks (CNNs) as the generator and the discriminator networks.

DCGANs were proposed in 2015 as an extension of GANs to address the challenge of generating high-quality images. The generator in a DCGAN is a deep CNN that maps a random noise vector to an image, and the discriminator is also a deep CNN that maps an image to a probability of being real or fake.

The use of convolutional layers in the generator and discriminator helps to capture the spatial dependencies in images and produce more realistic and sharp images. DCGANs have been successfully used for tasks such as image synthesis, image inpainting, and style transfer.

7.1.6

Deepfake

Deepfake is a technique used to create synthetic media, such as videos, images, or audio recordings, that are either entirely fake or manipulated to be false. Specifically, deepfake algorithms use artificial intelligence and deep learning techniques to generate or modify visual or audio content in a way that is convincing to humans. This has raised concerns about the potential misuse of the technology for malicious purposes, such as creating fake news, impersonating individuals, or creating non-consensual pornography. However, deepfake technology can also be used for positive applications, such as in film and entertainment or for creating more realistic simulations for training and education.

DCGAN can be used as a part of a pipeline to generate deepfake images or videos, but **it is not specifically designed for this task.**

7.2 Examples of generative models

7.2.1

Generating a text using RNN

This example shows text generator using RNN

```
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
import warnings
warnings.filterwarnings("ignore")
warnings.simplefilter(action='ignore', category=FutureWarning)

import io
import requests
url="https://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter09/Datasets/Articles.csv"
data = pd.read_csv(url)

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:

```
831
```

```
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']
```

```
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]]
```

```
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)

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()

model.fit(predictors, label, epochs=200, verbose=5)
```

Program output:

```
Epoch 1/200
Epoch 2/200
Epoch 3/200
Epoch 4/200
Epoch 5/200
Epoch 6/200
Epoch 7/200
```

```
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()

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))
```

7.2.2

Generating pictures using DCGAN

NOT WORKING needs data store

```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Program output:

```
Note: not using Google CoLab

import tensorflow as tf
from tensorflow.keras.models import Sequential, Model,
load_model
from tensorflow.keras.layers import Input, Reshape, Dropout,
Dense
from tensorflow.keras.layers import Flatten,
BatchNormalization
from tensorflow.keras.layers import UpSampling2D, Conv2D
from tensorflow.keras.layers import Activation, ZeroPadding2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import LeakyReLU
import zipfile
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from tqdm import tqdm
import os
import time
from skimage.io import imread
```

Program output:

```
ModuleNotFoundError
No module named 'skimage'

def time_string(sec_elapsed):
    hour = int(sec_elapsed / (60 * 60))
    minute = int((sec_elapsed % (60 * 60)) / 60)
    second = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(hour, minute, second)

gen_res = 3
```

```
img_chan = 3
gen_square = 32 * gen_res
img_rows = 5
img_cols = 5
img_margin = 16
seed_vector = 200
data_path =
"https://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter11/Exercise11.03/apple-or-
tomato/training_set/"
epochs = 1000
num_batch = 32
num_buffer = 60000

print(f"Will generate a resolution of {gen_res}.")
print(f"Will generate {gen_square}px square images.")
print(f"Will generate {img_chan} image channels.")
print(f"Will generate {img_rows} preview rows.")
print(f"Will generate {img_cols} preview columns.")
print(f"Our preview margin equals {img_margin}.")
print(f"Our data path is: {data_path}.")
print(f"Our number of epochs are: {epochs}.")
print(f"Will generate a batch size of {num_batch}.")
print(f"Will generate a buffer size of {num_buffer}.")
```

Program output:

```
Will generate a resolution of 3.
Will generate 96px square images.
Will generate 3 image channels.
Will generate 5 preview rows.
Will generate 5 preview columns.
Our preview margin equals 16.
Our data path is:
https://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter11/Exercise11.03/apple-or-
tomato/training_set/.
Our number of epochs are: 1000.
Will generate a batch size of 32.
Will generate a buffer size of 60000.
```

```
training_binary_path = os.path.join(data_path,\
f'training_data_{gen_square}_{gen_square}.npy')

print(f"Looking for file: {training_binary_path}")

if not os.path.isfile(training_binary_path):
    start = time.time()
    print("Loading images...")

    train_data = []
    images_path = os.path.join(data_path,'tomato')
    for filename in tqdm(os.listdir(images_path)):
        path = os.path.join(images_path,filename)
        images = Image.open(path).resize((gen_square,
            gen_square),Image.ANTIALIAS)
        train_data.append(np.asarray(images))
    train_data = np.reshape(train_data,(-1,gen_square,
            gen_square,img_chan))
    train_data = train_data.astype(np.float32)
    train_data = train_data / 127.5 - 1.


    print("Saving training images...")
    np.save(training_binary_path,train_data)
    elapsed = time.time()-start
    print (f'Image preprocessing time:
{time_string(elapsed)}')
else:
    print("Loading the training data...")
    train_data = np.load(training_binary_path)
```

Program output:

```
Looking for file:
https://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter11/Exercise11.03/apple-or-
tomato/training_set/training_data_96_96.npy
Loading images...
FileNotFoundError
[Errno 2] No such file or directory:
'https://raw.githubusercontent.com/PacktWorkshops/The-
TensorFlow-Workshop/master/Chapter11/Exercise11.03/apple-or-
tomato/training_set/tomato'
```

```
train_dataset = tf.data.Dataset.from_tensor_slices(train_data)
\
                  .shuffle(num_buffer).batch(num_batch)
```

Generator and Discriminator

```
def create_generator(seed_size, channels):
    model = Sequential()


model.add(Dense(4*4*256,activation="relu",input_dim=seed_size)
)
    model.add(Reshape((4,4,256)))

    model.add(UpSampling2D())
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    model.add(UpSampling2D())
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    # Output resolution, additional upsampling
    model.add(UpSampling2D())
    model.add(Conv2D(128,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    if gen_res>1:
        model.add(UpSampling2D(size=(gen_res,gen_res)))
        model.add(Conv2D(128,kernel_size=3,padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Activation("relu"))

    # Final CNN layer
    model.add(Conv2D(channels,kernel_size=3,padding="same"))
    model.add(Activation("tanh"))

    return model


def create_discriminator(image_shape):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=3, strides=2,
input_shape=image_shape,
                     padding="same"))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(64, kernel_size=3, strides=2,
padding="same"))
    model.add(ZeroPadding2D(padding=((0,1),(0,1))))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(128, kernel_size=3, strides=2,
padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(256, kernel_size=3, strides=1,
padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(512, kernel_size=3, strides=1,
padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    return model

def save_images(cnt,noise):
    img_array = np.full((
        img_margin + (img_rows * (gen_square+img_margin)),
        img_margin + (img_cols * (gen_square+img_margin)), 3),
        255, dtype=np.uint8)

    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5

    img_count = 0
    for row in range(img_rows):
        for col in range(img_cols):
            r = row * (gen_square+16) + img_margin
            c = col * (gen_square+16) + img_margin
            img_array[r:r+gen_square,c:c+gen_square] \
                = gen_imgs[img_count] * 255
            img_count += 1


    output_path = os.path.join(data_path,'output')
    if not os.path.exists(output_path):
        os.makedirs(output_path)

    filename = os.path.join(output_path,f"train-{cnt}.png")
    im = Image.fromarray(img_array)
    im.save(filename)

generator = create_generator(seed_vector, img_chan)

noise = tf.random.normal([1, seed_vector])
gen_img = generator(noise, training=False)

plt.imshow(gen_img[0, :, :, 0])

img_shape = (gen_square,gen_square,img_chan)

discriminator = create_discriminator(img_shape)
decision = discriminator(gen_img)
print (decision)

cross_entropy = tf.keras.losses.BinaryCrossentropy()

def discrim_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output),
real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output),
fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

```
def gen_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output),
fake_output)

gen_optimizer = tf.keras.optimizers.Adam(1.5e-4,0.5)
disc_optimizer = tf.keras.optimizers.Adam(1.5e-4,0.5)

@tf.function
def train_step(images):
    seed = tf.random.normal([num_batch, seed_vector])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as
disc_tape:
        gen_imgs = generator(seed, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(gen_imgs, training=True)

        g_loss = gen_loss(fake_output)
        d_loss = discrim_loss(real_output, fake_output)


        gradients_of_generator = gen_tape.gradient(\
            g_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(\
            d_loss, discriminator.trainable_variables)

        gen_optimizer.apply_gradients(zip(
            gradients_of_generator,
generator.trainable_variables))
        disc_optimizer.apply_gradients(zip(
            gradients_of_discriminator,
            discriminator.trainable_variables))
    return g_loss,d_loss

def train(dataset, epochs):
    fixed_seed = np.random.normal(0, 1, (img_rows * img_cols,
                                       seed_vector))
    start = time.time()

    for epoch in range(epochs):
        epoch_start = time.time()

        g_loss_list = []
        d_loss_list = []

        for image_batch in dataset:
            t = train_step(image_batch)
            g_loss_list.append(t[0])
            d_loss_list.append(t[1])

        generator_loss = sum(g_loss_list) / len(g_loss_list)
        discriminator_loss = sum(d_loss_list) /
len(d_loss_list)

        epoch_elapsed = time.time()-epoch_start
        print (f'Epoch {epoch+1}, gen
loss={generator_loss},disc loss={discriminator_loss},'\
           f' {time_string(epoch_elapsed)}')
        save_images(epoch,fixed_seed)

    elapsed = time.time()-start
    print (f'Training time: {time_string(elapsed)}')

train(train_dataset, epochs)

a = imread('/content/drive/MyDrive/Datasets'\
           '/apple-or-tomato/training_set/output/train-0.png')
plt.imshow(a)

a = imread('/content/drive/MyDrive/Datasets'\
           '/apple-or-tomato/training_set/output/train-1.png')
plt.imshow(a)

a = imread('/content/drive/MyDrive/Datasets'\
           '/apple-or-tomato/training_set/output/train-
25.png')
plt.imshow(a)

a = imread('/content/drive/MyDrive/Datasets/apple-or-tomato'\
           '/training_set/output/train-999.png')
plt.imshow(a)
```

PRISCILLA

priscilla.fitped.eu