|  |
| --- |
| IUBH |
| Deep Learning |
| DLBDBDL01 |

# Learning Objectives

Recent technological advances mean that a number of concepts and methods derived from artificial intelligence can now be applied in practice. Neural networks are a key concept affected by this progress. Thanks to faster and more cost-effective GPUs combined with freely available, well-documented frameworks, neural networks are now being used to solve a wide variety of problems, ranging from pattern recognition in text and images to the automated assessment of insured losses. In the **Deep Learning** course, students are introduced to the fundamental principles of this technology and equipped with the skills to apply these principles using simple examples.

# Unit 1—Introduction to Deep Learning

Study goals

After completing this unit, you will know ...

… how the term “artificial intelligence” is defined.

… what the subareas of artificial intelligence are.

… what “machine learning” means.

… which subgroups machine learning is divided into.

… what deep learning is and how it has developed.

… what types of deep learning frameworks have been created.

# 1. Introduction

### Introduction

As a concept, deep learning has long been viewed as the driving force behind research into artificial intelligence. In fact, the terms “artificial intelligence” and “deep learning” are often used synonymously, with no distinct boundaries between the two subject areas. So what exactly is artificial intelligence? What defines deep learning and how does it relate to machine learning?

The aim of this unit is to answer these questions and provide a clear definition and breakdown of the various terms used, giving the reader an overview of the various aspects of artificial intelligence and illustrating the most common methods used in machine learning. In the next stage, deep learning is subsequently presented in a historical context, the key concept is explained, and key application areas are discussed. Finally, the reader is given an overview of established deep learning frameworks, to lay the groundwork for a more in-depth study of this subject.

## 1.1 Artificial Intelligence

Artificial intelligence (AI) is omnipresent in everyday life and, in today’s world, almost impossible to imagine living without. As a subarea of information technology, AI focuses on solving problems which are deemed—from a human perspective—to require intelligence or attempts to imitate human problem-solving behavior. But how exactly do we define intelligence? There is no categorical answer to this question as no precise or uniform definition actually exists for this term. In the context of artificial intelligence, however, there are a number of features which characterize it:

* Making decisions: complex decisions can be made correctly, optimally (based on previously defined rules), and precisely on the basis of input factors.
* Verifying results: decisions which have been made can be substantiated.
* Logical thinking: problems which are not substantiated mathematically can be assessed using logical thinking.
* Learning and improving: the system learns from existing or new data in order to bring about self-improvement.

This enables an approximate description of the characteristics a system must possess in order to be regarded as “intelligent”.

The field of artificial intelligence can be divided into three large subgroups, on the basis of competence type, which each have various facets (Kaplan/Haenlein 2019):

**Cognitive**

Derived from the Latin word “cognoscere”, cognition describes the processing of information.

* **Analytical AI:** the majority of AI systems encountered today belong to the group of analytical AI systems, which are only consistent with **cognitive** intelligence. In these systems, a cognitive representation of the environment is generated and learned for the future on the basis of decisions made in the past. Image and speech recognition, autonomous vehicles, and strategic game-solving are among the key examples found in this category.
* **Human-inspired AI:** in addition to cognitive intelligence, this category also includes emotional intelligence, i.e., the comprehension and analysis of human emotions, in the decision-making process. Emotions, such as joy or sadness, can be detected through face or speech recognition, thus enabling the use of emotionally intelligent applications. Virtual customer service products, which can detect customer mood and take specific countermeasures as required, represent one such example of this application.
* **Humanized AI:** humanized AI is set to take the next step toward “human behavior” and add a social component to cognitive and emotional intelligence, with the aim of rendering reliable machine-human and human-machine interactions possible in the future. This type of artificial intelligence is not yet available and requires further in-depth research. Possible future applications of humanized AI could include use in autonomous assistance systems, which require a high degree of social intelligence.

In addition to grouping based on the competence levels of an intelligent system, a fundamental distinction can also be made between “weak” and “strong” AI. These terms were coined by **John Searle** and can be illustrated using a thought experiment known as the Chinese Room (Searle 2009). The assumption behind this experiment is that science has advanced to such a degree that it has taught a computer to understand Chinese. The computer accepts Chinese characters as input values and, following the instructions of a computer program, produces other Chinese characters as output. Suppose the computer then performs this task with such conviction as to convince a Chinese-speaking person that it is itself a live Chinese speaker. It gives appropriate responses to all of the questions that the person asks, leading the Chinese speaker to assume that they are speaking to another Chinese-speaking person.

**John Searle**

The American philosopher John Searle is regarded as one of the most prominent critics of strong artificial intelligence.

The question John Searle is looking to answer is whether the machine literally understands Chinese or is merely simulating the ability to understand Chinese. The former stance is known as strong AI and the latter as weak AI. Weak AI looks to solve specific application problems without generalizing them to a large degree. As part of this process, “anthropomorphic” qualities and thus intelligence are merely simulated.

The majority of current AI applications feature a weak level of AI. Speech recognition systems controlled using a limited number of pre-programmed examples are one such example. A weak AI system maps input values to a specific output. In contrast, strong AI systems go beyond the scope of simply simulating intelligence and acquire the ability to act with human intelligence, possibly even acquiring consciousness. In this instance, input data is not strictly mapped to output data. Instead, it is processed on flexible terms by **clustering** semantic relationships and assignments, with the result that a system creates associations autonomously from related data that it has not previously been taught. Furthermore, this system is not confined to a specific remit, but is intended to operate universally and autonomously, as encountered with human intelligence. Strong AI does not exist to date. Scientists have long been unsure whether it would actually be achievable but are now largely confident that such a system will be developed in the future.

**Cluster**

In terms of data analysis, a cluster describes the grouping of objects with similar properties.

Artificial intelligence can be taken as a generic term which covers a variety of different methods, the most important of which are shown in the following figure.

Diagram

Description automatically generated

* Machine learning: the term “machine learning” refers to a system that uses sample data to generate a static model, which can then be applied to unknown data for problem-solving purposes, such as in classification or regression. Not only are assignments “committed to memory” during this process, but patterns in the data are also recognized and learned so that the solution can be **generalized**.

**Generalizability**

A model is regarded as generalizing if it exhibits a high performance level with unknown data.

* Speech recognition: speech recognition is concerned with the detection of human speech, whereby acoustic signals are used to determine which words have been spoken.
* Computational linguistics: computational linguistics verges on speech recognition, whereby the aim is for the **semantics** of text and speed data to be “understood”. Factors, such as different dialects and incorrect grammar, for example, play a key role in this.

**Semantics**

Semantics describes the content or meaning of texts.

* Vision: machine vision refers to the handling of problems which originate from the field of human vision. Important applications relating to this area can be found in object recognition and localization.
* Expert systems: expert systems assist people with problems that would normally require assistance from human experts. These are interactive decision-making systems, which are largely based on “if-then” steps but can also include **heuristics** in decisions.

**Heuristics**

Derived from the Greek word “heuriskein”, meaning “to find”, heuristics describe strategies for making decisions with limited knowledge.

* Planning and optimization: the subarea of automatic planning and optimization deals with finding solutions to complex control problems which can be difficult to solve using conventional methods due to the need for optimization within high-dimensional spaces.
* Robotics: robotics as a subgroup of artificial intelligence deals with the fusion of physically present, programmable machines and intelligence in order to develop highly autonomous and (in the future) emotionally and socially intelligent robots.

The subgroups of artificial intelligence overlap and also merge to some extent, with the result that the areas are not clearly defined.

### Self-Check Questions

1. Please mark the correct answer(s) with a cross.

* Analytical artificial intelligence has a higher level of competence than humanized artificial intelligence.
* *Weak artificial intelligence maps input values to output values based on what has been learned.*
* Artificial intelligence is a subgroup of machine learning.

## 1.2 Machine Learning

As a subgroup of artificial intelligence, machine learning is the overarching term for different algorithms which learn to recognize relationships within data on the basis of examples. The systems taught in this manner are able to apply the statistical models developed during the **training phase** to new, unknown data, and, using generalization, to recognize previously learned patterns within them, for the purpose of carrying out classifications or regressions, for example. Such algorithms are motivated by the fact that many problems can only be solved with great difficulty, or not at all, using traditional methods. The fundamental idea is to accumulate data which describe a specific problem, and then learn a way to solve the problem using these data.

**Training phase**

During the training phase, the output of a model is approximated in stages to the ground-truth values.

Machine learning can essentially be divided into three subgroups, which will be covered in greater detail below.

### Supervised Learning

The first subgroup of machine learning is known as supervised learning. As the name suggests, the learning process is carried out under supervision, meaning that for each input value of the input data x ∈ X, there is a corresponding output value y ∈ Y. The known output data are referred to as **ground truth data**. The aim of a supervised learning algorithm is to approximate a mapping function of the input data to the output data f: X → Y in such a way that unknown data can also be approximated correctly. In order to evaluate the performance of a supervised algorithm, a cost function is defined, which is optimized during the learning process. The problems to be solved in supervised learning can essentially be divided into two different groups: classification problems and regression problems.

**Ground truth**

The quality of ground truth data determines how a model performs.

In the case of classification, the output data are represented by discrete values that can be described as different classes. A typical example would include the classification of objects in images (for example, cat, dog, etc.) or even the classification of components (defective/not defective). If a dataset is represented by two classes only, this is known as a binary classification problem, while the term “multi-class classification” is used for datasets with a larger number of classes. In both cases, each input value is assigned exactly one class, whereas with “multi-label classification”, several classes can describe a single input value at the same time. This applies, for example, if an image contains several objects that are to be classified (e.g., image 1: cat and dog, image 2: cat and horse, etc.). The left-hand part of the following figure shows a binary classification of input data illustrated in the form of a schematic diagram.

In the case of regression problems, the value range of the output data is continuous, meaning that the input data map to continuous numerical values. The simplest example is linear regression. In this, linear mapping is sought in the event of one-dimensional input data, which can optimally approximate the mapping of the input values to the output values (least squares method). A linear regression of one-dimensional data is shown as a schematic representation on the right-hand side of the following figure.

Chart, scatter chart

Description automatically generated

### Unsupervised Learning

In contrast to supervised learning, in which the ground truth data must be known in order to learn relationships within data, the aim of unsupervised learning is to recognize patterns or anomalies solely on the basis of the available input data. Learning takes place in a self-organized manner, without the inclusion of any information on the actual target values. One of the most widespread methods used in conjunction with unsupervised learning is cluster analysis, in which elements of a dataset are examined for similarities, and attempts are made to define subgroups within them that represent the clusters. Different algorithms are then available depending on whether the number of clusters is assumed to be known or unknown. Generally, there are five major subgroups of clustering methods that cluster the data in different ways (Madhulatha 2012).

#### Hierarchical

In hierarchical clustering methods, a distance measure is defined for the purpose of determining clusters, in which different **metrics** are used. Hierarchical clustering can take place on an **agglomerative** or a **divisive** basis. In the case of the former, each individual object in the dataset initially forms its own cluster. Adjacent clusters are then **merged** step by step using a fusion algorithm until either a minimum required number of clusters is achieved or the determined clusters reach a previously defined distance from each other. In divisive hierarchical clustering, cluster selection takes place using exactly the same method but in reverse. First, all objects are assigned to a common cluster, which is gradually broken down. There is no requirement to specify the number of clusters in the hierarchical clustering method: defining the required distance is sufficient. For a detailed description of the hierarchical methods, please consult the relevant literature (Johnson 1967).

**Metric**

A metric assigns a real value to two elements, rendering them quantifiable.

**Agglomerative and divisive**

Agglomeration refers to the simplification of a structure, and division refers to the elaboration of a structure.

**Fusion**

This refers to the joining of elements.

#### Partitioning

Knowledge of the number n of clusters within the data is fundamental to the partitional clustering method. In this process, the dataset is partitioned on an **iterative** basis using n cluster centers, so that the individual elements are assigned to the respective clusters closest to them. With each step, there is a shift both in the cluster centers, which represent the mean value of all cluster elements, and in the assignment of the individual elements to the clusters themselves.

**Iterative**

Denotes the gradually repeated calculation.

#### Density-based

As the name suggests, the characterizing parameter of the density-based clustering method is the density of elements within a specific region. Dense areas are defined as clusters, which are separated from each other by low-density areas. Elements which are not assignable to any density area of a cluster are labeled as **noise**.

**Noise**

Outliers are defined as noise.

#### Model-based

Model-based clustering methods use mathematical models to carry out clustering, in which certain probability/distribution functions are assumed. Gaussian mixture models, which are composed of several normal distribution functions with different mean values and standard deviations, are often used as the basis here. For a detailed description of Gaussian mixture models, please consult the relevant literature (Reynolds 2015).

#### Grid-based

In grid-based clustering methods, the data area is divided into a finite number of cells, which are used to perform the operations. Each element is, therefore, located in a certain cell. Clusters are identified on the basis of a pre-defined density of elements within a cell. Cells that fall below the density threshold are removed from the grid and adjacent cells with a sufficiently high number of elements are merged together to produce the final cluster.

### Reinforcement Learning

The methodology behind reinforcement learning can be explained using the following example. A toddler is attempting to learn to walk. For every “step” taken toward walking (standing up, standing still, etc.), the toddler is rewarded by his or her parents (applause, smile, etc.), with no reward given in the event of failure. In this way, the toddler independently learns the individual steps that lead to walking, simply through the reinforcing positive outcomes. If we apply the same example to a technical description of this methodology, an **agent** learns to solve a problem autonomously by being rewarded at certain times and by seeking to receive the greatest possible reward. We will close on this abstract definition of reinforcement learning, as a detailed description would exceed the scope of this course book. For more in-depth information, please consult the relevant literature.

**Agent**

An agent denotes a program that is capable of virtually independent behavior.

### Self-Check Questions

1. Name the three areas that machine learning is divided into.

Supervised learning, unsupervised learning, reinforcement learning

1. Which subgroup of machine learning do the following methods belong to?

Regression: supervised learning

Partitional clustering: unsupervised learning

Hierarchical learning: unsupervised learning

1. What is a ground truth value?

The ground truth value is the truth value of a target variable, i.e., the actual value that is to be approximated using a model.

## 1.3 Deep Learning

Following the introduction to the fundamental topic and to the fundamental terms of “artificial intelligence” and “machine learning” in the previous sections, we will now look at the methodology behind the major breakthroughs experienced in this area in recent years, and the significant media attention that this has received. The term “deep learning” is now omnipresent and is practically a synonym for artificial intelligence today, although a clear distinction must be made between the two. While machine learning is a subgroup of artificial learning, deep learning represents a method of machine learning that uses supervised and unsupervised learning procedures as well as mixed forms of teaching. Although the term appears to be relatively recent and appeared for the first time in the publication “Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications” (Aizenberg/ Aizenberg/Vandewalle 2000), the concept was actually introduced decades earlier. This methodology has, in fact, undergone three significant phases of research during the course of history (Goodfellow/Bengio/Courville 2015). This started with the appearance of the basic concept from 1940 to 1960 under the name **cybernetics**. The second revival, known as **connectionism**,came into effect between around 1980 and 1990, with the third and current phase coming into effect around 2006. The increased interest in reviving this concept is attributable not only to vast developments in hardware and software components but also to the availability of large amounts of training data.

**Cybernetics**

Cybernetics is generally defined as the science of control, regulation, and of the transfer of information.

**Connectionism**

Connectionism is principally concerned with the behavior of networked systems.

The fundamental models used in deep learning were originally designed as models for biological learning, which is why the term “artificial neural network” is also frequently used (Goodfellow/Bengio/Courville 2015). In symbolic terms, these models were, therefore, intended to imitate the behavior of the biological brain. During the phase known as cybernetics and inspired by the biological neural networks, linear models were initially developed for the brain function, consisting of a computing cell, which performed linear mappings of the form f: X, W → y with n input values x1, …, xn ∈ X, n **weights** , and the output value y. Having initially been defined manually by humans (see McCulloch-Pitts-Neuron [McCulloch/Pitts 1943]), algorithms were then developed on this basis. These algorithms were able to learn the weights independently using sample data, and thereby obtain the desired output value for new, unknown input values (see Perceptron [Rosenblatt 1958]). The confinement to linear relationships and resulting criticism of this concept prompted scientific interest in neural networks to wane for the first time.

**Weights**

The weights represent elements of the mapping matrix.

**Cognitive science**

Interdisciplinary cognitive science deals with the study of cognitive processes.

From around 1980, **cognitive science** developed an interest in perceptual models that could be built on actual neural implementations (Goodfellow/Bengio/Courville 2015). During this phase known as connectionism, a number of concepts were developed which are still of great significance in present-day research. The theory formulated at this time was that a distributed, networked structure of simple units (neural network) could lead to intelligent behavior. One of the associated core concepts of this theory, which plays a key role in deep learning today, is “distributed representation”, which can be illustrated using the following example. There is an image recognition system that can recognize different shapes (square, circle) and colors (red, blue, green), i.e., a total of nine different combinations. A trivial way of exploring this problem would be to generate the representation using nine memory units, which each represent a combination of shape and color. Another, more analytical, approach would be to store the information about shape and color in different cells, which not only reduces the number of cells from six to five but also allows semantically related information to be derived. The difference in object representation is shown in the Chart, scatter chart

Description automatically generatedfollowing figure.

If we apply this to a neural network, one key advantage of distributed representation lies in the saving of neurons and the resulting reduction in the number of mapping functions to be learned, with a further advantage being the autonomy between the respective classifications of the properties. By learning semantic meanings, a neural network in our example will learn to recognize color independently of shape and, in the same way, to recognize shapes independently of color.

A further concept developed during the connectionism phase, and one that remains an established concept to this day, is the training of neural networks using **backpropagation**, i.e., the feeding back of the error between predicted output and ground truth value. For a detailed introduction to the backpropagation algorithm, please consult the relevant literature (Dreyfus 1990). As scientists did not succeed in fulfilling unrealistic promises during this phase, leading to a drop in investors, and successes were achieved in other areas of artificial intelligence, support vector machines (Cortes/Vapnik 1995), and graphics models (Jordan 1998), interest in neural networks saw a further decline. The suggestion that deep neural networks were difficult to train was also regarded as a further disadvantage, however, this has since been refuted following the revival of the methodology under the term “deep learning” in 2006. Thanks to highly advanced hardware and the associated increase in computing capacity, efficient training was rendered possible in a fraction of the time previously required, and the actual efficiency demonstrated by neural networks in handling complex problems was showcased for the first time. The term “deep learning” became established through the development of new methods that made the efficient training of very deep neural networks a reality. Deep neural networks have since assumed the role of proxies within the supervised learning algorithms.

**Backpropagation**

Backpropagation is based on the partial derivation of a model’s error function.

Image and speech processing, in particular, are key application areas for deep learning. In convolutional neural networks, for example, which are used in image processing, features are extracted from the input data within the hidden layers and passed on from layer to layer. The more layers that have to be passed through, the more abstract the extracted features become. Other models, such as Long Short-Term Memory (LSTM) networks, use features of time-dependent sequences, for example, in the classification of video or speech data. Here, the neural network learns during the training phase which features need to be extracted in order to arrive at the desired output result. If we take the example of image recognition, the initial layers could extract simple features such as color, edges or similar “low-level” features, and deeper layers could extract abstract features that characterize specific object properties, known as high-level features. The key to a neural network performing well is the size of the training dataset used in its learning process. The buzzword **big data** is frequently used in connection with deep learning and refers to data that are too complex and in quantities that are too large to process using conventional data processing algorithms. Deep learning models require these vast quantities of data in order to achieve their full potential and produce excellent results. Any problem, regardless of the extent of its non-linearity, can be solved if the required quantity of training data is available. For this reason, data are the key component in solving a problem where deep learning is concerned.

**Big data**

This is the overarching term for data quantities that cannot be processed using conventional methods due to their size or complexity. It is assumed that the quantity of data is subject to exponential growth.

### Self-Check Questions

1. Please mark the correct answer(s) with a cross.

* The research phase of connectionism began earlier than that of cybernetics.
* Neural networks can approximate linear functions only.
* *The number of neurons within a neural network can be reduced using the principle of distributed representation.*

1. Please describe the fundamental structure of a neural network.

A neural network is composed of the input layer, via which the input values are transferred to the neural network, the output layer, via which the output variables are output, and one or more hidden layers, within which the function approximation is performed by adjusting the weights.

**Architecture**

The architecture of a neural network defines the number and type of neurons and layers, for example.

## 1.4 Deep Learning Frameworks

A variety of different frameworks are available for developing dedicated deep learning applications, which can be used to efficiently implement all of the necessary steps, from the definition of the **architecture** to training and testing. In the scientific arena, Python has dominated over other programming languages in the implementation of neural networks, which is largely attributable to its relatively simple, intuitive **syntax** and the direct entry point it provides for inexperienced developers. Compared with C++, for example, Python features a much steeper learning curve and also enables prototyping at a considerably faster rate as a result of being an **interpreted language**. For this reason, most published deep learning models are also publicly available as Python implementations. As a result of this popularity and the greater ease of entry for less advanced programmers, the contents of this course are confined to the development of deep learning applications using Python frameworks. It should also be noted that considerable progress is currently being made in the development of algorithms, methods, and applications of artificial intelligence through continuous publication (analog to open-source solutions). As a result, the community is continuously developing exceptionally high-quality solutions at a breathtaking rate.

**Syntax**

The syntax is the grammar of a (programming) language.

**Interpreted language**

Program code is processed sequentially using an interpreter and, in contrast to compiler languages, does not require prior conversion into machine code.

The optimal framework is chosen with reference to the intended area of application and the type of data to be processed. But what exactly is a deep learning framework? Let us look, by way of example, at the classification of image data. Now we could begin to implement the functionality of each individual component of a neural network, connect them to form larger structures, and implement the training of the weights manually. For experienced programmers, this may be achievable within a few weeks, for beginners, within months, if at all. It is precisely at this point that frameworks come into play. They allow models to be developed quickly and easily, without us having to concern ourselves with the underlying algorithms. They contain implementations of the necessary standard components, with it only being necessary to define their connections. This results in a development time that is greatly reduced. Development can take place at different abstraction levels, depending on the depth that is envisaged in the definition of a neural network structure. “High-level” frameworks enable rapid prototyping of neural networks, without requiring any understanding of the complex functionality and interactions between basic components. For initial development experiments involving deep learning applications, they provide a beneficial means of implementing functioning models and applying these to dedicated or standard datasets. “Keras” and “FastAI” are among the best-known, freely available high-level frameworks, with the latter now playing an increasingly important role.

In order to enable changes to lower-level neural network structures and allow fine adjustments, a deeper extraction level is required, which is made possible with the aid of “low-level” frameworks. These enable implementation down to the lowermost computing operations, which is why a deep understanding of the underlying algorithms should be available, to allow the power to be utilized to its full extent. In terms of frequency of use in publications, the best-known, freely available representatives of lower abstraction levels, given in descending order, are “Tensorflow”, “PyTorch”, “Caffe”, and “Theano” (Hale 2018). Low-level frameworks are frequently used as **backends** for higher extraction levels to create a more user-friendly interface. Keras, for example, runs on Tensorflow or Theano, while Pytorch acts as the backbone for FastAI. This results in a combination which permits the manipulation of underlying mathematical operations at low-level and user-friendly data handling at high-level.

**Backend**

The frontend works in opposition to the backend, as the interface to the user.

Diagram

Description automatically generated with medium confidenceThe fundamental development process of a deep learning application consists of the following steps, which are illustrated in the next figure.

The first step begins with the design of the neural network architecture. The type of data (for example, image, sound, text, etc.) and the prediction type on which training is to be based (classification, regression, etc.) play a decisive role here. The number, type, and size of the hidden layers of the network are defined, and the type and size of the input and output layers are specified to enable processing of the available data. In the second step, the **hyperparameters** are defined. These are all the parameters that are set before training of the neural network commences, and include the learning rate (Zuklifli 2018), i.e., the “extent” to which the weights of the neurons are supposed to change as a result of the training. A further hyperparameter is the batch size, which represents the amount of data that is fed into the network with each training cycle. The size of the current dataset divided by magnitude of the batch size gives the number of iterations needed to run the entire dataset through the network once. The formula for this is thus:

**Hyperparameters**

The prefix “hyper” reinforces the following word, which in this case refers to the important, overriding parameters.

A further hyperparameter is the number of epochs, or rather the number of cycles in which the entire dataset was used once to update the network weights. The required training period and the performance of a neural network are decisively influenced by the choice of underlying hyperparameters (Claesen/De Moor 2015). Gradual adjustments to these hyperparameters after individual training cycles, as a means of improving the results, are a normal part of the development process. The functionality of a neural network can also fail completely if the wrong hyperparameters are chosen.

The training step follows the definition of the hyperparameters, whereby the initial weights of the neurons are initialized as standard using either the value 0 or random (low) values (Yadav 2018). The input data of the training dataset are fed into the neural network and the predicted output values determined through the initialization of the neural weights are compared with the actual output values of the dataset. The difference is then used to recalculate the weights of the individual neurons for each layer, with backpropagation being the method most frequently used for this. The aim of the training is to minimize the error in the output values.

Chart

Description automatically generatedThe final step of the development process involves applying the fully trained network to previously unknown data and evaluating its performance. This serves to establish how well the actual output values are approximated and how well the network can generalize. Poor performance during the evaluation phase indicates a model which has been trained to an overly specific degree on the training data and has simply committed all actual outputs of the input data “to memory”. This behavior is known as **overtraining** and is illustrated in simplified form for a simple regression model in the following figure.

The figure shows that the red model performs a good regression, which also delivers a good result for unknown data points. The blue model, on the other hand, exhibits memorizing behavior, which regresses the training data with a high degree of accuracy, but produces significant deviations where new data are concerned. If the development model performs poorly, either the architecture can be adjusted by reducing or increasing the number of layers, for example, or the hyperparameters can be changed, which is known as “hyperparameter tuning”.

### Self-Check Questions

1. Which group of deep learning frameworks allows deep learning applications to be developed without in-depth knowledge of the fundamental mathematical operations: low-level or high-level frameworks?

High-level frameworks

1. What steps can the development of deep learning applications be divided into?

Architecture design, definition of hyperparameters, training and evaluations

|  |
| --- |
| Summary |
| Artificial intelligence (AI) is a subarea of information technology that is divided into a multitude of subcategories. The following abilities are the core features that characterize AI: making decisions, verifying results, logical thinking, learning, and improving. Depending on the level of competence, a distinction can be made between analytical, human-inspired, and humanized artificial intelligence, whereby most of the systems in existence today are assigned to the first two “weak” AIs. “Strong” AI remains a research topic for the future.  Machine learning is one of the main areas to emerge from artificial intelligence, and one in which distinctions can be made between various learning methods. In supervised learning, attempts are made through model training to approximate existing truth values as effectively as possible. In contrast, the aim of unsupervised learning is to find regularities in data without the availability of truth values. In reinforced learning, a model learns optimal behavior autonomously by receiving an award at specific times.  Deep learning is arguably the most popular subgroup of supervised machine learning today and is based on neural networks composed of several hidden layers of neurons, which enable the approximation of highly complex, non-linear functions. The fundamental concept behind this method has made several appearances in the past under the previous names of cybernetics and connectionism. The renewed popularity of this concept is due to the highly advanced hardware available today, which has now made it possible to train the neural networks efficiently.  A variety of different frameworks are available for developing deep learning applications, which can be divided up according to the degree of abstraction. These frameworks permit direct entry into the methodology, without the need for an in-depth understanding of the underlying operations. The workflow involved in creating deep learning applications with the aid of such frameworks follows the steps of architecture design, definition of hyperparameters, and training and evaluations. |

# Unit 2—Introduction to Neural Networks

Study goals

After completing this unit, you will know ...

… what linear regression is.

… how linear regression can be used to predict target variables.

… what logistic regression is.

… how logistic regression can be used as a linear classifier.

… what the basic functionality of a perceptron is and what problems are associated with its application.

… how a multilayer perceptron is structured and how it can be used to solve problems associated with the single-layer variant.

# 2. Introduction to Neural Networks

### Introduction

Inspired by biological systems, artificial neural networks operate as abstract models to process information and solve complex problems. They are structured according to the connectionist paradigm, meaning that they consist of a network of identical building blocks, which, in isolation, can only approximate simple non-linear functions, but, as a whole, can approximate complicated non-linear functions.

In order to gain a fundamental understanding of how such a neural network operates, we must first look at how these minute basic elements, known as neurons, work. The various types of regression used to solve simple regression and classification problems will, therefore, be addressed first. The subsequent discussion of the perceptron concept revisits the previously introduced topic of linear classification and uses a series of examples to work through associated problems, the solutions to which will be explained later in the unit with the expansion to multilayer perceptrons. A brief overview of the various subgroups is also provided.

## 2.1 Linear Regression

Linear regression is the simplest and most frequently used model in regression analysis. Here, the statistical relationships between continuous **influencing and target variables** are assumed to be linear and an attempt is made to parameterize a linear model in such a way that the absolute error between it and all data points used to describe the target variable as a function of the influencing variables is minimized. A fundamental distinction can be made between simple and multiple linear regression, with the former relating only one influencing variable and one target variable to each other at a time, and the latter using multiple influencing variables. The following material will be restricted to simple linear regression in the first instance, and subsequently expanded to include the multiple case. It should be noted that multiple and multivariate linear regression are not the same thing, as the latter also involves multiple target variables.

**Influencing and target variable**

The influencing variable is also referred to as the independent variable, and the target variable as a dependent variable.

In simple linear regression, the linear relationship between influencing and target variables can be described using two parameters, β0 and β1, and thus be represented by a linear function, i.e., a straight line, of the following form:

Here, y represents the target variable and x the influencing variable. The method of least squares is a sophisticated and easy-to-use procedure for determining the two model parameters, in which the sum of the squared differences between each data point and the regression line is minimized. The advantage of using the squared error over the mean error is that errors with a positive sign, i.e., where the data point lies above the line, and a negative sign, where a data point lies below the line, do not cancel each other out. The function to be minimized when using the least squares method can be specified using the following equation:

The target variable of the data points at the i-th position is represented by yi and the values of the regression line in identical position are represented by pi. Substituting the straight line equation as a function of the two parameters β0 and β1 into the function to be minimized, gives the equation:

**Elliptical paraboloid**

An elliptical paraboloid can be constructed by rotating a parabola about its y-axis.

For the purposes of illustration, this equation describes an **elliptical** **paraboloid** that has precisely one minimum. It has been established from analysis that the extreme points of a function can be determined analytically through differentiation and by equating with zero. As a multi-dimensional function is involved here, the partial derivatives are calculated relative to β0 and β1 and each equated with zero. This results in a linear equation system with two unknowns, from which the two parameters can be clearly determined:

Solving the equation system results in the following equations for the two regression line parameters. The full calculation path is available to interested readers in Eschler (2016), for example.

The regression line parameterized using β0 and β1 thus minimizes the quadratic error between this and all data points. A statement about the relationship between target and influencing variable can be made on the basis of the sign used with β0. If the sign is positive, the target variable increases with an increase in the influencing variable. If the sign is negative, an increasing influencing variable will lead to a reduction in the target variable. In graphical form, the parameter β0 represents the slope of the line and parameter β1 represents the y-intercept.

As an example, let us take an imaginary dataset which gives smartphone sales as a function of advertising investment. The following table shows sample values for ten consecutive years.

Sample Dataset

|  |  |  |
| --- | --- | --- |
| Year | Smartphones sold (in millions) | Investment in advertising (in million €) |
| 1 | 5.12 | 2 |
| 2 | 8.49 | 3 |
| 3 | 21.98 | 5 |
| 4 | 21.57 | 7 |
| 5 | 31.89 | 8 |
| 6 | 36.61 | 10 |
| 7 | 44.00 | 12 |
| 8 | 45.77 | 13 |
| 9 | 45.15 | 14 |
| 10 | 54.34 | 16 |

These data can be represented graphically using a scatter diagram, which can be generated using Python as follows:

1. # Importing the libraries
2. **import** numpy as np
3. **import** matplotlib.pyplot as plt
4. # Defining the data lists
5. X=[2,3,5,7,8,10,12,13,14,16]
6. Y=[5.12,8.49,21.98,21.57,31.89,36.61,44.00,45.77,45.15,54.34]
7. # Plotting
8. plt.figure()
9. plt.scatter(X,Y)
10. plt.show()

The following figure shows the result of the scatter diagram that has been generated, which suggests a linear relationship between advertising costs invested and smartphones sold. It should be noted that the subjective assumption of linearity, which appears clearly plausible in this example, requires further mathematical (statistical) methods to substantiate it and is only ever predicable in the context of an error interval to be defined. For further information about this, please refer to the corresponding statistics textbooks.

Chart, scatter chart

Description automatically generated

Using the parameter equations, the two parameters of a linear model can be determined as follows for this dataset:

1. # Generating the model
2. X\_mean = np.mean(X)
3. Y\_mean = np.mean(Y)
4. num = 0
5. den = 0
6. **for** i **in** range(len(X)):
7. num += (X[i] – X\_mean)\*(Y[i] – Y\_mean)
8. den += (X[i] – X\_mean)\*\*2
9. b0 = num/den
10. b1 = Y\_mean – m\*X\_mean

The following figure shows the parameterized regression line together with the scatter diagram of the data. The positive sign for parameter β0 indicates that the number of smartphones sold (linear) rises with increasing investment in advertising.

Chart, scatter chart

Description automatically generated

The Python library **scikit-learn** permits the regression parameters to be directly calculated from data, without the need for dedicated implementation. The same result can be obtained for the two parameters β0 and β1 for the sample dataset using the following lines of code:

**Scikit-learn**

The Python library scikit-learn contains numerous tools that are important for the field of data analysis.

1. # Importing the library
2. **from** sklearn.linear\_model **import** LinearRegression
3. # Calculation of the regression parameters
4. X=X.reshape((-1,1))
5. model = LinearRegression().fit(X, Y)
6. b0=model.coef\_
7. b1=model.intercept\_

The influencing variable format of the data must be adjusted here, as the algorithm requires a column vector. The parameters determined in this way correspond with the self-calculated parameters, allowing the number of lines of code required to be more than halved.

The restriction to a single influencing variable can be lifted through expansion to the multi-dimensional case. The general form of the linear function is retained here, with Y and β now represented by column vectors of the value () with number of target variables T, and X by a matrix of the value () with number of influencing variables M.

Each element of the target variable vector corresponds to one data point in each case. The resulting regression model represents a hyperplane in (M + 1)-dimensional space. **Hyperplanes** in two-dimensional space, i.e., in the presence of an influencing variable, are represented by the lines previously covered. The parameters that describe the regression hyperplane can also be determined in the multiple case using the least squares method. However, the computing effort involved in this is several times higher than for simple linear regression. For a deeper insight into the derivation of the necessary equations, please consult the relevant literature (Kronthaler 2015).

**Hyperplane**

The hyperplane generalizes the concept of the plane to spaces of any dimension.

### Self-Check Questions

1. What is the method of linear regression used for?

Linear regression allows predictions to be made about a target variable if linear relationships exist between this target variable and the input variable(s).

1. Please mark the correct answer(s) with a cross.

* Linear regression belongs to the unsupervised learning group.

**Discrete**

Discrete variables can only assume a finite number of values. Their opposites are continuous or steady variables.

* Linear regression predicts a probability.
* *Linear regression permits a forecast beyond the input data range.*
* Data can be classified using linear regression.

## 2.2 Logistic Regression

Logistic regression is a simple method used within regression analysis to predict relationships between a **discrete** target variable and one or more metric or categorical influencing variables. In contrast to linear regression, it is not the target variable as a function of the influencing variable that is predicted here, rather the probability that the target variable will assume a specific value due to the occurrence of one or more influencing variables. As a result, logistic regression provides a simple approach to solving classification problems. If the target variable in question is binary, this is referred to as a “binary logistic regression model”. The target variable is then also described as **dichotomous**. If the target value can assume more than two categories, this is referred to as multinomial logistic regression and the target variable is also called polychotomous. In this section, we will look at the binary case first, and subsequently expand the set of problems to the multinomial case.

**Dichotomous**

The term dichotomous essentially means having two parts and polychotomous means having several parts.

Chart

Description automatically generatedTo illustrate this better, let us look at the following simple example: a given dataset gives the dichotomous target variable “exam passed” as a function of the influencing variable “number of hours spent studying”. 20 data points from this dataset are shown in the following figure, where the output of the target variable “passed” is indicated by the value 1 and the output value “failed” is indicated by the value 0.

The data are clearly separated from each other and divided into two distinct groups. So how do we define a model that will make a prediction about the outcome of an exam for a given number of hours spent studying? The inexperienced user may take the initial approach of applying a linear regression model to the dataset. However, this will fail due to the fact that the model does not determine probabilities, but it assigns a specific output value of the target variable to each value of the influencing variable. A regression line for the available data would appear as shown in the figure below.

Chart, scatter chart

Description automatically generated

A number of characteristics of linear regression are visible from this representation, which rule out its suitability for **binary classification problems**. The first thing we can see is that the value range of the regression line, i.e., the quantity of values that the target value can assume, also delivers values outside the range that is valid for a probability assignment [0,1]. Furthermore, no threshold value can be specified which permits a distinction between the two possible outputs. The logistic function presented by Pierre François Verhulst in a series of publications addresses these very weaknesses and describes a suitable problem-solving approach to represent binary classifications, or saturation processes in general (Cramer 2002). This one-dimensional probability distribution, which belongs to the family of “sigmoid functions”, is described using the following equation:

**Binary classification problems**

In binary classification problems, there are just two classes to which the data are assigned.

Assuming that t represents a linear combination of two regression parameters β0 und β1, this gives:

The function t = β0 ∙ x + β1 is referred to here as linear predictor. The logistic function, therefore, carries out a non-linear mapping of a linear function onto the value range [0,1], thus enabling the assignment of probabilities. The zero of the linear predictor indicates the decision limit, i.e., the threshold that separates the two classes from each other. In contrast to linear regression, the regression parameters are not directly determinable in analytical terms using the least squares method but must be calculated iteratively. The maximum **likelihood** method is often used for this purpose. This estimates the regression parameters that the present data are most likely to occur for (Czepiel 2002). Further reading on this method is available to interested readers (Behnke 2014). The Python library scikit-learn provides a class for applying logistic regressions, which assists in the simple analysis of dedicated datasets and allows new data to be classified using the constructed model. The following code determines the regression parameters for the linear predictor of the logistic function for our sample dataset:

**Likelihood**

In probability theory, the likelihood indicates a probability of unknown values for parameters.

1. # Importing the library
2. **from** sklearn.linear\_model **import** LogisticRegression
3. **from** matplotlib **import** pyplot as plt
4. **from** scipy.special **import** expit
5. # Calculation of the regression parameters
6. model = LogisticRegression(C=1e5)
7. model.fit(X,Y) # X and Y contain the influencing and target variables of the data
8. # Estimated parameters
9. model.coef\_ # b0
10. model.intercept\_ # b1

Plotting the estimated logistic function with the calculated regression parameters β0 ≈ 8.4 and β1 ≈ –88.4 together with the data points of the sample dataset results in the curve shown in the figure below.

Chart

Description automatically generated

For our dataset, the zero point of the linear predictor is at approximately 10.5 hours, which means that the probability of passing and failing is 50% in each case. The probability of new data falling within the respective class can be determined using the following sklearn function:

model.predict\_proba(D) # D contains the data to be classified

If the target variable is dependent on more than one influencing variable, the linear combination of the linear predictor can be expanded, resulting in the following expanded equation:

Our sample dataset could be expanded to include factors such as the grade in previous exams, or rest time in hours prior to the exam, etc.

Let us now expand our analysis to more than two possible target variable categories. We know from the probability calculation that the sum of the probability of a variable and of the complementary probability is always 100%. In the binary case, the complementary probability can be derived directly by subtracting the probability of class 1 from 100%. Drawing a direct conclusion about the complementary probability is not as easy in the multinomial case, as this is produced from the sum of all probabilities for the other categories. With K possible categories, the probability distribution function expands to:

Expanding the target variable from our sample dataset to more than one category could mean separation into individual grade scales, for example. The representation of the probability distribution is no longer straightforward and requires K functions, which each represent the probabilities for the respective category and give a total of one at each point.

### Self-Check Questions

1. How can the output value of a logistic regression be interpreted for a specific input value?

Logistic regression is a probability function. It gives the probability of a specific input variable belonging to a specific class.

1. Please mark the correct answer(s) with a cross.

* *The linear predictor of the logistic regression equation can be used to determine the decision limit.*
* *Logistic regression belongs to the supervised learning group.*
* A logistic regression model predicts the target variable for a specific input value.

## 2.3 Perceptrons

The perceptron is a further linear classifier, alongside logistic regression, which can be used to carry out binary classifications of input data. Introduced by the psychologist and computer scientist Frank Rosenblatt in 1958, the perceptron is regarded as the first algorithm to be capable of learning using the **“trial-and-error” method** (Rosenblatt 1958). It was originally implemented as the “Mark 1 perceptron” in hardware as a standalone computer and was intended for use in image recognition (Bishop 2006). The initial euphoria surrounding the Rosenblatt perceptron quickly dissipated after it transpired that it was not suitable for the classification of many different classes, but only of **linearly separable** data, i.e., data that can be clearly separated from each other by hyperplane.

**Trial and error**

In the trial-and-error method, all possible permissible solutions are attempted until the correct outcome is reached.

**Linear separability**

If data in n-dimensional space can be separated from each other by an (n – 1)-dimensional hyperplane, then they are linearly separable.

The concept behind the Rosenblatt perceptron is the learning of a threshold function, which performs a non-linear mapping of input data to a binary output. This can be defined using the following equation, where x represents the input vector, w represents the vector of the weights, i.e., the factors of the input values, and b represents the bias:

The limit from which an input vector is assigned to the respective class is shifted with the aid of the bias. This gives the threshold from which the weighted sum of the input vector changes the sign. The term w ∙ x calculates the scalar product of the respective n-dimensional weight and input vector and is defined by:

Alternatively, the bias can be taken as the first element of the weight vector if the value 1 is added to the input vector in the first place, in order to limit the linear predictor to the scalar product of two vectors, which results in the following representation of the threshold function through summation:

where

In contrast to logistic regression, where the logistic function serves to restrict the value range to [0,1], a step function, also known as Heaviside function, is used with the perceptron as a non-linear transmission function. As a result, the output does not correspond to a probability assignment, but rather to the class assignment of the input vector only. The following figure shows the basic structure of a Rosenblatt perceptron.

Diagram

Description automatically generated

**Boolean algebra**

In Boolean algebra, laws are defined for calculations involving logical operations.

In order to be able to classify linearly separable data correctly in binary terms, it is necessary to adjust the weight vector, i.e., the perceptron must be taught on an iterative basis using training data. The training algorithm is explained below using the example of the binary OR operation of two 1-bit variables, with a description of implementation using Python. The corresponding truth table is known from **Boolean algebra** and is defined as shown in the following table.

Truth Table for OR Operation

|  |  |  |
| --- | --- | --- |
| **Input** | **Input** | **Output** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

The first step of the perceptron learning algorithm is concerned with initializing the weights of the weight vector. The type of initialization is variable and the chosen values do not influence the **convergence** of the algorithm. In this example, the three weights w0, w1, and w2 are initialized using the value 0. Each row of the truth table represents a training example consisting of an input vector

**Convergence**

When a function reaches a specific limit value after a specific time, this is described as convergence.

and a binary output value yw,i. First of all, the instantaneous output values ym,i are calculated from each input vector with the initialized weights, resulting in 0 for each xi from

The respective weights are now updated using the following equation by calculating the difference between the calculated and actual output value.

The factor l represents the **learning rate** here. For the purpose of this example, we will use the value 1 here, without further explanation. For more in-depth material and evidence that the initial choice of learning rate and weights has no effect on the convergence of the learning algorithm for linearly separable data, please consult the relevant literature (Novikoff 1963). Each weight is directly updated by each training input vector. This step is now carried out repeatedly until the outputs calculated correspond to the actual values, i.e., the difference between yw,i – ym,i is zero for each i. The perceptron learning algorithm for the OR operation can be implemented using the following Python code.

**Learning rate**

The learning rate indicates the extent to which the weights should be adjusted.

1. # Importing the library
2. **import** numpy as np
3. # Defining the input, output, and weight vectors
4. x0=np.array([1.,1.,1.,1.])
5. x1=np.array([0.,1.,0.,1.])
6. x2=np.array([0.,0.,1.,1.])
7. y=np.array([0.,1.,1.,1.])
8. w=np.array([0.,0.,0.])
9. ym=np.array([0.,0.,0.,0.])
10. # Iterate as long as the calculated and actual outputs are different
11. **while**(**not** np.array\_equal(ym,y)):
12. # Calculate instantaneous output value for all training data
13. **if** w[0]\*x0[0]+w[1]\*x1[0]+w[2]\*x2[0]>=0:
14. ym[0]=1
15. **else**:
16. ym[0]=0
17. **if** w[0]\*x0[1]+w[1]\*x1[1]+w[2]\*x2[1]>=0:
18. ym[1]=1
19. **else**:
20. ym[1]=0
21. **if** w[0]\*x0[2]+w[1]\*x1[2]+w[2]\*x2[2]>=0:
22. ym[2]=1
23. **else**:
24. ym[2]=0
25. **if** w[0]\*x0[3]+w[1]\*x1[3]+w[2]\*x2[3]>=0:
26. ym[3]=1
27. **else**:
28. ym[3]=0
29. # Weight update
30. **for** j **in** range(0,4):
31. w[0]=w[0]+1.\*(y[j]-ym[j])\*x0[j]
32. w[1]=w[1]+1.\*(y[j]-ym[j])\*x1[j]
33. w[2]=w[2]+1.\*(y[j]-ym[j])\*x2[j]

When the algorithm is run, the correct outputs and the corresponding weights of w0 = –1, w1 = 2, and w2 = 2 are obtained after five iterations. If the aim is to teach the perceptron on other logical operations, only the truth table must be adjusted, i.e., the values of output array y. For the logical AND, not AND, and not OR operation, the learning algorithm also converges to stable weights, but does not come to a solution with the **XOR operation**, where it alternates between different weights. It is here that the weakness of a single-layer Rosenblatt perceptron is demonstrated. The XOR operation is not linearly separable, which can be illustrated graphically using the following comparative representation with the AND and OR operation.

**XOR**

The XOR operation or “exclusive or” is true if precisely one of the two inputs only is true.

Chart, scatter chart

Description automatically generated

The blue dots represent the points at which the logic function outputs a 1 and the red dots represent the points at which this outputs a 0. One of the possible lines that clearly divides the data in spatial terms is shown in red. The figure shows that this must clearly be performed for both the OR and the AND function. This line cannot be found for the XOR function, which is indicated by the dashed line. The fundamental problem surrounding the need for linear separability of the Rosenblatt perceptron, taking the XOR problem as an example, was demonstrated in 1969, and not only triggered a fundamental discussion but also brought AI research to a standstill (Minsky/Papert 1969).

A major disadvantage of the presented learning algorithm for a Rosenblatt perceptron is the need for linear separability. If this criterion is not met by each element of the dataset, the algorithm will not converge. As a result, there is no solution available, aside from approximating the ideal division to the best possible extent. This problem can be approached using different approximating algorithms. Two established representatives of this group are the Maxover and Pocket algorithms (Wendemuth 1995, Stephen 1990). For a description of these alternative learning algorithms, interested readers should consult the relevant literature.

### Self-Check Questions

1. What is the fundamental criterion for the convergence of the perceptron learning algorithm?

The data must be linearly separable. In other words, for data in n-dimensional space, there must be an (n – 1)-dimensional hyperplane that completely separates the data from each other.

1. Which of the following logic functions are linearly separable?

* *AND*
* *OR*
* XOR
* *NAND*

1. How does the output of a perceptron differ from that of a logistic regression?

As a result of the different non-linear transmission function (logistic function vs. step function), the perceptron predicts the respective class, but the logistic regression predicts a probability.

## 2.4 Types of Perceptrons

The learning of a threshold function through iterative adjustment of a weight vector with the aid of a training dataset represents the fundamental structure of a single-layer Rosenblatt perceptron. This linear classifier, which is also referred to as an artificial neuron, enables the implementation of simple logic operations such as logical ANDs, ORs, or NOTs. The XOR problem discussed demonstrates the weakness and, as a consequence thereof, the associated limit in functionality. By linking several individual perceptrons to create a larger network, it is even possible to process data that are not linearly separable and, as a result, find theoretical solutions to any complex problem. Such a network is also known as a “multilayer perceptron” and in its basic form is composed of at least one input layer, one output layer, and a hidden neuron layer. The solution to the XOR problem using a multilayer perceptron, and implementation using Python, will be presented below.

The XOR operation, also known as the exclusive OR operation, outputs the value 1 when only one of the two input values assumes the value 1 and the other assumes the value 0. The associated truth table is shown in the following table.

Truth Table for XOR Operation

|  |  |  |
| --- | --- | --- |
| **Input** | **Input** | **Output** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

It is known from Boolean algebra that the XOR operation can be represented using the following expression:

Here, the operators and represent the links XOR, NAND, AND, and OR respectively. The operator, which as a whole is not linearly separable, has now been converted into two discrete terms with linear separability and with a linking function that is also linearly separable. We can, therefore, represent this function using three Rosenblatt perceptrons, which each learn an easily solvable threshold function. In this process, the inner terms and of two perceptrons are assigned to a hidden layer, the outputs of which are processed by an output layer perceptron using the AND operation. The conversion of the single-layer perceptron to a multilayer perceptron for the purposes of solving the XOR problem is illustrated in simplified form in the figure below.

Chart, bubble chart

Description automatically generated

Here, the blue circles represent the perceptrons of the input layer, the green represent those of the hidden layer, and the red represent those of the output layer. In contrast to the single-layer perceptron, nine (including the three biases of the respective perceptrons) weights must now be learned instead of three. Theoretically, each individual perceptron could now be taught to perform its specific function using the perceptron learning algorithm and then all of the perceptrons could be combined to solve the entire function. Although this approach may be feasible in relatively simple cases, it is impracticable for higher-dimensional and more complex problems. For this reason, the backpropagation method is used for multilayer perceptrons, which minimizes the squared error between the actual output and the calculated output. As a detailed presentation and implementation of this algorithm would exceed the scope of this chapter, we suggest you consult the relevant literature (Dreyfus 1990). In terms of our example, it should be sufficient to train the perceptron and show through networking that the XOR problem can be solved using a multilayer perceptron. We use the Python code from the previous section to learn the weights and acquire the following weights for the NAND, OR, and functions:

Calculated Weights of the Individual Perceptrons

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| **P1** | 1 | -1 | -1 | **NAND** |
| **P2** | -1 | 2 | 2 | **OR** |
| **P3** | -2 | 1 | 1 | **AND** |

The entire function can now be modeled as follows by calculating the networked forward path of the input data combinations:

Chart

Description automatically generatedThe outputs of the multilayer perceptron we have developed correspond to the anticipated values from the truth table for the XOR operation. In graphical terms, the function separates the data as shown in the figure below:

You can see that the data are separated by the conjunction of two linear separations, with the green line marking the division of the NAND function and the red line marking the division of the OR function. The multilayer perceptron that has been developed represents a feedforward neural network, as the data flow propagates exclusively from front to back, i.e., from the input to the output neurons. This property distinguishes the network from another class known as recurrent neural networks, which also involve reverse connections from subsequent to previous neurons. This feedback enables the processing of **data sequences**, as it represents a form of short-term memory. Furthermore, the model presented is fully connected, as each output of a previous neuron is networked with the input of a subsequent neuron. Alternatively, outputs of a neuron can also skip certain layers and be used as inputs of a neuron of a subsequent layer. As they effectively contain short circuits, models with this property are known as short-cut networks.

**Data sequence**

Data sequences include videos, i.e., sequences of images, or other sequences in which the current value is determined by previous values.

### Self-Check Questions

1. Please mark the correct answer(s) with a cross.

* Neurons in a neural network that is fully connected network their output with a neuron from the subsequent layer.
* Short-cut networks have a network from each output of a neuron with each input of the neurons of the following view.
* *In recurrent neural networks, the data also propagate from the back to the front neuron layers.*

1. Can multilayer perceptrons only be taught on linearly separable data?

No. Theoretically, any complex non-linear function can be approximated using multilayer perceptrons if sufficient data are available.

|  |
| --- |
| Summary |
| Linear regression is the simplest model for approximating data and enables the target variable of unknown data to be predicted using suitable parameterization based on sample data. The ideal parameters can be found using the least squares method, in which the sum of the squared differences between all target variable values in a dataset and parameterized model are minimized. The regression model represents a hyperplane in the (M + 1)-dimensional space, where M represents the number of influencing variables. The product in the event of a single influencing variable is a straight line.  Logistic regression is a simple method for solving simple classification problems, whereby a distinction is made between binary and multinomial logistic regression. The former refers to the existence of two classes, the problem is also known as dichotomous. The latter represents an expansion to more than two classes and is also known as polychotomous. Logistic regression models predict the probability of an input variable belonging to a specific class. The logistic function and a linear predictor form the basis of this.  The Rosenblatt perceptron is another linear classifier that can perform binary classifications of linearly separable data by learning a threshold function, along the same lines as neurobiological models. The criterion of linear separability has a significant impact where solving problems is concerned. The XOR operation, for example, cannot be learned using a single perceptron. As a result, networks composed of several perceptrons, called multilayer perceptrons, are used, which make it possible to approximate complex, non-linear functions. This network is known as a neural network and, in addition to an input and output layer, consists of at least one hidden layer. The direct linkage of all outputs from a previous layer with all inputs from a subsequent layer is known as “fully connected”, and it contrasts with short cuts where layers are skipped. If the propagation of input data runs from front to back only, this is referred to as a feed-forward network. Recurrent networks have feedback from neurons located further back in previous layers, which enables data sequence processing. |

# Unit 3—Training Neural Networks

Study goals

After completing this unit, you will know ...

… what the mean squared error represents and how to distinguish it from other errors.

… what the gradient method is and how it is used.

… how multilayer functions work and can be used.

… what function the backpropagation algorithm fulfils and how it can be used to train multilayer perceptrons.

… how the backpropagation algorithm can be implemented.

# 3. Training Neural Networks

### Introduction

The aim of this unit is to provide an understanding of the functions that form the basis of neural network training. We will look initially at an elementary cost function that is paramount in solving regression problems. We will then take a more in-depth look at the gradient method, which represents the basic mathematical method of finding an optimum for a cost function, and we will subsequently move on to the subject of multilayer perceptrons and the problems involved in teaching them. As a solution, we present the basic backpropagation algorithm and demonstrate its function in practical terms based on an internal implementation.

## 3.1 Mean Squared Error

Solving a problem using a statistical model requires a cost function that defines precisely what improvements need to be made in order to achieve the desired result, i.e., the desired target values for a set of input values. The mean squared error assumes an important role here as its minimization represents a suitable solution for a multitude of different problems. In order to explain the meaning of this error metric in greater detail, let us consider the set of tuples Y = {(1,–4), (2,–2.5), (3,–2), (4,–1.5), (5,0), (6,1.5), (7,2.5), (8,3), (9,4)}, with elements that represent pairs of influencing and target variables, and which can be illustrated in graphical form as follows.

Chart, scatter chart

Description automatically generated

The mean target value of the set can be easily determined as μ = 0. In order to draw conclusions about how well the mean value represents the set elements, an error metric is required, which indicates the average deviation of the target values from the mean value. The simplest approach to this would be to form the mean value of the differences between the determined mean value μ and each of the n target values of the set yi ∈ Y.

This gives a mean error of εm = 0, which denotes the fact that the mean value μ is a perfect estimator under the error metric used. It is, however, obvious that the mean value μ = 0 is by no respect optimal. The problem with using the mean deviation as the error metric is that the negative and positive deviations cancel each other out. In logical terms, the absolute value can be used to offset equivalent positive and negative values, with the result that the error can only assume positive values. This is known as the mean absolute error and is calculated as follows for the set Y:

This clearly shows that the mean value in no way represents an optimal estimator for the set . So why is this error function used so infrequently in the development of statistical models, particularly in the design of neural networks? The answer to this question is dependent on the manner in which a neural network is trained. The weights are adjusted using gradient methods, in which partial derivatives of the error function are required. The derivative of the absolute value function f(x) = |x| results in

and is undefined at the point x = 0. The property of the gradient of the absolute value function to remain constant for the values x > 0 and x < 0 is also a major disadvantage in the training of neural networks. This can be illustrated by the following figure.

Chart, line chart

Description automatically generated

Here, the red circles represent sample values of the absolute error function (blue) during consecutive training cycles. The figure clearly shows that the absolute error jumps back and forth in the immediate vicinity of the minimum and does not converge to any limit value. Consequently, the optimum is never reached and the statistical model designed in this way never represents an optimal solution. As a result of these disadvantages, which are associated with the use of the absolute error, the squared mean error is used instead of the absolute value, which can be calculated using the following equation:

For the set Y used in our example, this gives a value of εms = 16. Due to the squared error, no quantitative statement can be made about the actual deviation; this is only a qualitative measure of it. A comparison of two models is, however, possible on the basis of the respective mean squared errors, to determine how well they estimate the data in question. The mean squared error function is continuously differentiable and the gradient converges toward zero as values of the mean squared error endeavor to reach the minimum. In graphical terms, this property Diagram

Description automatically generatedcan be illustrated as follows:

Despite the advantages that the mean squared error has over the mean absolute error, it also has a number of negative properties. Values that deviate significantly from the usual values of a set, known as outliers, result in much higher squared errors than absolute errors and, consequently, have a much greater impact on the statistical model. There are cases in which this is desired, but also cases in which this effect should be avoided. If a training dataset is adulterated by outliers that represent unrealistic values, and these do not appear in the data that the statistical model that has been developed is to be applied to, the use of the absolute error should be considered. However, if these outliers represent reality, the mean squared error is the better problem-solving approach.

The mean squared error of the set from the example in question represents the expected value for its mean error. Due to the fact that only the mean was estimated, the mean squared error is also the variance of the data Y:

Let Y now be a function of the form Y : X → Y, which assigns a y ∈ Y to every x ∈ X. Let Ŷ be the function estimating Y, then the mean squared error can be indicated by:

where ε represents the deviation of the function to be estimated from the estimator function and is assumed to be normally distributed with a standard deviation σε and a mean value με. The following property of the anticipated value is helpful in making a statement about the mean squared error in terms of the variance and the **bias**. These values play a key role in statistical statements about models.

**Bias**

The bias of an estimator function is a parameter for quantifying the systematic overestimation or underestimation.

Multiplying out the squared function and careful solving with due regard to the mean value με and the variance σε of ε results in the following representation of the mean squared error:

This can, therefore, be represented by the sum of the variance of the model, i.e., of the estimated values of the true data values, the squared bias of the model, and the variance of the error ε. The complete solution is available to interested readers in Kurt (2019). The variance of the error σε is the parameter that limits the quality of the developed statistical model. This is constant and cannot be influenced. In practice, it is necessary to strike a compromise between low model variance and low model bias, as it is not normally possible to minimize both at the same time (with the exception of relationships that are easy to model, such as linear functions). Variance and bias can be used to analyze various problems relating to the model with the aid of test data that were not used to train the statistical model. A high variance of the model for the test data means that the training data have only been committed to memory and the estimator function is not regressing, but precisely mapping, the true function. The model is then described as “overfitted”. By contrast, a high bias of the model for the test data means that the model is unable to capture the relationship between the input and output data and, as a result, does not provide good estimates. Such a model is known as “underfitted”.

### Self-Check Questions

1. Please mark the correct answer(s) with a cross.

* The derivative of the mean squared error function is constant.
* Continuous differentiability is only present with the mean absolute error.
* *The magnitude of the derivative of the mean squared error function reduces with approximation to the minimum.*

1. Why is it sometimes advisable to use the mean absolute error instead of the mean squared error?

Outliers bring about a much higher squared error than absolute error. If outliers occur increasingly in a training dataset, but do not feature in the test dataset, it is beneficial to use the mean absolute error. However, if these outliers reflect reality, then the mean squared error should be used.

## 3.2 Gradient Method

Optimization is an essential element in the development of statistical models. Cost functions represent the error of a model and the aim is almost always to adjust the model’s parameters in such a way as to minimize this. In other words, the aim is to reach the lowest point of the function (equivalent to “lowest costs”).

The gradient method is a numerical method of solving optimization problems in which, as the name suggests, the gradient is used to find the solution. As established from multidimensional analysis, the gradient represents an expansion of the derivative term and contains the partial derivatives (directional derivatives) of a function in accordance with all variables. If we look at the function f(x0, x1, …, xi), the gradient represents a vector of the form

in which the gradient formation can be represented by the Nabla operator ∇. The result of applying the gradient to a function is provided by a gradient field, a vector field with a vector in a point P pointing in the direction of the greatest change and exhibiting an absolute value that represents the extent of the maximum change.

There are a multitude of different optimization problems. In the simplest case, a function is to be minimized or maximized, which results in a minimization or maximization problem. It is almost exclusively minimization problems that are encountered in machine learning, as these are usually modeled in such a way that the aim is almost always to minimize a cost function. For this reason, we will limit the gradient method to this type of optimization problem within this unit, and optimization will be used synonymously with minimization in this case.

The gradient method runs iteratively and, with each step, calculates the descent direction and the step size of the independent variables in order to reach the optimal point of the cost function. Following initialization of the variables, mostly using random values, each iteration is composed of the following steps:

1. calculation of the gradient, in order to determine the descent direction, and
2. updating of the variable values with the determined gradients, in order to approach the optimum.

These steps are repeated until no further improvement is achieved by updating the variables, i.e., the cost function has reached its minimum. If we consider this from a neural network perspective, we can view the independent variables as the weights of a neuron, in other words, the weights that best approximate a function are sought. For an n-dimensional function, the gradient method provides a sequence of points of the form

where the parameter αk represents the step size, known as the learning rate, and the parameter dk represents the descent direction of an iteration. The limit value of the sequence of points represents a stationary point xs of the function, at which the function exhibits a gradient of ∇f(xs) = 0. Each extreme point of a function f is a stationary point, i.e., both maxima and minima. In order to guarantee convergence to a minimum, the function in question must be convex, i.e., each connection between two random points of f must be located above the function curve. The concave function represents the opposite of this, whereby the function curve is always located above the connection between two random points. The two properties can also apply mutually to different intervals of a function. The following figure shows the different types of function curves in principle (l.—r.: convex, concave, convex and concave at different intervals).

A picture containing text, sky, different, various

Description automatically generated

Let us look first at the cost function f to find the linear regression parameters m and b for a data series D consisting of n pairs of values xi ∈ X and yi ∈ Y of the form

where

As the cost function is dependent on two variables (weights), it represents a two-dimensional function, viewed graphically as an elliptical paraboloid, with a minimum that represents the solution to the optimization problem. The function is strictly convex, meaning that a stationary point found will always represent a minimum. The problem is visualized in the following figure, in which the red Chart, surface chart

Description automatically generatedpoint represents the minimum of the function.

Let us initialize the two regression parameters with random values first, in this case setting m = b = 1. These initial values are now used to calculate the value of the cost function, which in this case means that the mean squared value must be minimized. Summation over the value pairs of the dataset D results in an error of f(m = 1, b = 1) ≈ 35. We now calculate the partial derivatives of the cost function relative to the two parameters and evaluate these at the current positions for m and b. The partial derivatives result in:

The derivative of the squared factor always results in a factor of 2. For this reason, the cost function for the mean squared error is usually expanded by a factor of , as this is cancelled out by the factor of 2 resulting from the derivative. This has no effect on the minimum of the cost function, as the position is not changed by multiplying by a constant. If we follow this procedure and assume a factor of in the cost function, we arrive at the mean squared errors with the initial weight parameterization of and . Using the gradient that has been determined, i.e., the step size in the first iteration step and a learning rate of , the rule for the weight update results in the new weights of m = 0.41 and b = 0.88. With the updated weights, the mean squared error has reduced from 17.5 to ~5.24 compared with the previous value (following the introduction of the factor of ). The steps that were implemented are then repeated until there is no further change in the weights, in other words, until the cost function has reached a stationary point, or the mean squared value is below a certain value.

In this example, we iterate until the mean squared error reaches a value of f(m, b) < 0.1, which is the case with the present configuration after around 516 iterations, and delivers the weights m ≈ 0.9 and b ≈ –4.24. The algorithm converges relatively slowly toward the stationary point. The convergence speed can be influenced with the aid of the learning rate, although caution should be exercised here. At a learning rate of 0.04, the algorithm reaches the defined mean squared error after 258 iterations and, at a value of 0.06, after 172. However, if the learning rate is too high, which is already demonstrated in this example by the value 0.08, the algorithm diverges, as it can never approximate the stationary point. Instead, it repeatedly jumps from one side to the other side of the minimum due to the magnitude of the weight update. For this reason, it is advisable to start with a lower learning rate at the expense of the convergence speed and then gradually adjust this, as it is not usually possible to conduct an analytical calculation of the optimal learning rate prior to starting the training (Russel 1999). The algorithm with a given learning rate lcan easily be reconstructed using the following Python code skeleton.

1. **import** numpy as np
2. **def** degrees(x,y,m,b,l): # X and Y values as lists, initial values for m and b, learning rate l
3. f=1
4. **while** f>0.1: # Iterate until error is reached
5. fm=0
6. fb=0
7. f=0
8. **for** j **in** range(0,len(x)):
9. f=f+0.5\*(y[j]-(m\*x[j]+b))\*\*2 # Calculation of the mean squared error
10. fm=fm+(-x[j]\*(y[j]-(m\*x[j]+b))) # Calculation of the gradient
11. fb=fb+(-(y[j]-(m\*x[j]+b)))
12. m=m-l\*fm/len(x) # Update the weights
13. b=b-l\*fb/len(x)
14. f=f/len(x)
15. **return** m,b
16. xd=[1,2,3,4,5,6,7,8,9]
17. yd=[-4,-2.5,-2,-1.5,0,1.5,2.5,3,4]
18. m,b=degree(xd,yd,1,1,0.02)

Naturally, the gradient method is not limited to cost functions with two factors to be determined but can be applied in theory to any high-dimensional problems.

### Self-Check Questions

1. What does the gradient of a function describe?

The gradient of a function results in a gradient field, with vectors that point in the direction of the greatest descent and with an absolute value that corresponds to the extent of the maximum change.

1. For which of the two function types is the gradient method guaranteed to converge to an absolute minimum: convex or concave functions?

Convex functions

1. Please give a brief description of the gradient method in your own words.

From a starting point, a drift occurs in the direction of the maximum descent until no further numerical improvement is achieved and a stationary point is reached.

## 3.3 Multilayer Perceptron

In order to learn non-linear problems, which cannot be achieved with a single neuron, these neurons are networked to form multilayer perceptrons. In addition to the input and output layers, which form the interface to the input and output of the data that are to be processed, the neural networks formed in this manner are also composed of at least one hidden layer, whereby each layer itself can be constructed from several neurons. With a sufficiently large quantity of training data, it is theoretically possible to solve any non-linear problem, irrespective of its extent, using a suitably large number of neurons and a hidden layer. A multilayer perceptron learns the relationships between a training dataset’s input and output data using the supervised learning method, and it is able to generalize this knowledge and apply it to unknown data if the training is carried out correctly, i.e., a good compromise is reached between the variance and bias of the model. Learning takes place by minimizing a cost function that must be defined on the basis of the existing problem.

To avoid processing from being confined to linearly separable data, non-linearities must be introduced, which are used to define the relationships between inputs and the degree of activation of the contained neurons. Activation functions are used for this purpose.

Originally, the threshold function, also known as the step or Heaviside function, was used for the perceptron, on the basis of which a threshold value was defined at which the output changed state. This activation function has proved problematic for most of the learning methods used today due to the lack of continuity at the point of the threshold value, as the derivative is not defined at this point and, as a result, the function is not differentiable in the traditional sense. Gradient methods for teaching neural networks require the activation functions of the neurons to be differentiable. For this reason, sigmoid functions constitute a suitable choice. Sigmoid functions can be defined by

and map the set of real numbers to the interval . Very deep neural networks predominantly use rectifier functions (also described as ReLu for “Rectified Linear Unit”) today due to the “vanishing gradient problem”, in which the weight update of the front layers of a neural network progresses very slowly or, in some cases, comes to a complete standstill. These are defined by

Diagram

Description automatically generatedand thus map the positive part of the input value linearly to the output, with the negative part mapped to the value zero. The following figures represent the fundamental course of the presented activation functions.

The functionality of a multilayer perceptron can be modeled with basic knowledge of linear algebra. Let us take the following simple neural network consisting of an input layer, an output layer, and a hidden layer, where a function that maps two input values to one output is to be approximated, and three neurons are to be used for the hidden layer:

Diagram

Description automatically generated

A fully interconnected structure is assumed, meaning that each output of each neuron in the previous layer is networked with each input of the neurons in the subsequent layer. The output of the neurons in the hidden layer can be calculated using the following matrix representation, where ϕ represents the activation function:

The output Y is produced using:

The implementation of the forward path, i.e., the flow of the input data from front to back, can be tracked using the following Python code skeleton.

1. **import** numpy as np
2. **def** relu(x)://Implementation of the ReLu activation function
3. **return** max(0,x)
4. **def** multilayer\_perceptron(x,W\_h,b\_h,w\_o,b\_o):
5. //Arguments: Input values x=[2,1]
6. // Weights hidden layer w\_h=[3,2]
7. // Bias hidden layer b\_h=[3,1]
8. // Bias output layer b\_o=[1,1]
9. h=relu(W\_h.dot(x)+b\_h)
10. y=relu(w\_o.dot(h)+b\_o)
11. **return** y

The function of a simple multilayer perceptron can, therefore, be implemented using just a few lines. With known weight matrices, the output value can be predicted for the input data, which is achieved by simple matrix and vector multiplication. The expansion of the problem to several outputs and inputs can be generalized by the following representation:

,

where n is the number of neurons in the hidden layer, m is the number of inputs, and k is the number of outputs. The use of deeper networks, i.e., the addition of further hidden layers, requires additional intermediate steps to be calculated for the transition from previous to subsequent layer.

But how can the weights of such a multilayer perceptron be calculated for a specific dataset? This calculation can be performed without issue for an individual neuron using the gradient method by defining and minimizing the cost function of the m inputs, whereby the weights (including bias) are updated iteratively. However, if the network contains hidden layers, then the derivatives of the previous layers are dependent on those of the subsequent layers, which is why the gradient method can no longer be applied directly, as the extent to which the individual weights are to be adjusted is not defined. In such cases, backpropagation is used.

### Self-Check Questions

1. Please name three different activation functions.

For example, the sigmoid, ReLu, step (Heaviside) functions

1. Why is the ReLu activation function used in preference with deep neural networks?

Due to the “vanishing gradient problem”, where the gradient strives for 0 in the front layers, causing training to stagnate.

1. Which of the following activation functions is continuously differentiable: the sigmoid, step, or ReLu function?

The sigmoid function

## 3.4. Backpropagation

In order to train deep neural networks, it is necessary to incorporate the effect that a change in the weight of each neuron has on the output value. Backpropagation is a special case of the gradient method and represents the standard in neural network training today. This method represents one of the key concepts developed during the heyday of connectionism and was introduced for the first time in the 1980s (Nesterov 2003).

In the gradient method, a cost function is minimized by updating the model parameters on the basis of partial derivatives. Let us consider the cost function of the mean squared error of the form

where the factor is used to offset the factor of 2 resulting from the derivative and has no effect on the extreme values of the cost function. The backpropagation algorithm now runs iteratively as follows: first, the input values are introduced to the network from the front, following random initialization of the weights, and the actual output is calculated using the available weight configuration. The actual output value is then compared with the target output value and the mean square of the difference is determined as the current error. This path is called the forward path and represents the later data flow of the trained network. The weights of the neurons are subsequently updated from back to front as a function of the influence on the error calculated, thereby approaching the minimum of the cost function. This path is referred to as the backward path, as it is based on the reverse flow of data. It is only used during training to determine the optimal weight configuration.

In order to update the individual neuron weights, all partial derivatives of the cost function f are required relative to each weight wj of the layer k, i.e.:

For the weights that impact directly on the output, i.e., are directly networked with the output layer, a direct calculation is possible. For the deeper layers, the chain rule must be applied, which, as established from the differential calculation, defines the differentiation of interlinked, i.e., inner and outer, functions of the form y = u(v(x)). To this end, the inner and outer derivative are formed and interlinked according to the rule (for the simple case of one variable). The following detailed structure shows the individual components of a neural network with a hidden layer, in order to illustrate the partial derivatives required for the backpropagation algorithm:

Chart, diagram

Description automatically generated

For the purposes of this illustration, the biases of the neurons have been omitted. Let us look first at the weight w3.1, which is directly networked with the output neuron. The partial derivative can be determined by applying the chain rule twice as follows:

The summation ΣY of the products of the weights w3,j, and of the output values of the neurons in the previous hidden layer contains only one term, which includes the weight w3,1, and is the reason why the partial derivative delivers the output of the previous neuron ϕ(H1) directly. For the second factor, the partial derivative , it is necessary to calculate the derivative of the activation function used for the respective neuron. For this reason, differentiability is an extremely important criterion in the choice of neuron activation. In this case, we are using the sigmoid function, the derivative of which is continuously differentiable and defined as follows:

The last factor of the partial derivative of the cost function in accordance with the weight w3,1 corresponds to the derivative of the cost function in accordance with the output of the output neuron yist. If we assume the gradient is calculated after each element of the dataset, the result is:

The entire partial derivative now results from multiplying the individual terms:

Equivalent to the gradient of the cost function relative to the weight w3,1, all gradients of the weights of an output neuron are calculated. Here, the value from updating the weight, which represents the connection between two neurons i and j, can be defined as follows:

Diagram

Description automatically generatedwhere α indicates the learning rate—a positive factor that determines the influence of the update on the previous weight. The gradient calculation from the back to the front layers takes place recursively for the weights of the neurons in the hidden layers, which do not directly influence the output of the network, as illustrated in the following figure.

For this, we define the following case distinction:

Here, k indicates the index of the neurons that follow the neuron j. The updating of a weight for iteration step t now takes place on equivalent terms to the gradient method using:

**Vanishing gradient problem**

Here, the cost function gradient becomes so small or disappears entirely, making further training very difficult. The reason for this is that a large change in input only effects a small change in the output of the sigmoid function.

It should be noted that despite the important criterion of continuous differentiability, the ReLu function is used very frequently as it represents a solution to the “**vanishing gradient problem**”, which occurs in deep neural networks when the sigmoid function is used as a non-linear activation function. At the point x = 0, the derivative of the ReLu function is not defined, which is why is normally set.

Typically, the weight gradients should not be calculated after each run of an element of the training dataset. Instead, a batch size is defined, which indicates how many elements the weights should be updated after. This changes the partial derivative of the cost function after the output of the output neuron yist with the batch size n to:

### Self-Check Questions

1. In which direction are the neuron weights updated in the gradient method?

From back to front

1. Please describe the backpropagation algorithm in your own words.

An input value is introduced to the network from the front and the outputs are compared with the ground truth value. The error is now propagated through the network, from front to back, and the neuron weights are updated based on the effect on the determined error.

## 3.5 Implementing Backpropagation

Although the backpropagation algorithm is already pre-implemented as a learning method for neural networks in all common deep learning frameworks and must only then be applied to a specific problem, an implementation approach is presented below to help explain this functionality. Let us look initially at a neural network consisting of two input neurons, three neurons in the hidden layer, and an output neuron. Once again, we will use the mean squared error as the cost function, and the sigmoid function as neuron activation. The following illustration shows the structure of the network and the associated notation for the implementation, with the parameter nk indicating the number of neurons, wk the weight matrix including bias weight (factor for the on-neuron with constant 1 as output), Σk the weighted output vector, and ϕk the activated weighted output vector of the respective layer k.

Diagram

Description automatically generated with medium confidence

We define the sigmoid function as activation for all neurons of the output and hidden layer, with the result that the output of a neuron can range from 0 to 1. As a result, the problem that is to be solved using this neural network must reflect this value range.

First, we define a Python function, which generates a new neural network with randomly initialized weights and receives, as function parameters, the number of neurons in the input, output, and hidden layers.

1. **import** numpy as np
2. np.random.seed(0) # Seed key for repeatable results of random variables
3. **def** neural\_network(n\_input, n\_hidden, n\_output):
4. # n\_input: number of input neurons
5. # n\_hidden: number of neurons hidden layer
6. # n\_output: number of output neurons
7. network=[] # Network represents a list of matrices and vectors
8. w1=np.random.rand(n\_hidden, n\_input+1) # Randomly initialized weight matrix hidden layer
9. network.append({'w': w1}) # Append weights as dictionary
10. w2=np.random.rand(n\_output, n\_hidden+1) # Randomly initialized weight matrix output layer
11. network.append({'w': w2}) # Append weights as dictionary
12. **return** network

The activation function is given a numpy matrix and the sigmoid function is applied to each of its elements.

1. **def** activate(x): # x: Numpy matrix
2. **return** 1/(1+np.exp(-x))

As already mentioned in the previous section on the derivation of the backpropagation algorithm, this consists of the forward and backward propagation steps. Let us consider the forward propagation of the network first. The transition from one neuron layer to the next can be indicated using our notation as follows:

The absence of an activation in the input layer replaces the vector ϕk–1 with the input vector x during the transition from input layer to hidden layer. The following Python function implements the forward path of an input vector through an initialized network.

1. **def** forward\_propagation(network, input): # Network: initialized network, input: input vector
2. **for** layer **in** network: # Iterate over layers of network
3. input=np.concatenate((input,np.array([[1]])),axis=0) # On-neuron with constant 1 for bias weight
4. layer['o']=activate(layer['w'].dot(input)) # Calculate output of layer and insert result in dictionary
6. input=layer['o'] # Set activated output as new input of next layer
7. **return** input # Return output of last layer

This function calculates the output vector of each neuron layer, adds this to the generated network, and returns the output from the output layer. This concludes the implementation of the forward propagation. As a test, we will generate a new network using the number of neuron layers previously defined and allow a sample input vector to propagate through it.

1. network=neural\_network(2, 3, 1)
2. input=np.array([[1],[2]])
3. forward\_propagation(network,input)
4. **print**(network)
5. """[{'w': array([[0.5488135 , 0.71518937, 0.60276338],
6. [0.54488318, 0.4236548 , 0.64589411],
7. [0.43758721, 0.891773 , 0.96366276]]), 'o': array([[0.92969121],
8. [0.88473832],
9. [0.9602581 ]])}, {'w': array([[0.38344152, 0.79172504, 0.52889492, 0.56804456]]), 'o': array([[0.89405734]])}]"""

The output of the network is shown in lines 6 to 10. With randomly initialized weights and biases, this is still unusable and merely serves to illustrate the content of the network structure.

In order to train the network that has been generated and thereby adjust the weights, the backward path of the backpropagation algorithm must now be implemented. To this end, we will first define the function for the required derivation of the activation function of the neurons .

1. **def** derivative\_activate(x):
2. **return** x\*(1-x)

The following function converts the case distinction for δjdefined in the previous section to calculate the gradient:

1. **def** back\_propagation(network, output):
2. # Network: initialized network, output: anticipated result
3. **for** i **in** reversed(range(0,len(network))):
4. # Iterate from back to front
5. layer=network[i] # Current layer
6. **if** i==len(network)-1: # If last layer
7. layer['d']=
8. (layer['o']-output)\*derivative\_activate(layer['o']) # Case 1
9. **else**:
10. # Case 2
11. layer\_f=network[i+1]
12. error=layer\_f['w']\*layer\_f['d']
13. layer['d']=error[0,0:3].transpose()\*derivative\_activate(
14. layer['o'])[0:3,0]

**Dictionary**

A dictionary consists of pairs of keys and objects. The associated object can be addressed via the key.

This function is used to add a new **dictionary** element to the network to represent the error delta δj, which is necessary for updating the neuron weights. The weight update is implemented in the following function:

1. **def** update\_weights(network,input,lr): # Network: initialized network, input: input vector, lr: learning rate
2. **for** i **in** range(0,len(network)): # Iterate from front to back
3. layer=network[i] # Current layer
4. layer\_p=network[i-1] # Previous layer
5. **if** i!=0: # If not the first layer
6. layer['w'][0,0:3]-=((lr\*layer['d']\*layer\_p['o']).transpose())[0,0:3] # Multiply outputs from previous layer by error delta
7. layer['w'][0,3]-=lr\*layer['d'] # Bias weight
8. **else**: # If first layer
9. layer['w'][:,0:2]-=(lr\*layer['d']\*input).transpose() # Multiply by input values
10. layer['w'][:,2]-=(lr\*layer['d']).transpose() # Bias weights

Here, the error deltas δj with the partial derivatives

are multiplied in this case by the output vectors of the previous layer, or, in the case of the first layer, directly by the input vector, multiplied by the defined learning rate, and subtracted from the current weight value.

After implementing the basic building blocks of the propagation algorithm, we can now construct a framework for training a network, within which the forward propagation, backward propagation, and weight update steps are to be executed iteratively. As parameters, this framework will receive an initialized network, a training dataset, the number of training runs (epochs), and the learning rate. This is realized within the following function:

1. **def** train(network,data,epochs,learning\_rate): # Network: initialized network, data: training dataset, epochs: passes, learning\_rate: learning rate
2. **for** i **in** range(0,epochs):
3. error=0
4. **for** j **in** range(0,len(data)):
5. input=data[j][0]
6. output=data[j][1]
7. forward\_propagation(network,input)
8. error+=(network[len(network)-1]['o']-
9. output)\*\*2 # Calculate mean squared error
10. back\_propagation(network,output)
11. update\_weights(network,input,learning\_rate)
12. **print**("Epochs "+str(i)+" error: "+str(error/len(data)))

We can now test the structure that has been implemented using a dataset consisting of two-dimensional input vectors and one-dimensional output values with the value range [0,1]. Here, our aim is to train the neural network so that it can decide which of the two input values is the larger of the two. We, therefore, initialize our network and our dataset, and then call up the training method with a learning rate of 0.9 and with 1,000 epochs.

1. network=neural\_network(2, 3, 1)
2. data=[[np.array([[1],[2]]),1],
3. [np.array([[3],[4]]),1],
4. [np.array([[2],[8]]),0],
5. [np.array([[5],[4]]),0],
6. [np.array([[3],[7]]),1],
7. [np.array([[1],[0]]),0],
8. [np.array([[0],[7]]),1],
9. [np.array([[6],[2]]),0],
10. [np.array([[10],[2]]),0],
11. [np.array([[1],[5]]),0]]
12. train(network,data,1000,0.9)

Graphical user interface, chart

Description automatically generatedIn the following figure, the mean squared error is shown as a function of the respective epoch.

It can be clearly seen that the mean squared error is minimized as required during training. The trained network can now be used for prediction by performing forward propagation and reading the output of the last neuron. This results in the following output for the test vector :

1. """[{'w': array([[ 3.752586 , -3.63410586, -0.66544748],
2. [ 6.58892285, -2.6878607 , 0.9095939 ],
3. [ 0.23974779, 1.17127276, 0.89752745]]), 'o': array([[0.01900038],
4. [0.99995106],
5. [0.99817046]]), 'd': array([-1.80843916e-07, 1.02122150e-03, 8.37463458e-06])}, {'w': array([[-8.10503368, 4.80319464, 0.04404601, -0.82305061]]), 'o': array([[0.97956532]]), 'd': array([[0.07928383]])}]"""

The output strives for the desired value of 1, as the second number is the larger of the two, but does not represent this exactly. The problem in question actually corresponds to a binary classification, as the value range is discrete. The chosen cost function has, however, a continuous range of values between 0 and 1, which is more likely to be used for regression problems. Nevertheless, the function of the backpropagation algorithm can also be demonstrated well with this.

### Self-Check Questions

1. Is the backward path also required while testing a neural network on unknown data?

No, this is only required for training the network, where the error is propagated from back to front through the network and the weights are updated accordingly.

1. How can the weights and biases of a neuron layer be represented in simple mathematical terms?

Using matrices and vectors

|  |
| --- |
| Summary |
| The development of statistical models by means of supervised learning in order to solve problems, regardless of whether these are regression or classification problems, requires cost functions that are used to define the objective of the optimization. The mean squared error is used particularly frequently in this context for regression problems, as this constitutes a mathematically suitable measure for deviation. As negative and positive deviations of equal size cancel each other out, it is not practical to use the mean error. Likewise, using the absolute value instead of squaring is also not a suitable option due to the continuous differentiability. Furthermore, the constant derivative for values greater than and less than 0 poses a stumbling block in finding a minimum of the cost function.  The gradient method is the fundamental component for determining the optimal model parameters on the basis of training data. Here, a stationary point, which represents the global minimum in the optimal case, is iteratively approximated through calculation of the partial derivatives of the cost function by moving in the direction of the steepest descent relative to the current point on the cost function. For single-layer perceptrons, the gradient can be used directly to calculate the optimal parameter configurations. This is not as readily achievable with multilayer perceptrons, as the error gradients of the layers are dependent on those of the subsequent layers. Generally speaking, the forward path of a multilayer perceptron can be traced using simple operations from linear algebra. In terms of updating parameters to optimize a cost function, backpropagation constitutes the basic algorithm as an expansion of the gradient method. This can be implemented by means of simple Python code using matrix and vector operations, and consequently a multilayer perceptron aimed at minimizing the cost function can be developed on the basis of training data. |

# Unit 4—Getting Started with Deep Learning Frameworks

Study goals

After completing this unit, you will know ...

… which deep learning frameworks are available and how they are divided up.

… how the Tensorflow environment is set up and used.

… what the fundamental concepts of the Tensorflow framework are.

… how mathematical functions can be implemented using Tensorflow.

# 4. Getting Started with Deep Learning Frameworks

### Introduction

This unit gives an introduction to working with deep learning frameworks, as a means of enabling entry into application development. First, an overview of existing frameworks is provided, after which the first steps involving Tensorflow are covered. We give a detailed description of the installation process and setup under Windows and Ubuntu, and then use examples to explain how the framework works. The lesson concludes with the presentation of a number of mathematical functions and application thereof in the implementation of a simple sample neural network, which will enable you to handle more complex problems with Tensorflow.

## 4.1 Overview

Due to the steady rise in the popularity of deep learning, a large number of different deep learning frameworks have been developed in recent years, covering all levels of expertise, from newcomer to expert. As a result of high-level frameworks, users can quickly familiarize themselves with the methodology of deep learning application development, without requiring in-depth knowledge of the underlying mathematical operations. Keras and PyTorch are just two of the frameworks that belong to this group. With their help, neural network architectures can be easily constructed, trained and tested, whereby the underlying structure is viewed as a kind of **black box** and only its functionality is defined.

**Black box**

In the case of a black box, it is not the internal structure that is of interest, only its interfaces to the outside world.

However, in order to gain a deep understanding of how neural networks work, we need to take a look behind this curtain and understand the core elements. Low-level frameworks allow the developer to access all of the underlying mathematical operations that take place in the construction, training, and testing stages. These frameworks include Tensorflow, Theano, and FastAl, with the former currently ranked as the most popular among these according to activity measured on Stack Overflow and GitHub (Hale 2018). The following comparison of the most common frameworks illustrates the leading role played by Tensorflow as measured on the basis of Amazon books, GitHub and KDnuggets activity, Medium and ArXiv articles, and job advertisements in 2018 (ibid.).

Chart, bar chart

Description automatically generated

Tensorflow was released by Google in 2015 and is available under an open-source license. One of the huge advantages of Tensorflow is the fact that it is available across all common platforms, such as Windows, Linux, macOS, and Android. Libraries are available for a multitude of programming languages, including Python, C++, C, and Java, which provides a high level of flexibility in development. The majority of Tensorflow users use Python as the preferred development language, due to the relatively simple syntax and faster training time compared to C++, for example. As a result, application development can get underway more quickly, without the need for a complicated syntax to be incorporated first. Tensorflow follows the paradigm of data stream-oriented programming. A continuous data stream is assumed here, for example, image or audio data, which is manipulated in real-time by the application that has been developed (Morrison 2010). Other supervised learning methods, such as support vector machines, for example, can be designed in addition to neural networks, and can be used to solve both regression and classification problems.

Many well-known applications are based on developments supported by Tensorflow, such as Google’s Gmail, the search engine Google, Google speech recognition, and even Google Streetview. A detailed explanation on getting started with the Tensorflow framework, and an introduction to the fundamental concepts and mathematical functions, is provided below.

### Self-Check Questions

1. What kind of deep learning frameworks enable neural networks to be developed without in-depth knowledge of the underlying mathematical operations?

High-level frameworks

1. Which group of deep learning frameworks does Tensorflow belong to?

Low-level frameworks

## 4.2 Getting Started with Tensorflow

In the following sections, we will limit ourselves to installation and development under the Tensorflow library for Python, and additionally to Windows and Ubuntu as underlying platforms. It is generally advantageous to set up a virtual development environment using **Anaconda**, for example, in order to develop applications in isolation from each other and to separate required dependencies, such as library versions, from each other. To this end, the installer for the corresponding platform must be downloaded from the Anaconda website and run. A detailed installation and user manual is also available on the website. In the following, we assume a newly set-up development environment, in which no packages have yet been installed.

**Anaconda**

This is regarded as one of the virtual standard platforms in the field of data science.

There are two different Tensorflow interfaces. The first is restricted to the use of the processor (CPU) for training and testing machine learning applications. This is practical if only small neural networks are to be trained that do not require an extremely large amount of computing capacity. The latest CPU version of Tensorflow, which at the time of creating this book was version 1.15, can be easily installed using the Python package management program pip as follows:

$ pip install tensorflow

The correct installation and the current version can be checked in the Python console using

1. **import** tensorflow as tf
2. tf.\_\_version\_\_
3. '1.15.0'

The second version of the Tensorflow interface supports the use of the graphics card (GPU) for training and testing. This is essential for deep neural networks that require an extremely large amount of computing capacity. The most computing-intensive part of neural networks, both during training and testing, lies in the numerous matrix calculations that are required for both forward and backward propagation. The advantage of the graphics card over the processor lies in its structure and functional principle. While processors are made up of a few complex cores, graphics cards have a large number of simple cores. As a result, matrix multiplications can be calculated in parallel on a large scale, which reduces the required computing time several times over. Use of the GPU version of Tensorflow requires an NVIDIA GPU with CUDA Compute Capability 3.5. A list of supported graphics cards can be found on the NVIDIA website. In the event of an existing supported NVIDIA graphics card, the required drivers and libraries can be installed as follows. Under Ubuntu, the following repository must first be added to the package management program apt and subsequently updated.

$ sudo add-apt-repository ppa:graphics-drivers/ppa

$ sudo apt-get update

The latest NVIDIA drivers can then be installed.

$ sudo apt-get install nvidia-driver-418

Pressing the tabulator key twice after the package name nvidia-driver- will bring up a list of all available versions, from which the required one can be selected. Following a restart, all drivers should be correctly installed. This can be tested using the tool nvidia-smi, which can also be installed. The next step involves installing the interface technology CUDA, which is used to establish communication with the graphics card. CUDA version 10.0 is currently recommended. This can be downloaded and installed from the NVIDIA website for both Ubuntu and Windows. The installation of Visual Studio Express is required under Windows, which can be sourced directly from the Microsoft Visual Studio website. Following subsequent installation, the CUDA environment variables must be checked under the system properties and added if they are not present. The following environment variables must be present:

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\bin

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\libnvvp

Once again, the absolute path can vary depending on the installation directory selected. Under Ubuntu, the following commands for exporting the environment variables must be written to the file ~/.bashrc following installation:

**export** PATH=/usr/local/cuda-10.0/bin:$PATH

**export** LD\_LIBRARY\_PATH=/usr/local/cuda-10.0/lib64

It must be ensured that the path specified during installation is correct. The updated profile ~/.bashrc can then be reloaded using the source command. The cuDNN version that is compatible with CUDA 10.0 must now be installed as the deep learning library for CUDA. The latest version for CUDA 10.0 is cuDNN 7.6 and can be downloaded directly from the NVIDIA website. Under Windows, the following header files and libraries contained within the downloaded zip file must be copied to the corresponding CUDA installation directories as follows:

..\cuda\bin\cudnn64\_7.dll --> C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\

..\cuda\ include\cudnn.h --> C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\include\

..\cuda\lib\x64\cudnn.lib --> C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\lib\x64\

The directory structures may vary depending on the selection made during installation. In the case shown, the standard installation paths were specified. Under Ubuntu, the following directories of the zip file are copied to the local header or library directory as follows:

$ sudo cp -P lib64/\*/usr/local/cuda/lib64/

$ sudo cp -P include/\*/usr/local/cuda/include/

Following successful installation, all necessary drivers and libraries for the GPU variant of Tensorflow are set up and the latest version can be installed with the Python package management program using

pip install tensorflow-gpu

It is important that the CPU and GPU variant of Tensorflow do not coexist within the same virtual development environment, as this will cause problems. Should you wish to switch from the CPU variant to the GPU variant, for example, the old package must be deinstalled first and then the new package installed.

pip uninstall tensorflow

pip install tensorflow-gpu

The correct installation of the GPU variant can be tested using the following simple command sequence within the Python console.

1. **import** tensorflow as tf
2. sess = tf.Session()

A Tensorflow session is started here, the significance of which is explained in the next section. If the information contained in the resulting output includes the following,

Found device 0 with properties:

name: **GeForce GTX 1060 6GB** major: 6 minor: 1 memoryClockRate(GHz): **1.759**

then the installation was successful. The details shown in bold will vary depending on the GPU used.

### Self-Check Questions

1. Please mark the correct answer(s) with a cross.

* The GPU variant of Tensorflow can be used with any graphics card.
* Neural network training is faster when using the CPU.
* The most computing-intensive part of training neural networks lies in the differentiation of the activation function.
* *The propagation of small neural networks in the CPU is most definitely possible.*

1. Please give a brief explanation of why a graphics card is well suited to training neural networks.

The advantage of the graphics card over the processor lies in its structure and functional principle. While processors are made up of a few complex cores, graphics cards have a large number of simple cores. As a result, matrix multiplications can be calculated in parallel on a large scale, which reduces the required computing time several times over.

## 4.3 Fundamental Concepts

Having presented the installation and setup of the various Tensorflow variants in the previous section, the fundamental concepts on which this framework is based will now be outlined below and explained using simple examples.

The core component of a Tensorflow application is the graph. Fundamentally, a graph is nothing more than the linking of Tensorflow operations, known as nodes, which, in their entirety, describe a model. Let us look first at the following two-dimensional mathematical function, in order to explain the concepts in more detail:

This can be represented graphically as follows.

Diagram

Description automatically generated

The red and blue squares represent variables or constants and the green circles represent mathematical operations. All of these components belong to the group of nodes that, connected by edges, represent the graph of the model. Each mathematical function is represented by Tensorflow using such a graph, thus defining the structure of the calculations. The nodes of the model are divided into tensor-generating operations (such as variables and constants) and tensor-manipulating operations (mathematical functions), and they are known as OpNodes (short for operation nodes). But what exactly is a tensor? Tensors could generally be described as multidimensional matrices, as illustrated in the following figure.

A screenshot of a computer

Description automatically generated with low confidence

All of these arrangements are tensors. A one-dimensional tensor can be represented as a vector (left), a two-dimensional tensor as a matrix (center), and a three-dimensional tensor as a matrix of vectors. This definition as a generalization of a matrix does not, however, cover the most important property. Tensors are dynamic structures that can adapt by interacting with other mathematical structures (Steinke 2017). Let us take, as an example, the activated output of a layer with two neurons of a neural network, which can be represented as a vector as follows:

This layer is followed by a second layer, which is connected to the first through the following weight matrix:

The output vector of the second layer without activation function can now be calculated using the matrix vector product

Let us now assume that the activation functions of the two neurons in the first layer are to be adjusted, for example, the first is to be extended or compressed by a factor of 2 and the second by a factor of 0.5. This corresponds to a left-sided multiplication by the matrix T:

Trivially, different outputs of the second layer are produced if weight matrix W remains constant. However, if we multiply this on the right-hand side by the inverse of the transformation matrix T–1, then identical outputs of the second layer are produced before the transformation of the activation functions.

This interaction between the vectors and matrices in the form of transformations constitutes the core property of tensors. The name Tensorflow itself demonstrates the significance of tensors in the underlying mathematical calculations.

Tensor-forming and tensor-manipulating nodes define a graph through mutual connection, without the need for any data. A graph, therefore, represents the calculation structure only. Placeholders define tensor-generating operations within the graph. These are “fed” the actual data when a graph is run. A placeholder carries information about the data type and the form of the data that will be used later, and it differs from Tensorflow variables in the sense that it is not trainable. Variables are used for the weights and biases of a network, for example.

The running of a defined graph takes place in Tensorflow within sessions, in which the placeholders of the graph are fed with data and the defined calculations are then carried out on this basis. The following code sample shows how the placeholders and sessions work.

1. **import** tensorflow as tf
2. x=tf.placeholder("float", None)
3. y=x+2
4. with tf.Session() as session:
5. result=session.run(y, feed\_dict={x: [1, 2, 3]})
6. **print**(result)
7. """[3. 4. 5.]"""

We, therefore, define a placeholder x first, which can be fed with floating-point numbers during a session. The second placeholder parameter specifies the quantity of data that can be added to this placeholder. “None” means that there is no restriction on quantity. The graph is defined using the function y = x + 2 and a new session is generated using “with tf.Session() as session:”, which is started using “session.run”. Within this session, placeholder x is fed three values. It is, of course, possible that suitable placeholders are defined for all possible types of tensors. If, for example, the parameter [None,80,80,3] is specified, this can be fed with any number of 80 x 80 x 3 tensors, for instance, with image data.

Tensorflow’s visualization program Tensorboard can be used to gain insights into the underlying structures of a Tensorflow application, such as the visualization of the generated graph. To this end, the “FileWriter” of a session can be used to create an event file, within which information on the structure and events during the session are written. The following line of code generates an event file in the “graphs” folder.

writer=tf.summary.FileWriter('./graphs', session.graph)

This can be opened with Tensorboard using the command

tensorboard --logdir="./graphs" --port 6006

Diagram

Description automatically generatedwhich launches a local server on the specified port that can be accessed via a browser. The graph of the application can now be displayed in the Tensorboard window that has been opened, which gives the following visualization for our example.

The oval components represent the OpNodes of the graph, which in our case are the tensor-generating placeholders and the tensor-manipulating addition. The constant, in our example the value 2, is shown as a circle.

A number of new concepts were introduced with Tensorflow 2.0, which was released during the creation of this book. The paradigm shift from programming based on data flow to imperative programming is significant, as operations can be evaluated directly without creating and calculating a graph (Gulli/Kapoor/Pal 2019). Graphs, sessions, and placeholders are no longer explicitly defined as before. Sessions are replaced by Python functions with parameters that represent the original placeholders and the data to be fed.

### Self-Check Questions

1. What is a tensor?

Tensors are dynamic structures that can adapt by interacting with other mathematical structures. Vectors and matrices are one-dimensional and two-dimensional tensors respectively.

1. What is the difference between tensor-generating and tensor-manipulating nodes?

Tensor-generating nodes are placeholders for variables or constants that are fed values at a later stage. Tensor-manipulating nodes change these values through mathematical operations.

1. Consider the function y = 3x. What can be regarded here as tensor-generating and tensor-manipulating node?

Multiplication: tensor-manipulating node, variable x, y: tensor-generating node

4. What is the graph of a Tensorflow application?

In the graph, all tensor-generating and tensor-manipulating nodes are connected to each other. The functionality is defined here and the calculations are started at a later stage by running the graph.

## 4.4 Mathematical Functions

All mathematical functions required for the application development and applicable to tensors are implemented in the Tensorflow framework. The example from the previous section could also be implemented with the same result as follows:

1. **import** tensorflow as tf
2. x=tf.placeholder("float", None)
3. y=tf.add(x,2)
4. with tf.Session() as session:
5. result=session.run(y, feed\_dict={x: [1, 2, 3]})
6. **print**(result)
7. """[3. 4. 5.]"""

The use of the Tensorflow function “add” produces the same result as the Python additions operator. However, a major advantage of using Tensorflow functions for mathematical operations lies in their own additional parameter, which allows for explicit naming. Having these labels helps to make the underlying graph clearer and thus easier to analyze.

Let us look first at a number of other mathematical operations in the Tensor framework. To this end, we define two constants with arbitrary values.

1. x=tf.constant([1,2,3,4,5,6],shape=(2,3))
2. y=tf.constant([2,3,4,5,6,7],shape=(3,2))

Constants are defined by the “constant” command and receive a list of values and information about the format as parameters. In this case, we have defined the constant x as a 2 x 3 matrix and the constant y as a 3 x 2 matrix. In addition, a name can be assigned to the constant using the “name” parameter. The matrix product is calculated using the “matmul” function, whereby the number of columns in the first matrix must correspond to the number a of rows in the second matrix.

1. result=tf.matmul(x,y)
2. """<tf.Tensor 'MatMul\_13:0' shape=(2, 2) dtype=int32>"""

The output on executing the command in the Python console can be seen in the second line, which contains information about the output format of the mathematical operation. Once again, the functionality that Tensorflow works with is clearly demonstrated here. By defining the Tensorflow nodes, as the definition of the constants and mathematical operations, only the graph is generated and no calculation is carried out. When a session is started, the graph is calculated and the result can be output.

1. with tf.Session() as session:
2. **print**(session.run(result))
3. """[[28 34]
4. [64 79]]"""

By contrast, in version 2.0 of Tensorflow, the calculation is output directly after the “matmul” function.

1. tf.matmul(x,y)
2. """<tf.Tensor: id=6, shape=(2, 2), dtype=int32, numpy=
3. array([[28, 34],
4. [64, 79]])>"""

Within this section introducing mathematical functions, we will confine ourselves henceforth to the Tensorflow 2.0 conventions. The same results can also always be achieved with previous versions by running a session. The application of the “multiply” function to two matrices executes an element-wise multiplication and is also known as the **Hadamard product**. This requires both matrices to have the same format, i.e., the number of columns and rows must match. To this end, let us first perform a “reshape” operation on matrix y to transform this into the format of matrix x, and then calculate the Hadamard product.

**Hadamard**

The Hadamard product was named after the French mathematician Jacques Hadamard.

1. y=tf.reshape(y,[2,3])
2. tf.multiply(x,y)
3. """<tf.Tensor: id=11, shape=(2, 3), dtype=int32, numpy=
4. array([[ 2, 6, 12],
5. [20, 30, 42]])>"""

The same result can be achieved in this case by using the “transpose” function instead of the “reshape” function, as the 3 x 2 matrix is also transformed into a 2 x 3 matrix here. Some Tensorflow operations require specific data formats for input data, such as integer or floating-point numbers. A type conversion can be carried out using the “dtypes.cast” operation. This receives the value to be changed and the new data format as parameters. The conversion of the data format of matrix y from integer (int32) to floating-point number (float32) can be achieved as follows:

1. tf.dtypes.cast(y, tf.float32)
2. """<tf.Tensor: id=13, shape=(2, 3), dtype=float32, numpy=
3. array([[2., 3., 4.],
4. [5., 6., 7.]], dtype=float32)>"""

You can see that the data type has changed from int32 to float32. Naturally, the reverse is also possible, whereby decimal places are lost. We can now apply an activation function to the matrix for testing purposes. The Tensorflow framework contains all of the common activation functions, such as the sigmoid function, for example.

1. tf.sigmoid(y)
2. """<tf.Tensor: id=15, shape=(2, 3), dtype=float32, numpy=
3. array([[0.880797 , 0.95257413, 0.98201376],
4. [0.9933072 , 0.9975274 , 0.999089 ]], dtype=float32)>"""

The function returns the output value of the sigmoid function for each element of the matrix y.

A picture containing bubble chart

Description automatically generatedBased on what we have learnt to date, we can already implement the forward path of a simple neural network manually using Tensorflow operations. Let us, therefore, consider the following network consisting of an input layer, an output layer, and a hidden layer, for which we assume sigmoid activation functions ϕ.

The input x can be represented here as a 2 x 1 vector, the weights and biases of the hidden layer W1 and b1 as a 2 x 2 matrix and a 2 x 1 vector respectively, the weights and biases of the output layer W2 and b2 as 2 x 1 vectors in each case, and the output y as scalar. The following function can thus be established for the output:

Here, we define the weights and biases as trainable Tensorflow variables. The forward path of the network is defined in a Python function that is passed the input vector as a parameter. The return value of the function corresponds to the output of the network. Within the following code, a sample vector is passed to our network and the output is displayed.

1. **import** tensorflow as tf
2. W1 = tf.Variable([[1.,2.],[3.,4.]], name="W1")
3. b1 = tf.Variable([[2.],[0.5]], name="b1")
4. W2 = tf.Variable([[1.],[1.5]], name="W2")
5. b2 = tf.Variable([[2.],[1.]], name="b2")
6. @tf.function
7. **def** forward\_path(x):
8. o=tf.sigmoid(tf.add(tf.matmul(W1,x),b1))
9. y=tf.add(tf.multiply(W2,o),b2)
10. **return** y
11. y = forward\_path([[2.],[1.]])
12. **print**(y)
13. """tf.Tensor(
14. [[2.9975274]
15. [2.4999588]], shape=(2, 1), dtype=float32)"""

The Python Decorator “@tf.function” enables optimal performance by transforming the code that follows it into a high-performance Tensorflow graph (Grogan 2020).

### Self-Check Questions

1. What is the difference between the Tensorflow functions tf.matmul and tf.multiply?

While tf.matmul calculates a matrix product of two matrices A ∙ B, where the number of columns of B must match the number of rows of A, tf.multiply calculates the Hadamard product, i.e., each element of a matrix by the element of a second matrix at the same position.

1. Does the Python decorator @tf.function have to be executed before every Tensorflow function?

No, it can be executed to achieve optimal performance by transforming the subsequent code into a high-performance Tensorflow graph.

|  |
| --- |
| Summary |
| Deep learning frameworks contain all of the functions that are needed to develop and evaluate statistical models. With the aid of high-level frameworks, applications can be designed without in-depth knowledge of the underlying mathematical operations. In terms of gaining an appreciation of how statistical models function as a whole, low-level frameworks are the appropriate tool, as they permit development at the lowermost level.  Tensorflow is currently the most widely used framework and was equipped, on the release of version 2.0, with new features that optimally support the development process. The basic functionality that Tensorflow works with is defined by an underlying graph that contains all operations, including tensor-generating operations such as constants or variables, and tensor-manipulating operations such as mathematical functions. All mathematical functions required for working with statistical models are integrated within the framework, which, when interconnected, define the Tensorflow graph. Tensors can be simplified but cannot be defined exactly as generalizations of matrices. They can change their structure, however, by interacting with other mathematical objects.  With Tensorflow, statistical models can be developed using just a few lines of code. With the introduction of version 2.0, certain previously fundamental elements of the development, such as sessions and placeholders, have been dropped and replaced with simple function calls. |

# Unit 5—Classification and Optimization

Study goals

After completing this unit, you will know ...

… what a linear classifier is and what it is used for.

… which cost functions can be used for linear classifiers.

… what a support vector machine and a Softmax classifier are.

… which different gradient methods exist and what the advantages of the batch-mini-batch and stochastic gradient method are.

… what epochs are used for in the training of linear classifiers.

# 5. Classification and Optimization

### Introduction

In this unit, we will cover the subject of classification and optimization. A linear classifier is continuously developed and optimized in the individual learning sections, which can ultimately classify the established MNIST dataset of images with handwritten numbers from 0 to 9. We look at different cost functions and methods used to configure the model parameters, and we describe their advantages and disadvantages in practice. An overview of important hyperparameters, which play a key role in the development of a statistical model, is also provided.

## 5.1 Linear Classifier

In the following section, we will show how a linear classifier works with the aid of a sample dataset and is implemented using Tensorflow. A linear classifier classifies data based on a linear combination of its special characteristics, known as “features”. During the training phase of a classifier, the elements of a weight matrix and a bias vector, which represent the factors of the linear combination and a linear threshold value respectively, are adjusted by minimizing a cost function. In order to decide which class a corresponding object belongs to, a decision function is required which maps the value of the linear combination to a specific value. Generally, a linear classifier can be mathematically described as follows:

where the multiplication represents the scalar product from the weight matrix with the input vector and f() represents the decision function. The number of columns of W must correspond to the number of entries of the input vector. The value for the number of rows of W is the product of the number of possible classes. For example, if our dataset consists of three-dimensional vectors belonging to four different classes, then W will assume the form of a 4 x 3 matrix. The number of rows of the bias vector corresponds to that of the weight vector.

**MNIST**

This acronym stands for Mixed National Institute of Standards and Technology.

A picture containing qr code

Description automatically generatedAs an example, let us consider the **MNIST** dataset, which contains 60,000 pieces of training image data and 10,000 pieces of test image data of handwritten digits from 0 to 9 in a 28 x 28 pixel format (LeCun 1998). Examples from the dataset are shown in the following figure. This publicly available dataset is also a highly popular tool for determining and comparing the efficiency of different machine learning methods.

There are ten different classes, since ten different digits occur within the dataset. For our purpose, we convert the images in 28 x 28 pixel format into single-column vectors with a length of 28 ∙ 28 = 784 by successively stacking the individual rows on top of each other in a column vector, which, for reasons of clarity, is illustrated in the following figure using 3 x 3 pixels.

Chart, box and whisker chart

Description automatically generated

These vectors can be regarded as points in 784-dimensional space. The MNIST dataset can be loaded directly from the Tensorflow framework and used for test purposes. To this end, the “tensorflow-datasets” package must first be installed with the aid of package management program pip using

pip install tensorflow-datasets

The library can then be imported directly and the dataset can be loaded:

1. **import** tensorflow\_datasets as tfds
2. mnist=tfds.load(name="mnist", split="train")

The “split” parameter indicates that the training part of the dataset is to be loaded. Tensorflow downloads the dataset and saves it on the local computer when the lines of code are executed. The following code can be used to iterate over the loaded dataset and read out the information about the input vector and the associated class. Here, the “take” function of the data object is executed with the number of required elements.

1. **for** sample **in** mnist.take(1):
2. image, label = sample["image"],sample["label"]

The variable “image” now contains a tensor of the size 28 x 28 x 1, which contains one element of the dataset, and the variable “label” contains a one-dimensional tensor with the associated class. The conversion to a tensor of the size 784 x 1 is necessary for our example, in order that the linear classifier can use it. This is carried out using Tensorflow’s “reshape” function. The image data are in unsigned integer format, with the pixel values, therefore, ranging between 0 and 255. We scale these to the range between 0 and 1 by dividing each element of the vector by the value 255, resulting in floating-point numbers that all Tensorflow functions are compatible with. Scaling and reshaping are implemented using the following codes:

1. image=tf.reshape(image,shape=(1.784))
2. image=image/255

The weight matrix of the linear classifier must have 784 columns equivalent to the length of the input vectors and, due to the number of classes, must have ten rows, in common with the bias vector. In Tensorflow, we can convert the previous model as follows, whereby the input vector x and the bias vector are implemented as row vectors for reasons of code clarity, and the former is multiplied from the left by the weight matrix. The result is equivalent to the notation previously presented. To this end, we create a “LinearClassifier” class using the two methods “init”, which initializes the weight matrix and the bias vector of the transferred variable with random values, and “call”, which returns the result from W ∙ x + b.

1. shape\_W=(784,10)
2. shape\_b=(1,10)
3. **class** LinearModel:
4. **def** \_\_call\_\_(self, x):
5. **return** tf.matmul(x,self.W) + self.b
6. **def** \_\_init\_\_(self,shape\_W,shape\_b):
7. self.W = tf.Variable(dtype=tf.float32,
8. initial\_value=tf.random.normal(shape\_W), name="W")
9. self.b = tf.Variable(dtype=tf.float32,
10. initial\_value=tf.random.normal(shape\_b), name="b")

First, we define the variables of the weight matrix and the bias vector. These are initialized via the “initial\_value” parameter with random normally distributed floating-point values using Tensorflow’s “random.normal” function within the “init” method. The linear classifier equation is implemented within the “call” function, which receives the input vectors as parameters and returns a ten-dimensional output vector. The “init” method of a class is executed when the class is called up in order to generate an **instance**, whereas the “call” method is called up through instantiation. The linear classifier currently only equates to a simple mapping of . We can generate an instance of the linear classifier class and call this up by transferring the previously loaded element of the MNIST dataset.

**Instance**

An instance is also called an object and is an example of the class generated during the runtime.

1. model=LinearModel(shape\_W,shape\_b)
2. y=model(image)
3. """tf.Tensor(
4. [[ 4.7491984 8.353248 -22.879402 -17.482237 4.278392 -4.606733
5. -16.356066 -20.450083 -11.04475 -4.575908 ]], shape=(1, 10), dtype=float32)"""

Naturally, due to the randomly initialized weights and biases, the output vector does not yet have a pattern. These must be adjusted first so that the respective output representing the class to which a transferred element belongs corresponds to the ground truth output.

### Self-Check Questions

1. What is the equation of a linear classifier with weight matrix W and bias vector b?
2. A linear classifier executes the mapping . How many classes are there here?

Three

1. What does the MNIST dataset contain?

This consists of digitized images of handwritten numbers from 0 to 9.

## 5.2 Cost Functions

In order that a linear classifier can learn, it must be told exactly what is to be learned. Supervised training using training data presents an optimization problem, which minimizes a defined cost function indicating the deviation between the outputs predicted by the model and the ground truth data. In the simplest case, the optimization problem can be indicated as follows.

Here yi represents the i-th ground truth output, xi the i-th input vector, W the weight matrix, and b the bias vector. The function L is the error function that measures the deviation between the ground truth output and the model’s prediction. The solution to the problem is obtained using the parameter configuration of W and b, which minimizes the sum of all cost functions over the training dataset. Let us return to the example we started in the previous section. The ground truth outputs imported from the MNIST dataset currently represent integers between 0 and 9, which corresponds to the appropriate associated class of the input image. For multi-class classification problems, as in this case, the multi-class support vector machine cost function (multi-class SVM) is a frequently used approach (Weston/Watkins 1998). In the case of a support vector machine, which constitutes a large margin classifier, the weights and biases are adjusted in such a way that the distance between all classes is maximal. This can be easily illustrated in the two-dimensional Chart, scatter chart

Description automatically generatedcase.

The two classes (blue and red circles) can be completely separated from each other using several different straight lines. The objective of the support vector machine is not to find any straight line, but the one that maximizes the distance from it to the two classes. In the figure, the red line represents the optimal solution over the green line. For a dataset with vectors of dimension n, the separation represents an (n – 1)-dimensional hyperplane. The multi-class SVM cost function can be defined as follows:

Here, yi represents the ground truth output for the i-th input vector and syi the associated scoring function, i.e., the output predicted by the model for the correct class. The second scoring function sj represents the output predicted by the model for all classes j that do not correspond to the ground truth class. The higher the value of the corresponding scoring function, the more secure the model with the classification. The cost function returns the maximum value of 0 and the term sj – syi + 1. In the event of a correct classification, the latter term has a negative sign and thus the result of the maximum value is 0. In the event of an incorrect classification, the term gives a positive value, which provides the result of the maximum value function. The multi-class SVM cost function, therefore, adds up the incorrect classifications only and ignores correct classifications. Using this method, it attempts to increase the scoring value for the correct class relative to the scoring values for the incorrect classes. The maximum value formation max(0, sj – syi + 1) is also called the hinge cost function. For the sake of simplicity, let us first consider a binary classification problem that represents the prediction of the color (red or blue) of an object. The output vector is two-dimensional because of the two classes present. The following table shows the sample outputs of a notional model with randomly initialized values for the weight matrix and bias vector.

Chart, bar chart

Description automatically generated

The value of the multi-class SVM cost function can now be calculated as follows. The label for the first input value (first column) is “red”, thus the scoring function syi assumes the value 2.8. Accordingly, the value for the scoring function sj is that which is not equal to the ground truth class, i.e., –1.4. Inserting the values into the cost function gives:

The class was thus correctly predicted. For the second object, the ground truth value is “blue”, hence syi = –5.3 and sj = 1.7, leading to the following cost function result:

The class was, therefore, incorrectly classified and the result was included in the totaling process. The entire classification error over a dataset is obtained by totaling across all elements using:

For this simple example consisting of two sets of training data, the total error is obtained from .

If more than two classes are involved, the results of the multi-class SVM cost function are totaled over all classes. If we return to our original example of the 10-class classification of the MNIST dataset, this results in nine terms for totaling per input vector. Let us assume that the ground truth class belonging to an input vector corresponds to the class “2”. Now, the differences in all results of the scoring functions for classes that are not equal to 2 are formed using the value of the scoring function of class 2 and the results are totaled.

**Softmax function**

Another name for the Softmax function is normalized exponential function.

The total over the entire training data divided by their number gives the total classification error.

The **Softmax** classifier represents another linear classification method, alongside the multi-class state vector machine, which is easier to interpret. This classifier calculates the probabilities of a specific input vector belonging to the respective class, where the sum of all probabilities is 1. The following table shows sample predicted values for a trained multi-class SVM and a Softmax classifier for an input vector belonging to class “2” of the MNIST dataset.

Text

Description automatically generated

This is where the advantage of the Softmax classifier becomes clear. Although the multi-class SVM does indeed predict the correct class, the result is not particularly authoritative as the SVM’s algorithm only seeks to maximize the scoring value for the correct class. For this reason, no statement can be made as to which represents the next most probable class, for example. By contrast, the Softmax classifier enables a more precise analysis of the result as it represents a probability distribution. The underlying Softmax function represents a generalization of the logistic function and is defined as

where K contains the number of classes. In our example of the MNIST dataset, the Softmax function is applied to the output vector of the model y = W ∙ x + b. As the output of this function is in the form of a ten-dimensional vector, the ground truth outputs must be identical in form. For this reason, we transform the ground truth values into the one-hot coding. In the case of ten different classes, a ten-dimensional vector is created in which precisely one element only has the value 1 and the rest have the value 0. The following figure illustrates the transformed Table

Description automatically generatedground truth values.

We achieve this coding using the “one\_hot” Tensorflow function, which receives the value to be encoded and the length of the one-hot vector as parameters. The following code implements the coding of the MNIST label:

1. label=tf.one\_hot(label,10)

The most frequently used cost function for Softmax classifiers is the **cross-entropy** cost function, which calculates the distance in the form of the difference between the outputs generated by the Softmax function and the one-hot encoded ground truth label. The cross entropy between probability vector σi and ground truth vector yi,true is defined using

**Cross entropy**

Cross entropy is a measure of the quality of a modeled probability distribution.

and the resulting cost function is defined using

We implement a cost function “loss” for our model, which, with the aid of the function “nn.softmax\_cross\_entropy\_with\_logits”, applies a Softmax function directly to our output Y and simultaneously defines a cross-entropy cost function between the outputs and ground truth labels, which receive this as a parameter. As our aim is to minimize their value, we call up Tensorflow’s “reduce\_mean” function with the previously defined cost function.

1. **def** loss(label,predicted):
2. **return** tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(
3. labels=label, logits=predicted))

If we call up this function with the ground truth label of a loaded dataset object and the output of our model, the result of the cost function can be output.

1. loss(label,model(image))
2. """tf.Tensor(20.847282, shape=(), dtype=float32)"""

We have now defined exactly what our linear classifier should learn, but we have not yet defined by what means.

### Self-Check Questions

1. Please mark the correct answer(s) with a cross.

* The output of a multi-class support vector machine gives a probability distribution.
* A Softmax classifier constitutes a large margin classifier.
* *The output of a support vector machine can also be negative*.

1. Which cost function can be used to train a Softmax classifier?

For example, using the cross-entropy cost function

1. Please describe the output of a Softmax classifier.

The output of a Softmax classifier constitutes a probability distribution that represents the probability of affiliation to a specific class. The outputs total 1 here.

1. What is one-hot coding?

Here, a vector is defined, the length of which corresponds to the number of classes. The element in the place of the class number has the value 1, while all other elements are set to 0.

## 5.3 Parameter Configuration and Cross-Validation

In the last two sections, we implemented the framework for a linear classifier and now want to adjust its parameters, i.e., the elements of the weight matrix and bias vector, in such a way as to minimize the cost function. To this end, we define a training function that is used to optimize the parameter configuration by calculating the error gradient. The following code shows the implementation of the training algorithm:

1. **def** train(model, x, y, lr=0.5):
2. with tf.GradientTape() as t:
3. t\_loss = loss(y, model(x))
4. dW, db = t.gradient(t\_loss, [model.W, model.b])
5. model.W.assign\_sub(lr \* dW)
6. model.b.assign\_sub(lr \* db)

As parameters, the function receives our model, an input tensor, and the associated ground truth label tensor, as well as a value for the learning rate. The gradient of the cost function and consequently the error deltas of the current parameter configuration are calculated using a Tensorflow “GradientTape” object. The products of the error deltas and learning rate are subtracted from the weight matrix and the bias vector of the model to obtain the new configuration. Here, we use the batch gradient method to configure the model parameters. The gradients of the entire training dataset are added together and their average gradients are formed, which then determine the error deltas of the parameters. This variant of the gradient method has the advantage that it is guaranteed to converge to the global minimum for convex cost functions and to a local minimum for non-convex cost functions. Furthermore, the cost function has a relatively smooth curve with several iterations to the parameter update, which can be attributed to be averaging of the gradients. The following code implements the batch gradient method for the entire MNIST training dataset:

1. **for** data\_train **in** mnist\_train.take(1):
2. images=data\_train["image"]
3. labels=data\_train["label"]
4. images\_train,labels\_train=format\_data(data\_train["image"],
5. data\_train["label"])
6. **print**(loss(labels\_train,model(images\_train)))
7. """tf.Tensor(13.45889, shape=(), dtype=float32)"""
8. train(model,images\_train,labels\_train)
9. **print**(loss(labels\_train,model(images\_train)))
10. """tf.Tensor(10.683774, shape=(), dtype=float32)"""

Here, we define a batch size of the entire number, i.e., 60,000, in advance when loading the training dataset. We then load the batch and format the input vectors and the ground truth labels, in order to be able to use these subsequently for the parameter update. In lines 6 and 9 we output the value of the cost function before and after updating the model parameters respectively, and we identify a decrease in the error. If we repeat this step several times in succession and write the value of the cost function to a “writer” object, we can visualize this as a function of the iterations using Tensorboard. To this end, the “FileWriter” is initialized at the beginning of the code and the path that is to be written to is specified.

1. writer = tf.summary.create\_file\_writer("./session")

The lines of code involved in the writing must then be embedded using a “with” statement and the value of the cost function written to the “writer” object. We define an iteration variable that tells the object when a new value is to be written.

1. tf.summary.scalar('loss\_train', loss(labels\_train,model(images\_train)), step=i)

Chart, line chart

Description automatically generatedFirst, we iterate ten times over the entire training dataset, which gives the following cost function curve:

The smooth curve is clearly visible. The error of our model is indeed decreasing, but very slowly. The major disadvantage of using the batch gradient method is the high computing intensity that is needed to calculate the average gradients in an update step. After presenting another important test method for trained statistical models below, we will look at problem-solving approaches to the slow parameter update, and at alternatives to the batch gradient method, in the subsequent sections.

Evaluating the quality of a statistical model using small samples is problematic. For this reason, the method that has been developed must be validated, with the cross-validation procedure providing an effective method for this. There are different approaches for applying this method. First, an entire dataset could be divided into training and test data in a certain ratio, with the training portion used to configure the model parameters and the test portion used for their validation. Although this method is easy to implement, it has the disadvantage that potentially interesting data from the dataset, which are valuable to training, are not used for training, therefore, resulting in its reduced effectiveness. At the start of this section, we used the training portion of the MNIST dataset to configure the model parameters. The test portion of the dataset can then be used for validation. The following figure shows the principle of the training/test division.

Chart, pie chart

Description automatically generated

One alternative to this variant of validating a statistical model is k-fold cross-validation, in which the entire dataset is divided into k parts of equal size, one part of each is retained for validation, and the rest is used to train the model. This procedure is repeated k times until each part has been used once as a validation set. The following figure illustrates this procedure.

Table

Description automatically generated

Here we assume a subdivision of the dataset into five folds of equal size, resulting in five different validation errors. The total error of the model is produced from the mean of all individual errors. A dataset is frequently subdivided into ten folds, although this choice is not substantiated. A subdivision into k = n folds, where n corresponds to the number of elements in the dataset, results in the leave-one-out cross-validation. Here, the model is trained n times on the entire dataset, with the exception of one element, and validated using the one retained element only. For large datasets in particular, this variant requires extensive computing resources. The choice of suitable subdivisions k leads, with consistent model complexity, to a bias-variance tradeoff (Geman/Bienenstock/Doursat 1992). Compared to k-fold cross-validation with a smaller k, such as 10, for example, leave-one-out cross-validation has a lower bias but a higher variance, as almost the entire part of the dataset is used for training and the training datasets in each iteration are consequently almost identical, which is why the validation errors correlate to a high degree, as the model parameters are also almost identical as a result. When k is smaller, the training datasets differ to a larger degree, which is why the variance decreases. In principle, this problem can be illustrated as follows.

Chart, line chart

Description automatically generated

We present two further approaches to configuring the model parameters below and show how cross-validation can be used to validate the model quality.

### Self-Check Questions

1. A 10-fold cross-validation is carried out to validate a statistical model. How many runs are required for this?

Ten

1. How many elements of a training dataset are used when applying the batch gradient method to update the model parameters?

All elements of the dataset

1. What is the meaning of bias-variance tradeoff?

Compared to k-fold cross-validation with a smaller k, such as 10, for example, leave-one-out cross-validation has a lower bias but a higher variance, as almost the entire part of the dataset is used for training and the training datasets in each iteration are consequently almost identical. This is why the validation errors correlate to a high degree, as the trained models also have almost identical parameters as a result. When k is smaller, the training datasets differ to a larger degree, which is why the variance decreases.

## 5.4 Stochastic Gradient Descent

Up to now, the entire dataset has been used to update the model parameters. The other extreme is to calculate the errors for each individual element and use these directly for the parameter update. This method is known as stochastic gradient descent, or SGD for short, and offers a number of advantages over the batch gradient method. Firstly, a step is calculated very quickly, as only one element is used to calculate the gradient, and, secondly, the amount of memory required is not as great as for the batch gradient method, which is no longer practical for extremely large datasets. The convergence of the optimization is theoretically faster for large datasets using SGD but is associated with increased noise as the variance of the errors is higher. In addition, the high noise level can again lead to a slower convergence. Elements are selected from the dataset randomly here and the total gradients are approximated by these. This is, therefore, a stochastic process, from which the name of the method is derived.

We now adapt our previous implementation to the training of the linear classifier by defining a shuffle and a batch size of 1 when loading the dataset, as follows:

1. mnist\_train=tfds.load(name="mnist", split="train",batch\_size=1,
2. shuffle\_files=True)

We additionally load the test portion of the MNIST dataset with batches of the size 64, in order to validate our model performance on this basis.

1. mnist\_test=tfds.load(name="mnist", split="test",shuffle\_files=True,
2. batch\_size=64)

As the model parameters are updated with each element of the training dataset, the training cost function oscillates significantly. This property helps in training a statistical model insofar that it becomes possible to leave local minima which do not represent the global optimal configuration. We validate our model after 128 steps using a randomly selected portion of the test dataset and visualize the corresponding values of the cost function using Tensorboard. After approximately 10,000 updates of the parameters, which equates to 1/6 of the training dataset, the following cost function curve is produced for the training:

Chart

Description automatically generated with low confidence

Until now, we have taken the learning rate parameter as given and have not addressed its meaning in greater detail. The learning rate is a hyperparameter that determines the extent of the influence of the calculated error deltas on the parameter update. If the selected learning rate is too high and constant, the minimum of the cost function cannot be achieved, as its values will either skip back and forth around the minimum or completely diverge, i.e., become progressively larger. If the selected learning rate is too small, training will progress very slowly, as movement toward the minimum is only occurring in very small steps. Furthermore, with cost functions that are not convex over the entire definition range, it may transpire that the optimizer stagnates in a local minimum and can no longer strive toward the global minimum. It is, therefore, advisable to start with a learning rate that is higher, but not too high, and then reduce this either after a fixed number of iterations or when the value of the cost is not minimized further. Some optimizers also offer the option of an adapting learning rate by adjusting this independently. We add an adaptive learning rate to our previous implementation by multiplying this by a factor of 0.1 every 5,000 iterations.

1. **if** i%5000==0:
2. lr=lr/10

The adaptive learning rate brings about a significant decrease in the value of the cost function compared with the constant learning rate, as the following figure shows.

Chart

Description automatically generated

We define an appropriate measure in order to be able to assess the classifier’s performance based on its classification accuracy. The accuracy of the classification can be calculated by counting the number of matches between the predicted class and the ground truth class and dividing this by the total number of classifications.

To this end, we implement a new function and also write the accuracy for the test dataset to the Tensorboard “writer” object every 128 steps.

1. **def** accuracy(label,predicted):
2. correct=tf.equal(tf.argmax(predicted, 1), tf.argmax(label, 1))
3. accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))
4. **return** accuracy

Chart

Description automatically generatedThe course of the validation accuracy within the first 10,000 iterations exhibits the following curve:

We have seen that our model converges at a faster rate using the stochastic gradient method than with the batch gradient method. Frequently updating the model parameters does, however, require extensive computing resources, which significantly increases the training time. For this reason, it is practical to use another alternative for parameter configuration.

### Self-Check Questions

1. A dataset consists of 20,000 elements. How often does a parameter update take place with the stochastic gradient method so that each element of the dataset is used once?

As only one element is ever used for the parameter update in the stochastic gradient method, a total of 20,000 runs are required.

1. Which gradient method requires more memory: the batch or stochastic method?

As the entire dataset is used for a parameter update in the batch gradient method, the memory requirement is at its highest here.

## 5.5 Mini-Batching

The two extremes involved in updating the model parameters, the batch and the stochastic gradient method, in which either the entire or only one element of the training dataset at a time is used, have their own advantages and disadvantages. Mini-batching constitutes an alternative to both these methods. Here, small parts of the dataset are used, in order to calculate the error gradients and the parameter update. The aim of this method is to achieve equilibrium between the positive and negative properties of the SGD and batch gradient method. The following figure provides an excellent illustration of the various curves of the three gradient methods for reaching a minimum (Dabburra 2017).

Diagram

Description automatically generated

Chart

Description automatically generatedHere, the black ellipses represent an imaginary parameter hyperspace and the red point its minimum. Mini-batching is the most commonly used parameter update method, particularly in deep learning. The choice of mini-batch size introduces a new hyperparameter to the development of a statistical model that must be determined to optimal effect. In many cases, the maximum size is, however, already determined by the available (GPU) memory. Based on the underlying computer architecture, this should represent a power of two, in order to permit optimal storage in the memory. A batch size of 32 is often recommended as a guide value (Bengio 2012; Masters/Luschi 2018). We adapt our implementation accordingly and define a batch size of 32 when loading the MNIST dataset. The following curves of the cost function are the product of the training (left) and test dataset (right).

The corresponding curves of the training (left) and validation accuracy (right) are shown below.

Chart

Description automatically generated

The noise in the training data curves is significantly reduced in contrast to the SGD by using several elements of the dataset to calculate the gradients. If we compare the computing times required to use each element of the MNIST dataset once for the parameter update, we can see clear differences between the three methods when the training was only carried out on one CPU. While the training time between the batch gradient method and the mini-batch gradient method is still close at approximately 13.9 seconds and 15.4 seconds respectively, SGD requires a multiple of the time at more than eight minutes. If we consider the convergence speed and the required computing time, the mini-batch gradient method represents the best ratio.

### Self-Check Questions

1. A dataset consists of 5,000 elements. If a batch size of 10 is selected, how many iterations are required in order that each element is used once for the parameter update?
2. What is often the limiting factor in the selection of the batch size for the mini-batch gradient method?

The memory capacity of the graphics card

1. Which of the following gradient methods performs the most frequent parameter updates: batch, stochastic or mini-batch gradient method?

Stochastic gradient method

## 5.6 Epochs

So far, we have trained our model for a specific number of iterations, without addressing another important hyperparameter for stochastic model development: a run, in which each element of a dataset was used once to update parameters, is known as an epoch. When using mini-batching with a batch size k and a dataset of the size n, iterations are required within an epoch in order to run through the entire dataset once. In practice, no fixed number of epochs is initially defined in most cases, rather iteration is carried out as often as is necessary until the error is no longer minimized. Using the method of early stopping, a model can be trained until it reaches the threshold of overfitting (Yao/Rosasco/Caponnetto 2007).

The model we have designed shows no further significant improvement on using mini-batching after six epochs. The classification rate of the trained linear classifier on the entire MNIST test dataset is around 82.5%. However, validating the quality of our model on the basis of this sample is not optimal, as already mentioned in the previous section. Instead, we now want to implement a 10-fold cross-validation and thereby divide the training portion of the MNIST dataset into ten folds of equal size. We will then use the test portion to determine the accuracy of our finished model. In order to parameterize the distribution of the data using percentage information, it is necessary to use the “mnist:3.\*.\*” version of the MNIST dataset.

1. data\_val = tfds.load('mnist:3.\*.\*', split=[
2. 'train[{}%:{}%]'.format(k, k+10) **for** k **in** range(0, 100, 10)
3. ],batch\_size=32)
4. data\_train = tfds.load('mnist:3.\*.\*', split=[
5. 'train[:{}%]+train[{}%:]'.format(k, k+10) **for** k **in** range(0, 100, 10)
6. ],batch\_size=32)

Here we generate a list of parts of the training dataset using the integrated for-loop, which parameterizes the “split” parameter with the percentage. The validation list, therefore, contains the following ranges:

Division of the Dataset for Cross-Validation

|  |  |
| --- | --- |
| 0-10% | data\_val[0] |
| 10-20% | data\_val[1] |
| 20-30% | data\_val[2] |
| 30-40% | data\_val[3] |
| 40-50% | data\_val[4] |
| 50-60% | data\_val[5] |
| 60-70% | data\_val[6] |
| 70-80% | data\_val[7] |
| 80-90% | data\_val[8] |
| 90-100% | data\_val[8] |

The corresponding remaining folds of the division can be found in the training list. We now iterate over all ten folds within ten runs so that each of these has been used once for validation.

1. **for** train\_el,val\_el **in** zip(list\_train,list\_val):
2. i=0
3. lr=0.2
4. model=LinearModel(shape\_W,shape\_b)
5. **for** e **in** range(0,7):
6. **for** data\_train **in** train\_el:
7. images\_train,labels\_train=format\_data(data\_train["image"],
8. data\_train["label"])
10. train(model,images\_train,labels\_train,lr)
12. **if** i%5000==0:
14. lr=lr/10
15. i+=1
16. images\_val,labels\_val=format\_data(val\_el["image"],val\_el["label"])
17. acc.append(accuracy(labels\_val,model(images\_val)))
18. """[0.801,0.814,0.802,0.803,0.802,0.808,0.801,0.812,0.824]"""

With each new fold, the model is reinitialized so that the previously learned model parameters have no influence on the further process. We now iterate using the training data over seven epochs and then validate the trained model using the respective validation dataset. The respective values for the validation accuracy are then added to a list for further processing at a later stage.

The cross-validation results in ten validation accuracy values, which are shown in line 18 of the previous code extract and are now to be processed statistically. Box plots are a particularly good means of visualizing these values, enabling authoritative position and scatter measures to be clearly summarized. For this purpose, it is necessary to calculate a number of measures of the cross-validation result, which are represented by the median, the lower and upper quartile, and the minimum and maximum of the data. The lower and upper quartiles contain the data that belong to the lower and upper quartile of the entire data respectively. Box plots from data series can be visualized particularly well using numpy. The following figure can be used to evaluate the conducted cross-validation statistically.

Chart, box and whisker chart

Description automatically generated

The median of the data is represented by the orange line within the box. This is only just above the minimum. Approximately 50% of the cross-validation results, therefore, lie within this range, with the spread of the remaining values being significantly larger and extending over approximately two percentage points.

Within this unit, we have now worked, step by step, through the development and validation of a linear classifier, which is able to classify the MNIST dataset correctly to around 80% without any further optimization. This relatively simple statistical model constitutes a good basis for further work involving more complex models. The fundamental steps of model definition, training, and validation are part of the standard procedure involved in their development.

### Self-Check Questions

1. 1,000 iterations are required in order to use a complete dataset once for the parameter update to a model. A batch size of 32 was selected. How many elements does the dataset contain?



1. Please mark the correct answer(s) with a cross.

* *A box plot contains information on the quartiles of a data series.*
* *Within an epoch, each element of a dataset is used once for the parameter update.*
* More epochs always result in a higher classification rate.

|  |
| --- |
| Summary |
| Linear classifiers can be used to adjust their parameters, i.e., the weights and biases, on the basis of training data in such a way that even new, unseen data can be correctly classified. They consist of a weight matrix and a bias vector and map input vectors of a certain length to an output vector that corresponds to the number of existing classes. Using the multi-class support vector machine, a cost function can be minimized, which attempts to separate the present classes from each other in such a way that their distances from the class-separating hyperplane are maximal. The support vector machine thus constitutes a large margin classifier. The output of a multi-class support vector machine is not interpretable as it does not carry any information on how certain it is about the classification. The Softmax classifier is a more suitable alternative. This is an expansion of the logistic function and outputs probability values for affiliation to the respective classes, with all class probabilities totaling 1. The suitable cost function for training a Softmax classifier is cross entropy, where the predicted output and ground truth label must be identical in form. One-hot coding of a class label is used for this purpose, which is as long as the number of classes and carries a 1 at the position of the correct class, with all other entries being 0.  The gradient method is used to configure the parameters of a statistical model, the functionality of which can be implemented directly using integrated Tensorflow functions. There are three concise variants that can be used and which each offer their own advantages and disadvantages. The batch gradient method calculates the error gradient for each element of a training dataset and generates the mean value from this, which is then used for the parameter update. This variant requires extensive computing and memory resources, thus rendering it impractical for large datasets. The second extreme, the stochastic gradient descent, uses individual, randomly selected elements of the dataset to calculate the gradient and carries out a parameter update in each case. The convergence of this variant is considerably greater but is associated with high noise. In addition, extensive computing resources are also needed to calculate the gradient for each individual element and the training time is significantly increased. Mini-batching provides a suitable alternative, in which the training dataset is divided into small batches of fixed size, which are used to update the model parameters.  Cross-validation can be used to validate a statistical model. In k-fold cross-validation, the training dataset is divided into k folds of equal size, each of which are used once within k runs for validation and the rest for training. The information obtained about the performance of the individual runs can be used to carry out a statistical evaluation. Box plots are particularly suitable for this, as they are able to present position and scatter measures in concise form. |

# Unit 6—Multilayer Neural Networks

Study goals

After completing this unit, you will know ...

… what the motivation is behind using neural networks.

… what advantages neural networks have over linear classifiers.

… what the fundamental structure of multilayer neural networks is and what mathematical operations they are subject to.

… how multilayer neural networks can be implemented and trained using Tensorflow.

… what overfitting is and how this can be addressed using different methods.

# 6. Multilayer Neural Networks

### Introduction

This unit covers the topic of multilayer neural networks. First of all, we will look at the advantages over linear classifiers and discuss the fundamental structure, including the underlying mathematics and biological inspiration.

Implementing dedicated models using the Tensorflow framework will make up a large part of this unit. Both the use of pure Tensorflow functions and the high-level interface Keras will be discussed and we will demonstrate how multilayer neural networks can be implemented using just a few lines of code. We will show how a classification network can be developed that can be used to classify the MNIST dataset to a high level of accuracy. We will also discuss the saving and loading of models, as essential processes for a reliable development process. To close, we provide an insight into the topic of model overfitting and possible problem-solving approaches.

## 6.1 Introduction and Motivation

Linear classifiers are particularly suitable for data that meet the criterion of linear separability, i.e., where clear boundaries are to be drawn between the various underlying classes. However, if the data exhibit a higher degree of noise, i.e., if the classes overlap and, as a result, cannot be clearly separated from each other, then their performance will decrease considerably. Intuitively, one approach to factoring in non-linearities could be to link several linear classifiers in succession, in order to be able to map the underlying functions. This will not, however, achieve an increase in performance, which can be directly understood from the following equation:

Here, W1 and b1 represent the weight matrix and bias vector of a first linear classifier, and W2 and b2 the respective parameters of a second linear classifier. The function ϕ represents a non-linear function of the output, for example, a Softmax function, in order to perform the classification. The equation can be rearranged to:

This gives a new weight matrix W12, a new bias vector b12 and thus, once again, a linear classifier y12, which features the same restrictions as the individual linear classifiers y1 and y2. Regardless of how many linear classifiers are concatenated, the result remains a linear classifier.

The solution to the problem lies in the addition of non-linear activation functions to each stage. This prevents the chain from being reduced to another linear function, which is visible from the following equation:

The first stage of our model thus defined represents the hidden layer and the second stage represents the output layer of a neural network. Compared with the linear matrix, a non-linearity is now inserted into the linear mapping defined by the weight matrix W1 and the bias vector b1 using the activation function ϕ1, by which means the aforementioned restrictions are lifted. Non-linear functions can, therefore, be approximated using neural networks and represent function approximators. The universal approximation theorem states that a neural network with only one hidden layer can, in fact, theoretically replicate any real continuous function using a sufficient number of neurons (Csáji 2001). In order to illustrate the universal function approximation, let us consider the following function, for which the equation and thus the relationship between input x and output y is unknown.

Chart, line chart

Description automatically generated

How can this function now be approximated using a neural network? To this end, we can divide this function into small sections, which, in themselves, represent simpler functions and, in their entirety, represent the mapping to be approximated.

Chart, histogram

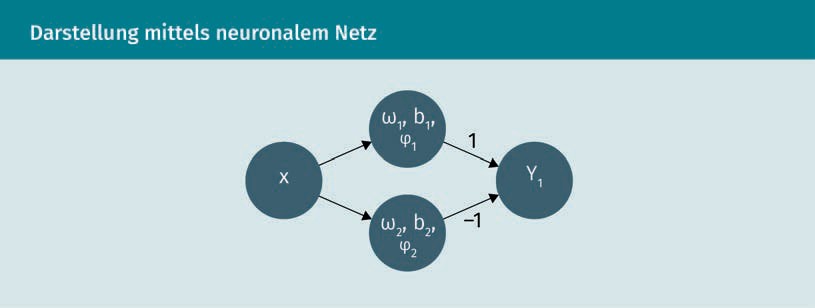
Description automatically generated

Equivalent to the forming of a function’s integral, a perfect approximation of the function is produced from a number of subdivisions of n → ∞. When using sigmoid functions as the activation function of the neurons, such a square-wave signal can be achieved by combining two neurons. Chart, box and whisker chart

Description automatically generated

Chart, box and whisker chart

Description automatically generatedBy selecting suitable parameters, the slope of the sigmoid function can be set very steeply, thus allowing a square-wave function to be replicated. One of the two neurons deflects here just before the value τ and one just after. Subtracting the two signals from each other gives the required signal.

This subtraction can be represented as a neural network using suitable weights as follows:

By adding neuron pairs to the hidden layer, which each map a piece of the function, the entire function can be approximated in this way. Furthermore, this universal function approximation is not restricted to one-dimensional functions but is theoretically valid for any high-dimensional mapping. This raises the question of how efficiently such a network can be trained. It has been shown that neural networks consisting of several hidden layers require significantly fewer neurons for the majority of functions than those with just one hidden layer and can be trained more efficiently (Håstad 2014). Neural networks that contain more than one hidden layer are known as deep neural networks and can increase performance due to the increasing abstraction of the data (Kozma/Ilin/Siegelmann 2018; Khan et al. 2019).

In addition to the previous structure of the feedforward neural network, in which the flow of data runs exclusively from front to back, feedback to previous layers is also possible, which is referred to as a recurrent neural network. This recurrent neural network enables sequences, such as video or audio data, to be processed by giving the network a memory of sorts. Recurrent networks are considerably more difficult to train than feedforward networks and, as a result, will not be covered further in this unit. For a detailed description, please consult the relevant literature (Sak/Senior/Beaufays 2014; Li/Wu 2015).

### Self-Check Questions

1. When are linear classifiers no longer sufficient for classification problems? What is the solution to this?

If the underlying data are not linearly separable, then the use of linear classifiers is not suitable. Adding non-linear activation functions, which represent neural networks, is the solution to this.

1. What is the result of concatenating several linear classifiers?

The result is again a linear classifier.

1. What does the universal approximation theorem say?

The universal approximation theorem states that a neural network with only one hidden layer can theoretically approximate any real continuous function using a sufficiently high number of neurons.

## 6.2 Structure and Mathematics

In addition to an input and output layer, neural networks consist of at least one hidden layer in between. With the aid of non-linear activation functions, they represent universal function approximators and can be used for a variety of different tasks through the suitable training of a dataset. The respective layers consist of neurons, which each carry out a non-linear mapping of an input vector to an output value as follows:

The activation function ϕ represents the non-linearity, whereby a variety of different function types exist. The artificial neurons were originally motivated by biological nerve cells that process signals in the brains of mammalsDiagram

Description automatically generated. The following figure shows the core components of a nerve cell:

The dendrites are responsible for picking up signal inputs and transferring these to the cell body, the soma. If the incoming signal exceeds a certain threshold value, the nerve cell “fires” and transmits a signal to the output via the axon to adjacent nerve cells. The interconnection of billions of these cells forms the basis of all signal processing performed by the brain (Churchland/Sejnowski 2013). If we apply this to the artificial neuron, the timing of the signal transmission is represented by the activation function. In contrast to the biological variant, current implementations do not work to the all-or-nothing principle of the threshold value but exhibit a variable slope. The first model, known as the McCulloch-Pitts neuron, did, however, work exactly like the biological variant (McCulloch/Pitts 1943). The following illustration shows a selection of common activation functions that are largely used today for multilayer neural networks.

Diagram

Description automatically generated

The sigmoid function is shown on the left, which is defined by

and has a value range of (0,1). This is largely used for neural networks that are intended to predict probability, as the value range corresponds precisely to this output. In the middle is the hyperbolic tangent function (tanh) related to the sigmoid function, which is defined by

and has a value range of (–1,1), meaning that it is symmetrical about the value zero. This property is advantageous in relation to training using backpropagation (Nwankpa et al. 2018). These two activation functions perform poorly in very deep neural networks, as the gradient in the front layer either becomes very small or disappears altogether (vanishing gradient problem). The solution to this is the ReLu function (rectified linear unit), which is shown on the right and is defined as follows:

This means that the output is always set to the value 0 when the input value is negative, and a positive input value is output unchanged. In Tensorflow, the various activation functions can be easily implemented using the functions

1. tf.math.sigmoid # Sigmoid function
2. tf.math.tanh # Hyperbolic tangent function
3. tf.nn.relu # ReLu function

Naturally, different types of activation functions can also be used within the same neural network. For example, the hidden layers are often activated using the ReLu function and the output layer is activated using the sigmoid function (or Softmax function in the event of multiple outputs) based on the required value range. The activation functions effectively act as a link between two levels.

A neural network in its entirety can be viewed as a single mathematical function, with the successive layers representing nests of the non-linear mappings from the previous layers. Let us consider, for example, the following structure of a neural network in which only the layers and not the individual neurons are shown.

Chart, table

Description automatically generated

The output of the network corresponds to the following function only:

This nesting is very important for training neural networks using backpropagation. As the output of a layer is dependent on the output of the previous layer, the chain rule is required to form the gradient, by means of which the differentiation of interlinked, i.e., inner and outer, functions of the form y = u(v(x)) is defined. To this end, the inner and outer derivative are formed and linked according to the rule in order to obtain the entire derivative.

### Self-Check Questions

1. Which of the following terms are components of the biological nerve cell?

* *Axon*
* *Dendrites*
* *Soma*
* Adam

1. Which of the following activation functions has a value range of (–1,1)?

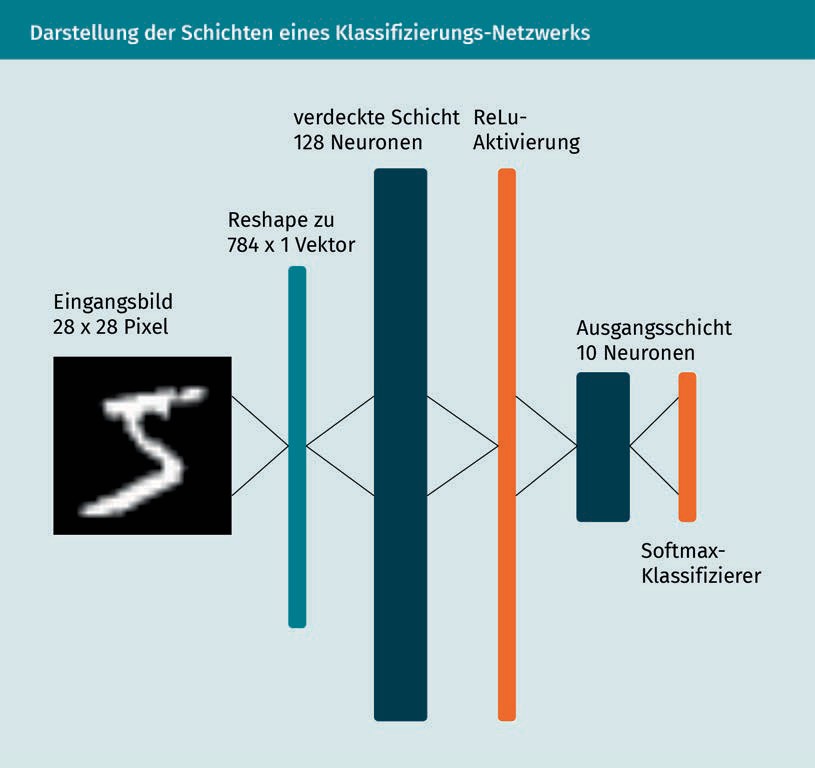
* Heaviside function
* *Hyperbolic tangent function*
* ReLu function
* Sigmoid function

3. Which activation function is a solution to the vanishing gradient problem?

*The ReLu function*

## 6.3 Implementation with Tensorflow

Following the introduction to the structure and underlying mathematics of neural networks in the previous two sections, our focus in this section will be on the implementation of a simple neural network using Tensorflow 2.0. To this end, we will design a feedforward network for use in classifying the MNIST dataset, which consists of 60,000 pieces of training data and 10,000 pieces of test data in the form of handwritten numbers between 0 and 9.

We will start by defining the architecture of our neural network. In addition to an input and output layer, we first want to implement a hidden layer consisting of 128 neurons and activate these using ReLu activation functions. For the output layer, we will carry out the classification using the Softmax function instead. The following structure shows the levels of the neural network that are to be implemented:

The image data in the dataset are composed of 28 x 28 pixels and must first be converted into a 784-dimensional vector for entry into the network. The resulting vector is applied to a hidden layer with 128 neurons and its output is activated by a ReLu function. In order to represent the ten classes present, the activated vector is subsequently applied to the output layer consisting of ten neurons and the probability of affiliation to the respective classes is defined using the Softmax function. We will start implementing the individual levels from front to back.

The MNIST dataset can be easily loaded into the Tensorflow environment using “tensorflow\_datasets”, a preproduced compilation of Tensorflow datasets.

1. **import** tensorflow as tf
2. **import** tensorflow\_datasets as tfds
3. dataset\_train=tfds.load(name="mnist:1.\*.\*",split=tfds.Split.TRAIN)
4. dataset\_train=dataset\_train.shuffle(1024).batch(128)
5. dataset\_test=tfds.load(name="mnist:1.\*.\*",split=tfds.Split.TEST)
6. dataset\_test=dataset\_test.repeat().batch(128)

Here, we select the training or test part of the dataset using the split command, blend the training part with a **buffer size** of 1,024 elements, and set the batch size for both parts to 128.

**Buffer**

A buffer is used to store data on a temporary basis.

The reshaping of the image matrices into an input vector is commonly referred to as flattening and can be implemented as follows using the Tensorflow function “reshape”.

1. **def** flatten(inputs):
2. **return** tf.reshape(inputs,shape=(tf.shape(inputs)[0],-1))

We define the “flatten” function, which expects a tensor as a parameter and returns the corresponding column vector. Transferring the parameter -1 to the “reshape” function always generates a single-column vector with a length that is equal to the number of elements of the input tensor. Next, we define a function for the neuron layers. These are implemented as fully connected layers, also known as dense layers, by transferring the input tensor and a weight matrix to the function. The output of the neuron layer is calculated using the Tensorflow function tf.matmul and returned by the function.

1. **def** dense(inputs,weights):
2. **return** tf.matmul(inputs,weights)

For the weights, we generate a list consisting of the matrices of the individual layers. Here, it is important that the “trainable” parameter is set to “True” for the variable definition, in order to be able to train the weights.

1. weights=[]
2. weights.append(tf.Variable(initializer([784,128]),name="d1",trainable=True,dtype=tf.float32))
3. weights.append(tf.Variable(initializer([128,10]),name="y",trainable=True,dtype=tf.float32))

The first entry in the list represents the weight matrix of the hidden layer, with its number of rows corresponding to the length of the input vectors and its number of columns corresponding to the number of neurons in the layer. The weight matrix of the output layer is defined by the number of neurons in the previous layer and the number of classes. We are now in a position to define the model of the network architecture.

1. **def** model(x):
2. x=tf.cast(x,dtype=tf.float32)
3. x=flatten(x)
4. d1=dense(x,weights[0])
5. d1=tf.nn.relu(d1)
6. y=dense(d1,weights[1])
7. **return** y

The outputs of the individual functions can be easily chained together to create the network. The first function “tf.cast” performs a type conversion so that the input tensors are of the “float” type, thus enabling the other functions to process them. This brings the implementation of the forward path to a close and the functions for the backward path can now be defined. To this end, it is first necessary to define a suitable output for our network and the associated cost function. We want to use a Softmax classifier and minimize its error using the cross-entropy cost function, which can be implemented within the following function:

1. **def** loss(label,pred):
2. tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(label,pred))

In order to update the parameters of the neuron layers, we must first define an optimization algorithm. Tensorflow provides a range of different methods for this purpose, which each have their own advantages and disadvantages. In this case, we will select the Adam optimizer, which represents a variant of the gradient method that has been expanded to include an additional parameter. For a detailed description of how the algorithm works, please consult the relevant literature (Kingma/Ba 2017). The optimizer can be defined as follows by transferring a value for the learning rate:

1. optimizer=tf.optimizers.Adam(lr)

The process of updating the weights of the respective layers is implemented within the following function:

1. **def** train(model,inputs,outputs):
2. with tf.GradientTape() as tape:
3. current\_loss=loss(outputs,model(inputs))
4. gradient=tape.gradient(current\_loss,weights)
5. optimizer.apply\_gradients(zip(gradient,weights))
6. **return** current\_loss

The gradients are calculated using a Tensorflow “GradientTape” object, which is applied to the current weights via the defined optimizer. We now have all the necessary components for training our network. We can start the training and testing on the basis of the training and test dataset respectively using the following loops:

1. lr=0.01
2. **for** e **in** range(0,5):
3. optimizer=tf.optimizers.Adam(lr)
4. **for** data\_train,data\_test **in** zip(dataset\_train,dataset\_test):
5. image\_train,label\_train=data\_train['image'],data\_train['label']
6. image\_test,label\_test=data\_test['image'],data\_test['label']
7. label\_train=tf.one\_hot(label\_train,10)
8. label\_test=tf.one\_hot(label\_test,10)
9. loss\_train=loss(label\_train,model(image\_train))
10. loss\_test=loss(label\_test,model(image\_test))
11. lr/=10

Chart

Description automatically generatedWe initially set the learning rate value to 0.01. The network is trained for five epochs and the learning rate is divided into tenths after each epoch. As the labels of the dataset only bear the number of the correct class, they still need to be converted into the correct format for the Softmax classifier using the Tensorflow function “tf.one\_hot”. The values of the cost functions for training and testing can be visualized effectively using Tensorboard, as shown in the figure below.

The minimization of the errors can be clearly seen, although there is no further improvement from an error of around 0.3. We can determine the classification rate using the following function:

1. **def** accuracy(label,predicted):
2. correct=tf.equal(tf.argmax(predicted, 1), tf.argmax(label, 1))
3. accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))
4. **return** accuracy

This gives a comparison of how often the predicted and ground truth label match and divides the result by the number of overall classifications. The following curves are produced for the training and test classification rates.

Chart

Description automatically generated

Chart, scatter chart

Description automatically generatedA test accuracy of around 92% can be achieved with just one hidden layer. This represents an increase of more than ten percentage points over a linear classifier. We will now add another hidden layer to the network and see whether this brings about an improvement in the classification rate. The following figure shows the accuracies for the training and test data respectively.

We achieve an improvement to approximately 95% within the test data. The neural network can generalize better with two layers than with just one layer and achieves a higher degree of accuracy as a result. The addition of further layers would not bring about a further improvement, however: it would lead to a problems that will be addressed in the following sections. There is a trick we can use to gain a few more percentage points in classification accuracy in the form of normalization. To this end, we divide all pixels of the input data by the maximum possible value, which in this case is 255, as the color values range between 0 and 255, with the result that all pixel values are now between 0 and 1. As a result, the variance is reduced. Repeating the network training with normalized input values achieves an accuracy of approximately 98%.

With Tensorflow 2.0, Keras became the official high-level interface, rendering network implementation much more user-friendly. Our neural network can be reduced to the following code using the Keras interface:

1. **import** tensorflow as tf
2. mnist = tf.keras.datasets.mnist
3. (image\_train,label\_train),(image\_test,label\_test)=mnist.load\_data()
4. image\_train,image\_test=image\_train/255.0,image\_test/255.0
5. model = tf.keras.models.Sequential([
6. tf.keras.layers.Flatten(input\_shape=(28, 28)),
7. tf.keras.layers.Dense(128, activation='relu'),
8. tf.keras.layers.Dense(10, activation='softmax')
9. ])
10. model.compile(optimizer='adam',
11. loss='sparse\_categorical\_crossentropy',
12. metrics=['accuracy'])
13. model.fit(image\_train,label\_train,epochs=5)
14. model.evaluate(image\_test,label\_test,verbose=2)

From line 3 to 6, the MNIST dataset is loaded via the dataset library of the Keras interface and normalized. In lines 8 to 12, the network architecture is created using preproduced layers, which can already be parameterized with the required number of neurons and activation function. In lines 14 to 16, the training is defined, complete with optimization algorithm and cost function. The metrics for outputting, such as the classification accuracy, for example, can also be specified. Training is started using the “fit” method and testing is started using “evaluate”. The advantage of implementation using the Keras interface lies in the direct implementability enabled by pre-produced functions, and the immediate logging of training and test metrics.

### Self-Check Questions

1. What is the purpose of the “model.fit()” function of the high-level Tensorflow interface Keras?

* *To train a model*
* To configure a model
* To evaluate a model

2. What is the purpose of the Adam algorithm?

Adam represents an optimization algorithm for training neural networks.

## 6.4 Adapting Existing Models

After explaining the development of a new neural network “from scratch” in the last section, we will now address the adaptation of existing models using Tensorflow. But first of all, why are Tensor models stored at all? The answer is, just as with other documents, to save the current training status and to be able to access this again at a later stage. Neural networks can be saved with Tensorflow during and after training using checkpoints or SavedModels. Here, either just the trained parameters (each tf.variable) or the complete model can be saved. It is this process that renders it possible to exchange deep learning applications between several people, and thus collaborate, in the first place. The checkpoints do not contain a description of the mathematical operations for the saved model, whereas in SavedModels the full calculation descriptions are serialized. Serialization refers to mapping structured data, in this case the network model, to a sequential form of representation. This allows the entire states of objects, including all referenced objects, to be saved and then converted back to the structured file format at a later stage, which is known as deserialization.

First of all, let us refer to the code example used in the last section to classify the MNIST dataset using Keras interface implementation, which is relisted below.

1. **import** tensorflow as tf
2. mnist = tf.keras.datasets.mnist
3. (image\_train,label\_train),(image\_test,label\_test)=mnist.load\_data()
4. image\_train,image\_test=image\_train/255.0,image\_test/255.0
5. model = tf.keras.models.Sequential([
6. tf.keras.layers.Flatten(input\_shape=(28, 28)),
7. tf.keras.layers.Dense(128, activation='relu'),
8. tf.keras.layers.Dense(10, activation='softmax')
9. ])
10. model.compile(optimizer='adam',
11. loss='sparse\_categorical\_crossentropy',
12. metrics=['accuracy'])
13. model.fit(image\_train,label\_train,epochs=5)
14. model.evaluate(image\_test,label\_test,verbose=2)

Now what happens when the code is started? With each run, the training and test dataset is loaded first, and the model is defined, parameterized, trained, and subsequently evaluated. Should there be a requirement for a new image to be classified by the network at a later stage, the network would first have to be retrained over and over. This is not necessary, of course, and for very deep network architectures in particular, with training durations of up to several weeks, is completely impractical. In our case, the simplest course of action is to execute the “save” method of the model object after the “fit” method. The model is serialized in the “Hierarchical Data Format 5” (HDF5) with the file extension .h5.

1. model.save('new.h5')

We will now create a new Python model, load the saved model into our environment as follows, and reevaluate this with the aid of the MNIST test dataset:

1. **import** tensorflow as tf
2. mnist = tf.keras.datasets.mnist
3. (image\_train,label\_train),(image\_test,label\_test)=mnist.load\_data()
4. image\_test=image\_test/255.0
5. model = tf.keras.models.load\_model('new.h5')
6. model.evaluate(image\_test,label\_test,verbose=2)

If we compare the classification rates of the evaluation directly after training the network and after loading the stored model, we will see that they agree. This shows that it is easily possible to implement fully trained models at a later stage. But what is the situation if this model is to undergo further training at a later stage? When a model is saved within an HDF5 file, not only the weights and architecture are saved but also information on the optimizer used and the cost function. This allows interrupted training to be resumed without issue at a later stage. Once again, this is carried out using the “fit” method of the loaded model object. The model can also be stored in the Tensorflow-specific SavedModel format instead of in the HDF5 file format. To this end, the additional parameter “save\_format= ‘tf’” is transferred instead of the file extension h5.

1. model.save('new', save\_format='tf')

The advantage here lies in the additional storage of the underlying Tensorflow graph.

A further option for saving a model is to use checkpoints, which only contain all values of trainable parameters for a model. Information on the model architecture, the optimizer used, or the cost function is lost in this process. For this reason, the use of checkpoints is only advisable if the corresponding Tensorflow code that was used to generate them is available. They are used as standard during the training in order to enable access to various configurations of a network at a later stage, if, for example, this has been trained and overfitted for too long. The previous weights can simply be reloaded when using checkpoints, meaning that the process does not have to be restarted entirely from scratch. A checkpoint is saved by calling up the “save\_weights” method, either in HDF5 format using

1. model.save\_weights('model\_check.h5')

or in Tensorflow format using

1. model.save\_weights('model\_check', save\_format='tf')

These methods can only be executed after, and not within, the “fit” method, which is why they can only be used to save the weights after the training run. The use of callbacks constitutes a more sophisticated solution. Callbacks are a cluster of functions that can be executed at a certain point in the training. They are transferred to the “fit” method as parameters. We can define a callback to store a checkpoint after each epoch as follows:

1. callback=tf.keras.callbacks.ModelCheckpoint(
2. filepath='model\_check.{epoch:03d}',
3. save\_weights\_only=True,
4. verbose=1,save\_frequency=1)
5. model.fit(image\_train,label\_train,epochs=5,callbacks=[callback])

In this process, the file path and file name are first transferred to the ModelCheckpoint. A formatted specification is also possible, as in this case using “{epoch:03d}”, whereby the file name contains the number of the epoch. Setting the parameter “save\_weights\_only” to “True” only saves the model weights, i.e., the “save\_weights” method is executed in the background, whereas setting the parameter to “False” will execute “save”. The frequency of the output on the console can be controlled using the “verbose” parameter, while “save\_frequency=x” allows storage to be set for every x epochs. If we run the expanded code, a checkpoint will be saved after each epoch. Each checkpoint consists of an “index” file and one or more “data” files. The former contains information on the name and format of the saved variables, while the latter contain the actual values and, for this reason, are considerably larger. The required weights can then be loaded into a model using

1. model.load\_weights(checkpoint\_name)

which, as the storing model, must feature the exact architecture. An additional useful parameter of the ModelCheckpoint callback is “save\_best\_only”. Setting this to “True” means that only the weights that bring about an improvement in performance over the previous weight configurations, as measured by the metric transferred via the “monitor” parameter, are stored. The following callback definition, for example, stores the model weights each time the validation weight reduces:

1. callback=tf.keras.callbacks.ModelCheckpoint(
2. filepath='model\_check.{epoch:03d}',
3. save\_weights\_only=True,
4. verbose=1,monitor='val\_loss',
5. save\_best\_only=True)

This facility, which allows model parameters to be stored at any desired time, now ensures more reliable and organized training and testing of neural networks.

### Self-Check Questions

1. What is the purpose of callbacks in the training of neural networks?

* To update the weights
* *To execute functions at specific times*
* To calculate the error gradients

1. What are saved when a SavedModel is stored?

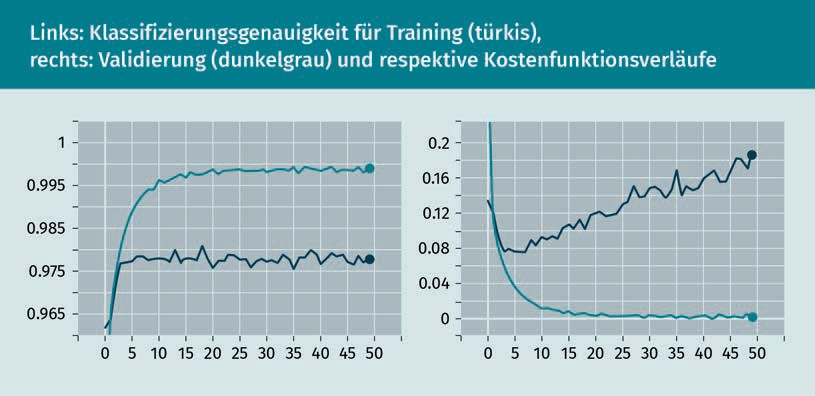
* *Weights*
* *Cost function*
* *Optimizer*
* *Calculation graph*

## 6.5 Overfitting and Problem-Solving Approaches

In order for neural networks to learn problems, they must be taught using training data. So why not just train a model until it achieves an accuracy of 100% on this data, as it should then achieve maximum performance? Let us imagine that our network learns to classify data into ten different classes and achieves a classification accuracy of 100%. However, if we now try to classify new, unseen data, we will find that the accuracy is many times lower, for example, below 50%. What has happened here? The key word here is “overfitting”. Our network has simply committed the training data to memory and adjusted the weights perfectly so that they are correctly classified with one hundred percent accuracy and with no generalization. Generalization plays an essential role in the development of statistical models, without which it would be impossible to apply it to data outside of the training dataset. A generalizing model learns the underlying “signal” and ignores the noise around it. The following figure illustrates this issue using the example of a binary classification problem.

Chart

Description automatically generated

The blue and red objects represent elements from two different classes. While the model represented by the black line recognizes the basic signal and learns the class division accordingly, the dashed line represents an overfitted model that learns the noise around the signal. The impact of this is the failure to classify data that have not been used for training. Let us revisit our example from the start of this unit on the classification of the MNIST dataset and now run the training over 50 epochs instead of five. The following figures show the training and validation accuracies (orange and blue respectively on the left) and the training and validation errors (orange and blue respectively on the right).

The figures very clearly show that the training error continues to decrease with ongoing training and the training accuracy continues to increase, whereas the validation error reaches its minimum at around the seventh epoch and then increases at a steady rate. The validation accuracy also stagnates or reduces from this point in time. It is precisely this inflection point in the validation curve that can be used to identify the point in time of overfitting, from which a steady deterioration in the generalization of the model occurs. What can we do now to counteract this problem? One possible problem-solving approach is the dropout method, which belongs to the group of regularization methods (Srivastava et al. 2014). Here, as the name suggests, certain neurons of the neural network are ignored and not updated during the forward and backward path. The number of affected neurons is random here and the frequency of the dropout is defined with a given probability. The following figure illustrates the dropout method using a sample neural network:

Shape, bubble chart

Description automatically generated

The neural network is represented without dropout on the left and with dropout on the right. Neurons affected by dropout have no effect on either the preceding or subsequent neurons. This prevents the neurons from developing mutual dependencies during the training and avoids more robust features being learned. All neurons are then used normally during the test phase. With the incorporation of the Keras interface, the dropout method can be easily executed by inserting

1. tf.keras.layers.Dropout(0.2)

Chart

Description automatically generatedafter the affected layer, with the parameter indicating the probability of application. Without a Keras interface, this can be applied using “tf.nn.dropout”. The following figure shows the accuracy of training and validation classification (left) and the training and validation error (right) with the use of the dropout method, with a probability of 20% for the first and 30% for the second layer of our network.

It can be clearly seen that the validation error does not increase further compared to the training without dropout but continues to decrease or stagnate. As a result, we have been able to gain about half a percentage point in validation accuracy. Generally, the optimal dropout value for the respective layers must be determined by means of testing.

Two further regularization methods that are often used to avoid overfitting are L1 and L2 regularization. In this respect, a further term, known as the regularization term, is added to the cost function as follows:

Here, “error” represents the chosen cost function of the model, such as the mean squared error or cross entropy, for example. The L1 and L2 regularizations differ in the type of regularization term. The former adds the sum of the weight amounts, while the latter adds the sum of the squared weights. The following equations represent the respective regularization terms:

Here, the impact of the regularization term on the total error is determined by the factor λ, whereby the selection of a value that is too large leads to an underfitting model. The aim of these regularization methods is to ensure that the weights of a model do not become too large. Large weight values lead to large regularization terms and consequently to larger overall errors. The main difference between L1 and L2 regularization lies in the fact that the former “penalizes” high weight values only, whereas the latter “penalizes” all values through squaring. For a detailed description of how the methods work, please consult the relevant literature (Shanmugamani 2018).

With Tensorflow, the regularization can be easily added to the respective layer to be considered using the additional parameter “kernel\_regularizer=tf.keras.regularizers.l1” or “kernel\_regularizer=tf.keras.regularizers.l2”.

So far, we have only looked at adaptations to the network itself, with the aim of addressing its overfitting, and have disregarded an important mechanism affecting the training data used. Data augmentation is a sophisticated method of generating more training data by artificial means, without the need for new data. This involves changing the training data using various methods, such as rotations, shifts or noise. The following figures show a number of data augmentation variants applied to an element of the MNIST dataset (l.—r.: original, translation, rotation, contrast).

Graphical user interface, website

Description automatically generated

Even small variations in the training data can have a major impact on a model’s tendency toward overfit. For the application of data augmentation to a dataset, an ImageDataGenerator can simply be created, which is parameterized with the required augmentations—in this example with a rotation range of 20°, a vertical and horizontal translation range of 20%, and horizontal mirroring.

1. data\_aug = tf.keras.preprocessing.image.ImageDataGenerator(
2. rotation\_range=20,
3. width\_shift\_range=0.2,
4. height\_shift\_range=0.2,
5. horizontal\_flip=True)

This augmentation object must then be parameterized with the dataset so that the manipulations can be applied to its elements. The “fit\_generator” method must then be applied to the model instead of “fit” as the training data are only regenerated during the runtime.

1. data\_aug.fit(image\_train)
2. model.fit\_generator(data\_aug.flow(image\_train, label\_train),
3. epochs=50,validation\_data=(image\_test,label\_test),
4. callbacks=[tensorboard\_callback ])

In most cases, it is possible to gain a few percentage points in validation accuracy by such means.

### Self-Check Questions

1. Which of the following terms does not represent a regularization method?

* Dropout
* L1 regularization
* L2 regularization
* *Data augmentation*

1. Which of the following operations belong to the data augmentation method?

* *Rotation*
* *Noise*
* Dropout
* *Translation*

1. How can the overfitting of a statistical model be recognized from the cost function?

If the value of the training cost function decreases further while the value of the validation cost function increases again.

|  |
| --- |
| Summary |
| While linear classifiers are particularly suitable for data that meet the criterion of linear separability, they quickly reach their limits when the data do not. Concatenating multiple linear classifiers does not change their performance, as this also results in a new linear classifier. The introduction of non-linearities in the form of activation functions represents the solution to this, allowing neural networks to be defined mathematically. Important representatives of activation functions are the sigmoid, ReLu, and hyperbolic tangent functions, with the ReLu function having the ability to eliminate the vanishing gradient problem.  In general terms, it is theoretically possible to approximate any function using a neural network with a hidden layer with a sufficiently high number of neurons, as stated by the theorem of universal function approximation. However, training such a network quickly becomes inefficient.  The layers of neural networks can either be implemented manually in Tensorflow by simulating the mathematical operations with Tensorflow operations or integrated directly via the high-level interface Keras. This makes it very easy to define models and configure them using the required parameters. There are various modalities available for saving and loading models. On the one hand, it is possible to save entire models, including hyperparameters, information on the optimizer, and the cost function, via SaveModel. On the other hand, learnable parameters only can also be stored via checkpoints. Callbacks can be used to perform a backup at certain times, for example, when a new minimum of the cost function is reached.  Longer model training does not always result in better performance but can very quickly lead to overfitting. In such cases, generalization ceases; the model simply commits the training data to memory and cannot be applied to new, unknown data. The point of overfitting can be readily determined from the cost function curves for the training and validation, which is marked by a renewed rise in the validation error after a minimum has been reached. Problem-solving approaches to this include L1 and L2 regularization, the dropout method, and data augmentation. |

# Unit 7—Convolutional Neural Networks

Study goals

After completing this unit, you will know ...

… what convolutional neural networks are and what they can be used for.

… how convolutional neural networks are structured and what mathematical operations they are subject to.

… how convolutional neural networks can be used for text and image analysis.

# 7. Convolutional Neural Networks

### Introduction

In this unit, we will look at the development of convolutional neural networks, which currently represent the state of the art for most text and image analyses. We present a number of application areas and explain the motivations behind this approach. A general overview of the functionality, including important mathematical operations, is provided, which helps to lay a secure foundation for further work.

Following the introduction to the subject, two key areas of application—text and image analysis using convolutional neural networks—will be presented together with the implementation of suitable networks.

## 7.1 Motivation and Areas of Application

For relatively simple and low-dimensional input data, very good results can be achieved with multilayer neural networks in terms of both classification and regression problems. However, if the complexity of the data to be processed increases, the use of such networks ceases to be practical. One extreme disadvantage associated with multilayer neural networks is that the number of parameters to be trained increases dramatically, resulting in a vast memory requirement. This high dimensionality of the network is highly inefficient, as it contains a large amount of redundant information. In addition, a large number of model parameters can very quickly lead to the problem of overfitting.

A further problem associated with the use of multilayer neural networks is the manner in which the input data is applied to them. These must take the form of a vector, i.e., matrix-like data must first be transformed into a column vector. This renders the processing of spatial information in the data impossible, which is particularly important in computer vision applications, for example, in the classification of objects in images, when the position of an object can vary significantly within an image. Convolutional neural networks, or CNNs for short, represent the current state of the art for tasks of this kind. In comparison with multilayer neural networks, far fewer parameters are required for training, which is because small, convolution matrices are moved over the input of the network and the inputs of the neurons are calculated using an operation known as discrete convolution. CNNs are particularly suitable for feature extraction within data. This means that they find the key features for the specific problem within the training by autonomous means, which is not achievable with multilayer neural networks.

These advantages of CNNs have made them the current standard solution for a variety of complex problems, which are either unsolvable or can only be solved to an insufficient degree using classic image processing methods or multilayer neural networks. One key area of application is face recognition, for example, in which faces can be extracted and recognized, i.e., classified, within image data. This technique is used in many smartphones, for example, for the purpose of unlocking. Numerous popular apps, such as Snapchat or FaceApp, also use convolutional neural networks.

In addition to a wide range of image data applications, CNNs also come into their own in text recognition, text classification, and even text generation. For example, they can be used to conduct mood analyses, i.e., assessing the feelings that a text should convey. Many online platforms and social networks also use the technology to identify hate speech or other undesirable content, for example. Chatbots, or virtual chat partners, also work using a combination of text recognition and text generation by analyzing incoming messages and generating appropriate answers.

In the following sections, we will explain the structure and functionality of convolutional neural networks and develop implementations for image and text analysis. Naturally, due to the extensive nature of this topic, it cannot be covered in full in just one unit, and so only a brief introduction is provided. For detailed information, please consult the course literature.

### Self-Check Questions

1. Name two areas of application in which convolutional neural networks achieve particularly good results.

For example, computer vision and computational linguistics

1. Do the input data have to be transformed into a column vector in advance for processing by CNNs?

No

## 7.2. Structure

In this section, we will describe the structure of convolutional neural networks, discussing mathematical operations in the process. Essentially, different types of layers are used here, with the functionality resulting from their mutual combination. The first operation applied to the input data is discrete convolution, using a convolution matrix within convolutional layers. In mathematical terms, convolution refers to the combination of two functions to create a new function. In other words, two information sources are linked together. With reference to CNNs, convolution matrices, also known as filters or kernels, are used to move over the input data, and the matrix products between these and the corresponding image areas are calculated and processed further. This results in feature maps. The process is shown in the figure below for the purposes of illustration.

Diagram

Description automatically generated with medium confidence

In green we see a sample filter, which performs the calculation first in the upper left-hand corner of an input matrix. The entry in the feature map is the product of the calculation of:

A picture containing diagram

Description automatically generatedThe filter now moves one step to the side and recalculates the entry of the feature map.

The entire input matrix is processed in this way until the feature map is completely filled. Numerous different filters are used to extract a multitude of features from the data. In the case of image data, for example, it must also be ensured that they consist of multiple channels if color images are involved (red, green, blue channels). Accordingly, the number of filters needed per feature extraction will be the same as the number of channels present. Activation functions are applied to the outputs of the neurons of a conventional layer, as equivalent to the multilayer neural networks, in order to generate a non-linearity. The ReLu function is largely used for this purpose.

As standard, a convolutional layer is followed by a pooling layer, in which information is removed from the feature maps by reducing their dimension. Various approaches are available here, with max pooling being the most frequently used variant. Here, only the maximum value from an area of the feature map is used for further processing. The size of the area is also defined by a filter matrix. The procedure of max pooling is shown on a sample feature map in the following figure.

Diagram

Description automatically generated with medium confidence

The max pooling with a area, therefore, performs the calculation

with the index i representing the upper left entry of the area under consideration. As a result, only the activation of the neuron with the most pronounced response is maintained and the size of the data is reduced, leading to an increased processing rate. This makes it possible to create very deep networks, as the number of parameters does not rise as dramatically as in multilayer neural networks. Another less effective pooling variant is average pooling, i.e., the averaging of an image area.

**Receptive field**

In the receptive field of sensory receptors, information is passed to a single neuron.

Concatenating several blocks of convolutional layers and pooling layers results in extracted features of increasing complexity. In the analysis of image data, for example, the first layers extract basic features such as edges or corners and the rear parts of objects to be classified. This functional principle is heavily inspired by biology. It has been demonstrated, for example, that the **receptive fields** of the visual cortex, a part of the visual system located in the brains of mammals, work in accordance with this principle (Hubel/Wiesel 1962).

The last layers of a CNN correspond to the fully connected layers of multilayer perceptrons, particularly in the case of classification problems. They are used to assign the previously extracted features to the corresponding classes. An alternative is available in the form of global average pooling, in which the overall averages of the last feature maps are calculated and classified using the Softmax classifier. This completely eliminates the need to use fully connected layers, which in turn can save memory and reduce the risk of overfitting (Lin/Chen/Yan 2013). The following representation shows the overall structure of a sample convolutional neural network for analyzing image data:

Diagram

Description automatically generated

When using fully connected layers for classification, it must be ensured that the last feature maps are flattened, i.e., that a column vector is generated from them so that it can be processed.

### Self-Check Questions

1. What are convolutional layers used for? Which mathematical operations are they based on?

Convolutional layers are used to extract features from data. They perform discrete convolution and generate feature maps.

1. Name two pooling variants that you are familiar with.

Max pooling, average pooling

1. Please mark the correct answer(s) with a cross.

* Pooling layers are used to extract features.
* *Pooling layers are used to reduce dimensionality.*
* *Convolutional layers are used to extract features.*
* Convolutional layers are used to reduce dimensionality.

## 7.3 CNNs for Text Analysis

Our aim within this section is to use a convolutional neural network for the analysis of text files. To this end, a Tensorflow implementation will be derived, which will enable the correct classification of texts based on previously learned patterns. This method of Natural Language Processing (NLP) has played an important role in recent years. For the purposes of our example, we will focus on **mood analysis** here, which involves classifying the attitude expressed in a text into a positive or negative mood. We will use a standard dataset of IMDB (Internet Movie Database) comments, which can be loaded directly from the Tensorflow environment and contains 50,000 different film comments, with 25,000 intended in each case for training and testing. As the number of positive and negative data is balanced here, we also refer to this as a balanced dataset. 88,585 different words are used in total. We can load the dataset using the Keras submodule “keras.datasets.imdb”, which is built into Tensorflow, and split this into training and test data accordingly.

**Mood analysis**

The term sentiment analysis is also frequently used.

1. **import** tensorflow as tf
2. **from** tensorflow **import** keras
3. imdb = keras.datasets.imdb
4. (train\_data, train\_labels), (test\_data, test\_labels) = imdb.load\_data()

We also divide the training portion into a training part and a validation part, in order to leave the test portion completely untouched during the training and to use it for the subsequent evaluation only.

1. val\_data = train\_data[:10000]
2. train\_data = train\_data[10000:]
3. val\_labels = train\_labels[:10000]
4. train\_labels = train\_labels[10000:]

In order to obtain an overview of the format of the underlying dataset, we can display an element of it by addressing the corresponding position via an index.

1. **print**(train\_data[0])
2. """[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 2
3. 56, 5, 25, 100, 43, 838, 112, 50, 670, 22665, 9, 35, 480, 284, 5, 150, 4, 172, 1
4. 12, 167, 21631, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192,
5. 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43,
6. 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 19193, 5, 62, 3
7. 86, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16,
8. 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52,
9. 5, 14, 407, 16, 82, 10311, 8, 4, 107, 117, 5952, 15, 256, 4, 31050, 7, 3766, 5,
10. 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 12118, 1029, 13, 104, 88, 4,
11. 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134,
12. 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 1
13. 6, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345, 19, 178,
14. 32]"""

What does this list of numbers have to do with a text? The underlying dataset is already preprocessed, meaning that it is already encoded in such a way that it can be processed using a convolutional neural network. The first important step in text analysis is to find a suitable coding for the data, as otherwise the entire process may fail if the representation is incorrect. In this case, a unique number is assigned to each word and serves to represent it. The numbers-words dictionary can be accessed via the “get\_word\_index()” method of the imdb object. Within this dictionary, the object—in this case, the encoded number—can be searched for using a key—in this case, a word. In order to decode our sample element, we must reverse the dictionary so that we arrive at the respective word object via a number key. This is easily achievable using the following code.

1. word\_number = imdb.get\_word\_index()
2. # First indices must be preserved
3. word\_number = {k:(j+3) **for** k,j **in** word\_number.items()}
4. number\_word = {word\_number[i]:i **for** i **in** word\_number}

We can now implement a help function, which will return us to the original text for an encoded data element.

1. **def** decode(code):
2. text=""
3. **for** i **in** code:
4. text=text+" "+number\_word.get(i, '?')
5. **return** text
6. """? this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert redford's is an amazing actor and now the same being director norman's father came from the same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for retail and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been good and this definitely was also congratulations to the two little boy's that played the part's of norman and paul they were just brilliant children are often left out of the praising list i think because the stars that play them all grown up are such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all that was shared with us all"""

The “?” at the beginning of the data element represents the start sequence. Now that we have gained an overview of the underlying data and the applied encoding, we can now take a look at the architecture design of our neural network. A second important point following the suitable encoding is that the input data must be equal in size. In this case, the number of words in the respective elements of the dataset, which varies, is the key point. We solve this problem by taking the length of the longest data element and bringing all other elements to this length by padding them with zeros at the end. This process is known as padding and can be applied directly to the training, validation, and test data using the Keras preprocessing function “keras.preprocessing.sequence.pad\_sequences”, whilst taking into account the maximum element length of 256 words.

1. train\_data = keras.preprocessing.sequence.pad\_sequences(
2. train\_data,value=0,padding='post',maxlen=256)
3. val\_data = keras.preprocessing.sequence.pad\_sequences(
4. val\_data,value=0,padding='post',maxlen=256)
5. test\_data = keras.preprocessing.sequence.pad\_sequences(
6. test\_data,value=0,padding='post',maxlen=256)

Now that have made the input data a uniform size, we can start to define our model. To this end, we use Tensorflow’s Keras interface for a simple architecture description. The first layer of our network is an embedding layer. In simple terms, this layer projects the words into a continuous vector space, whereby the position of a word within this vector is learned on the basis of the training data and is dependent on the surrounding words. As a result, semantically related words are projected into equal areas of the vector space during the training. The learned position is referred to as embedding. The embedding layer receives the desired batch size and number of occurring words—in this case 88,585—as parameters, in order to generate the associated vector space accordingly. We create a sequential cell model, i.e., a concatenation of different layers, and define an embedding layer as the first layer using the following code.

1. model=tf.keras.Sequential()
2. model.add(tf.keras.layers.Embedding(88585,16))

We then perform the feature extraction from the generated vector space using a convolutional layer, following by an averaging of the resulting feature map using **global average pooling**. We use 512 different convolution filters with a size of 3. The resulting vector of each element of the dataset is then added to a fully connected layer with 16 neurons, which is networked with the output layer consisting of an output neuron. Classification takes place using the sigmoid function in the output layer. For all other activations, the ReLu function is used. The following code shows the implementation of the entire model:

**Global average pooling**

In global average pooling, the average of an entire feature map is calculated, as opposed to average pooling with a specific window size.

1. model = tf.keras.Sequential()
2. model.add(tf.keras.layers.Embedding(88585,16))
3. model.add(tf.keras.layers.Conv1D(512,3,activation='relu'))
4. model.add(tf.keras.layers.GlobalAveragePooling1D())
5. model.add(tf.keras.layers.Dense(units=16,activation=tf.nn.relu))
6. model.add(tf.keras.layers.Dense(units=1,activation=tf.nn.sigmoid))

As the cost function for the binary classification, we use binary cross entropy, and as the optimizer we again use the Adam optimizer (Kingma/Ba 2017). We will also be interested in the classification accuracy later and will configure our model accordingly.

1. model.compile(optimizer='adam',
2. loss='binary\_crossentropy',
3. metrics=['accuracy'])

We can now train our model by running the training for 30 epochs with a batch size of 256 elements.

1. model\_train = model.fit(train\_data,
2. train\_labels,
3. epochs=50,
4. batch\_size=256),
5. validation\_data=(val\_data, val\_labels),
6. verbose=1)

With the configuration used, we achieve a validation accuracy of approximately 87%. We then test our training model on the untouched test dataset.

1. model.evaluate(test\_data, test\_labels)

Here, we achieve an accuracy of around 86%. It is, therefore, already possible to carry out text data classifications using a relatively simple convolutional neural network if a few key points relating to data coding are observed. An improved network architecture can increase the accuracy on the test dataset, for example, by concatenating several blocks of convolutional and pooling layers to achieve a high-level feature extraction. For a more detailed insight into this complex subject area, please consult the relevant literature (Ganegedara 2018).

### Self-Check Questions

1. What is an embedding layer used for?

With the aid of an embedding layer, data are projected into a vector space, in which related data are located in the same areas.

1. What is essential for the processing of text data using CNNs?

The data must be encoded in order that they can be processed. Furthermore, they must be of equal length (e.g., have the same number of words).

1. What name is given to a dataset that contains the same number elements for each class?

Balanced dataset

## 7.4 CNNs for Image Analysis

Shape

Description automatically generatedOne of the main application areas of convolutional neural networks is in the field of computer vision and, in association with this, in areas including object detection, object classification, object localization and **segmentation** in image data. The differences between the terms are illustrated in the following figures.

**Segmentation**

Segmentation generally refers to breaking down a whole into individual parts.

Classification is solely concerned with classifying the content of an image, i.e., which objects this image contains. With localization, the position of the object within the image is determined. In the case of images with several objects, the term “object detection” is used when the objects contained within the image are classified and their position is determined. Segmentation describes the determination of the areas of an image that belong to the objects in question.

Our aim within this section is to develop a convolutional neural network that enables image data to be classified. To this end, we will use the standard dataset Cifar10, which contains 60,000 color images, divided into 50,000 pieces of training data and 10,000 pieces of test data from ten object classes in a 32 x 32 pixel format (Krizhevsky/Hinton 2009). The classes included are planes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. This dataset can also be loaded directly into the Tensorflow environment via the Keras dataset module.

1. **import** tensorflow as tf
2. **from** tensorflow **import** keras
3. cifar = keras.datasets.cifar10
4. (train\_data, train\_labels), (test\_data, test\_labels) = cifar.load\_data()

We normalize the pixel values between 0 and 1 and divide the training part again into a training and a validation dataset in a ratio of three to two.

1. train\_data, test\_data = train\_data/255.0, test\_data/255.0
2. val\_data = train\_data[:20000]
3. train\_data = train\_data[20000:]
4. val\_labels = train\_labels[:20000]
5. train\_labels = train\_labels[20000:]

Once we have prepared our input data accordingly, we can now start defining the network architecture. To this end, we first define a sequential model and create a convolutional layer as the first layer, the input dimension of which corresponds to the size of the image data and the number of color channels, i.e., 32 x 32 x 3. First, we select a number of 32 filters with a size of 3 x 3 pixels. As the image data are, in spatial terms, a two-dimensional representation, the layer “Conv2D” is now required. As an activation function, we use the ReLu function for all hidden layers. A max pooling layer with a kernel of 2 x 2 follows this convolutional layer, which reduces the feature map generated. We then add two more convolutional layers, each with 64 feature maps, in order to be able to extract “higher-level” features, and add a classifier head to the network, which carries out the actual classification of the extracted features. The prior flattening of the two-dimensional features into a vector is important here so that the fully connected layers can be processed. The following code shows the definition of the model:

1. model = tf.keras.models.Sequential()
2. model.add(tf.keras.layers.Conv2D(32, (3, 3),
3. activation='relu', input\_shape=(32, 32, 3)))
4. model.add(tf.keras.layers.MaxPooling2D((2, 2)))
5. model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
6. model.add(tf.keras.layers.MaxPooling2D((2, 2)))
7. model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
8. model.add(tf.keras.layers.Flatten())
9. model.add(tf.keras.layers.Dense(64, activation='relu'))
10. model.add(tf.keras.layers.Dense(10))

As a multi-class classification problem is involved, we require a corresponding cost function, with the “sparse categorical cross entropy” cost function being the appropriate choice. Using the following lines of code, we configure our model with the appropriate parameters and start the training over ten epochs:

1. model.compile(optimizer='adam',
2. loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True), metrics=['accuracy'])
3. model\_training = model.fit(train\_data, train\_labels, epochs=10,
4. validation\_data=(val\_data, val\_labels))

We achieve a validation accuracy of approximately 65.9% here, with a subsequent evaluation conducted on the basis of the test dataset resulting in an accuracy of approximately 66.5%. This value does not seem particularly high, which raises the question of whether it cannot also be achieved by a multilayer perceptron. To this end, we implement a fully connected neural network with four hidden layers in order to be able to compare the performance with that of the CNN.

1. model = tf.keras.models.Sequential()
2. model.add(tf.keras.layers.Flatten())
3. model.add(tf.keras.layers.Dense(128, activation='relu'))
4. model.add(tf.keras.layers.Dense(128, activation='relu'))
5. model.add(tf.keras.layers.Dense(32, activation='relu'))
6. model.add(tf.keras.layers.Dense(10))

Here, we achieve an accuracy of approximately 46% on the test dataset. If we look at the number of parameters to be trained, which can be displayed using the “summary()” method of the model object, this gives values of 414.314 for the multilayer perceptron and 93.248 for the convolutional neural networks, which is four times lower. The addition of further fully connected layers brings about a rapid increase in the number of parameters, which is not the case with convolutional layers. For this reason, there will be a very rapid tendency toward overfitting when a multilayer perceptron is expanded. Our aim within this section is to present the functionality of a CNN and show its application only, and, therefore, not to make any further adjustments to the model to achieve a better classification accuracy.

What about using CNNs for other image analyses, such as object localization? In the case of classification, classes, i.e., discrete values, are predicted, whereas with localization, continuous values are output. If we take the example of bounding boxes, which frame objects in the smallest possible rectangle, four numbers are predicted, namely the x and y position of the box and its length and width. This is, therefore, a regression of the bounding box parameters. The expansion to multiple objects within an image and the associated regression of multiple bounding boxes is known as object detection. Prominent architectures of object detection networks include Faster RCNN, YOLO, and SSD (Ren et al. 2015; Redmon et al. 2016; Liu et al. 2016).

The segmentation of image data is an even more complex task. Here, no bounding boxes are regressed, but the global context of the image scene is analyzed and the areas of the objects within the image are estimated. Semantic segmentation is an extreme variant in which each individual image pixel is classified. On the basis of this, an attempt is made to gain a complete understanding of the scene. In the context of training, **masks** are created for all objects, which define their contours with pixel accuracy—a process that is extremely time-consuming. A well-known and frequently used network architecture for semantic segmentation is the RCNN mask (He et al. 2017).

**Masks**

The marking of segments within images is carried out using masks, which are used as ground truth data in training.

### Self-Check Questions

1. What are convolutional layers used for in image analysis?

Convolutional layers carry out feature extraction. Their complexity increases with increasing depth.

1. Please describe the semantic segmentation method.

Through semantic segmentation, each pixel of an image is assigned a specific class to which it belongs. The aim of this is to completely separate all the components of an image from each other and thereby gain a complete understanding of the scene.

1. Which of the following methods is used to determine the location of objects within image data?

* Segmentation
* Classification
* *Localization*
* Semantic segmentation

|  |
| --- |
| Summary |
| Fully connected neural networks reach their limits at a very rapid rate in the presence of more complex data. One extreme disadvantage of this architecture is the rapidly increasing number of model parameters to be trained, which can quickly cause overfitting problems. The architecture can also experience difficulties when objects are situated in different locations within images, as the input data are applied to the network as a column vector and, as a result, the data’s position within it is fixed. The use of convolutional neural networks (CNNs) provides a solution to this. They perform discrete convolution operations and determine feature maps, which contain features extracted from the data. If several convolutional layers are used in succession, increasingly complex features will be extracted. The convolution operations are followed by pooling layers, which reduce the dimension.  CNNs can be used for many purposes, including problems in computational linguistics and computer vision. It is always important to encode the input and output data appropriately first, so that they can actually be processed. For example, embedding layers can be used to project text data into a vector space, in which words that frequently occur together are located in similar areas. Convolutional layers can be used to extract the features from the vector spaces, which are then classified using a classification head.  There are a variety of problems connected with image analysis, including object classification, object localization, object detection, and segmentation. In contrast to classification, no discrete numbers, i.e., classes, are predicted in localization. Rather, continuous numbers are output. This is, therefore, a regression problem. Segmentation is concerned with the exact areas of the objects in images, with semantic segmentation representing the extreme case by assigning a specific object class to each image pixel. The aim of this is to gain a complete understanding of the scene. |