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

Earlier 3D shape descriptors were mostly handcrafted based on a particular geometric property of the actual shape's surface or volume. Those descriptors can be divided into two groups. On the one hand, there are model-based 3D descriptors, that are directly working with the available 3D representation of objects that have been modeled using polygons, voxels, or point clouds, among others. *Osada et al.* [1] finds dissimilarities between sampled distributions of simple shape functions measuring global geometric properties of an object like distances of points. On the other hand, there are view-based 3D shape descriptors. Those are created using multiple 2D views of an object instead of the 3D representation of an object. *Chen et al.* [2] introduced Lighting Field descriptors, the first typical view-based descriptor. This uses an collection of 10 views. *Shu et al.* [3] presented the Principal Thickness Images descriptor, that creates three gray-scale images by encodes the boundary surface and the voxelized components of the actual 3D shape. In general, the advantages of view-based shape descriptors are their low dimensionality compared to model-based ones and, hence, the efficiency for processing. With the introduction of convolutional neural networks like AlexNet [4] and their improvement in image processing, in particular in object detection, they could be used for creating descriptors. This performance was further improved with famous architectures like VGG-Net [5], ResNet [6] and Inception-v4 [7]. According to the ImageNet [8] challenge results the last two architectures outperform humans regarding the top-5 classification error. Using a variation of the VGG architecture, the so-called VGG-M [9], *Su et al.* [10] make their MVCNN network learn a compact shape representation of the actual object by collecting information from any number of input views without a specific order. Previous methods combine the information of views with simple strategies like pairwise comparisons of descriptors or concatenating descriptors from ordered, consistent views. They, however, perform a maximum pooling operation across all views for collecting discriminative features in a single shape descriptor. This leads to an accuracy of 89.9% for a network trained on the ImageNet1k dataset and fine-tuned with the ModelNet40 one with 12 views per object. In contrast, learning a single-view classification with an identical training and fine-tuning yields an accuracy of 88.6%. Here the accuracies of the corresponding 12 views are averaged before the overall accuracy is calculated. Moreover, this outperforms the 3D ShapeNets [11] with an accuracy of 77.3%, which is

using a convolutional network on raw CAD data. Hence, it is pre-trained and fine-tuned with the ModelNet40 dataset. In the comparison of 2D- and 3D-representations from *Su et al.* [12] the MVCNN architecture outperforms architectures working on point clouds and voxels. Furthermore, they could improve to performance of the vanilla MVCNN to 95.0% per instance by using a deeper network and a better object centering. However, *Hegde et al.* introduce their FusionNet [13] that combines a multi-view architecture, in particular [10], with a volumetric convolution neural network for obtaining each representation's advantages. The 2D-representation is used for local spatial correlations, while the 3D-representation is used for long range spatial correlations. Their architecture achieves an accuracy of 93.11% on the ModelNet10 dataset with 20 views and 90.80% on ModelNet40 with 60 views. According to *Feng et al.* [14], view-to-shape descriptor methods like the one from *Su et al.* are a milestone for 3D shape recognition and reflects the state-of-the-art. Because in [10] all views are weighted equally, their goal is to exploit the discriminability among views and their intrinsic hierarchical correlation. Hence, they add a module that divides views with similar features into the same group. Views inside a group are mean pooled for creating a group descriptor for each group. Furthermore, a group with more discriminative views is associated a higher weight than groups with less discriminative views. Finally, a single weighted group descriptor is computed representing the shape descriptor of the object. This yields an accuracy of 92.6% with an identical training and configuration as before and the GoogLeNet or Inception-v1 architecture [15], respectively. Using only 8 views results in an accuracy of 93.1%. With transferring the MVCNN concept to the GoogLeNet architecture, an accuracy of 92.1% is achieved. Another view grouping approach is presented by *Cyr et al.* [16], however, they are using handcrafted descriptors. They define similarity metrics based on curve matching for performing the view grouping. Because views are redundant in a large part they can be reduced to a minimal set due to performance. They introduce the aspects graph representation. The theory behind is, that a small change in the vantage point of an object results in only a small change in the view projection. However, for some views that change is large. Those views represent a transition, the views inbetween an aspect. Hence, it is supposed, that the first describes the object satisfiable.

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. General knowledge is taken from [17] and [18].

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

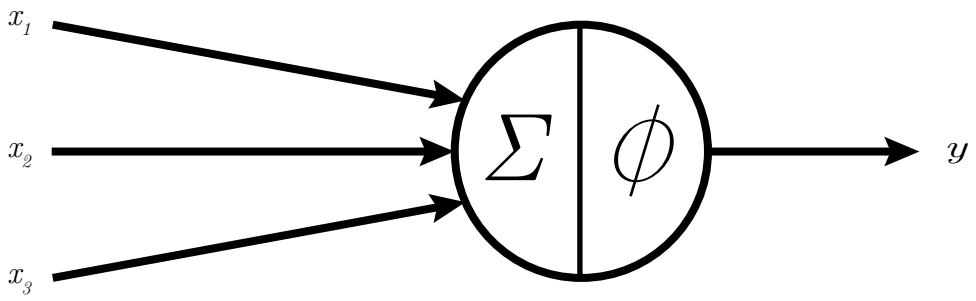


Figure 3.1: Model of a neuron. The inputs x_i are summed up and put into the activation function ϕ whose result is the neuron's output y.

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[19] 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. 3.19. 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[20], 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."[21] 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[22]. 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. 3.19b 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.

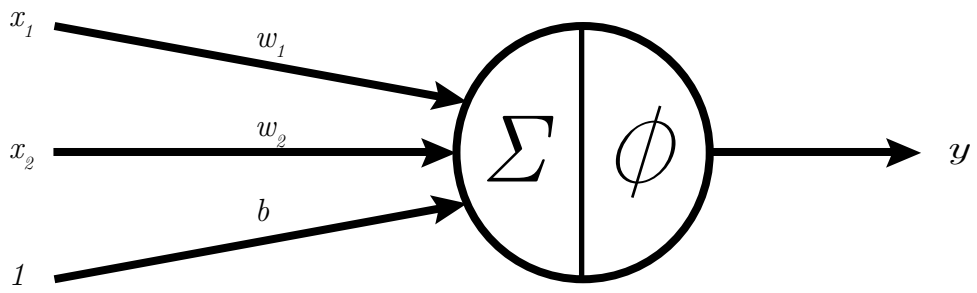


Figure 3.2: Model of a perceptron. The inputs x_i are weighted by w_i and are summed up with the bias b. This sum is put into the activation function ϕ whose result is the perceptron's output y.

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[23]. It is an universal approximator for every function[24] regardless of the activation functions used[25]. 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.

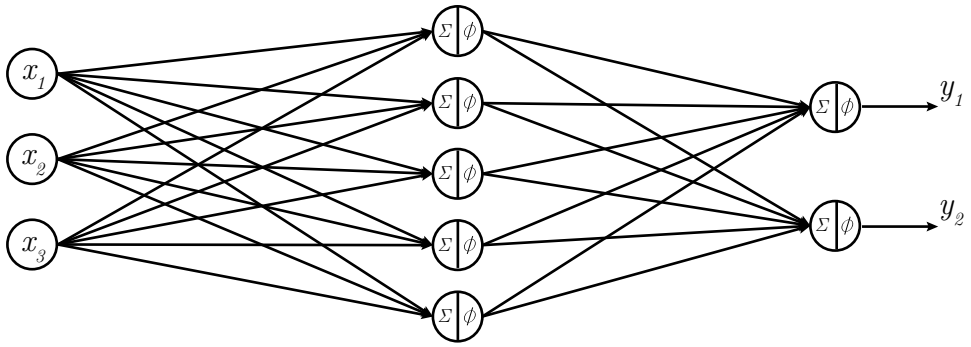


Figure 3.3: Multilayer perceptron with three layers. Each layer consists of multiple perceptrons. The input layer transfers real world data into the network. The output layer makes the computations of the network interpretable for humans. The layers inbetween, the hidden layers, perform calculations 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

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

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

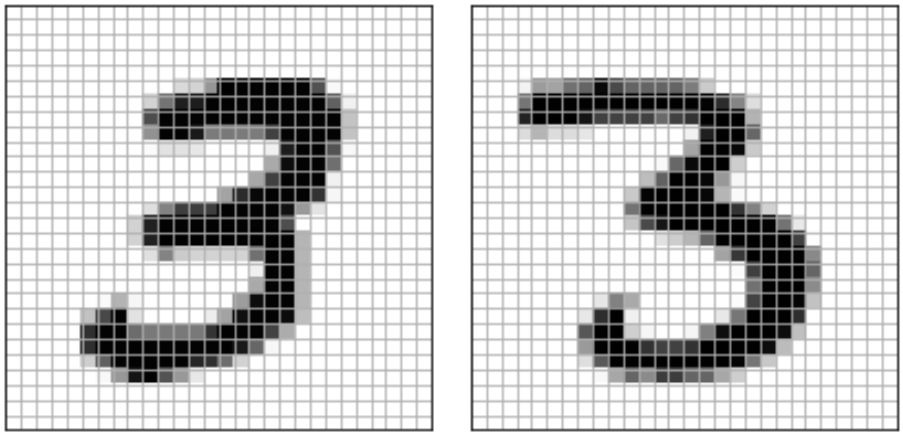


Figure 3.4: Handwritten Digits from the MNIST Digit Dataset. Represented as a 28×28 pixel matrix. Each cell represents a pixel.

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[26]. 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[26][27]. 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

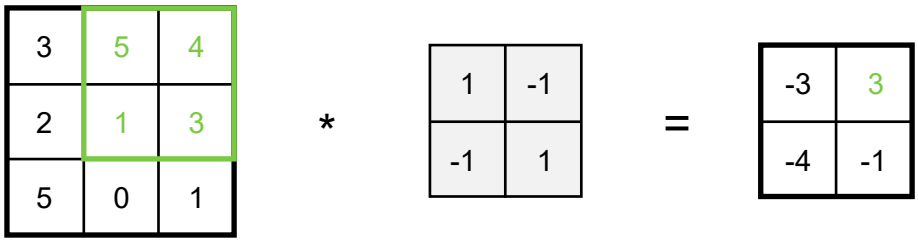


Figure 3.5: Convolution of an image and a kernel. The 2×2 kernel is sled across the 3×3 image and performs a dot product multiplication within its window each time. Here, the kernel moves with a stride of 1, which results in the shown image on the right, the so called feature map.

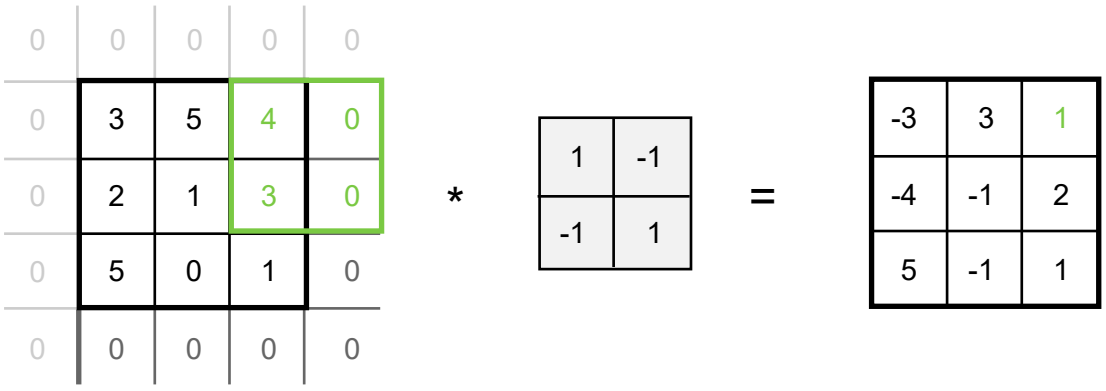


Figure 3.6: Convolution of a padded image with a 2×2 kernel. The 3×3 image is surrounded with zeros for inducing its size to the feature map. However, the convolution operates like usual, just on a larger input. If the amount padding is odd, a padding at the right and at the bottom is preferred.

$\boldsymbol{F} \in \mathbb{R}^{m-i+1 \times n-j+1}$. 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. 3.19 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[28].

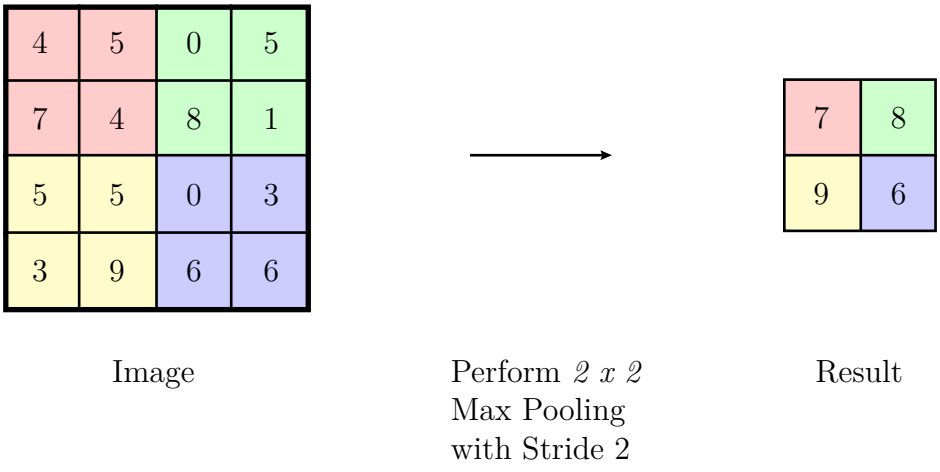


Figure 3.7: Max pooling with 2×2 filter across a 4×4 input with a stride of 2. Within each window the maximum of its values is computated. Finally, this yiels a matrix with each maximum at its corresponding position.

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 the previous convolutional or pooling layer is used. However, fully-connected layers can be stacked, so the outputs of one can be the inputs of another one. Every edge has a weight and every node has a bias. The values flowing over an edge are multiplied with the related weight and summed up. To this weighted sum the bias of the neuron is added and the result is fed into 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. Due to the weights, some attributes are more significant than others. For example, if four legs and a long nose are found, there is a dog in the image and not a cat. If the task is to distinguish only between cats and dogs, the nose feature is weighted more than the leg feature. For preventing overfitting and improving generalization, a fully-connected layer can be combined with the dropout regularization technique[29]. This drops out nodes randomly during training depending on a given probability, i.e. changing their incoming and outgoing weights temporarily to zero. Hence, those nodes are excluded from the current classification and from backpropagation as well. So their weights experience no change, which achieves the desired effect. The interpretation of the outputs of a fully connected layer 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,

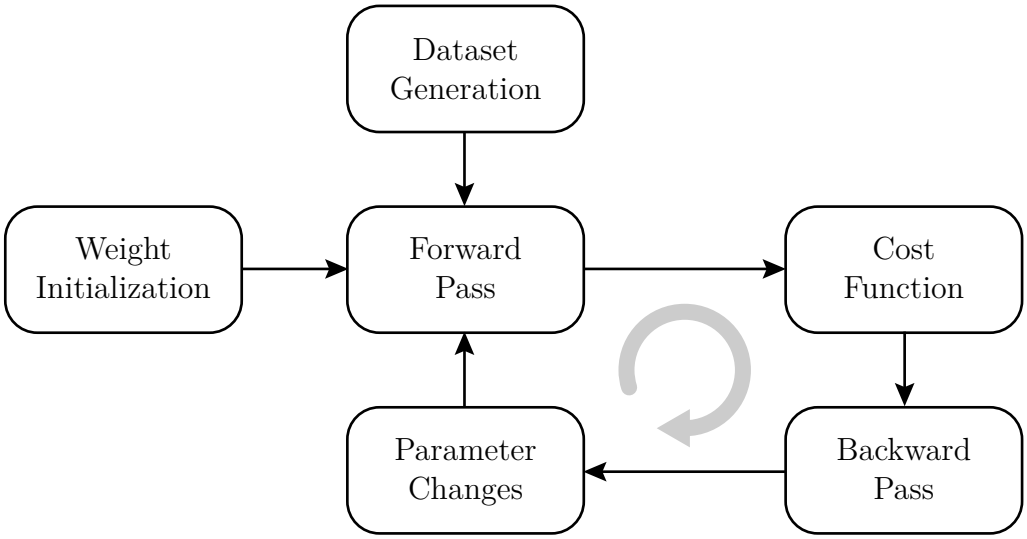


Figure 3.8: Training Process

respectively[30]. 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 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 with different samples 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 slightly 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[31]. 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[32] 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

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	$\boldsymbol{f}_2 = (0,0,1)^T$

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 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[33]. 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 3.1.4.4 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 up-

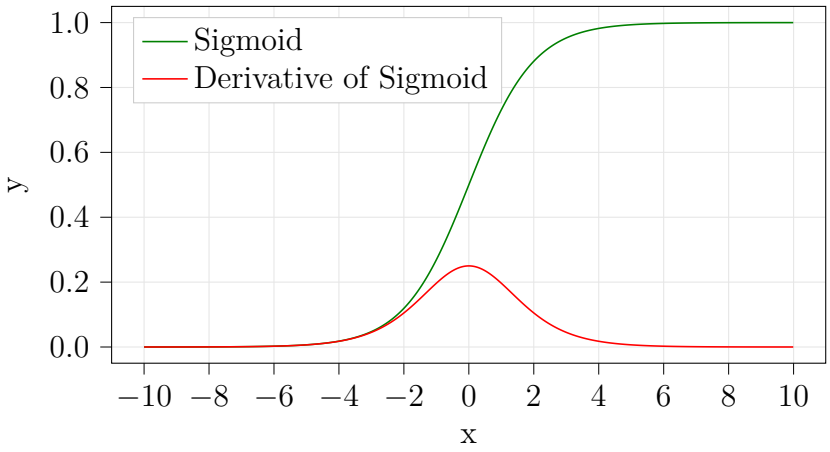


Figure 3.9: Sigmoid Function and its Derivative

dates. 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* [34] 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.* [35] states the initialization

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

especially for ReLU neurons.

3.1.4.3 Feed-Forward Pass

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 and assuming in general flattened input matrices yields

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

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, forming matrices. 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

$$\boldsymbol{w}_j^{[l]} = \begin{pmatrix} w_{j,1}^{[l]} & w_{j,2}^{[l]} & \cdots & w_{j,n_h^{[l-1]}}^{[l]} \end{pmatrix}^T \tag{3.21}$$

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 of the j-th neuron in the l-th layer

Table 3.4: Example comparison of an one-hot encoded ground truth label and a softmax 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	$\hat{\boldsymbol{y}} = \begin{pmatrix} 0.54 & 0.28 & 0.2 & 0.63 & 0.96 \end{pmatrix}^T$

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

$$\boldsymbol{W}^{[l]} = \begin{pmatrix} \boldsymbol{w}_1^{[l]} & \boldsymbol{w}_2^{[l]} & \cdots & \boldsymbol{w}_{n_h^{[l]}}^{[l]} \end{pmatrix}^T \tag{3.22a}$$

$$\boldsymbol{b}^{[l]} = \begin{pmatrix} b_1^{[l]} & b_2^{[l]} & \cdots & b_{n_h^{[l]}}^{[l]} \end{pmatrix}^T \tag{3.22b}$$

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

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

as the network's prediction for the i-th data sample $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n_y}$. 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 loss value. The most common loss function for comparing two probability distributions of mutually exclusive classes is the cross entropy loss function. It is defined as

$$H(y, p) = -(y \log(p) + (1 - y) \log(1 - p)) \tag{3.26}$$

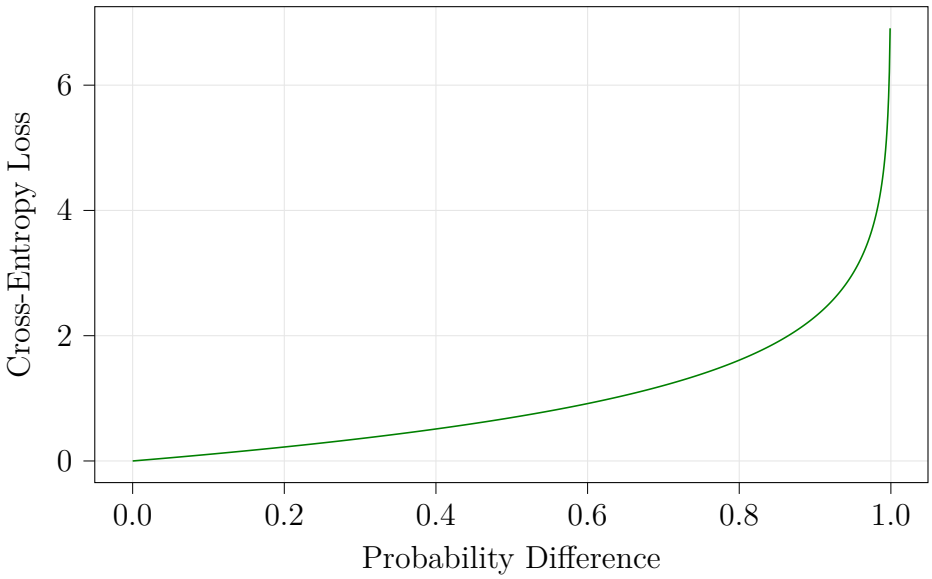


Figure 3.10: Cross-Entropy Loss. Large deviations in probability are strongly penalized.

for a single output representing two classes and as

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

for multiclass classification, where C is the number of classes, i the moving index, $\boldsymbol{y}$ the ground truth vector and $\boldsymbol{p}$ the predicted probabilities[36]. Applying the previous notation to this yields

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

$$\boldsymbol{p} = \hat{\boldsymbol{y}}^{(i)} \tag{3.28b}$$

$$C = n_y \tag{3.28c}$$

as analogies. Due to the one-hot encoding of the labels, only the positive class C_p is taken into account in the loss computation. Hence, Theorem 3.27 is reduced to

$$H(\boldsymbol{y}, \boldsymbol{p}) = -y_p \log(p_p) \tag{3.29}$$

where y_p and p_p denote the probability of the positive label and its corresponding prediction, respectively. A visualization of this expression is shown in Fig. 3.10. It can be seen, that large deviations in probability are strongly penalized. In the range of small deviations the slope of the graph is small which leads to little changes in loss, if the probability difference changes only slightly. Therefore, both kind of errors are penalized. Using this loss function for computing a cost that averages all losses yields

$$J(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}, \boldsymbol{y}) = J(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{m} \sum_{i=1}^{m} H(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) \tag{3.30}$$

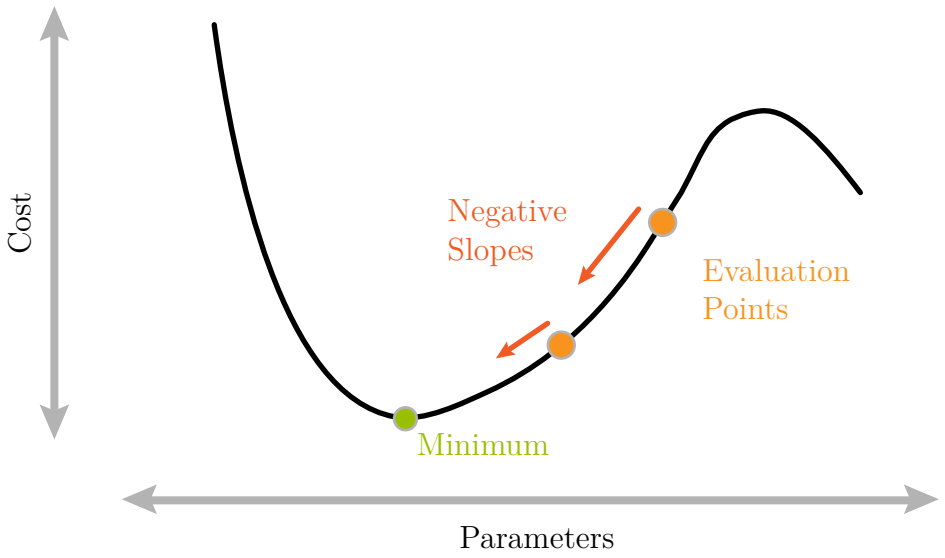


Figure 3.11: Schematic of gradient descent. Cost is evaluated, its negative gradients are computed and the parameters are moved along this direction until the minimum is reached.

as the cost function. This function depends on all weights and biases as regression parameters, hence, it is highly dimensional. Minimizing it with respect to the weights and biases yields optimal parameters.

3.1.4.4 Backpropagation with Gradient Descent

Due to the non-linearity in the network the minimum of the cost function cannot be computed analytically but numerically. Hence, it is evaluated at a certain point. Using backpropagation[37][17] results in a gradient for each parameter. Depending on these values representing the slope of the parameters, and, hence, their impact on the output, the parameters are changed to move closer to the minimum. This optimization is done by using the gradient descent algorithm[38][39]. Fig. 3.11 illustrates this approach. The gradients point into each direction of steepest ascent. Because a minimum needs to be found, those are negated for pointing into the direction of steepest descent. Finally, the parameters are moved along this direction. Following this approach means for the whole network that the cost function is backpropagated layer for layer to the beginning of the network using partial derivatives and the chain rule. When this is completed, it is known how the parameters are influencing the output and therefore how they need to be changed depending on their slope.

Let's fill these statements with mathematical expression. The goal of backpropagation

is to compute the partial derivatives

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \frac{1}{k} \sum_{i}^{k} \frac{\partial J^{(i)}}{\partial w_{jk}^{[l]}} \tag{3.31a}$$

$$\frac{\partial J}{\partial b_{j}^{[l]}} = \frac{1}{k} \sum_{i}^{k} \frac{\partial J^{(i)}}{\partial b_{j}^{[l]}} \tag{3.31b}$$

of the cost function w.r.t. the parameters by averaging the partial derivatives of cost functions of k samples that were passed through the network. These k samples build a batch. Fig. 3.12 recaps the data flow in a neuron in the last layer. By checking step wise how a parameter directly influences a subsequently one, Theorem 3.31 can be written as

$$\frac{\partial J}{\partial w_{jk}^{[L]}} = \frac{\partial z_{j}^{[L]}}{\partial w_{jk}^{[L]}} \frac{\partial a_{j}^{[L]}}{\partial z_{j}^{[L]}} \frac{\partial J}{\partial a_{j}^{[L]}} \tag{3.32a}$$

$$\frac{\partial J}{\partial b_{j}^{[L]}} = \frac{\partial z_{j}^{[L]}}{\partial w_{jk}^{[L]}} \frac{\partial a_{j}^{[L]}}{\partial z_{j}^{[L]}} \frac{\partial J}{\partial a_{j}^{[L]}} \tag{3.32b}$$

for the last layer. How much the activations of the second to last layer influence the cost function is expressed by

$$\frac{\partial J}{\partial a_{k}^{[L-1]}} = \sum_{j=1}^{n_y} \frac{\partial z_{j}^{[L]}}{\partial a_{k}^{[L-1]}} \frac{\partial a_{j}^{[L]}}{\partial z_{j}^{[L]}} \frac{\partial J}{\partial a_{j}^{[L]}} \tag{3.33}$$

where all weighted sums and activations of the output layer are considered. This needs to be done because layers are fully connected and the activation of one neuron influences every neuron in the next layer. Especially in the last layer, the activation of one neuron in the second to last layer influences directly the activations of all neurons in the last layer. Those are directly related to the cost, which is computed by comparing them with the ground-truth values. Hence, the influences of the activations of the neurons in the second to last layer need to be summed up. Once Theorem 3.33 is calculated for the second to last layer, this process can be repeated for all the weights and biases feeding into that layer. This goes on layer for layer until the first one is reached. In general,

$$\delta_{j}^{[l]} = \frac{\partial J}{\partial a_{j}^{[l]}} \frac{\partial a_{j}^{[l]}}{\partial z_{j}^{[l]}} \tag{3.34}$$

defines the error of the j-th neuron in the l-th layer. Combining Theorem 3.32 and

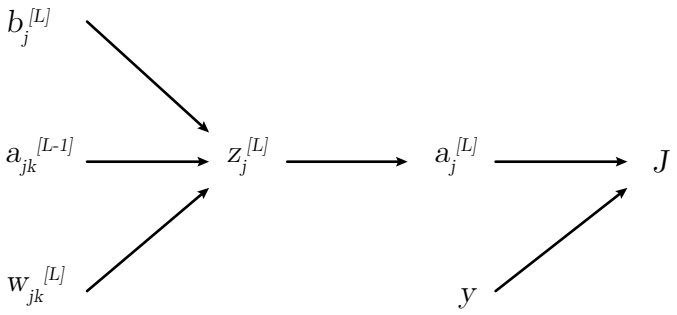


Figure 3.12: Data Flow in a Last-Layer Neuron for Backpropagation

Theorem 3.34 yields

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial J}{\partial a_j^{[l]}} = \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} \delta_j^{[l]} \tag{3.35a}$$

$$\frac{\partial J}{\partial b_j^{[l]}} = \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial J}{\partial a_j^{[l]}} = \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} \delta_j^{[l]} \tag{3.35b}$$

as the general expressions for backpropagation where

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_j^{[l-1]} \tag{3.36a}$$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1 \tag{3.36b}$$

$$\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \phi'(z_j^{[l]}) \tag{3.36c}$$

are the derivatives. Summarizing the gradients of the cost function yields the compact representation

$$\nabla \boldsymbol{J} = \begin{pmatrix} \frac{\partial J}{\boldsymbol{W}^{[1]}} & \frac{\partial J}{\boldsymbol{b}^{[1]}} & \frac{\partial J}{\boldsymbol{W}^{[2]}} & \frac{\partial J}{\boldsymbol{b}^{[2]}} & \cdots & \frac{\partial J}{\boldsymbol{W}^{[L]}} & \frac{\partial J}{\boldsymbol{b}^{[L]}} \end{pmatrix}^T \tag{3.37}$$

containing the influences of all parameters. Each element points into its direction of steepest ascent with a magnitude. Hence, each gradient is inverted for pointing into its direction of steepest descent for finding a minimum. Along each direction depending on its magnitude each parameter is changed. This can be expressed by

$$w_{jk}^{[l]}(\tau + 1) = w_{jk}^{[l]}(\tau) - \gamma \nabla \boldsymbol{J}(w_{jk}^{[l]}(\tau)) \tag{3.38a}$$

$$b_j^{[l]}(\tau + 1) = b_j^{[l]}(\tau) - \gamma \nabla \boldsymbol{J}(b_j^{[l]}(\tau)) \tag{3.38b}$$

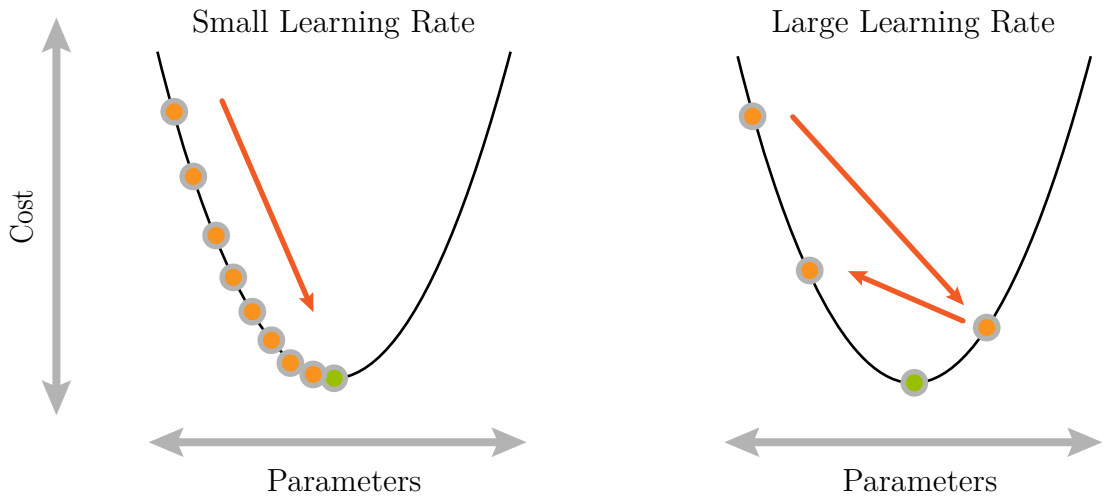


Figure 3.13: Comparison of Learning Rates. A small learning rate finds the minimum slowly but more exactly than a high learning rate. However, the latter tends to overshoot the minimum.

where the hyperparameter γ is the learning rate and τ the iteration step. This update procedure is done for every batch that is passed through the network. The updates resulting from a batch are called an iteration. This is repeated for the whole training set, which is called an epoch. The objective of the learning rate is to control how much the parameters are adjusted. The smaller it is, the slower the parameters are moving along the graph to the minimum. On its way, the slope steadily gets smaller which intensifies this effect. A similar effect arises by moving over plateaus. Surely, the minimum will be found more exactly than with a large learning rate, but it would make learning really slow. However, a large learning rate can steadily overshoot the minimum leading to no convergence. These effects are illustrated in Fig. 3.13. Thus, a trade-off must be found or an adaption of the learning rate to certain circumstances like the magnitude of the slope is necessary. According to *Bengio* [40] a traditional default value for the learning rate is $\gamma_0 = 0.1$ or $\gamma_0 = 0.01$ for standard multilayer neural networks. However, "it would be foolish to rely exclusively on this default value". The learning rate should be greater than 10^{-6} and less than 1.0. *Goodfellow et al.* [17] state, "in practice, it is common to decay the learning rate linearly until iteration τ. After iteration τ it is common to leave [the learning rate] constant." The underlying idea is to move quickly to close proximity to the minimum and then carefully to it. Another common approach is an exponential decay like

$$\gamma(\tau) = \gamma_0 \exp(-k\tau) \tag{3.39}$$

where γ_0 is the initial learning rate, k a factor and τ the time step. The concept of changing the learning rate over time is called learning rate schedule. Its values are

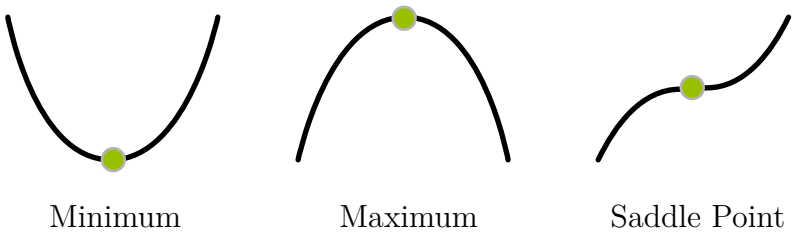


Figure 3.14: Types of Critical Points

arbitrary and do not inevitably need to be results of a decay but can be fixed values that are active depending on the time step τ, for example.

Due to the number of parameters and the use of hidden layers, cost functions are highly dimensional and neither convex nor concave. An explanation for the latter is, that several combinations of parameters can result in the same loss value. Hence, there are several local minima in the cost function. According to *Choromanska et al.*[41] almost all local minima have very similar function values to the global minimum. Hence, finding a local one is sufficient. Having neither a convex nor concave cost function, the function probably has saddle points. These points are no optimum, but have a gradient of $\nabla \boldsymbol{J} = \boldsymbol{0}$. The types of critical points are shown in Fig. 3.14. With this gradient the algorithm would get stuck. Hence, adding some noise to Theorem 3.38 yields

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \gamma \nabla \boldsymbol{J}(w_{jk}^{[l]}) + \boldsymbol{\epsilon}(w_{jk}^{[l]}) \tag{3.40a}$$

$$b_{j}^{[l]} := b_{j}^{[l]} - \gamma \nabla \boldsymbol{J}(b_{j}^{[l]}) + \boldsymbol{\epsilon}(b_{j}^{[l]}) \tag{3.40b}$$

where $\boldsymbol{\epsilon}$ is a noise vector with mean 0. Because saddle points are very unstable, adding some noise helps to overcome them.

3.1.4.5 Adam: Adaptive Moment Estimation

Learning rate schedules have the problems of defining their parameters in advance and applying the same learning rate to every weight and bias. Hence, the RMSProp (Root Mean Square Propagation) optimizer was developed[42]. Its objective is illustrated in Fig. 3.15. The ellipses represent contour lines. The orange line visualizes the process of gradient descent. Using it can result in parameters oscillating in one direction while making progress in another one moving to the minimum. The objective of RMSProp, represented by the blue line, tries to dampen the oscillations by slowing down learning of the responsible parameter and fasten up the other one. Hence, it can use a larger learning rate and reaches the minimum more quickly. RMSProp adapts a learning rate to each of the parameters. Furthermore, it divides the learning rate for a parameter by a weighted running average of the magnitudes of its previous gradients. The weighted

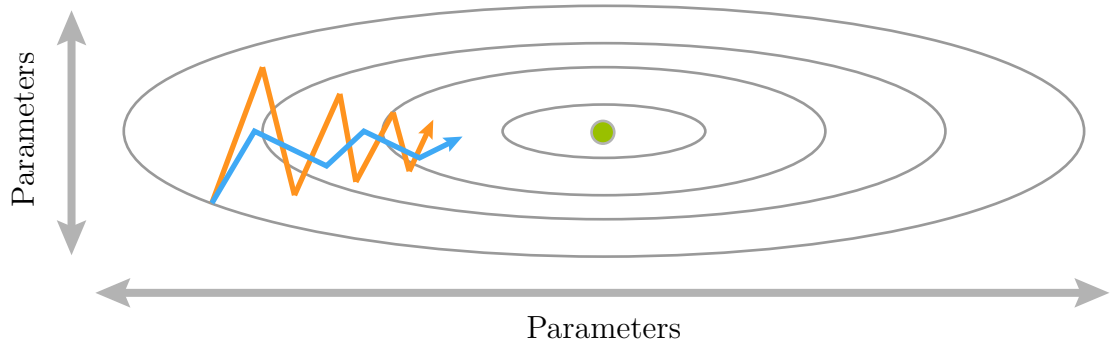


Figure 3.15: Process of gradient descent and RMSProp using contour lines. The orange line illustrates the process of gradient descent. It oscillates and moves slowly to the minimum. The blue line shows the process of the RMSProp algorithm. While one parameter changes slower the other one changes faster. Hence, it dampens oscillations and moves faster to the minimum.

running average is calculated by

$$v(w_{jk}^{[l]}, \tau) = \beta_2 v(w_{jk}^{[l]}, \tau - 1) + (1 - \beta_2)(\nabla \boldsymbol{J}(w_{jk}^{[l]}))^2 \tag{3.41a}$$

$$v(b_j^{[l]}, \tau) = \beta_2 v(b_j^{[l]}, \tau - 1) + (1 - \beta_2)(\nabla \boldsymbol{J}(b_j^{[l]}))^2 \tag{3.41b}$$

where hyperparameter β_2 is the forgetting factor. As a value its author suggests $\beta_2 = 0.9$. The subscript 2 is for later purposes and could be omitted for now. The squaring operation is done element-wise. So this expression adds kind of an inertia to the update procedure dampening oscillations and building up speed on flat surfaces. Finally, the parameters are updated by

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \frac{\gamma}{\sqrt{v(w_{jk}^{[l]}, \tau)}} \nabla \boldsymbol{J}(w_{jk}^{[l]}) \tag{3.42a}$$

$$b_j^{[l]} := b_j^{[l]} - \frac{\gamma}{\sqrt{v(b_j^{[l]}, \tau)}} \nabla \boldsymbol{J}(b_j^{[l]}) \tag{3.42b}$$

using the weighted moving average just calculated. Let's say the horizontal parameter is w_1 and the vertical one w_2 referring to Fig. 3.15. Due to the oscillations the gradient $\nabla \boldsymbol{J}(w_2)$ is much larger than $\nabla \boldsymbol{J}(w_1)$. Hence, v_2 is larger than v_1 resulting in a smaller update for w_2 and a larger one for w_1. This yields a more direct moving to the minimum. However, this approach does not make it easier to configure the learning rate as the step size is independent of it. Though, it improves the speed of the optimization process because a better set of weights is discovered in fewer training steps than with pure gradient descent. The Adam (Adaptive Moment Estimation) optimization is an update

to the RMSProp algorithm by adding a momentum

$$m(w_{jk}^{[l]}, \tau) = \beta_1 m(\tau - 1) + (1 - \beta_1) \frac{\partial J}{\partial w_{jk}^{[l]}} \tag{3.43a}$$

$$m(b_j^{[l]}, \tau) = \beta_1 m(\tau - 1) + (1 - \beta_1) \frac{\partial J}{\partial b_j^{[l]}} \tag{3.43b}$$

for each parameter using a weighted average of latest gradients[43] where β_1 is the forgetting factor. This momentum can be imagined as a ball in a bowl-shaped cost function rolling downwards and building up speed depending on the gradients. The hyperparameter β_1 represents friction. This approach can be summarized by using running averages of both the gradients and the second moments of the gradients. Due to the initialization $\boldsymbol{v} = \mathbf{0}$ and $\boldsymbol{m} = \mathbf{0}$ these values are biased towards zero, especially during the first few iterations. A correction is desirable, because the first and second moments are only estimations. In general, an estimation should equal the parameter that is tried to be estimated. Hence, the property

$$E[m] = E[g] \tag{3.44a}$$

$$E[v] = E[g^2] \tag{3.44b}$$

needs to be fulfilled, where $E[\cdot]$ represents the expectation of a variable. These properties only hold true, if unbiased estimators are used. Hence, the corrected values are expressed by

$$\hat{m}(w_{jk}^{[l]}, \tau) = \frac{m(w_{jk}^{[l]}, \tau)}{1 - \beta_1^\tau} \tag{3.45a}$$

$$\hat{m}(b_j^{[l]}, \tau) = \frac{m(b_j^{[l]}, \tau)}{1 - \beta_1^\tau} \tag{3.45b}$$

$$\hat{v}(w_{jk}^{[l]}, \tau) = \frac{v(w_{jk}^{[l]}, \tau)}{1 - \beta_2^\tau} \tag{3.45c}$$

$$\hat{v}(b_j^{[l]}, \tau) = \frac{v(b_j^{[l]}, \tau)}{1 - \beta_2^\tau} \tag{3.45d}$$

using the expression from before. Finally,

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \gamma \frac{\hat{m}(w_{jk}^{[l]})}{\sqrt{\hat{v}(w_{jk}^{[l]})} + \epsilon} \tag{3.46a}$$

$$b_j^{[l]} := b_j^{[l]} - \gamma \frac{\hat{m}(b_j^{[l]})}{\sqrt{\hat{v}(b_j^{[l]})} + \epsilon} \tag{3.46b}$$

Figure 3.16: Optimal Range of Learning Rates

updates the parameters, where ϵ is a small constant for preventing a division by zero. As for other optimization algorithms the learning rate γ needs to be tuned. The remaining hyperparameters are recommended by *Kingma et al.* to be $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. However, the latter has no big impact on the optimization.

3.1.5 Improving Performance

Earlier sections make general recommendations on hyperparameters. Nevertheless, there is room for improvement. Finding well-suited hyperparameters is a trial-and-error method and requires much time. However, there are methods that aim to find a good starting point. Those are going to be presented in the following.

3.1.5.1 Optimal Learning Rate

The learning rate is the most important hyperparameter in a neural network. If it is too small, learning converges exactly, though, really slow. If it is too large, the minimum is steadily overshot and learning maybe diverges.

Smith introduces the cyclical learning rate [44]. For being cyclical, the learning rate needs a lower and upper bound. This builds a range of optimal learning rates. With these values inbetween, the evaluations of the cost function are effectively decreased over time. Hence, for finding that optimal range, the learning rate is initialized with a very small value and slightly increased after each training step. For each of the steps the cost function is evaluated. Fig. 3.16 shows a typical plot of the cost function over the learning rate. It can be seen, that if the learning rate is small, the cost function does not improve noticeably. When the learning rate gets into its optimal range, the cost function suddenly becomes smaller. This continues with a large gradient, until the learning rate exits its optimal range. This passage is detected when the cost function starts to oscillate. If the learning rate still gets increased, the cost function starts to diverge eventually. *Smith* suggests a triangular learning rate policy which first increases the learning rate from the lower bound to the upper bound and then decreases it back to the lower bound. Each change happens linearly. The actual learning rate depends on the time step.

Another approach is doing warm restarts of the learning rate after several iterations, hence, it is called Stochastic Gradient Descent with Warm Restarts[45]. This is especially combined with the gradient descent algorithm. The idea behind it is to swing out of a tight local minimum if the algorithm seems to get stuck there. If well-suited parameters are found for a cost function, the latter can slightly change if the dataset changes. Hence, the once well-suited parameters lead to a worse cost now. Therefore, a minimum in a flatter region needs to be found where a slight change has no big impact, i.e. a

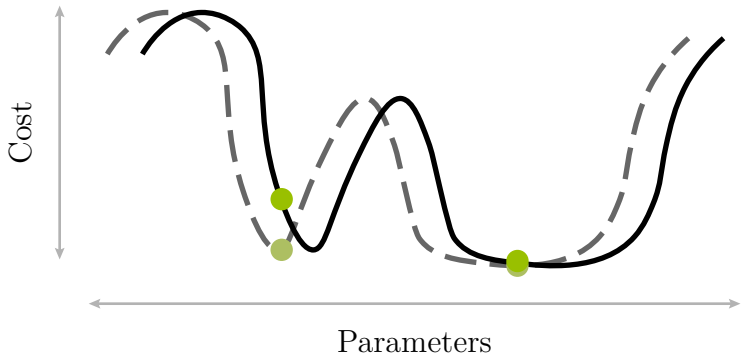
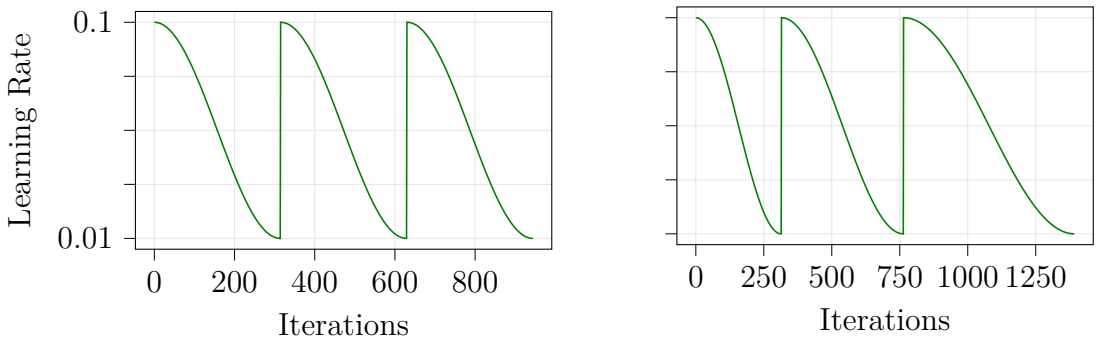


Figure 3.17: Cost Function of Different Datasets. If the parameters represent a minimum of a dataset's cost function, the same parameters do not inevitably represent the minimum of a different dataset's cost function.



(a) Cosine Annealing

(b) Extended Cosine Annealing

Figure 3.18: Annealing of Stochastic Gradient Descent with Warm Restarts

solution that is more generalized across datasets. Fig. 3.17 illustrates this scenario. This algorithm uses a cosine annealing of the learning rate. This means the learning rate is decreased in the form of half a cosine curve. This is called a cycle. After this, it is set to its initial value and the annealing is repeated. This process is visualized in Fig. 3.18a. So, first, the cost in decreased until the learning rate reset happens. Then, it is possible to overshoot the local minimum that much, that a different one is targeted. However, this minimum is flat enough that another learning rate restart does not change the targeted minimum. Due to the objective of finding flat regions it is advisable to steadily extend each learning rate cycle like it is shown in Fig. 3.18b. It is assumed, that the more iterations are done, the flatter the found region gets. Thus, the longer its minimum is searched.

3.1.5.2 Optimal Batch Size and Number of Epochs

Let's do a brief recap of the terminology. The batch size defines how many samples are propagated through the network at once. For each sample within the gradients of the cost function with respect to all parameters are calculated and finally averaged over all samples. The updates resulting from a batch are called a iteration. Processing all samples from the training set is called an epoch.

For finding the optimal batch size, first examine the two extremes. On the one hand, the whole training set can form a batch. This way the best direction to a minimum can be calculated. In terms of number of iterations this methods is the best. However, it is very expensive in terms of resources, because usually the amount of data can not be held in the RAM or GPU. This means, either more memory needs to be bought or a continual reload of data happens, which slows down the overall training process. An even more significant downside is that large batch sizes like 512 result in an impaired quality of the model in terms of generalization[46]. Because the parameter updates follow the best direction to a minimum does not mean, that this minimum is well-suited for other data. Usually the model converges to a sharp minimizer similar to Fig. 3.17. On the other hand, a batch size of one is used, which is called stochastic. The parameter updates are noisy and can point into a completely wrong direction, while still pointing along the steepest descent. Hence, they wander around the cost function and eventually reach the minimum after a long training time. However, the cost of computing the gradients of a single sample is quite trivial. Thus, a trade-off must be found for a so called mini-batch. One requirement is that the training needs to converge in a reasonable amount of time. This includes averaging out the noise of the gradients for more accurate steps leading to an earlier convergence. Hence, the batch size also depends on the learning rate. A smaller batch size is better suited for a small learning rate due to the noise. A good balance is found, if the batch is small enough to avoid the poor minima but stays in the flatter, better performing ones. Another requirement is the computational cost. Fortunately, vector computing is optimized almost perfectly in most frameworks, resulting in only marginally higher computation cost for a few samples compared to a single one. Hence, a batch size should be larger than one, usually, and less than the whole training set but the optimal size is only found by trial and error. Common batch sizes are 32, 64, 128 and 256.

When using mini-batches the number of epochs is a hyperparameter, that can be tuned, as well. In general, smaller learning rates require more epochs because more parameter updates are necessary. But more important is the generalization of the network. The objective is to prevent underfitting or overfitting, which is explained in Section 3.1.4.1). Thus, the number of epochs cannot determined beforehand, but depends on the data. However, the training set needs to be shuffled every epoch, so that different batches are created compared to earlier epochs. This improves generalization due to the computation of different batch gradients.

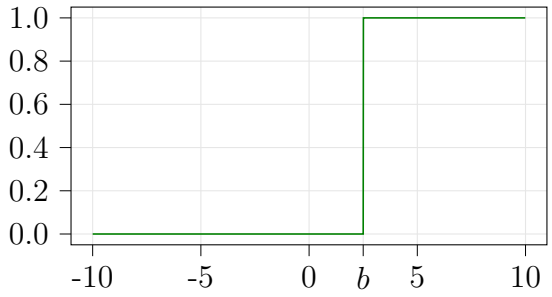
3.1.5.3 Activation Functions

The activation functions of a neural network affect its performance and convergence as well. Common activation functions are shown in Fig. 3.19. The most common ones are going to be examined. The sigmoid function recently has fallen out of favor. One disadvantage is its saturation which leads to a very small gradient. If this small gradient is often multiplied because of several layers, the gradient gets very small. This vanishing gradient leads to very small parameter updates. Furthermore, a well-suited weight initialization is required. If they are too large, related neurons become saturated and the network will barely learn. Hence, in both cases convergence takes a very long time. Another disadvantage is, that outputs are not zero centered. That means, that, for example, if all data is positive, all related gradients point into the same, either positive or negative, direction. This leads to an undesired zigzag pattern of the parameter updates for several samples. However, the use of mini-batches smooths out this effect. The tanh activation function has a zero centered output, though, the saturation of the output remains. The ReLU activation function accelerates the convergence process of stochastic gradient descent compard to sigmoid and tanh. According to *Krizhevsky* it is six times faster on the CIFAR10 dataset compared to tanh activation functions[4]. This is due to its linear, non-saturation form. Another advantage is its simple and light computation. Furthermore, it makes the activations sparse from the perspective of a neural network. This means not all neurons are active due to the ReLU being zero for input values below zero. Hence, the overall network is lighter. However, its property of the evaluation to zero is also a disadvantage. This results in a gradient of zero as well and therefore in no related parameter updates. If the weights are initialized badly or an unfavorable update is applied, for example, due to a too large learning rate, a ReLU unit can die, because its evaluation and the gradient are zero for all further computations. In this case, a unit is very unlikely to recover. A solution forms the leaky ReLU, which adds a slight slope of like $s = 0.01$ to the horizontal line of 0, preventing the gradient to become zero. Hence, the unit can recover. However, the results of this approach are very inconsistent.

Taking all that information into account yields the recommendation of the ReLU activation function, if the weights and learning rate are chosen carefully Additionally, the fraction of dead units should be monitored. If the number is still concerning, leaky ReLU activations should be applied. The tanh activation should be generally preferred over the sigmoid one.

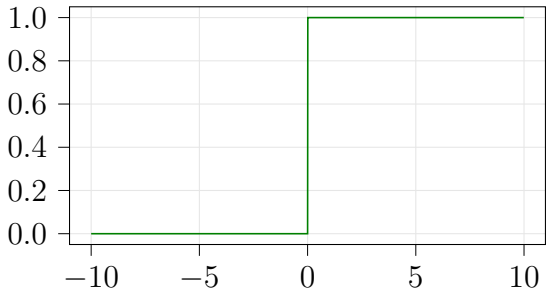
3.1.6 Metrics for Performance Evaluation

When the training of a network is finished, its performance needs to be expressed in numbers. There are several common metrics available, that depend on some definitions that will be introduced first. A true positive TP is a correct prediction of a sample, so the positive class is predicted correctly. A true negative TN is a correct rejection



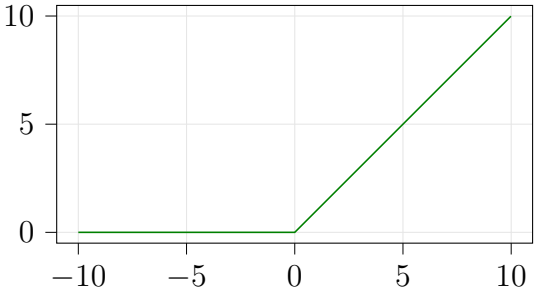
$$\phi(x) = \begin{cases} 1 & x \geq \text{Bias } b \\ 0 & x < \text{Bias } b \end{cases}$$

(a) Threshold



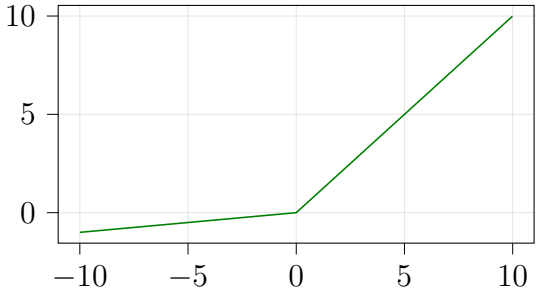
$$\phi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

(b) Heaviside



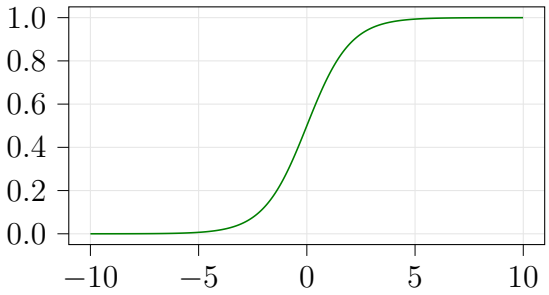
$$\phi(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

(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 3.19: 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 $s = 0.01$.

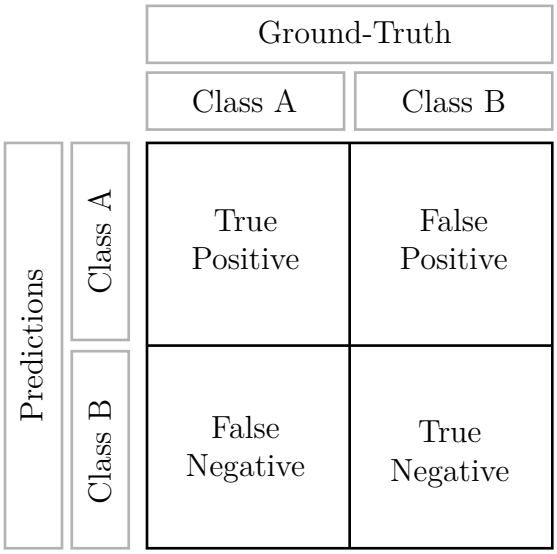


Figure 3.20: Confusion Matrix for Class A

of a sample, i.e. the network knows that a certain sample does not belong to a certain category. Hence, it predicts correctly the negative class. Furthermore, a false positive FP is a wrong prediction of a sample. The network incorrectly predicts the positive class. The last definition is a false negative FN. This means the network incorrectly predicts the negative class. Their connection is visualized in Fig. 3.20 showing the so called confusion matrix[47] for the example class Class A. One metric is the accuracy

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.47}$$

of the model. This states how many samples the network correctly classifies. However, this metric's results are not reliable for the real performance, because it highly depends on the dataset. If it is unbalanced like if there are more samples in a well classifiable category than in a bad one, the accuracy is shifted. The precision or positive predicted value of a network measures how accurate the predictions are and is calculated by

$$PPV = \frac{TP}{TP + FP} \tag{3.48}$$

where the denominator refers to the total positive results. Furthermore, the recall or true positive rate metric measures how good all positives are found by

$$TPR = \frac{TP}{TP + FN} \tag{3.49}$$

where the denominator refers to the actual positives.

3.2 Software

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

3.2.1 Tensorflow

Tensorflow is a free framework in particular for machine learning task. It was originally developed by Google Brain for internal Google use and got finally licensed for open source. Mathematical operations are designed as a symbolic graph. After its creation any operation inside can be executed and only needs the computation of its dependent operations. Every computation involves tensors. A tensor is a generalization of scalars, vectors and matrices independent on their dimension. Hence, a definition of every tensor's size is important for a cost-effective creation and computation of the graph. Due to the symbolic graph, it is possible to define a neural net with an input and output layer and steadily feed and compare different tensor values.

3.2.2 Blender

Blender is a free and open source 3D creation suite to model, texture and animate objects. The integrated algorithms like lighting and shadowing offer default parameters that can be changed according to one's wishes. Furthermore, it supports importing existing models and manipulating them. Additionally, an API interface is provided, that can be used with the programming language Python, to control every function of Blender. This eases repetitive tasks tremendously.

Every object in Blender has its own coordinate system. Hence, a way of expressing points of one coordinate system in another would be useful. Let's define a world coordinate system $\mathcal{W}$ with the origin $\boldsymbol{o}_{\mathcal{W}} = (0,0,0)^T$ and the rotation $\boldsymbol{r}_{\mathcal{W}} = (0,0,0)^T$, where each element of the latter represents a rotation around the x-, y- or z-axis, respectively, in radians. This system contains every other system. Now another coordinate system $\mathcal{L}$ is created, of course, inside the world coordinate system. However, $\mathcal{L}$ can be translated and rotated in comparison to $\mathcal{W}$. Hence, every coordinate system has a rotation matrix $\boldsymbol{R}$ in euler representation and a translation vector $\boldsymbol{t}$ that stores how they are rotated and translated to every other coordinate system. Considering $\mathcal{L}$ and $\mathcal{W}$ yields

$$\boldsymbol{R}_{\mathcal{L}\to\mathcal{W}} = \boldsymbol{R}_{\mathcal{W}\to\mathcal{L}}^{-1} \tag{3.50}$$

$$\boldsymbol{t}_{\mathcal{L}\to\mathcal{W}} = -\boldsymbol{t}_{\mathcal{W}\to\mathcal{L}} \tag{3.51}$$

as properties, where the subscript indicates the transfer. Transferring the coordinates of an arbitrary local point $\boldsymbol{x}_{\mathcal{L}}$ into corresponding coordinates $\boldsymbol{x}_{\mathcal{W}}$ of the reference coordinate system is done by using

$$\boldsymbol{x}_{\mathcal{W}} = \boldsymbol{R}_{\mathcal{L}\to\mathcal{W}} \cdot \boldsymbol{x}_{\mathcal{L}} + \boldsymbol{t}_{\mathcal{L}\to\mathcal{W}} \tag{3.52}$$

as the general expression.

Chapter 4

Methods

This chapter explains the generation and preparation of the dataset and the implementation of the neural network architecture for classifying objects depending on their color feature. The input consists of multiple views of an object, while percentages of class memberships are outputted. The creation of the dataset is supported by the Blender API interface, while the model is written in Python using the tensorflow framework.

4.1 Dataset Generation

4.1.1 Choosing a Dataset

One requirement of the dataset is, that there are multiple views of the same object available. Optimally, these views can be arbitrarily chosen for having as much freedom as possible for training and evaluating the network. Hence, three dimensional objects are necessary. The related object categories are preferably discriminative to each other due to the objective of classifying real world objects and an easier evaluation of the model. This means, the dataset should not contain only flowers or faces for example. These constraints bring up two possibilities. The first one is creating new CAD objects. These can be modeled in a way that supports the own needs the most. However, creating plenty of models for every category would take a huge amount of time. The second option is to use CAD models of an existing dataset. Referring to the first possibility, this one just takes the time for finding a suited dataset and perhaps some slight modifications. The time for manipulating this one according to the wanted color features compared to the self-created one is probably identical and, therefore, not taken into account. Another advantage is the competitive ability, because if this one is a popular dataset, other researches probably used it as a benchmark for their neural network as well. Hence, a qualitative evaluation would be possible. Taking all these arguments into account, an existing dataset is the best choice.

There are three different techniques for building a three-dimensional shape. One uses triangles, where each one is defined by the position of its edges. These positions are called vertices and contain an x-, y- and z-coordinate. Additionally, each triangle has a normal vector and other information like a color and a texture. Combining several

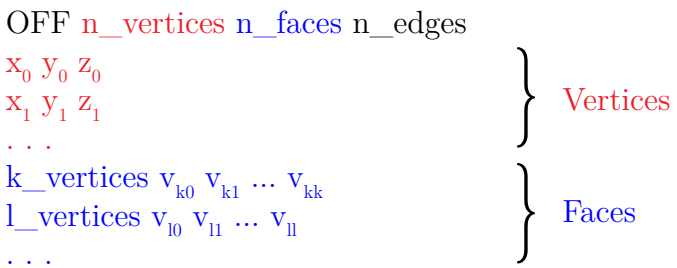


Figure 4.1: CAD Model in Object File Format

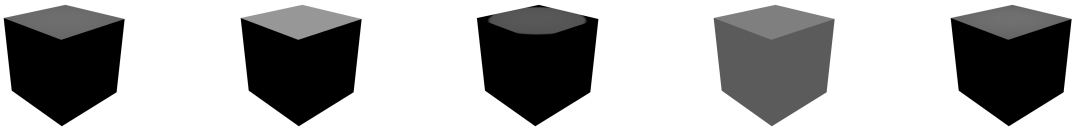
triangles results in the final shape that is called a mesh object. Another one uses voxels. This word is a portmanteau of "volume" and "pixel" and represents a cell in a three-dimensional grid. Each voxel behaves like a pixel and has a color. The last technique is a point cloud. Each point is represented by a vertex. Such a cloud is often created by 3D scanners that measure a large number of points in a scene, like distances and sometimes color for digitalizing it. A comparison of these techniques yields that the mesh object is the best suited one. It has the smoothest surface because it is continuous, hence, representing real world objects most likely. Using enough voxels could result in a similar shape, but the data size would be much bigger. Furthermore, applying a color feature to a single triangle is easier than applying it to several voxels, where connected ones need to be found first.

The most popular dataset containing CAD objects in polygon mesh representation is ModelNet[11]. It contains 127,915 CAD models divided in 662 object categories for now. For convenience, there are a 10-class and 40-class subset containing 10 or 40 popular categories, respectively. Both are cleaned in respect to a wrong category sorting and then split into a training and a testing set. Furthermore, the orientations of the models of the first one are aligned as well.

4.1.2 Rendering Views of CAD Models

Each CAD model of ModelNet is saved in object file format with the extension ".off". This file contains among others all definitions of faces necessary for modeling the related shape as ASCII text. Faces refer to an arbitrary number of triangles building a plane. However, the file starts with the keyword "OFF" and each number of vertices, faces and edges. The latter can be ignored because they are not relevant in this representation. Then, all vertices are listed with each x, y and z coordinates, where every vertex is written in a separate line. This way every vertex has a related index starting at 0. Afterwards, the faces are listed with one per line. Every line starts with the number of vertices of this face and then the corresponding vertex indices. A schematic of this file is shown in Fig. 4.1.

The folder structure of ModelNet is as follows. First it is divided into training and



(a) Point (b) Sun (c) Spot (d) Hemi (e) Area

Figure 4.2: Lamp types available in Blender. Each light source is placed above the cube pointing directly towards it.

testing set. Each set contains identical categories represented as folders. Within each category folder the related model files are stored having a unique identifier as their file name. It is important to mention, that both sets contain different models, hence, if the network is trained on the training set, the models from the test set were never seen before. Both the sets are recursively searched for models which are then subsequently processed the same way. How a single model is processed is explained in the following. All executed operations in Blender are called through an related API call, which allows running those commands for each model in an automated script.

First the file is imported into Blender, where it is interpreted with respect to its vertices and faces and drawn into a so called scene. The origin of the coordinate system of the mesh is set to its center of mass depending on the face areas. Furthermore, for simplicity, the mesh coordinate system is considered the world coordinate system $\mathcal{W}$. For adding lighting to the scene a lamp needs to be placed inside. Blender offers several lamp types, whose effects are shown in Fig. 4.2. The point lamp emits light omni-directional from its origin. That means, the same amount of light is emitted along the radial direction in all directions. This is useful for providing local lights like light bulbs. Another type is the sun lamp. It is placed very far away and provides light of constant intensity emitted parallel into a single direction. The spot lamp emits light in a cone-shaped beam into a given direction. The area lamp is similar to the point lamp, except it emits light from a surface like a TV screen. This produces shadows with soft borders. Finally, the hemi lamp emits light radially from a plane. Similar to the sun lamp, the location is independent, just the direction is important. One requirement is, that the object should be illuminated on all sides due to the rendering of several views. Another one is simplicity. It would get quite complex adding several point lamps for achieving the first requirement. Hence, the hemi lamp is chosen, assured by Fig. 4.2 to provide a well-suited illumination. Furthermore, although, all models have different heights, a related change of the lamp's location is not necessary due to its properties. Hence, the lamp is assigned only a rotation of $\boldsymbol{r}_l = (0, 0, 0)^T$, i.e. pointing directly onto the model along the negative z-axis.

For rendering views a camera object $\mathcal{C}$ is necessary and, thus, added to the scene. Its parameters are left to the default values, except of its view distance, which is set very high to work with all models. Hence, only its location and rotation needs to be set. Due to the verification from *Su et al.* [12] of the setup from *Su et al.* [10], it is adopted. The

camera is elevated 30 degrees from the ground plane, pointing towards its origin. This results in the first rotation vector

$$r_{c,0} = \left(\frac{r_{x,deg} \cdot \pi}{180}, 0, 0\right)^T = \left(\frac{60 \cdot \pi}{180}, 0, 0\right)^T \tag{4.1}$$

in radians, where $r_{x,deg}$ is the rotation around the x-axis in degrees. Because the camera points along the negative z-axis by default, $r_x = 60$ corresponds to the just mentioned setup. The next step is fitting the camera view to the model just by changing the location of the camera. Fortunately, Blender supports this with a single API call. Just like in [12] an image canvas is applied, to support valid convolutions in the future. This is achieved by moving the camera coordinate system away from the mesh coordinate system along the line of their centers. However, from the perspective of the camera coordinate system, itself needs to be moved along its z-axis, because by definition cameras point towards its negative z-axis by default and it is already aligned with the mesh. Expressing the change of the local camera coordinate system in the mesh coordinate system is done by

$$\boldsymbol{x}_c = \boldsymbol{R}_{\mathcal{C}\to\mathcal{W}} \cdot (\boldsymbol{o}_{\mathcal{C}} + \boldsymbol{d}) + \boldsymbol{t}_{\mathcal{C}\to\mathcal{W}} \tag{4.2}$$

where the origin $\boldsymbol{o}_{\mathcal{C}}$ is moved by $\boldsymbol{d}$ first and then rotated and translated according to the difference in both the coordinate systems. The rotation matrix $\boldsymbol{R}_{\mathcal{C}\to\mathcal{W}}$ and translation vector $\boldsymbol{t}_{\mathcal{C}\to\mathcal{W}}$ are properties directly available in Blender. The local translation is set to

$$\boldsymbol{d} = \left(0, 0, \frac{\boldsymbol{c}_{\mathcal{W},z}}{10}\right)^T \tag{4.3}$$

where the z element is a fraction of the original z position $\boldsymbol{c}_{\mathcal{W},z}$ of the camera in the world, i.e the distance from the world origin to the camera object. The advantage of a fraction is, that the padding is independent of the model's size. Finally, this camera view is rendered with the following properties. The resolution is defined to be 224×224 pixel with a resolution percentage of 100%. The latter sets a fraction of the chosen resolution. This is useful for the development process to save resources and time, but for the final rendering the full resolution is desired. Furthermore, it needs to be coped with aliasing. Because every pixel can only have a single color edges usually have a step pattern. This is not realistic, hence, it is smoothed out by anti-aliasing techniques. This works by rendering the related image region in a higher resolution, taking several samples of pixel values and averaging them to get the value of the pixel in the desired resolution. The best available sample size in Blender is 16, hence, it is chosen. The background color is left at the default rgb color $\boldsymbol{c} = (64, 64, 64)$ for adding some kind of noise to the views. A black background would yield inputs of 0 and resembles in general no real world views. Finally, this view is saved as a PNG file with a trailing view index in its file name, while preserving the original folder structure. For gathering multiple views, the camera needs to be repositioned. Hence, the following steps are repeated for the desired number of

views. The rotation of the camera is set to

$$\boldsymbol{r}_c = \left(\frac{60 \cdot \pi}{180}, 0, \frac{k \cdot s \cdot \pi}{180} \right)^T \tag{4.4}$$

where k is the view index, originally starting at 0, and s the moving interval in degrees. The latter is set to

$$s = \frac{360}{n} = \frac{360}{12} = 30 \tag{4.5}$$

where n equals the number of views. For the ability of comparing this work to related researches $n = 12$ is defined. The number of objects per set corresponds with [10]. That means, 90 objects per category for the training set and 30 for the testing set. However, for simplicity and considering further manipulations with material features, only four categories are picked from the ModelNet10 dataset. Those embrace bathtubs, dressers, sofas and monitors.

4.1.3 Applying Material Feature Manipulations

The objective is to clone every model and apply a color feature to every clone to be able to distinguish same models. Cloning is a trivial task, because this can be done by rendering the native object first and then the color featured one. One method for the color feature manipulation is preparing a color feature like a colored circle as an image and putting it on the model. This feature can be scaled dependently on all the face areas for achieving a similar scale across all models. However, clinging this image to a model considering all its edges is problematic. It would be necessary to find a way to unfold the model along its edges. This can be done by hand, but is very time-consuming and complex. In fact, Blender supports a automatic way of unfolding, but its results are not satisfying. Therefore, this approach is not further pursued. Another method is to color vertices. However, for achieving reasonable features, a high number of vertices is necessary. There is an option to subdivide existing faces and add additional vertices, but this increases the data size extremely, which needs much more computer performance, and has the risk of inducing unwanted geometry into the model. Hence, coloring single faces becomes the chosen approach due to its simplicity. In CAD modeling objects are assigned a material with a color that reacts to light which produces shadows. In blender the default material's color is a slightly darkened white. Changing the color of a face is performed by changed the color of its material.

For following this approach, it is advisable to pre-define material presets. Those presets consist of the default material but have different colors. After making those presets available to the Blender scene, a well suited face for being colored needs to be found. In the following this faces is referred to as the optimal face. It is important to filter all faces of an object by their area size. Fig. 4.3 shows why and the results of several thresholds. If faces are not filtered at all, any face can result in being the optimal one. However, it is not guaranteed that this face has a descent size and can be seen and

(a) Unfiltered (b) Face Area Mean (c) Max Face Area Dependent

Figure 4.3: Threshold Setup for Filtering Faces by their Area Size

recognized by the network easily. This assumption was later verified by a not changing loss value during training. Like in Fig. 4.3a the optimal face is comparatively small to the whole model. Accepting only faces as the optimal face, that have at least the size of the mean of all faces lead to a result like in Fig. 4.3b. The optimal face can be seen more easily than before, but this threshold often results in long and slender optimal faces. This is due to the fact, that the chosen object categories mostly contain objects that are built using such faces, because they have longish surfaces. Hence, filtering by the mean of the faces amplifies the probability to choose such a face as an optimal one. Therefore, a threshold of above the mean is desired to skip all those lathy faces like it is shown in Fig. 4.3c. Additionally, larger faces should be preferred. Thus, the area of the optimal face have to satisfy

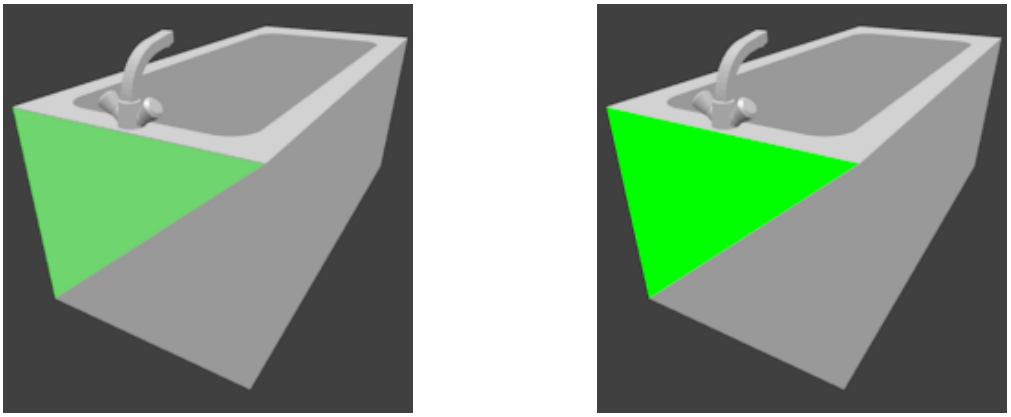
$$a_{opt} > (a_{mean} + a_{max}) \cdot s \tag{4.6}$$

where a are the related areas and s a scalar. With $s = 0.3$ satisfying results are achieved.

Furthermore, it needs to be guaranteed, that the optimal face is visible in at least one view and not visible in at least one view. This is validated by casting rays from the camera center onto the possible optimal face for every camera position that was defined in Section 4.1.2. In brief, if the rays hit the face, the face is visible. Fortunately, there is a function in Blender performing this approach and returning among others the index of the face that is hit. However, this cannot be performed for every pixel of a face due to performance issues. Thus, the trade-off against accuracy is only checking the vertices defining the face. This can raise errors, though, because a vertex can define multiple faces and it is not certain which face index the function returns. Hence, each checkpoint $\boldsymbol{p}_i$ is moved slightly to the center of the face by

$$\boldsymbol{p}_i = \boldsymbol{v}_i + 0.05 \cdot (\boldsymbol{f}_0 - \boldsymbol{v}_i) \tag{4.7}$$

where $\boldsymbol{v}_i$ is the related vertex and $\boldsymbol{f}_0$ the center of the face. If the ray cast is valid for at least one checkpoint the face is supposed to be visible. If all rays hit the wrong face,



(a) Single Optimal Face (b) All Optimal Faces

Figure 4.4: Material Manipulation on Duplicated Optimal Faces

the current face is supposed to be not visible. As soon as both conditions are satisfied for the examined face among all views, it becomes the optimal face. Otherwise, the next possible face is investigated. If the conditions are never satisfied for each possible face, the object is skipped at all.

It is found, that a single optimal face is not enough, because some models have several duplicated faces. Those are different faces where the coordinates of all the describing vertices are identical to the ones of other faces. Hence, there are faces laying into each other. This results in rendering issues like it is shown in Fig. 4.4, because the rendering engine does not know, which material should be rendered on the surface. In Fig. 4.4a only one of two identical faces is colored, which leads to the noticeable transparency effect. That is one of the brighter effects, though. It is also possible that one optimal face is shown normally and only on the edge rendering issues are visible like a dotted line with the colors of all optimal faces. Nevertheless, any of this effects could induce false correlations into the dataset that are not available on real-world objects, hence, leading to a not practical network. Thus, in Fig. 4.4b the materials of all identical faces are changed, which leads to a realistic color representation.

Regarding a validation of the later model, an examination of two material features per object would be interesting. Hence, for applying another material change on a model, another optimal face or faces, respectively, needs to be found. A requirement for this is the presence of only one material feature in a single view. Due to the automation of this task, a reliable solution is necessary. One approach would be using the other side of the surface of the first optimal face. However, this fails if the surface has a thickness of more than a single face. Then the next face along its normal needs to be found by ray casting and then the visibility of this face needs to be verified. Thus, this process leads to excessive ray cast validations and therefore not followed up on. However, it is considered, that faces at a similar location as the original optimal face are well-suited as well. Thus, the choices of further optimal faces are narrowed down by sorting all

remaining faces by their distance from their center to the center of the first optimal face in a descending order. The intention is that choosing faces with the largest distances result in either a kind of opposite face or in a face that is that far away to not be visible at the same time. as the first optimal face. Of course, the tasks of checking the visibility and finding identical faces are performed on the new face as well. If no additional optimal face is found, the model is skipped, although, this never happened during execution.

4.2 Preparing the Dataset

4.2.1 Single-View to Multi-View Conversion

The views created in Section 4.1.3 exist in a single view representation. Thus, a multi-view classification should be performed by the network. This means, each input represents a model with all its corresponding views. Hence, each model's single views need to be converted into a related multi-view representation. For achieving this the single views needs to be collected first. From a given custom file path to the dataset all model views are collected recursively in a sorted order. This is necessary for keeping related views in the order by which they are created. Due to the self-explanatory folder structure, models for the training and testing set can be handled independently. To each of the sets belongs a list $\tilde{\boldsymbol{X}}_{set}$ for storing all related views in RGB color representation. Now, each view's pixel values are read, normalized to a range between 0 and 1 and then stored in a matrix or array $\boldsymbol{X}_{set}^{(i)}$, respectively, which is added to one of the just mentioned lists. Simultaneously to the view gathering each label needs to be created as well and put into a list $\tilde{\boldsymbol{Y}}_{set}$ similar to the views. Getting the category is quite simple, because the file path of the view is known. Hence, the second to last element of the view's file path represents the category label. The file name of a view is build like *category_object-id_material-id_view-id.ext*. Here is the material index label extracted by splitting the file name by "_" first and then taking the third split element. Depending on the classification task of the model, those two labels possibly need to be combined. The single-label classification case embraces the following configurations. If categories and materials are classified, both labels are appended to each other. This results in $n_l = n_c \cdot n_m$ labels, where n_c is the number of categories and n_m the number of materials. If only categories or materials are classified, the final label is the category label or the material label, respectively. Logically, the number of labels is either $n_l = n_c$ or $n_l = n_m$. In the multi-label classification case both labels are used independently. Hence, the number of labels equals $n_l = n_c + n_m$. An example of those configurations is shown in Table 4.1. When the number of labels is known, a one-hot encoding is performed on each label. In conclusion,

Table 4.1: Label Generation with example categories "bathtub" and "sofa" and materials "0" and "1" for different cases of classifications.

Classification	Single-Label	Multi-Label
Category + Material	bathtub_0 bathtub_1 sofa_0 sofa_1	bathtub sofa 0 1
Category	bathtub sofa	n/a
Material	0 1	n/a

this process results in four lists

$$\tilde{\boldsymbol{X}}_{train} = \begin{pmatrix} \boldsymbol{X}_{train}^{(1)} & \boldsymbol{X}_{train}^{(2)} & \cdots & \boldsymbol{X}_{train}^{(m_{train})} \end{pmatrix} \tag{4.8a}$$

$$\tilde{\boldsymbol{X}}_{test} = \begin{pmatrix} \boldsymbol{X}_{test}^{(1)} & \boldsymbol{X}_{test}^{(2)} & \cdots & \boldsymbol{X}_{test}^{(m_{test})} \end{pmatrix} \tag{4.8b}$$

$$\tilde{\boldsymbol{Y}}_{train} = \begin{pmatrix} \boldsymbol{y}_{train}^{(1)} & \boldsymbol{y}_{train}^{(2)} & \cdots & \boldsymbol{y}_{train}^{(m_{train})} \end{pmatrix} \tag{4.8c}$$

$$\tilde{\boldsymbol{Y}}_{test} = \begin{pmatrix} \boldsymbol{y}_{test}^{(1)} & \boldsymbol{y}_{test}^{(2)} & \cdots & \boldsymbol{y}_{test}^{(m_{test})} \end{pmatrix} \tag{4.8d}$$

where $\boldsymbol{X}^{(i)}$ and $\boldsymbol{Y}^{(i)}$ of the same set are building an input-output pair.

Those single view and label representations need to be converted to a multi-view representation. Because sorted data was read in, it is known that each 12 elements belong together and need to put together somehow. Looking at common definitions yields, that images are a three-dimensional matrix with the shape definition $Height \times Width \times Channels$. Furthermore, Tensorflow mostly uses the shape definition $Batch \times Height \times Width \times Channels$ for tensors. Hence, assuming each view as a batch element is a reasonable approach. If an actual batch dimension becomes necessary it is inserted as a new first dimension. This yields a reduction of Theorem 4.8a and Theorem 4.8b to a n_v-th of its size, where n_v is the number of views. The labels can be processed similar, however, much easier. Because each n_v labels are identical, it is sufficient to just keep every n_v-th label. For a later lookup of the labels they are saved to the disk as a text file.

4.3 Multi-View Network Architecture

Because convolutional neural networks are well-suited for image classification task, this approach is pursued. Furthermore, in [10] the VGG-M[9] architecture is used and in [14] GoogLeNet[48]. Both are convolutional neural networks, where the first uses 8 layers and the latter, that is more recent, 22, however, both of them yield satisfying results.

The number of parameters of both is too large for the available resources for this work, though. Hence, the AlexNet architecture[4] is chosen. It is very similar to the VGG-M one, but uses larger filters for convolutions and less nodes in the fully connected layers. However, the lesser parameters are a trade-off for accuracy.

In [14] it is shown that the performance of the network from [10] can be improved even more by grouping views with similar informational content and giving their descriptors more weight during the classification. It is supposed that in particular the grouping process suits the task of distinguishing same models with different materials very well. Because the material is the only difference, the related views should be weighted the most while all others should only play a marginal role in the classification process. This would reduce noise and thus results in a better performance. Hence, the network is divided into three important modules as illustrated in Fig. 4.5. The feature module takes all views of an object and calculates a descriptor for each. Those are fed to the grouping module, that groups views dependent on their information content and generates a descriptor for every group and additionally their weight. Dependent on those outputs a single descriptor is calculated, that describes the inputted shape or object, respectively, and is used for the classification. Each of the modules is examined in detail in the following sections.

4.3.1 Feature Module: Generating View Descriptors

The objective of the feature module is the generation of a descriptor for each view by using five convolutional layers. Each one is referred to as a view descriptor $\boldsymbol{V}_i$ in the following. Fig. 4.6 shows the basic concept. This module is the first one in the feed-forward chain. It is connected with the real world by having all views $\mathbb{V}$ of an object as its input. Furthermore, because tensorflow supports batch execution, i.e. all its operations can be applied to a batch of data, the input tensor is extended with a batch dimension for multiple multi-views. This yields an input tensor of shape $Batch \times Views \times Height \times Width \times Channels$. In the following, if a tensor has a batch dimension, which is usually the case, it is assumed that any mentioned operation or approach is applied to every batch element. This module consists of five main convolutional layers. A main convolutional layer is supposed to have a convolutional layer and an optional pooling layer. The first main layer performs a valid convolution with 96 filters of size 7×7 and a stride of 2 on each $224 \times 224 \times 3$ input. Every filter extracts different features. In comparison, the original AlexNet uses filters of size 11×11 and a stride of 4. However, it is assumed that smaller filters and a smaller stride are collecting more information that can be used for a classification of the object and material at once. A convolution operation is done by flattening the filter tensor to a 2D matrix of size $Filter\_Height \times Filter\_Width \times Channels\_In \times Channels\_Out$. Here, $Channels\_in = 3$ because of the RGB channels of the input image and $Channels\_Out = 96$ because of the defined number of filters. Then, image patches are extracted from the input tensor depending on padding and stride that

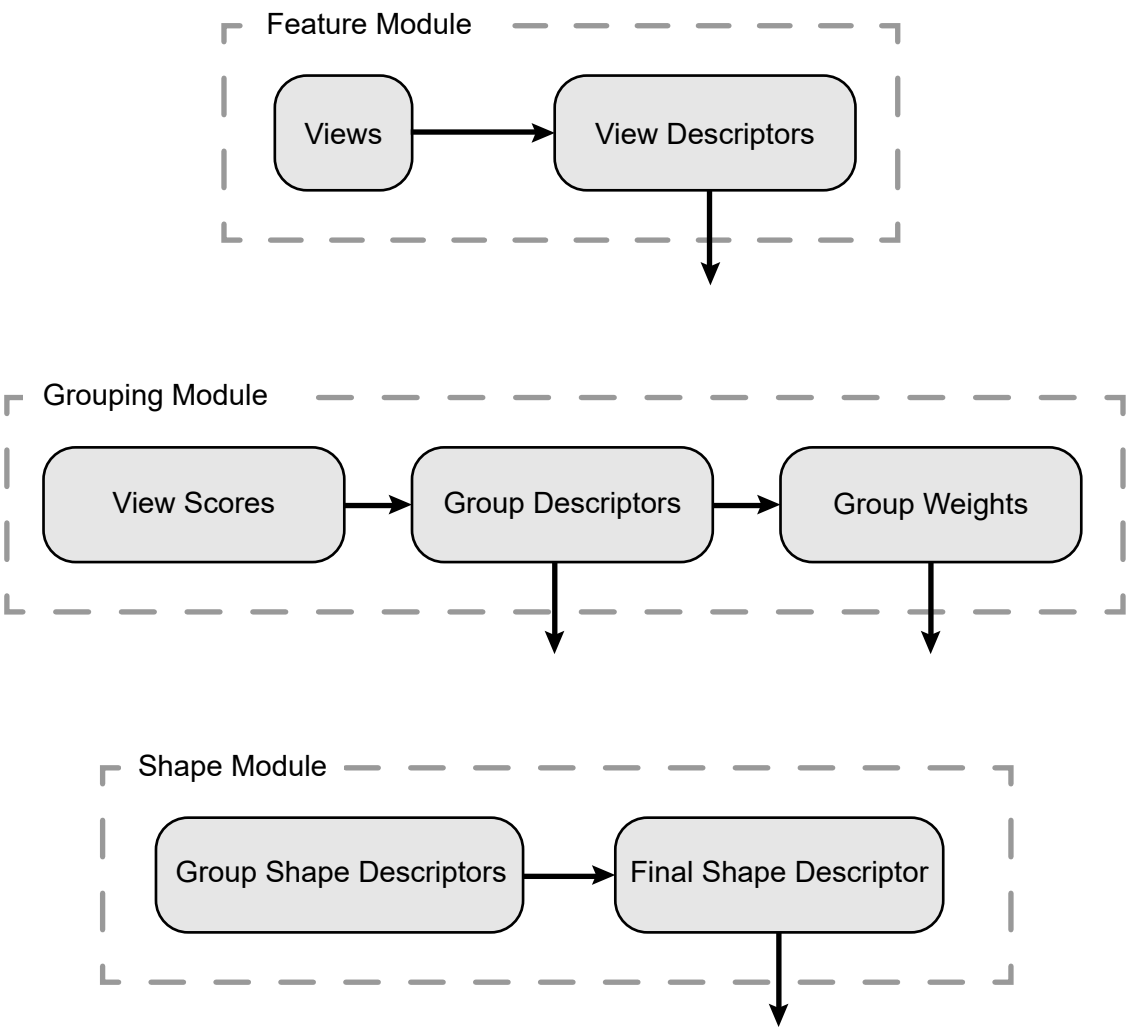
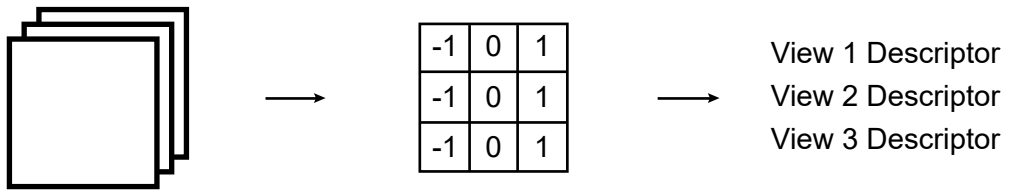


Figure 4.5: Modules of the Multi-View Architecture



Views | Apply Convolutions, Poolings and Activations | Results in a Descriptor per View

Figure 4.6: Basic Concept of the Feature Module

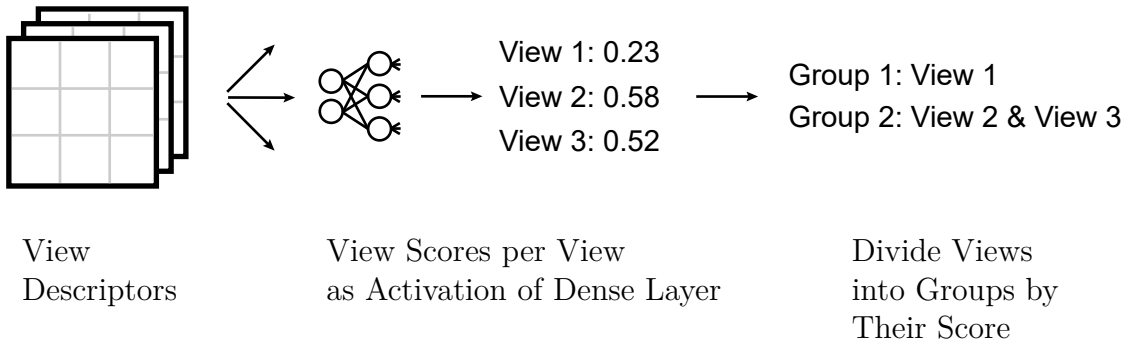


Figure 4.7: Group Creation and View Sorting in Grouping Module

match the number of elements in a column in the flattened filter matrix. Multiplying the filter matrix and the image patch vector yields the convolution results for the current window. This is repeated for the whole input resulting in a matrix with a size of $Batch \times Height\_Out \times Width\_Out \times Filter\_Height \cdot Filter\_Width \cdot Channels\_In$. Furthermore, to each convolution output the corresponding bias is added. This result is fed into a ReLU activation function. Finally, the outputs are max-pooled with a window of size 3×3 and a stride of 2. No padding is applied as well. It is defined, that the max-pooling is always performed on the last dimension, i. e. the one containing each feature. The next layers are similar including the bias addition and the ReLU activation function. Hence, only the operations and their parameters will be mentioned. Furthermore, the layers are added sequentially. That means, the activations of the previous main layer are the input of the current main layer and so on. The second main layer performs a convolution with 256 filters of size 5×5 and a stride of 2. However, this time the input is padded in a way, that the output has the original input's size. The max-pooling uses again a window of 3×3 and a stride of 2. The valid padding technique is applied. The third and fourth main layer use 384 filters of size 3×3 for the convolution task with a stride of 1 each and the padding technique same. However, no pooling is performed. For the last main layer, the fifth one, a convolution with 256 filters of size 3×3 is performed. The stride is 1 and the padding technique same. The output's dimension is reduced with a valid max-pooling of size 3×3 and a stride of 2. In the end, this whole process results in a tensor containing each view descriptor $\boldsymbol{V}_i$ of size $6 \times 6 \times 256$ of every batch element. Hence, this tensor's shape is $Batch \times Views \times 6 \times 6 \times 256$.

4.3.2 Grouping Module: Generating Group Descriptors

The objective of the grouping module is grouping several view descriptors of an object depending on their information content. The view descriptors $\boldsymbol{V}_i$ of each group $\mathbb{G}_i$ are then combined to a group descriptor $\boldsymbol{G}_i$. Hence, the module's input are the view descriptors coming from the feature module. First, the informational content of each

view needs to be calculated. The simplest and most intuitive way is to give every view a single number representing its score of discrimination. One approach would be to make the score directly depend on the pixel values. This could be performed with a fully-connected layer with the pixels as inputs and the score as output. However, like with fully-connected neural networks, this leads to a stiffness of the network due to its translation-variance of input values. This could be overcome by using convolutions first for extracting features. However, such features are already extracted, presumably much more accurate than a few convolutions for the score would do. Hence, the followed approach is that each view's score depends on its latest descriptor. It is worth noting, that in [14] the view scores do not depend on the last convolution. However, their network architecture uses more than five convolutional layers, but their scores depend on the fifth one. This is probably because later features are too detailed and would result in too divergent discrimination scores for a reliable group generation. Each latest view descriptor is fed into the same single fully-connected layer with 1 node. Therefore, the view descriptor matrix is flattened into a row-wise vector, which is multiplied with the corresponding weight matrix $\boldsymbol{W}^{[d4]}$ of shape $6 \cdot 6 \cdot 256 \times 1$. This results in a single value to which a bias $b^{[d4]}$ is added. A dropout is not performed, because with only one node it is not desirable. Dropping out this node represents a view discrimination score of 0. Of course, this would generalize the view score, but a kind of overfitting on detailed features is desirable for evaluating views. It is supposed that those discriminative features are reoccurring in different views, if the latter is actually discriminative. Thus, the more discriminative features a view has, the more discriminative it is. Hence, they should be kept and trained on. Finally, the weighted sum of a node is fed into its activation function. It was found during training, that the unit died at the beginning of the training most of the time when using ReLU activation functions. This is due to its characteristic. If the weighted sum is zero or below at the beginning of the training due to an unsuited weight initialization, the activation is zero, hence, the neuron dies immediately. Another reason could be a too large learning rate, allowing the weights to update in too big steps leading to a weighted sum of 0 or below. This results in a view score of 0 for every view that is not changed during later training steps due to a gradient of 0. The learning rate should suit the whole network and not only this layer, though. So, in contrast to every other layer in this network, leaky ReLU activation functions are applied to those units. Their gradient is always unequal to zero, hence, it is solving the dying ReLU unit problem.

For interpretation purposes the activation is squeezed into a range from 0 to 1 using the sigmoid function according to [14] representing a probability of discrimination. Because the sigmoid function saturates at values higher than around $|5|$, but the activation of the fully-connected layer are assumed to be larger, the natural logarithm of the activation is computed first. This shifts the saturation to values that are presumably not in the range of the activation. For having a continual function the absolute value of the activation is

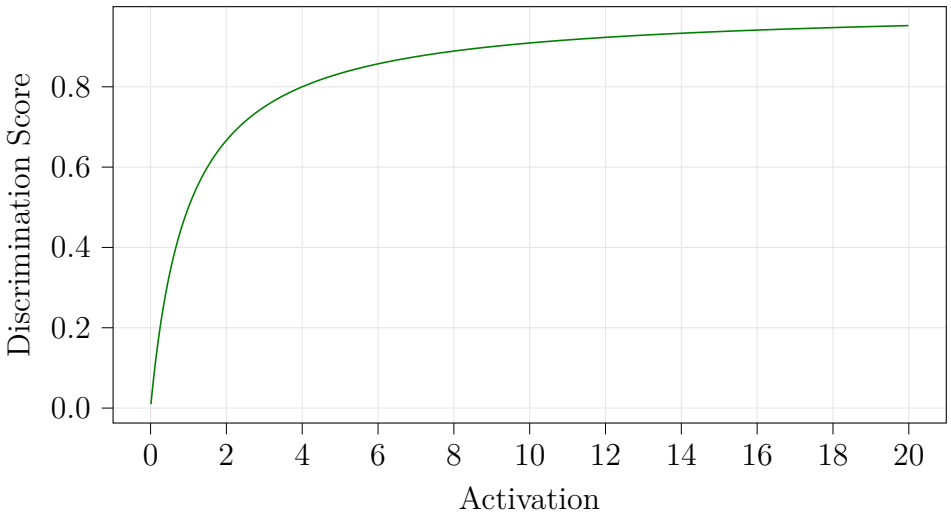


Figure 4.8: View Discrimination Score Function Plot

taken beforehand. This yields the expression for a view's score

$$s = \text{sigmoid}\left(\log\left(\left|a\right| + \varepsilon\right)\right) \tag{4.9}$$

where a is the output or activation, respectively, of the fully-connected layer. The small constant $\varepsilon = 10^{-6}$ is added for numerical stability for avoiding $\log(0)$. A plot of this function is shown in Fig. 4.8. All view discrimination scores for a batch are stored in a tensor with shape $Batch \times Views$.

Dependent on each view score, the related views are divided into groups. For maximum flexibility the number of groups n_g equals the number of views n_v. Hence, the size of each group r_g is related to the number of views n_v per object and the range of possible view scores r_s. With the limit of (4.9)

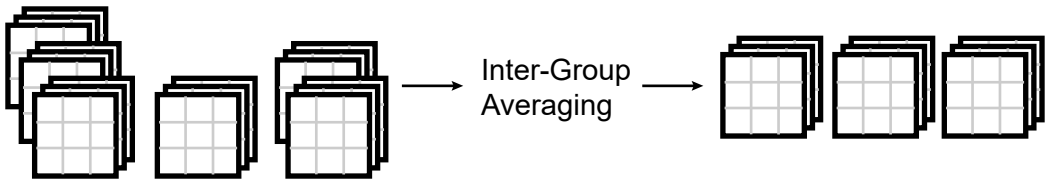
$$\lim_{x \to \infty} \text{sigmoid}\left(\log\left(\left|a\right| + \varepsilon\right)\right) = 1 \tag{4.10}$$

and only positive values from the ReLU, scores are in the range $r_s = 1 - 0 = 1$. Dividing r_s in equal sized parts yields

$$r_g = \frac{r_s}{n_v} = \frac{1}{12} \approx 0.083 \tag{4.11}$$

as each group's size. Hence, a group $\mathbb{G}_g$ contains views with scores of $(g-1) \cdot r_g \leq s < g \cdot r_s$ where $g = 1, 2, \ldots, n_v$. Fig. 4.7 illustrates the basic view score calculation and group dividing. This example assumes a group size of 0.33. The group to which each view $\mathbb{V}_i$ belongs is calculated by

$$g = \text{floor}\left(\frac{s_i}{r_g}\right) \tag{4.12}$$



View Descriptors per Group

Single Group Descriptor per Group

Figure 4.9: Generating Group Descriptors by Calculating the Average of Related Features

yielding the group's index. Those are stored for every view resulting in the group index vector $\boldsymbol{g} = (g_1, g_2, \cdots, g_{n_v})^T$. The subscripts correspond to the view indices, e.g. g_2 belongs to $\boldsymbol{V}_2$. With this approach it is possible for groups to remain empty. Based on the group indices, related view descriptors are averaged across their first dimension. This means, every channel of one view descriptor is averaged element-wise with the corresponding channel of the other view descriptors. In brief, every feature is averaged. This is illustrated in Fig. 4.9. A normal average is chosen, because all views in a group should have similar extracted features, e.g. the left and right side of a car. If the maximum of all features were taken, the group would presumably contain all views where important features were extracted for creating a group descriptor containing as many features as possible. However, this is not desirable for the use case. This results in a group descriptor

$$\boldsymbol{G}_i = \frac{\sum_{\boldsymbol{D} \in \mathbb{G}_i} \boldsymbol{D}}{|\mathbb{G}_i|} \tag{4.13}$$

with the same size as a view descriptor, where $\boldsymbol{D}$ is a view of group $\mathbb{G}_i$. The addition and division is calculated element-wise. However, the views of different objects are not necessarily divided into the same number of groups, thus, leading to a different number of group descriptors. Due to tensorflow's constraint, that a tensor is not allowed to change its shape in a graph during execution, additional empty group descriptors need to be created for reaching the maximum possible number of group descriptors of n_v. So, the final shape of the tensor $\bar{\boldsymbol{G}}$ containing all group descriptors is $Batch \times n_v \times 6 \times 6 \times 256$.

Every group $\boldsymbol{G}_i$ gets a weight w_i assigned representing its discrimination. This depends on the scores of its contained views. Analog to before, views of a group are found by checking the group index vector $\boldsymbol{g}$. This time the related view scores are summed up and divided by their number for calculating a mean. In mathematical terms,

$$\hat{w}_i = \frac{\sum_{\boldsymbol{V} \in \mathbb{G}_i} s(\boldsymbol{V})}{|\mathbb{G}_i|} \tag{4.14}$$

calculates the weight of the i-th group, where $s(\boldsymbol{V})$ is the score of a view $\boldsymbol{V}$ in $\mathbb{G}_i$.

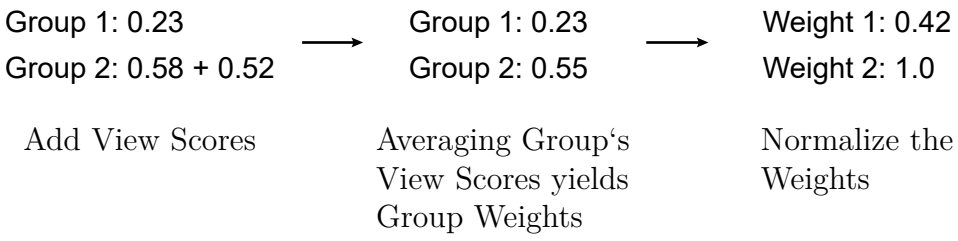


Figure 4.10: Calculation of Group Weights in Grouping Module

Furthermore, the weight of a group is normalized by

$$w_i = \frac{\hat{w}_i}{\max(s(\mathbb{G}_i))} \tag{4.15}$$

to a range of 0 and 1, where $\max(s(\mathbb{G}_i))$ yields the maximum score of the given group, for being able to compare it with the ones in [14]. The calculation of the group weights is illustrated in Fig. 4.10 referring to the example in Fig. 4.7. The weights of the padded group descriptors equal 0 for being ignored in a later matrix multiplication. Hence, the shape of the tensor $\bar{\boldsymbol{W}}$ storing all group weights is $Batch \times n_v$.

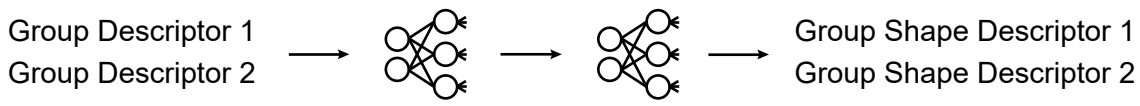
4.3.3 Shape Module: Generating a Shape Descriptor

The objective of the shape module is combining the group descriptors to a single shape descriptor that can be used for the classification. This descriptor contains every important feature from all views and groups. As usual, the tensor $\bar{\boldsymbol{G}}$ containing all the group descriptors of every batch is processed by each batch element. This batch element $\boldsymbol{G}$ of size $n_v \times 6 \times 6 \times 256$ contains the n_v group descriptors of an object. This tensor is split across its first dimension, i.d. across the group descriptors, resulting in a tensor for each group descriptor. Each group descriptor is then fed into a fully-connected sub-network with two layers. The first layer has $6 \cdot 6 \cdot 256$ edges per unit and 4096 units in total, while the second layer has 4069 edges per unit and 4069 output units as well. The inputs are processed like in the fully-connected layer before. First, each input $\boldsymbol{G}_i$ is flattened into a row-vector $\boldsymbol{g}_i$. Then, a matrix multiplication of the inputs and the corresponding weight matrix is performed and a bias vector is added. This result is fed into a ReLU activation function $\phi(\cdot)$ resulting in an activation for the particular layer. In conclusion, the two fully-connected operations

$$\boldsymbol{a}_i^{[d1]} = \phi(\boldsymbol{g}_i \boldsymbol{W}^{[d1]} + \boldsymbol{b}^{[d1]}) \tag{4.16a}$$

$$\boldsymbol{a}_i^{[d2]} = \phi(\boldsymbol{a}_i^{[d1]} \boldsymbol{W}^{[d2]} + \boldsymbol{b}^{[d1]}) \tag{4.16b}$$

are performed as illustrated in Fig. 4.11. Furthermore, both layers contain a dropout layer with a dropout probability of 0.5. This corresponds to the original AlexNet configuration. Hence, the final activations of the seventh layer in the network or second dense



Group Descriptors From Grouping Module

Dense Layer Operation Results in a Shape Descriptor per Group

Figure 4.11: Generate Group Shape Descriptors in Shape Module. Each group descriptor is fed into two fully-connected layers representing layer 6 and 7 of the network. The activation of layer 7 represents the shape descriptor of each group descriptor.

layer, respectively, represent the shape descriptor of every group descriptor. Those single shape descriptor vectors of each group are then again stacked along the first dimension for having a compact representation $\boldsymbol{S}_G$ with size $n_v \times 4096$. Expanding this with a batch dimension yields the tensor $\bar{\boldsymbol{S}}_G$ with size $Batch \times n_v \times 4096$.

Now the group shape descriptors need to be combined for generating the final single shape descriptor of the object. This is done by considering the group weights $\boldsymbol{w}$ with n_v elements calculated in the grouping module. As a reminder, they are the mean of all view scores of each group and, thus, an indicator for the group's discrimination. Hence, a weighted average is calculated for considering this relation. For having a valid matrix multiplication, the group shape descriptor $\bar{\boldsymbol{S}}_G$ needs to be transposed while keeping the batch dimension as the first one. Hence its size changes from $Batch \times n_v \times 4096$ to $Batch \times 4096 \times n_v$. Furthermore, the group weights tensor $\bar{\boldsymbol{W}}$ is expanded with a third dimension yielding the shape $Batch \times n_v \times 1$. For this weighted average calculation a tensor holding the sums of the weights in inevitable. Thus, $\bar{\boldsymbol{W}}_{sum}$ stores the sum of the group weights tensor along its first dimension, i.e. each element is the sum of all group weights of an object. For a valid matrix division a third dimension must be expanded as well. This yields a tensor $\bar{\boldsymbol{W}}_{sum}$ of the shape $Batch \times 1 \times 1$ with weight sums. Now the weighted average can be computed by

$$\bar{\boldsymbol{S}} = \frac{\bar{\boldsymbol{S}}_G \bar{\boldsymbol{W}}}{\bar{\boldsymbol{W}}_{sum}} \tag{4.17}$$

where the division is performed element-wise. Due to the matrix multiplication, the padded entries have no impact. The result has a shape of $Batch \times 4096 \times 1$ and represents the final single shape descriptor. This can be made more compact by changing the shape to $Batch \times 4096$ without loosing any information, because the number of elements stays the same.

This representation is directly compatible with the last fully-connected layer, which is responsible for calculating the predictions of the network, thus, $\bar{\boldsymbol{S}}$ is fed into it. The number of neurons in this layer equals the number of different labels or categories, respectively, where each neuron has 4096 edges. The performed operations are identical

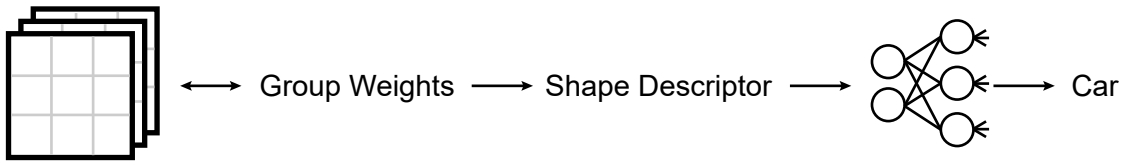


Figure 4.12: Basic Concept of the Shape Module Combined with a Classification. A weighted average of the group shape descriptors with the related group weights is calculated yielding a single shape descriptor. It is fed into the last fully-connected layer resulting in the prediction of the network.

to the ones earlier and are described by

$$\boldsymbol{z}^{[d3]} = \bar{\boldsymbol{S}}\boldsymbol{W}^{[d3]} + \boldsymbol{b}^{[d3]} \tag{4.18}$$

using the related weights and biases. However, no activation function is applied here. For making the predictions $\boldsymbol{z}^{[d3]}$ interpretable, they are fed into a softmax layer. This outputs a valid probability distribution $\hat{\boldsymbol{y}}$ depending on all its inputs $\boldsymbol{z}^{[d3]}$. Hence, this results in a membership probability for the network's input to each class. The class representing the index with the largest value in $\hat{\boldsymbol{y}}$ is considered the predicted class. The functionality of the shape module is summarized in Fig. 4.12.

4.4 Training the Architecture

The multi-view network architecture is trained by inputting the generated multi-view images $\boldsymbol{X}^{(i)}_{train}$ and $\boldsymbol{X}^{(i)}_{test}$ and comparing the prediction $\hat{\boldsymbol{y}}^{(i)}$ of the network with the corresponding one-hot encoded labels $\boldsymbol{y}^{(i)}_{train}$ and $\boldsymbol{y}^{(i)}_{test}$. However, the test set is only used for supervising the training process for now. This is the general idea that will be explained more detailed in the following.

First, a batch size needs to be chosen to define how many samples will be propagated through the network at once. Batch sizes of 1 or the full number of samples of the training set will be avoided due to issues like time of convergence and memory size. In a temporary training using only single-views a batch size of 128 could be achieved. Thus, using $n_v = 12$ views the batch size is reduced to $b = floor(128/12) = 10$. However, due to a limited memory size of 8GB of the experimental setup's GPU and the additional parameters for the multi-view training only a batch size of 8 supports training reliably. Because this is way less than the recommended sizes but the maximum possible, this size is chosen. This yields $n_{b,train} = m_{train}/8$ batches for the training set and $n_{b,test} = m_{test}/8$

for the testing set. Nevertheless, dividing each set into 8 samples will be odd in general. Thus, the last batch is filled with all remaining samples. After each epoch, the training set is shuffled so that batches do not contain the same samples as before. This is done by combining corresponding multi-view images and labels to a list like

$$\tilde{\boldsymbol{L}} = \left(\left[\boldsymbol{X}_{test}^{(1)}, \boldsymbol{y}_{test}^{(1)} \right], \left[\boldsymbol{X}_{test}^{(2)}, \boldsymbol{y}_{test}^{(2)} \right], \cdots, \left[\boldsymbol{X}_{test}^{(m_{test})}, \boldsymbol{y}_{test}^{(m_{test})} \right] \right) \tag{4.19}$$

where each pair builds a list element for experiencing the same operations. This list is then randomly shuffled element-wise and split again into multi-view images and labels. For simplicity a sample in the shuffled list is still referred to by its current index.

For calculating the cost and the derivatives of the parameters a softmax cross entropy is performed in the single-label classification case. For efficiency tensorflow applies a softmax internally, so the unscaled predictions need to be fed. Then, the softmax measures the probability error between the prediction and ground-truth, while assuming mutually exclusive labels. In the multi-label classification case, a sigmoid cross entropy is applied. Here the sigmoid is calculated internally and not mutually exclusive classes are assumed. For updating the parameters with the goal of minimizing the cost function the Adam optimizer is employed. One of its advantages is adapting a learning rate for each parameter, which is supposed to achieve better results in such a network with many parameters. Moreover, in many recent researches it outperforms the classical stochastic gradient approach due to fixing its downsides, hence, it is supposed to be valid here as well.

The prediction $\hat{\boldsymbol{Y}}^{(i)}$ of the network needs to be compared to the ground-truth labels $\bar{\boldsymbol{Y}}^{(i)}$ of the batch sample i for checking the network's accuracy. Due to the batch operations, the single dataset samples in a batch are referred to as batch samples. How the comparing is performed, though, depends on the type of classification. In the case of a single-label one, the index of the largest value in the batch element prediction $\hat{\boldsymbol{y}}^{(i)}$ is located. The same operation is performed on the corresponding ground-truth label $\boldsymbol{y}^{(i)}$. Each index represents a certain class, which is declared the predicted or actual one, respectively. Now a binary comparison of both indices is performed, resulting in 0 if they are different and 1 if they are equal. This is repeated for each batch element, while storing all results in a vector $\boldsymbol{e}$. Finally, the accuracy α is calculated by

$$\alpha^{(i)} = \frac{\sum_j e_j}{|\boldsymbol{e}|} \tag{4.20}$$

for the current batch i. In the case of a multi-label classification, a probability threshold needs to be defined when a predicted feature is actually considered predicted. In this case, the threshold is $p = 0.5$, hence, the values of the prediction vector can be rounded. Now an identical binary comparison as before can be applied to both label vectors resulting in a vector $\boldsymbol{e}$ as well. The accuracy is calculated with Theorem 4.20.

Furthermore, a starting learning rate is necessary. Because finding it by trail-and-error would be time-consuming, the approach of the cyclical learning rate is used for finding

an optimal learning rate. Hence, the learning rate is initialized with $\gamma = 10^{-5}$. After processing each batch it is exponentially increased according to

$$\gamma(\tau + 1) = \alpha\gamma(\tau) \tag{4.21}$$

where $\alpha = 1.1$ is the scaling factor and τ the iteration. Its value can be chosen arbitrarily, but should be in range for achieving a desirable precision in learning rates. For each learning rate, the related cost function evaluation is stored. Training is stopped when the last cost value is four times the second to last one, i.e. when a drastic deterioration in cost happens. For evaluation the cost values are plotted against the learning rates. On the basis of this, the range of optimal learning rates can be read where a steep descent in cost values happen.

4.5 Evaluating the Architecture

Plenty of data needs to be collected for evaluating the overall performance of the network. Fortunately, every tensor can be gathered, though, some need to be manipulated for being interpretable. The most important tensor contains the cost of every iteration of the training set, because this is attempted to be minimized, because it represents how well the network classifies the data. Each one is stored during training for being able to plot them afterwards. Furthermore, after each epoch the cost and accuracy of the whole training set and the whole testing set are calculated with current parameters. This is done by computing each one for each batch and averaging the results for each set. However, the last batch has in general less elements than the ones before. Hence, a weighted average is performed with the batch sizes as the weights. This yields the averaged cost

$$J\left(\hat{\bar{\boldsymbol{Y}}}, \bar{\boldsymbol{Y}}\right) = \frac{\sum_i^{n_b} n_{b_i} \cdot J\left(\hat{\bar{\boldsymbol{Y}}}^{(i)}, \bar{\boldsymbol{Y}}^{(i)}\right)}{|\boldsymbol{n_b}|} \tag{4.22}$$

and the averaged accuracy

$$\alpha\left(\hat{\bar{\boldsymbol{Y}}}, \bar{\boldsymbol{Y}}\right) = \frac{\sum_i^{n_b} n_{b_i} \cdot \alpha\left(\hat{\bar{\boldsymbol{Y}}}^{(i)}, \bar{\boldsymbol{Y}}^{(i)}\right)}{|\boldsymbol{n_b}|} \tag{4.23}$$

where $|\boldsymbol{n_b}|$ is a vector containing the batch sizes. As mentioned, this is performed separately on the training set and the testing set. Those results are also stored for plotting purposes. Comparing both related units can reveal if training should be continued or if overfitting or underfitting occurs. Because the cost value is more general and the accuracy rather for practical purposes, the first one is examined. The effect could be seen on both, though. If the training loss decreases while the training loss decreases as well, the network improves and training should be continued. However, if the training

loss decreases while the testing loss increases, overfitting occurs. The network does not generalize, but focuses on the features in the training set, hence, never seen data like the testing set cannot be classified properly. At the end of training, the accuracy of the training set is defined as the performance of the network. Furthermore, all plots are saved to disk. To plots, whose shown values $\boldsymbol{p}$ oscillate, a moving average $\boldsymbol{q}$ of the values is added. A single averaged value is calculated by

$$q_i = \frac{\sum_{j=\max(1,i-s+1)}^{i} p_j}{\max(0, i-s)} \tag{4.24}$$

where s is the window size, that defines how many values are taken into account for averaging a certain sample. If the window is larger than the available number of samples, in particular in the beginning, the window size is adapted temporarily. By default it is set to $s = \text{floor}\,(0.1\,|\boldsymbol{p}|)$ for a dynamical size.

For evaluating the whole network model regarding its practical use, the accuracies for each class, each category and each material are calculated. In general, the accuracy states how many samples are classified correctly. However, the overall accuracy can be misleading, although each class has almost the same number of objects, because it does not represent if certain classes are better classified than others. Hence, the precision and recall score is computed for each class as well. Furthermore, a confusion matrix is calculated containing every sample's prediction. It is plotted against the ground-truth classes, where predictions of identical classes are counted up. The grouping module needs to be evaluated as well. It is supposed that in a classification of only the materials, the views with visible material manipulations belong to the most weighted group. Hence, it is sufficient to examine if the most discriminative group only contains views with visible manipulations. Because the colors of the manipulated features are known, each pixel of a view in the top group is checked if it matches such a RGB value. If there is a match with at least one pixel of a view, the view is considered grouped correctly. All correctly grouped views TP and not correctly grouped views FP are counted. Based on them a percentage

$$\alpha = \frac{TP}{TP + FP} \tag{4.25}$$

is calculated representing the accuracy of the grouping module for a given multi-view input. This is repeated for all multi-views of the same material. The final accuracy for that material results from averaging the accuracies of every multi-view of that material. This is performed for every material to analyze if different colored material manipulations yield different results.

For predicting the classes and gathering information of certain inputs, the file name of a single view of each object needs to be given. By splitting the filename into three parts, where the second is the view index, the third the extension and the first the remaining part, it can be combined again to represent the filename of each view by replacing the view index. Those files are then combined to a multi-view image. Performing this on

each sample results in a common input tensor $\bar{\boldsymbol{X}}^{(i)}$ representing a batch. If the number of samples, that are going to be predicted, exceeds the defined batch size, they need to be divided into an appropriate number of batches. This tensor is now propagated through the network as usual. Meanwhile, the activations of the first convolutional layer are stored for visualizing the extracted features in each view. Therefore, the each view's activations are split across the last dimension that represents the features. Each feature's values $\boldsymbol{F}_i$ are normalized by

$$\boldsymbol{F}_{i,norm} = \frac{\boldsymbol{F}_i - \min(\boldsymbol{F}_i)}{\max(\boldsymbol{F}_i) - \min(\boldsymbol{F}_i)} \tag{4.26}$$

to a range of 0 and 1 for making it visualizable. Finally, each feature is saved as an gray-scale image. Moreover, each view discrimination score, group weight and group index is stored for showing them with their associated view afterwards. Furthermore, a saliency map $\boldsymbol{S}_j^{(i)}$ is computed for each view $\boldsymbol{V}_j^{(i)}$ showing how much each pixel influences the raw output of the network $\boldsymbol{z}^{(i)}$. Here, the direct output of the eighth layer is taken, that means no softmax is applied. A single saliency map can be defined as the derivative of the output with respect to a single view yielding

$$\boldsymbol{S}_j^{(i)} = \frac{\partial \boldsymbol{z}^{(i)}}{\partial \boldsymbol{V}_j^{(i)}} \tag{4.27}$$

as a general expression. For plotting each saliency map is normalized with Theorem 4.26 and visualized in gray-scale.

Chapter 5

Results

5.1 View to Group Classification

5.2 Overall Performance

5.3 Misclassified Predictions

Chapter 6

Discussion

6.1 Conclusions

6.2 Outlook

Bibliography

[1] Osada, Robert ; Funkhouser, Thomas ; Chazelle, Bernard ; Dobkin, David: Matching 3D Models with Shape Distributions. In: *Proceedings of the International Conference on Shape Modeling & Applications*. Washington, DC, USA : IEEE Computer Society, 2001 (SMI '01). – ISBN 0–7695–0853–7, 154–

[2] Chen, Ding-Yun ; Tian, Xiao-Pei ; Shen, Yu-Te ; Ouhyoung, Ming: On Visual Similarity Based 3D Model Retrieval. In: *Computer Graphics Forum* (2003). http://dx.doi.org/10.1111/1467-8659.00669. – DOI 10.1111/1467–8659.00669. – ISSN 1467–8659

[3] Shu, Zhenyu ; Xin, Shiqing ; Xu, Huixia ; Kavan, Ladislav ; Wang, Pengfei ; Liu, Ligang: 3D Model Classification via Principal Thickness Images. In: *Comput. Aided Des.* 78 (2016), September, Nr. C, 199–208. http://dx.doi.org/10.1016/j.cad.2016.05.014. – DOI 10.1016/j.cad.2016.05.014. – ISSN 0010–4485

[4] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. USA : Curran Associates Inc., 2012 (NIPS'12), 1097–1105

[5] Simonyan, K. ; Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: *International Conference on Learning Representations*, 2015

[6] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep Residual Learning for Image Recognition, 2016, S. 770–778

[7] Szegedy, Christian ; Ioffe, Sergey ; Vanhoucke, Vincent ; Alemi, Alex A.: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In: *ICLR 2016 Workshop*, 2016

[8] Russakovsky, Olga ; Deng, Jia ; Su, Hao ; Krause, Jonathan ; Satheesh, Sanjeev ; Ma, Sean ; Huang, Zhiheng ; Karpathy, Andrej ; Khosla, Aditya ; Bernstein, Michael ; Berg, Alexander C. ; Fei-Fei, Li: ImageNet Large Scale Visual Recognition Challenge. In: *Int. J. Comput. Vision* 115 (2015), Dezember, Nr. 3, 211–252. http://dx.doi.org/10.1007/s11263-015-0816-y. – DOI 10.1007/s11263–015–0816–y. – ISSN 0920–5691

[9] Chatfield, Ken ; Simonyan, Karen ; Vedaldi, Andrea ; Zisserman, Andrew: Return of the Devil in the Details: Delving Deep into Convolutional Nets. In: *CoRR* abs/1405.3531 (2014). http://dblp.uni-trier.de/db/journals/corr/corr1405.html#ChatfieldSVZ14

[10] Su, Hang ; Maji, Subhransu ; Kalogerakis, Evangelos ; Learned-Miller, Erik: Multi-view Convolutional Neural Networks for 3D Shape Recognition. In: *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*. Washington, DC, USA : IEEE Computer Society, 2015 (ICCV '15). – ISBN 978–1–4673–8391–2, 945–953

[11] Wu, Zhirong ; Song, Shuran ; Khosla, Aditya ; Yu, Fisher ; Zhang, Linguang ; Tang, Xiaoou ; Xiao, Jianxiong: 3D ShapeNets: A deep representation for volumetric shapes. In: *CVPR*, IEEE Computer Society, 2015. – ISBN 978–1–4673–6964–0, 1912-1920

[12] Su, Jong-Chyi ; Gadelha, Matheus ; Wang, Rui ; Maji, Subhransu: A Deeper Look at 3D Shape Classifiers. In: *CoRR* abs/1809.02560 (2018)

[13] Hegde, Vishakh ; Zadeh, Reza B.: FusionNet: 3D Object Classification Using Multiple Data Representations. In: *CoRR* abs/1607.05695 (2016)

[14] Feng, Yifan ; Zhang, Zizhao ; Zhao, Xibin ; Ji, Rongrong ; Gao, Yue: GVCNN: Group-View Convolutional Neural Networks for 3D Shape Recognition. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018

[15] Szegedy, Christian ; Liu, Wei ; Jia, Yangqing ; Sermanet, Pierre ; Reed, Scott ; Anguelov, Dragomir ; Erhan, Dumitru ; Vanhoucke, Vincent ; Rabinovich, Andrew: Going Deeper with Convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, S. 1–9

[16] Cyr, Christopher M. ; Kimia, Benjamin B.: A Similarity-Based Aspect-Graph Approach to 3D Object Recognition. In: *Int. J. Comput. Vision* 57 (2004), April, Nr. 1, 5–22. http://dx.doi.org/10.1023/B:VISI.0000013088.59081.4c. – DOI 10.1023/B:VISI.0000013088.59081.4c. – ISSN 0920–5691

[17] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[18] Li, Fei-Fei ; Karpathy, Andrej ; Johnson, Justin: *CS231n: Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/. Version: Spring 2019

[19] 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

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

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

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

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

[24] 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

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

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

[27] 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

[28] 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

[29] Srivastava, Nitish ; Hinton, Geoffrey ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In: *J. Mach. Learn. Res.* 15 (2014), Januar, Nr. 1, S. 1929–1958. – ISSN 1532–4435

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

[31] 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

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

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

[34] 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

[35] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; 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

[36] Murphy, Kevin P.: *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.] : MIT Press, 2013 https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020/ref=sr_1_2?ie=UTF8&qid=1336857747&sr=8-2. – ISBN 9780262018029 0262018020

[37] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning representations by back-propagating errors. In: *Nature* 323 (1986), Oktober, 533–. http://dx.doi.org/10.1038/323533a0

[38] Kiefer, J. ; Wolfowitz, J.: Stochastic Estimation of the Maximum of a Regression Function. In: *The Annals of Mathematical Statistics* 23 (1952), 09, Nr. 3, 462–466. http://dx.doi.org/10.1214/aoms/1177729392. – DOI 10.1214/aoms/1177729392

[39] Robbins, Herbert ; Monro, Sutton: A Stochastic Approximation Method. In: *The Annals of Mathematical Statistics* 22 (1951), 09, Nr. 3, 400–407. http://dx.doi.org/10.1214/aoms/1177729586. – DOI 10.1214/aoms/1177729586

[40] Bengio, Yoshua: Practical recommendations for gradient-based training of deep architectures. In: *CoRR* abs/1206.5533 (2012). http://arxiv.org/abs/1206.5533

[41] Choromanska, Anna ; Henaff, Mikael ; Mathieu, Michaël ; Arous, Gérard Ben ; LeCun, Yann: The Loss Surface of Multilayer Networks. In: *CoRR* abs/1412.0233 (2014). http://arxiv.org/abs/1412.0233

[42] Tieleman, Tijmen ; Hinton, Geoffrey: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.

[43] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015

[44] Smith, Leslie N.: No More Pesky Learning Rate Guessing Games. In: *CoRR* abs/1506.01186 (2015). http://arxiv.org/abs/1506.01186

[45] Loshchilov, Ilya ; Hutter, Frank: SGDR: Stochastic Gradient Descent with Restarts. In: *CoRR* abs/1608.03983 (2016). http://arxiv.org/abs/1608.03983

[46] Keskar, Nitish S. ; Mudigere, Dheevatsa ; Nocedal, Jorge ; Smelyanskiy, Mikhail ; Tang, Ping Tak P.: On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In: *CoRR* abs/1609.04836 (2016). http://arxiv.org/abs/1609.04836

[47] Fawcett, Tom: An Introduction to ROC Analysis. In: *Pattern Recogn. Lett.* 27 (2006), Juni, Nr. 8, 861–874. http://dx.doi.org/10.1016/j.patrec.2005.10.010. – DOI 10.1016/j.patrec.2005.10.010. – ISSN 0167–8655

[48] Szegedy, Christian ; Liu, Wei ; Jia, Yangqing ; Sermanet, Pierre ; Reed, Scott ; Anguelov, Dragomir ; Erhan, Dumitru ; Vanhoucke, Vincent ; Rabinovich, Andrew: Going Deeper with Convolutions. In: *Computer Vision and Pattern Recognition (CVPR)*, 2015