Master's Thesis

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyzing the Information Content of Multiple Views of an Object in Object Detection with Neural Networks

Dennis Kraus

Electrical Engineering and Information Technology

10.06.2019

Supervisor:
Sebastian Schrom, M.Sc.

REGELUNGSMETHODEN UND ROBOTIK
Prof. Dr.-Ing. J. Adamy

Eidesstattliche Erklärung

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Dennis Kraus, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.
Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Darmstadt, 10.06.2019

Dennis Kraus

(English translation of above declaration for information purposes only)

Thesis Statement
pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Dennis Kraus, have written the submitted Master's Thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§38 paragraph 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.
In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Abstract

english abstract

Zusammenfassung

deutscher abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

1.2 Motivation

Chapter 2

Related Work

Chapter 3

Fundamentals

This chapter covers the fundamentals necessary to understand the methods presented and their application. It is divided into a section on neural networks and one on the used software and frameworks. The former starts with the principle of a neural network. It continues with an explanation of the network architectures multilayer perceptrons and convolutional neural networks. The first one serves as an example how networks work in general. The second one is more suited for image processing and outperforms the first one in this task. It continues with an explanation of the required steps to train the network for achieving the wanted use case. Finally, methods and parameters are examined that improve the overall performance of a neural network. The latter explains which software and frameworks support building and training a model.

3.1 Artificial Neural Networks

This section examines the types of neural networks that are important for this work. Furthermore, it explains how these types are build and trained in order to achieve the wanted use case.

3.1.1 Overview

Artificial neural networks are vaguely inspired by the biological neural networks that constitute animal brains for recognizing patterns. Its task is being an universal approximator for any unknown function $f(x) = y$ where x is the input and y the output. There are two conditions that need to be fulfilled. One is the relation of x and y and the other is the presence of numerical data. So every data like images, text or time series must be translated. The complexity of the approximated function depends on the use case but usually it is highly non-linear. General use cases for neural networks embrace classification, clustering and regression.

Classification means the network divides given data like images into categories by recognizing patterns. This is the task used in this work. The correct category of each input is given as an additional label. Therefore, the network learns the correlation between data and labels. Kind of an downside here is that every input must be labeled

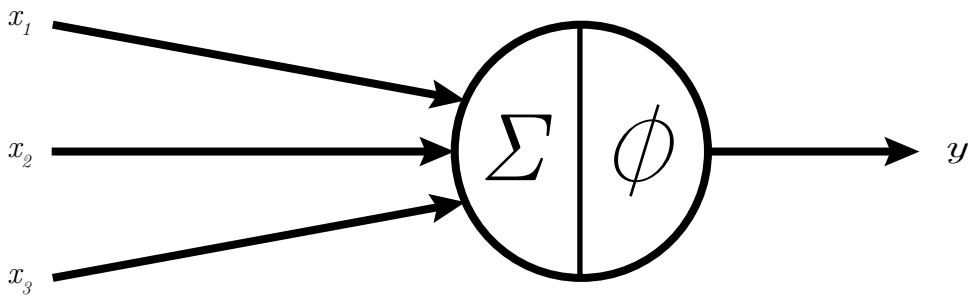


Figure 3.1: Model of a neuron. The inputs x_i are summed up and put into the activation function ϕ whose result is the neuron's output y.

by human knowledge beforehand. This kind of learning is called supervised learning, because each predicted category by the network needs to be compared with the ground truth label. Use cases are for example the classification of cars in images or even the type of car in an image or whether an email is spam. Again, it all depends on the wanted use case and given data.

Clustering divides data into clusters or groups, respectively, but without requiring labels. Therefore, this learning type is called unsupervised learning. So it is kind of an classification task with a dynamic category creation. Use cases are comparing data to each other and finding similarities or anomalies. Because unlabeled data occurs way more often than labeled data in real wold examples, a network can train on a broader range of related data and probably gets more accurate than a classification one.

Regression is the prediction of a future event by establishing correlations between past events and future events. A simple use case is the prediction of a price of a house given its size and the size-price data pairs of different houses. A more advanced use case is the prediction of hardware breakdowns by establishing correlations of already known data.

3.1.2 Multilayer Perceptron

This section starts with an explanation of a single computational neuron and its development to become a perceptron and ends with an overview of the multilayer perceptron architecture.

3.1.2.1 Pereptron

McCulloch and Pitts[1] were the first who defined a computational model of a neuron that corresponds to the functionality of one in neurobiology. This neuron has several logical inputs which can either be true or false and a logical output. Therefore, this neuron works as a linear classifier separating two categories where only one is the correct one. This is called binary classification. A schematic of this model can be seen in Fig. 3.2.

Table 3.1: Truth Tables of Logical Operations

(a) Logical AND

x_1	x_2	x_3	Thresh	Output
0	0		2	0
0	1		2	0
1	0		2	0
1	1		2	1
0	1	1	3	0
1	1	1	3	1

(b) Logical OR

x_1	x_2	x_3	Thresh	Output
0	0		1	0
0	1		1	1
1	0		1	1
1	1		1	1
0	0	0	1	0
0	0	1	1	1

(c) Logical XOR

x_1	x_2	x_3	Output
0	0		0
0	1		1
1	0		1
1	1		0
0	1	1	0
1	1	1	1

(d) Logical XNOR

x_1	x_2	x_3	Output
0	0		1
0	1		0
1	0		0
1	1		1
0	1	1	0
1	1	1	1

Because numerical values are required for later operations, every logical value is transformed to 0 if it is false or 1 if it is true. After summing up the inputs, a threshold activation function is applied. In a mathematical sense

$$y = \phi\left(\sum_{i}^{n} x_i\right) \tag{3.1}$$

describes this operation where n is the amount of inputs, x_i a single input and ϕ the used activation function, in this case the threshold activation function. Plots of several activation functions including their equations are shown in Fig. 3.10. That means, if a given threshold is reached the output of the perceptron is 1 and 0 otherwise. This corresponds to the neurobiological spike of a neuron. The truth tables for the logical AND and OR operations are shown in Fig. 3.2a and Fig. 3.2b, respectively. The first one outputs true if all inputs are true. The second one outputs true if at least one input is true. It can be seen, that by changing the threshold value a different logical operation can be represented. For two inputs a threshold of 2 represents the logical AND operation. If the number of inputs are changed, a related change of the threshold still models the same operation. In this example the number of inputs is changed to 3 whereas the threshold must be changed to 3. The same procedure is valid for the logical OR operation. But here the threshold value is always 1 regardless of the amount of inputs and therefore differs to other logical operations. One limitation of this neuron type is its inability to represent exclusive logical operations like XOR and XNOR[2], whose truth

tables are shown in Fig. 3.2c and Fig. 3.2d, respectively. The first one outputs true if the inputs are different. The second one outputs true if all inputs are identical which equals an inverted XOR operation. It can easily be seen, that no threshold value can be found for meeting the requirements. For example, in the first case if the threshold is at 1 the first three combinations of inputs could be covered. If all inputs are false the threshold value is not reached and therefore the neuron outputs a 0. If the inputs differ from each other, the threshold value is reached which outputs a 1. But as soon as the fourth one needs to be classified the threshold value is no longer valid. The sum of the inputs equals two which exceeds the threshold and would output a logical true like in the OR case. But according to the truth table a logical false should be outputted. The reasons for this misbehavior are the neuron's limitation to binary values and a threshold activation function and therefore the classification of only two categories.

Donald Hebb states "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."[3] on how neurons learn. This means, that if a neuron repeatedly and persistently stimulates a immediately subsequent neuron, i.e. the more often two wired neurons are active, their synaptic efficacy increases. This is known as the Hebbian Theory. Hebb summarizes this with his famous quote "neurons that fire together, wire together".

Frank Rosenblatt developed the first perceptron[4]. Considering the Hebbian Theory the original McCulloch-Pitts-Neuron needs to be modified by adding associated weights for the inputs in order to simulate the strength of a synapse. Thus, Theorem 3.1 changes to

$$y = \phi\left(\sum_{i}^{n} x_i \cdot w_i\right) \tag{3.2}$$

by considering the weights w_i. Furthermore, the perceptron allows the usage of real-valued inputs and weights and uses the Heaviside step function as the activation function. According to Fig. 3.10b the Heaviside function outputs 0 if its parameter is negative and 1 otherwise. Therefore, its difference to the threshold activation function is just an offset of the threshold or bias, respectively. Adapting Fig. 3.1 to this results in Fig. **??**. There is an input with the value 1 which is weighted by the bias. Thus, when multiplied representing the bias. This result is part of the weighted sum of the inputs which is fed to an activation function whose result is the output of the perceptron. Combining this with Theorem 3.2 yields

$$y = \phi\left(x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n + b\right) \tag{3.3}$$

$$= \phi\left(\left(\sum_{i}^{n} x_i \cdot w_i\right) + b\right) \tag{3.4}$$

where x_i are the inputs, w_i the weights, b the bias and ϕ the Heaviside activation function.

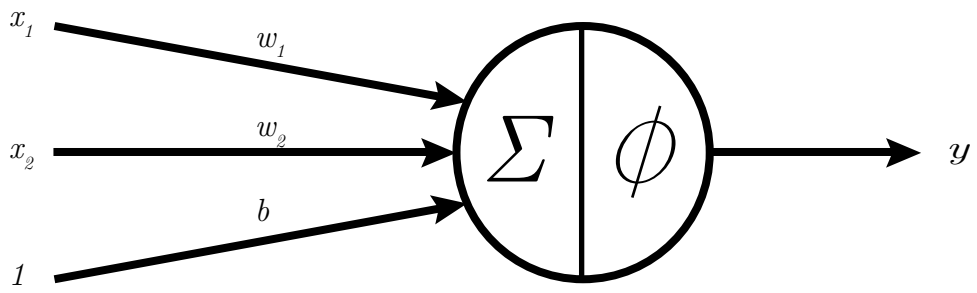


Figure 3.2: Model of a perceptron. The inputs x_i are weighted by w_i and are summed up with the bias b. This sum is put into the activation function ϕ whose result is the perceptron's output y.

Let's write the inputs and weights as vectors of

$$\boldsymbol{x} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}^T \tag{3.5}$$

$$\boldsymbol{w} = \begin{pmatrix} w_1 & w_2 & \cdots & w_n \end{pmatrix}^T \tag{3.6}$$

for simplicity. Inserting this in Theorem 3.3 results in

$$y = \phi\left(\boldsymbol{x} \cdot \boldsymbol{w^T} + b\right) \tag{3.7}$$

with the same parameters as before. However, this model still works as a linear classifier and thus is unable to represent logical exclusive functions. This can be solved by concatenating multiple perceptrons and building a multilayer artificial network.

3.1.2.2 Multilayer Perceptron

A multilayer perceptron consists of multiple perceptrons divided in layers and solves complex tasks[5]. It is an universal approximator for every function[6] regardless of the activation functions used[7]. Because of the multiple layers and the non-linear activation functions non-linearity is introduced into the network. Thus, it can distinguish data that is not linearly separable as most real world data is.

There are at least three layers. Each layer contains several perceptrons that are not connected to each other. However, every perceptron is connected to every perceptron of its subsequent layer. This type of connection is called fully-connected network. Because the data flow within the network is only in one direction and does not contain circles, the architecture is called feedforward neural network. A visualization of this is shown in Fig. 3.3. Although the weights and biases are not displayed for clarity, they still exist and follow the same principle as with a single perceptron. Like the single perceptrons every node in the networks still holds its activation, a single numerical value. In this kind of network architecture perceptrons are often referred to as nodes.

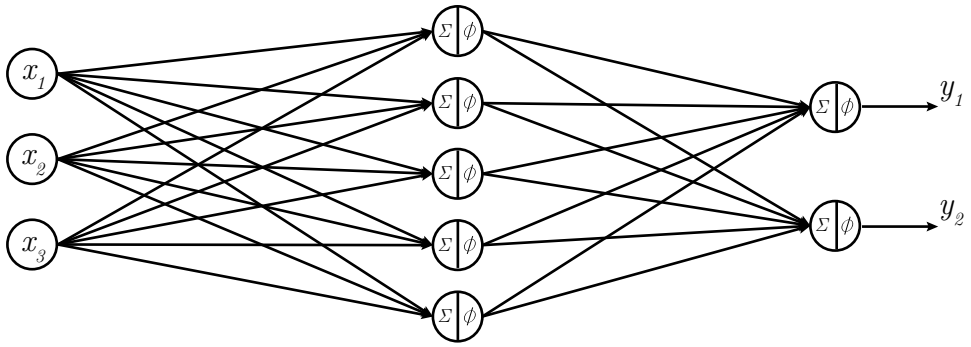


Figure 3.3: Multilayer perceptron with three layers. Each layer consists of multiple perceptrons. The input layer transfers real world data into the network. The output layer makes the computations of the network interpretable for humans. The layers inbetween, the hidden layers, perform calculations and feed forward the network data. Each connection between nodes has a weight, that is not displayed for clarity. Also every node has its own bias.

The input layer serves as an interface for the data. It does not perform any calculations and just passes the data to the next layer. The number of nodes in this layer depends on the data and how it can be divided. If the real world data is an image, for example, the number of nodes should be equal to the number of pixels, so that every node can hold the intensity value of one pixel.

The output layer is responsible for transferring the network data to the outside so that it can be interpreted and worked with. The number of nodes in this layer depends on the expected results. If kinds of animals need to be detected in an image, every output node would represent a single kind or category, respectively. Let's say there are three kinds of animals possible, then there need to be three output nodes. In theory the node representing the correct kind of animal holds a one and every other a zero, if the values are squashed within this range.

Every layer between the input and the output layer is a hidden layer. They have no direct connection to the outside, neither to the input nor the output, hence, their name. Their task is to transfer the input information to the output by performing calculations. With at least one hidden layer every continuous function can be approximated. So, the network models the function

$$f(\boldsymbol{x}) = \boldsymbol{y} \tag{3.8}$$

where

$$\boldsymbol{x} = (x_0, x_1, \cdots, x_n) \tag{3.9}$$

$$\boldsymbol{y} = (y_0, y_1, \cdots, y_m) \tag{3.10}$$

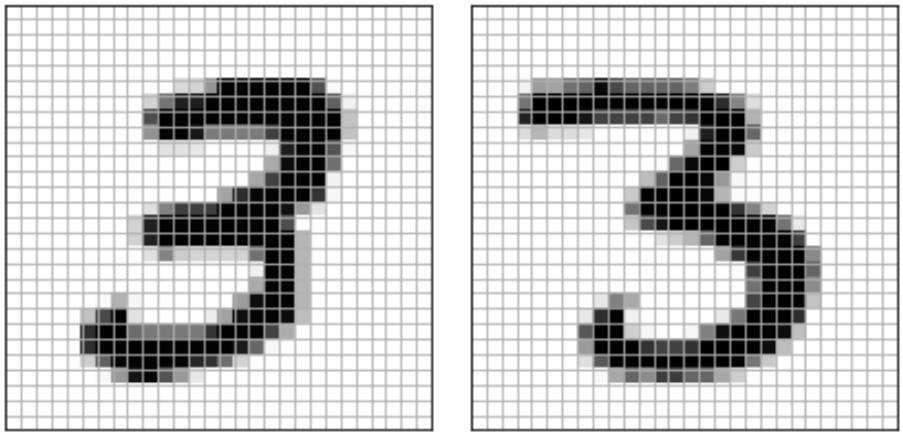


Figure 3.4: Handwritten Digits from the MNIST Digit Dataset. Represented as a 28×28 pixel matrix. Each cell represents a pixel.

are the input vector with n elements and output vector with m elements, respectively. Every element represents the activation of a neuron in the input layer or output layer, respectively.

Let's take again an image as an example. The task is to classify a handwritten digit from the MNIST dataset[8]. The digit can be seen in Fig. 3.4. Each grid cell represents a pixel. Now it is evidently that the intensity data of every pixel is relevant for classifying the digit. Thus, every pixel needs an associated node in the input layer of the network. This real world data is transferred into the network by flattening the intensity values of the image matrix to a vector. Therefore, the vector contains $28 \cdot 28 = 784$ elements which equals the number of input nodes. With the correct weights and biases the network knows which nodes are active for a certain digit. This means, that if in another image more or less the same pixels or nodes, respectively, have high intensities or activations, the same number needs to be classified. The downside of the flattening is, that every relation of pixels like position is lost, which means a loss in overall information. The consequence of this is, that if a digit has no similar position and shape like the digits the networks knows, the classification fails. If, for example, a digit the network classified correctly is not centered anymore and downscaled to take up only have its original size, completely different neurons are active. And, thus, the network cannot find any correlation to the original image or its knowledge of how digits look and returns a wrong classification result. Another severe downside is the huge number of parameters. If the image gets larger, the number of input neurons naturally needs to be adapted. Due to

this, additional weights and biases are introduced to the network, because every input neuron is connected to every node of its subsequent layer. This extends the finding of the optimal weights and biases and needs plenty of resources. A better solution is provided by convolutional neural networks that are covered in Section 3.1.3.

3.1.3 Convolutional Neural Networks

Convolutional neural networks are suited for image processing tasks, because they perform better than the multilayer perceptrons architecture regarding accuracy and the number of parameters[8][9]. The reason for the first one is most likely that they are invariant regarding the position of an object within an image. Convolutional neural networks do not have a as strict separation in multiple layers as multilayer perceptrons do. They rather have a pool of several layers which can be arbitrarily connected to fulfill one's needs. Combinations of these layers and their hyperparameters is called a architecture. There are different proposed architectures with their weights and biases available. The most common ones are AlexNet, VGG, GoogleNet and ResNet.

3.1.3.1 Convolutional Layer

The most important layer is the convolutional layer. As the name suggests it performs a convolution with a filter matrix of arbitrary size on an input matrix of arbitrary size. Let's say the input matrix is an image $\boldsymbol{X} \in \mathbb{R}^{m \times n}$. The filter matrix $\boldsymbol{K} \in \mathbb{R}^{i \times j}$ contains $i \cdot j$ weights of the network. The filter is now slid across the image and performs a dot product within its window. Fig. 3.5 illustrates the following operation. In reference to the figure the kernel covers the four elements in the top right corner of the input image. Hence, the dot product multiplication for this setup yields $5 \cdot 1 + 4 \cdot -1 + 1 \cdot -1 + 3 \cdot 1 = 3$. This result is stored in a new matrix at its corresponding place. At the end, this matrix will hold all values of the convolution operation. After each calculation of the dot product, the filter matrix moves. The corresponding step size is called stride. A stride of 1 moves the filter one pixel or element, respectively. There can be a different stride along the x-axis and y-axis. When the filter has moved across the whole input, the resulting matrix is completely filled up like the one in the figure. This resulting matrix is called a feature map, because the convolution extracted features from its input. Doing this operation the feature map is always smaller than the input. Sometimes this is not desirable, because this means a loss in information. If multiple convolutions are performed, the feature map constantly shrinks until no feature can be extracted anymore or only rough ones. Thus, a padding p can be applied to the input. This means surrounding the input with p rounds of zeros like it is illustrated in Fig. 3.6 with $p = 1$. The convolution operates like usual, just on a larger input. In practical terms, there are two common conventions for convolutions: valid and same. The former defines that no padding is applied and therefore a kind of valid convolution is performed, because only the real input is minded. This means that the feature map has a size of

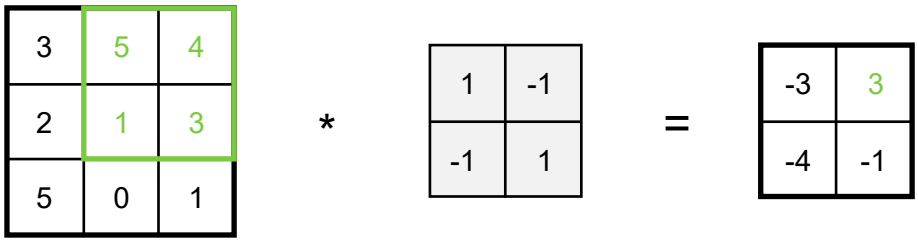


Figure 3.5: Convolution of an image and a kernel. The 2×2 kernel is sled across the 3×3 image and performs a dot product multiplication within its window each time. Here, the kernel moves with a stride of 1, which results in the shown image on the right, the so called feature map.

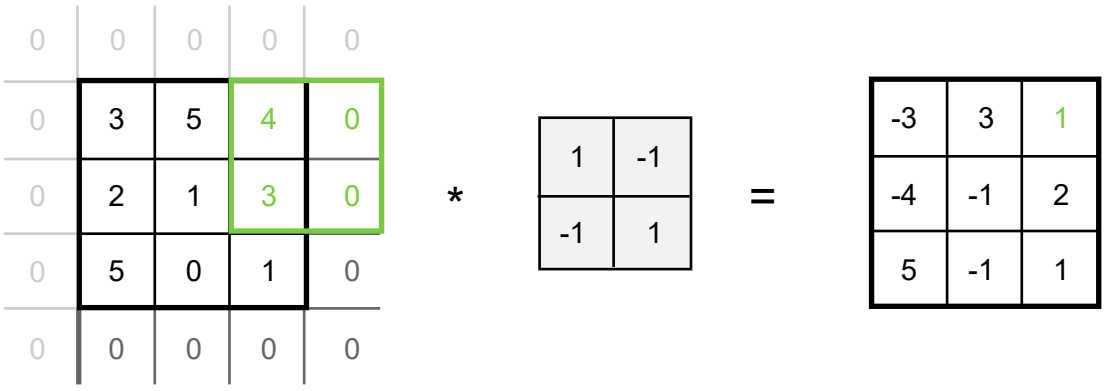


Figure 3.6: Convolution of a padded image with a 2×2 kernel. The 3×3 image is surrounded with zeros for inducing its size to the feature map. However, the convolution operates like usual, just on a larger input. If the amount padding is odd, a padding at the right and at the bottom is preferred.

$\boldsymbol{F} \in \mathbb{R}^{m-i+1 \times n-j+1}$. The latter means, that the size of the feature map equals the one of the input. How much padding p needs to be applied can be calculated by comparing each matrix shapes and using

$$m = m + 2p - i + 1 \tag{3.11}$$

$$p = \frac{i-1}{2} \tag{3.12}$$

as a general equation. However, this only covers the padding height. If the image or filter are not symmetrical, the padding along the width needs to be calculated as well by replacing m with n and i with j. Another remark is, that in computer vision filter sizes usually are odd. There can be two reasons for this. First, the filter has a center which helps to tell where exactly the filter points to. Second, the padding p is even. Otherwise, it needs to be rounded up for a correct mathematical representation, but only used on two sides of the image like it is shown in the figure with the help of transparency. Only the padding at the right and at the bottom are taken into account for creating a filter matrix with the same shape of the original input. For inputs with more than one channel, a convolution is performed almost identically. But instead of a kernel with a depth of one it is extended to a depth that matches the input. Then a common dot product multiplication is calculated for every input channel with its corresponding filter channel. This results in a matrix with the depth of the input and filter. Finally, the resulting depth channels are summed up element wise which results in a matrix with depth one. For the case of an RGB image, that is an image with three channels representing the colors red, green and blue, a filter would have a depth of three and a final convolution result would always have a depth of one. Usually, at the end of a convolutional layer a bias addition is performed for simulating the neurobiological spike of a neuron. This result is put into an activation function like the ones from Fig. 3.10 for introducing non-linearities into the network. Both the methods and their purposes are analog to the ones known from multilayer perceptron networks.

The kernel in the figure would find top-left to bottom-right diagonal lines, because those are the pixels weighted most. Like this but with slightly larger kernels and different weights more complex features can be found. It is also possible to perform multiple different convolutions on the same input for finding different features. They are stored as a matrix, where the number of different features represents the depth of the feature map. This whole process solves the limitation to a fixed position of features of the multilayer perceptrons architecture. Even if, for example, a digit covers only have the image, all features are found, because the kernel is moved over the image and not single neurons or filters are responsible for single pixels. Furthermore, because features are found by a moving filter, convolutional neural networks need way less weights and biases due to the possibility of reusing them. The accuracy compared to multilayer perceptron networks is improved by concatenating several convolutions. That means a convolution is performed on the feature map of an earlier convolution. First, rough features like edges are found, and the deeper it gets into the network, the finer the features get.

3.1.3.2 Pooling Layer

After obtaining features using a convolutional layer a pooling layer can be inserted. Pooling serves as a sample-based discretization process. This is done by moving a filter with an arbitrary stride over the input that compresses the information or values, respectively, within its window. The result is a reduction of the inputs spatial dimensions. However, the depth stays the same. The objective of this process is a gain in computational performance, because there is less spatial information. Hence, fewer weights and biases are needed which in turn improves training time. Another advantage is a less chance for overfitting due to the loss of spatial information. Moreover, the extracted features are translation invariant. This means, that features can be found even when they underwent a small positional displacement The reason for this is, that within the pooling window the found translated feature is still more important than another one For example, the network classifies perfectly round zeros by looking for four quarter circles as features next to each other. Though, the input is a wider zero and, thus, the network finds small lines as features between the quarter circle features. If the pooling window now checks one of these quarter circle features extended with a line feature, still the first feature is used for further calculations because it is more important. Fig. 3.7 illustrates the pooling process for a max pooling operation in practical terms. A max pooling filter yields the maximum within its window as the result. Moving such a 2×2 filter over the 4×4 input with a stride of 2 yields a matrix with each maximum at its corresponding position. The maximum of the red colored 2×2 window is 7, hence, this number comes up in the result. The other windows are processed identically. The size of the result of an arbitrary pooling operation with an input of size $w_1 \times h_1 \times d_1$ and a filter of size $f \times f$ is

$$w_2 = \frac{w_1 - f}{s} + 1 \tag{3.13}$$

$$h_2 = \frac{h_1 - f}{s} + 1 \tag{3.14}$$

$$d_2 = d_1 \tag{3.15}$$

where w is the number the rows, h the number of columns, d the depth and s the stride. As it can be seen, pooling layers do not have learnable parameters only hyperparameters. There are only two common variations of hyperparameters. Either $f = 3$ and $s = 2$, which models a overlapped pooling operations, because some values are part of different windows, or $f = 2$ and $s = 2$. Larger filters and strides take away too much information. Another pooling type is mean pooling. Hereby, the result of each window is the average of all its values instead of the largest one. However, its results are outperformed by the max pooling[10].

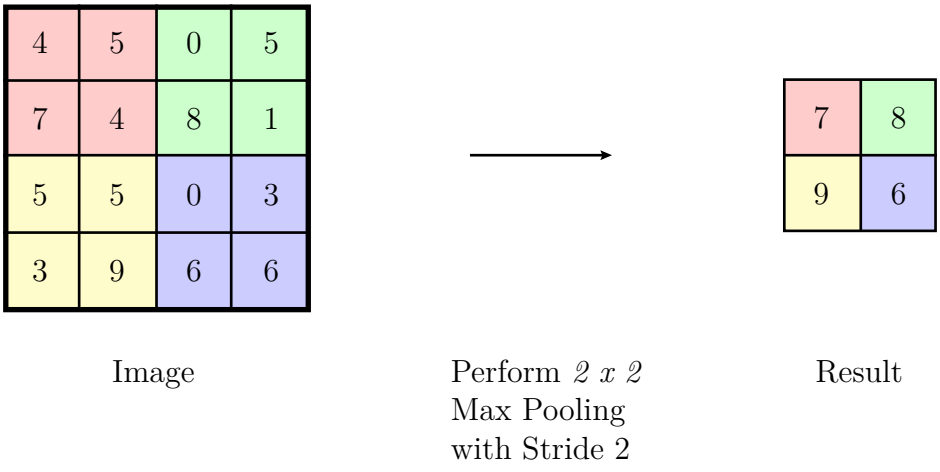


Figure 3.7: Max pooling with 2×2 filter across a 4×4 input with a stride of 2. Within each window the maximum of its values is computated. Finally, this yiels a matrix with each maximum at its corresponding position.

3.1.3.3 Fully-Connected Layer

The last part of a convolutional neural network consists of at least one fully connected layer. This is identical to Fig. 3.3 but instead of directly inputting real world values the output or activations, respectively, of a the previous convolutional or pooling layer is used. Every edge has weights and every node has a bias and outputs the result of an activation function. Describing this in a mathematical sense can be done with Theorem 3.7. The objective is to combine several features that were detected and use them as attributes for classifying the input. For example, if four legs and a long nose are found, there is a dog in the image and not a cat. The interpretation of the output is simplified by applying an additional softmax function that squashes the output into a range of 0 and 1, whereas the sum of all outputs equals 1, to represent percentages of confidence or a probability distribution, respectively[11]. The softmax function can be written as

$$S(y_i) = \frac{\exp(y_i)}{\sum_j \exp(y_j)} \tag{3.16}$$

where y are the outputs, i an index of an output and j a moving index over all outputs. This exploits the knowledge of mutually exclusive classes, i.e. that only one class is correct. Otherwise, for multi-label classification, a sigmoid function can be used, that squashes each output of the network into a range of 0 and 1.

3.1.4 Training

Thus far optimal weights and biases were assumed in all examples. But in practical terms they need to be found first. This starts by generating and preparing a dataset from which

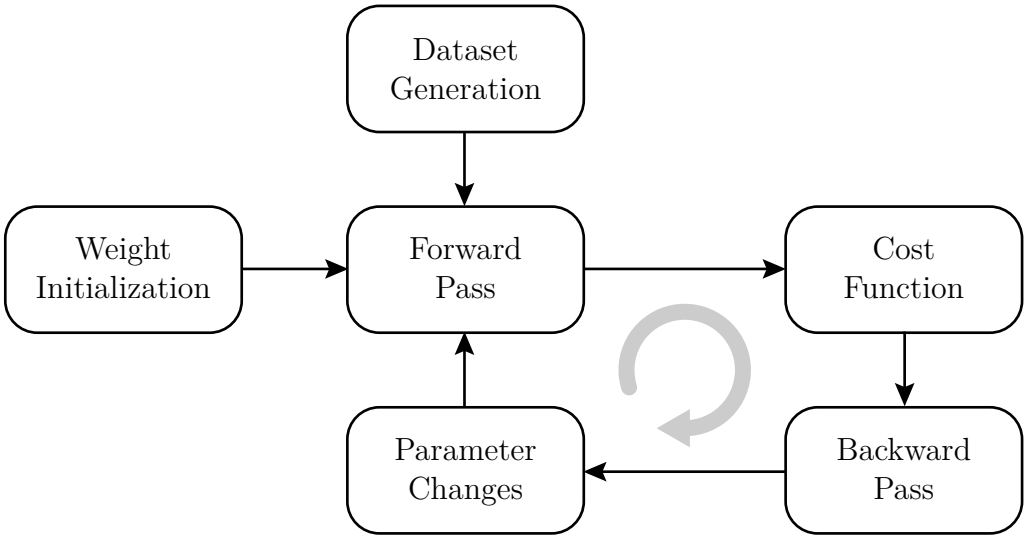


Figure 3.8: Training Process

the network can find correlations by changing the weights and biases. First, these are initialized. Then, a input is feed-forwarded through the network. This classification is put into a cost function. The result is back-propagated through the network by computing its gradients for changing the weight and biases. The forward pass and backward pass are repeated until an exit condition is satisfied. Fig. 3.8 illustrates this process. Each of these steps is covered in the following sections.

3.1.4.1 Dataset Generation

The whole training process is based on the dataset from which the network learns the correlations of input and label. Usually, a dataset consists of input-label pairs, where the input is the data that is fed into the network and the label is the ground-truth. In the case of a classification task, the label represents the category. However, there is no general rule for the amount of data. It can be said, that more data is better for generalization, but too many samples can lead to overfit the network to the shown data. The latter means, that the network is trained to long or to intensive on the shown data. The consequence is, that it adjusts its weights and biases to classify this data perfectly, but cannot reliably classify general, unknown data of same objects anymore, because they slighty differ. Basically, the amount of data depends on the objective of the network. For classifying whether an image is black or white, only a few training samples would be needed. Is the objective classifying objects within images, it depends on the number of possible objects and their complexity. If the objects are simple geometric shapes, then not as many samples are needed as if the objects are common objects like type of animals or cars. For the latter the number of samples would probably be

approximately 1000 examples per class. Luckily, there are several datasets available, that are already sorted and labeled, like the MNIST handwritten digits or the ImageNet dataset. Available datasets are not limited to images, but can contain CAD models like the ModelNet dataset which is used for this network architecture.

Samples of a dataset are split into a training and testing set, and sometimes a validation set[12]. The first contains data, the network trains on. From this data correlations of input and label are found. After arbitrary training steps where parameter changes happened, the performance of the network is tested on the validation set. This is data, the network is not trained on. The objective is to check if overfitting occurs. If the accuracy of the training set increases, the accuracy of the validation set has to increase as well. This shows that the network still learns and gets better. If the accuracy of the training set increases, but the accuracy of the validation set stays the same or decreases, it is an indicator for overfitting. The testing set is data the network is not trained on as well. It serves as a final performance check of the network to confirm its general accuracy. If no validation set is available, the testing set can be used. How the dataset is split depends again on the number and complexity of samples and the objective. However, a equal distribution of samples in each set should be minded. Otherwise the performance will not be satisfying. This means, if the network trains mostly on sweatshirts a test set with mostly pants would not yield an acceptable accuracy, because the network does not know these particular features.

For processing the dataset, a one-hot encoding[13] of the labels is recommended. Usually, the labels are categorical data. This means, they contain label or string values, respectively, instead of numeric values, that the networks needs. For example, there is a fashion variable with the values "boot", "sweatshirt" and "pants". The network would not know how to interpret these. Thus, these values need to be converted to numeric values. Furthermore, if these label values are outputs of the network, it should be easily possible to convert them back from numeric values. Hence, they are converted to integers that represent a category. Referring to the example, this results in the numeric values 0, 1 and 2 for the labels "boot", "sweatshirt" and "pants", respectively. But numeric values have a natural ordered relationship between each other, that neural networks could exploit. The index of "pants" is higher than the one of "boot", but neither of these categories is better or worse than the other. Therefore, the indices are one-hot encoded as well. This means removing the integer representation and inserting binary variables for simulating existing features. Applying this to the example results in the feature label vector $\boldsymbol{f}_1 = (0, 1, 0)^T$ for the "sweatshirt" label. This vector has a length of the number of different categories available, where every element is 0 except the one of the corresponding category which is 1. Table 3.3 summarizes this approach.

3.1.4.2 Weight Initialization

Before the actual training starts, the parameters, the weights and biases, of the network need to be initialized. If this is done right, i.e. the values are in a range that supports

Table 3.3: One-hot encoding of categorical data. First, categorical label values are transformed to numeric values representing a category index. Then, this is replaced with binary variables to represent features, that removes the natural relationship of numeric values to each other. This vector has a length of the number of different categories, where every element is 0 except for the corresponding category which is 1.

Categorical	Integer	One-Hot
"Boot"	0	$\boldsymbol{f}_0 = (1,0,0)^T$
"Sweatshirt"	1	$\boldsymbol{f}_1 = (0,1,0)^T$
"Pants"	2	$\boldsymbol{f}_2 = (0,0,1)^T$

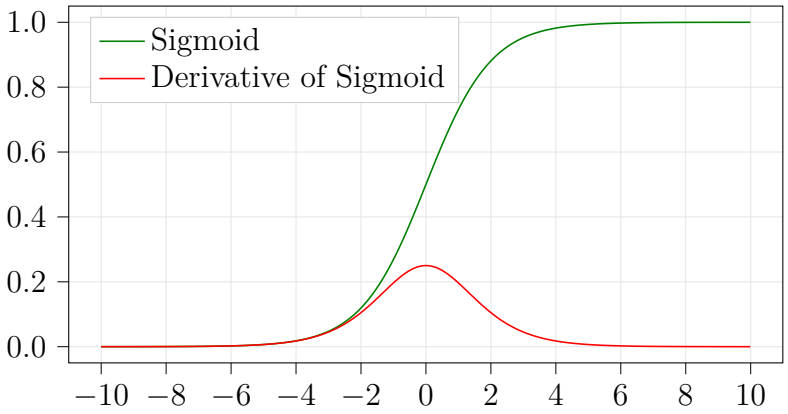


Figure 3.9: Sigmoid Function and its Derivative

training, optimization will be achieved in lesser or least time. In the other case, a converging to optimal values can be impossible. Reasons for this are the exploding or vanishing of gradients during backpropagation[14]. In the backward-pass the gradients are computed for every layer and are passed from end to beginning using the chain rule. For example, the derivative of the sigmoid function as it can be seen in Fig. 3.9 is in the range of $(0, 0.25]$. If this is multiplied several times, the gradients at the beginning are way smaller than at the end. If the weights are too small or too large, this effect is intensified. This is partly true for other activation functions like the ReLU as well. But here the gradients can become very large too, if the weights are really large. None of these scenarios is desirable, because the optimal weights are either not reached or skipped. This will become more clear in Section 3.1.4.3 when the expressions of backpropagation are presented.

If the weights are initialized with 0, every neuron would compute the same output. This leads to an identical gradient for each one and therefore identical parameter updates. All in all, this would reduce the network to a linear one. Hence, a common initialization approach is using a Gaussian distribution like $N(\mu, \sigma^2) = N(0, 0.01)$. However, this way the variance of this distribution of each neuron's output grows with the number

of its inputs. Therefore, a normalization of the variance of each neuron's output to 1 is performed. This is done by scaling its weights by the square root of its number of inputs. This can be derived with the n inputs $\boldsymbol{x}$ and weights $\boldsymbol{w}$ by

$$\begin{aligned} Var(z) &= Var\left(\sum_i^n w_i x_i\right) \\ &= \sum_i^n Var\left(w_i x_i\right) \\ &= \sum_i^n \left[E(w_i)^2\right] Var(x_i) + E\left[(x_i)\right]^2 Var(w_i) + Var(x_i)Var(w_i) \\ &= \sum_i^n Var(x_i)Var(w_i) \\ &= (nVar(\boldsymbol{w}))Var(\boldsymbol{x}) \end{aligned}$$

where zero mean inputs and weights are assumed and an identically distribution of all w_i and x_i. Now, z needs to have the same variance as all of its inputs $\boldsymbol{x}$, which yields $Var(w) = 1/n$ as every weights variance. Hence,

$$\boldsymbol{w} = \frac{N(0,1)}{\sqrt{n}} \tag{3.17}$$

initializes the weights. This is mostly universal, but must be used for tanh activation functions. A similar analysis is done by *Glorot and Bengio* [15] whose recommendation is

$$Var(\boldsymbol{w}) = \frac{2}{n_{in} + n_{out}}$$

where n_{in} and n_{out} is the number of neurons in the incoming and outgoing layer, respectively. Their motivation is, that by doing the earlier variance calculations for the backpropagated signal, it turns out that

$$Var(\boldsymbol{w}) = \frac{1}{n_{out}} \tag{3.18}$$

is needed for keeping the variance of the input and output the same. Because in general the constraint $n_{in} = n_{out}$ is not fulfilled, they make a compromise by taking the average. Though, these initializations are not valid for, for example, ReLU units, due to their positive mean. Fortunately, *He et al.* [16] states the initialization

$$\boldsymbol{w} = N(0,1) \cdot \sqrt{\frac{2}{n}} \tag{3.19}$$

especially for ReLU neurons.

3.1.4.3 Stochastic Gradient Descent

Let's recall the information, that are needed for the actual training step, i.e. the finding of optimal weights and biases. Everything starts with a dataset $\mathbb{D}$ containing m pairs of inputs and corresponding labels. Performing a one-hot encoding on the labels yields by assuming in general flattened input matrices

$$\mathbb{D} = \begin{pmatrix} \boldsymbol{X} & \boldsymbol{Y} \end{pmatrix} \tag{3.20}$$

where $\boldsymbol{X} \in \mathbb{R}^{n_x \times m}$ and $\boldsymbol{Y} \in \mathbb{R}^{n_y \times m}$ are representing each input and label as vectors, respectively. Hereby, n_x represents the size of each input and n_y the number of categories. Furthermore, there is a neural network with L layers each containing an arbitrarily number of perceptrons. Expressing the activation of every node with Theorem 3.3 would get very confusing for a whole network. Hence, a matrix notation is the way to go in the long run. First, for every j-th node in the l-th layer its weights are summarized in a vector

$$\boldsymbol{w}_j^{[l]} = \begin{pmatrix} w_{j,0}^{[l]} & w_{j,1}^{[l]} & \cdots & w_{j,n_h^{[l-1]}}^{[l]} \end{pmatrix}^T \tag{3.21}$$

containing single weights, where the superscript in square brackets denotes the layer and the subscript denotes the edge of (target neuron, preceding neuron). The number of hidden neurons in the l-th layer is represented by $n_h^{[l]}$. These conventions are maintained for all parameters for the rest of this thesis. The bias is just a scalar denoted as $b_j^{[l]}$. Now, every weight vector and bias can be combined to a matrix and vector, respectively, for each layer. This yields

$$\boldsymbol{W}^{[l]} = \begin{pmatrix} \boldsymbol{w}_0^{[l]} & \boldsymbol{w}_1^{[l]} & \cdots & \boldsymbol{w}_{n_h^{[l]}}^{[l]} \end{pmatrix}^T \tag{3.22a}$$

$$\boldsymbol{b}^{[l]} = \begin{pmatrix} b_0^{[l]} & b_1^{[l]} & \cdots & b_{n_h^{[l]}}^{[l]} \end{pmatrix}^T \tag{3.22b}$$

where $\boldsymbol{W}^{[l]} \in \mathbb{R}^{n_h^{[l]} \times n_h^{[l-1]}}$ and $\boldsymbol{b}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$.

Using these matrices and vectors data can easily be forwarded through the network by building up on Theorem 3.7. The weighted sum of all neurons of the l-th layer is computed as

$$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]} \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]} \tag{3.23}$$

with the activations vector $\boldsymbol{a}$. Putting this in an activation function yields

$$\boldsymbol{a}^{[l]} = \phi\left(\boldsymbol{z}^{[l]}\right) \tag{3.24}$$

for the l-th layers activations. Performing this for every layer results in

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}^{(i)}) \tag{3.25}$$

Table 3.4: Example comparison of an one-hot encoded ground truth label and a prediction. However, the actual ground truth feature has the second smallest probability in the prediction. Therefore, the parameters need to be adjusted.

Ground-Truth	$\boldsymbol{y} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \end{pmatrix}^T$
Prediction	$\hat{\boldsymbol{y}} = \begin{pmatrix} 0.54 & 0.28 & 0.2 & 0.63 & 0.96 \end{pmatrix}^T$

as the networks prediction, where $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n_y}$ is the i-th data sample. This superscript in round brackets is part of the used convention.

Let's assume that the result is already fed into a sigmoid function and therefore only contains values between 0 and 1. This prediction needs to be compared with the ground-truth label $\boldsymbol{y}^{(i)}$ for checking how good the network performs and, hence, how well the parameters fit. An example of how these vectors can look like is shown in Table 3.4. It can be clearly seen, that the prediction is completely wrong. The actual ground truth feature has the second smallest probability in the prediction. In theory, an identical representation would be desirable. However, in practical terms a slight deviation is normal. Because finding optimal parameters is a optimization problem, a metric for the performance of the networks is served by a loss function that maps these parameters to a cost. The most common loss function for mutually exclusive classes is the cross entropy loss function. It is defined as

$$-\left(y \log(p) + (1-y) \log(1-p)\right) \tag{3.26}$$

for two classes and as

$$H(\boldsymbol{y}, \boldsymbol{p}) = -\sum_{i}^{M} y_i \log(p_i) \tag{3.27}$$

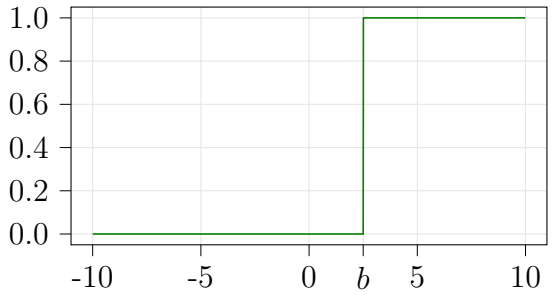
for multiclass classification, where M is the number of classes, i the moving index, $\boldsymbol{y}$ the ground truth vector and $\boldsymbol{p}$ the predicted probabilities. Adapting the previous notation to this yields

$$\boldsymbol{y} = \boldsymbol{Y}^{(i)} \tag{3.28}$$

$$\boldsymbol{p} = \hat{\boldsymbol{y}} \tag{3.29}$$

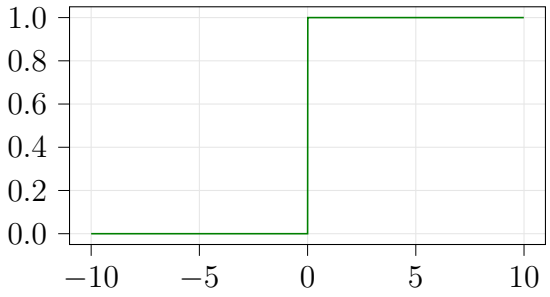
$$M = n_y \tag{3.30}$$

as analogies.



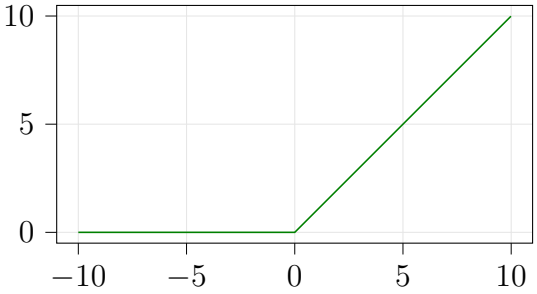
$$\phi(x) = \begin{cases} 1 & x \geq \text{Bias } b \\ 0 & x < \text{Bias } b \end{cases}$$

(a) Threshold



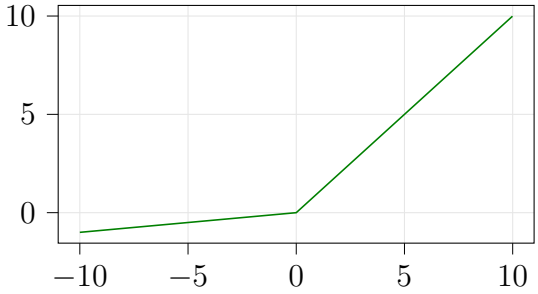
$$\phi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

(b) Heaviside



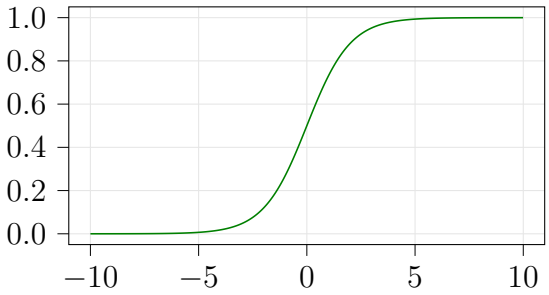
$$\phi(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

(c) Rectified Linear Unit



$$\phi(x) = \begin{cases} x & x \geq 0 \\ x \cdot \text{Slope } s & x < 0 \end{cases}$$

(d) Leaky Rectified Linear Unit



$$\phi(x) = \frac{1}{1 + \exp(-x)}$$

(e) Sigmoid

Figure 3.10: Plots and equations of common used activation functions. Where the Bias b is the threshold value and s adds a small slope. Usually, the latter is very small like $s = 0.01$.

3.1.4.4 Adam Optimization Algorithm

3.1.5 Improving Performance

3.2 Software

This section focuses on explaining which software and frameworks are used for implementing the network and generating the dataset.

Chapter 4

Methods

Chapter 5

Results

Chapter 6

Discussion

6.1 Conclusions

6.2 Outlook

Bibliography

[1] McCulloch, Warren S. ; Pitts, Walter: Neurocomputing: Foundations of Research. Cambridge, MA, USA : MIT Press, 1988. – ISBN 0–262–01097–6, Kapitel A Logical Calculus of the Ideas Immanent in Nervous Activity, S. 15–27

[2] Minsky, Marvin ; Papert, Seymour: *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA : MIT Press, 1969

[3] Hebb, Donald O.: *The Organization of Behavior: A Neuropsychological Theory*. Wiley, 1949 (A Wiley book in clinical psychology). – ISBN 9780471367277

[4] Rosenblatt, Frank: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. In: *Psychological Review* (1958), S. 65–386

[5] Bishop, Christopher M.: *Neural Networks for Pattern Recognition*. New York, NY, USA : Oxford University Press, Inc., 1995. – ISBN 0198538642

[6] Cybenko, G.: Approximation by superpositions of a sigmoidal function. In: *Mathematics of Control, Signals and Systems* 2 (1989), Dec, Nr. 4, 303–314. http://dx.doi.org/10.1007/BF02551274. – DOI 10.1007/BF02551274. – ISSN 1435–568X

[7] Hornik, Kurt: Approximation capabilities of multilayer feedforward networks. In: *Neural Networks* 4 (1991), Nr. 2, S. 251–257

[8] Lecun, Yann ; Bottou, Léon ; Bengio, Yoshua ; Haffner, Patrick: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE*, 1998, S. 2278–2324

[9] LeCun, Yann ; Bengio, Yoshua: The Handbook of Brain Theory and Neural Networks. Version: 1998. http://dl.acm.org/citation.cfm?id=303568.303704. Cambridge, MA, USA : MIT Press, 1998. – ISBN 0–262–51102–9, Kapitel Convolutional Networks for Images, Speech, and Time Series, 255–258

[10] Scherer, Dominik ; Müller, Andreas ; Behnke, Sven: Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In: *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*.

Berlin, Heidelberg : Springer-Verlag, 2010 (ICANN'10). – ISBN 3–642–15824–2, 978–3–642–15824–7, 92–101

[11] Bishop, Christopher M.: *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg : Springer-Verlag, 2006. – ISBN 0387310738

[12] James, Gareth ; Witten, Daniela ; Hastie, Trevor ; Tibshirani, Robert: *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. – ISBN 1461471370, 9781461471370

[13] Harris, David ; Harris, Sarah: *Digital Design and Computer Architecture, Second Edition*. 2nd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2012. – ISBN 0123944244, 9780123944245

[14] Hochreiter, S.: *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München*. 1991

[15] Glorot, Xavier ; Bengio, Yoshua: Understanding the difficulty of training deep feedforward neural networks. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010

[16] He, Kaiming ; Zhang, Shaoqing Xiangyuand R. Xiangyuand Ren ; Sun, Jian: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In: *CoRR* abs/1502.01852 (2015). `http://arxiv.org/abs/1502.01852`