**Master's Thesis**

TECHNISCHE UNIVERSITÄT DARMSTADT

# Analyzing the Information Content of Object Views in Multi-View Object Recognition 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

The objective of this work is to classify same objects with distinguishing color marks with a convolutional neural network. Each object is represented by a set of 2D views. That so-called multi-views outperform single object views is shown by [44]. The color marks include single and double marks. For each additional color mark, a new network is trained on the full dataset for being able to compare the results optimally. The core functionality of this work is based on the grouping mechanism from [12] that assigns each view a score depending on its discriminative content and divides them into groups of similar scores. For each group, a group descriptor is generated. Merging them yields a single compact shape descriptor of the object, that is used for classification. Based on those groups it is analyzed why views are more discriminative than others and which impact color marks have. Moreover, it is examined whether all networks share the same characteristics regarding the assignment of discriminative scores for views. The results show that the grouping mechanism works satisfiable and the networks classify the colors satisfiable with respect to accuracy. However, it is evident, that the networks get more complex with more classes, because the accuracies get slightly worse each time for the same number of epochs. It is noticed, that the networks weight same views but showing different color marks differently. Hence, it is assumed, that the final shape descriptor is divided into ranges representing each class.

---

# Zusammenfassung

Das Ziel dieser Arbeit ist es, identische Objekte mit unterscheidenden Farbmerkmalen mithilfe von Convolutional Neural Networks zu klassifizieren. Dabei ist jedes Objekt als Set von 2D Bildern beschrieben. Dass sogenannte Multi-Views einzelne Objektansichten übertreffen, zeigte [44]. Das Set an Merkmalen beinhalten einfache und zweifache Merkmale. Für jedes zusätzliche Farbmerkmal wird ein neues Netzwerk anhand des vollen Datensatzes trainiert, um die Ergebnisse bestmöglich vergleichen zu können. Das Herzstück dieser Arbeit ist ein Gruppierungsmechanismus nach [12], der jedem Bild eine Wertung anhand seines Informationsgehalts zuweist und es abhängig davon in eine Gruppe einteilt. Für jede Gruppe wird ein Gruppen-Deskriptor erstellt. Diese werden zu einem einzigen kompakten Form-Deskriptor zusammengeführt, der für die Klassifizierung verwendet wird. Anhand der Gruppeneinteilung wird untersucht, warum einige Ansichten einen höheren Informationsgehalt besitzen als andere und welche Rolle die Farbmerkmale spielen. Desweiteren wird untersucht, ob alle Netzwerke dieselben Charakteristiken bezüglich der Bewertung von Ansichten teilen. Die Ergebnisse zeigen, dass der Gruppenmechanismus zufriedenstellend arbeitet und die Farbmerkmale zufriedenstellend unterschieden werden hinsichtlich der Klassengenauigkeit. Allerdings fällt dabei auf, dass die Netzwerke bei mehr Klassen komplexer werden, da sich die Genauigkeiten bei gleicher Trainingszeit leicht verschlechtern. Weiterhin wird bemerkt, dass gleiche Bilder, jedoch mit unterschiedlichen Farbmerkmalen, unterschiedlich bewertet werden. Daher wird die These aufgestellt, dass der kompakte Formdeskriptor in Bereiche aufgeteilt wird, die einzelne Klassen darstellen.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations and Symbols

### General

| | |
|---|---|
| $x$ | Scalar |
| $\boldsymbol{x}$ | Vector |
| $\boldsymbol{X}$ | Matrix |
| $\bar{\boldsymbol{X}}$ | Tensor with a Batch Dimension as the First One |

### Sizes

| | |
|---|---|
| $m$ | Number of Samples in the Dataset |
| $m_{\text{set}}$ | Number of Samples in Subset "set" |
| $n_x$ | Input Size |
| $n_y$ | Output Size |
| $n_h^{[l]}$ | Number of Hidden Units in the $l$-th Layer |
| L | Number of Layers in the Network |
| $s$ | Stride |
| $p$ | Padding |
| $\gamma$ | Learning Rate |
| $n_v$ | Number of Views per Multi-View |

### Objects

| | |
|---|---|
| $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n_x}$ | $i$-th Sample Represented as Column Vector |
| $\boldsymbol{X} \in \mathbb{R}^{n_x \times m}$ | Input Matrix Containing Samples as Column Vectors |

| | |
|---|---|
| $\tilde{\boldsymbol{X}} \in \mathbb{R}^{n_v \times n_x \times m/n_v}$ | Multi-View Input Matrix |
| $\boldsymbol{y}^{(i)} \in \mathbb{R}^{n_y}$ | Label for $i$-th Sample |
| $\boldsymbol{Y} \in \mathbb{R}^{n_y \times m}$ | Label Matrix Containing Labels as Column Vectors |
| $\tilde{\boldsymbol{Y}} \in \mathbb{R}^{n_v \times n_y \times m/n_v}$ | Multi-View Label Matrix |
| $w_{jk}^{[l]}$ | Weight of Edge Connecting $k$-th Unit in Layer $l-1$ with $j$-th Unit in Layer $l$ |
| $\boldsymbol{W}^{[l]} \in \mathbb{R}^{\text{number of units in l-th layer} \times \text{number of units in previous layer}}$ | Weight Matrix of Layer $l$ |
| $b_j^{[l]}$ | Bias of $j$-th Unit in $l$-th Layer |
| $\boldsymbol{b}^{[l]} \in \mathbb{R}^{\text{number of units in l-th layer}}$ | Bias Vector of $l$-th Layer |
| $\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^{n_y}$ | Prediction of $i$-th Sample |
| $\boldsymbol{K}$ | Filter Matrix |
| $\boldsymbol{F}$ | Feature Map Matrix |

### Expressions

| | |
|---|---|
| $\text{floor}(\cdot)$ | Rounding off to Integer Representation |
| $\phi(\cdot)$ | Activation Function |
| $\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}$ | Weighted Sum of Units in $l$-th Layer |
| $\boldsymbol{a}^{[l]} = \phi^{[l]}\left(\boldsymbol{z}^{[l]}\right)$ | Activation of Units in $l$-th Layer |
| $J(\hat{\boldsymbol{y}}, \boldsymbol{y})$ | Cost Function |

# Chapter 1

# Introduction

Researches showed that handcrafted 3D descriptors of objects are outperformed by using views of an object and generating 2D descriptors with the help of convolutional neural networks. Hence, this work follows this approach for classifying objects by collecting multiple views of it and building a multi-view image from those single view images. That multi-view discretizes the 3D object. However, not all of those views are equally relevant to the actual classification task. Hence, a score per view is calculated that describes its discrimination and its weight in the classification process. All views are divided into groups depending on their score. Then for each group, a group descriptor is calculated by averaging the group's views. Each group gets a weight assigned with the mean of its views discrimination scores. Finally, those groups are weighted averaged depending on their weights for building a compact single shape descriptor that describes the object. With this descriptor, the final class is predicted. The grouping mechanism represents the core functionality of this work. This is extended with applying color features to each object so that an object exists with its blank views and additionally with its colored ones. A real-world example could be a robot driving through a scene and needs to classify the same objects, that only differ in a color feature. As it progresses more views of each object become visible. Thanks to the grouping system it knows if a view is discriminative enough for a desirable classification or if it needs to collect more for being sure. Hence, this work examines how each view is treated and what are features the network looks for. If this is known it could be manipulated for the certain use case.

This work is structured as follows. In Section 2, the fundamentals are explained. It covers the general idea and development of artificial neural networks, followed by the concept of convolutional neural networks, that are more suited for image processing tasks. Furthermore, it is stated how data needs to be prepared, how it is propagated through a network and how the actual learning process works. Moreover, it introduces hyperparameters and how they need to be chosen for achieving a satisfiable network performance and continues with metrics that examine that performance. It finishes with a brief overview of the used software and framework. The third chapter Section 3 summarizes recent researches building the fundamentals for this work and supplying the knowledge for being able to choose an approach for this work. Section 4 presents this

work's approach and its implementation. This includes the creation of the dataset, the applying of face material manipulations and the conversion from single-views to multi-views. Furthermore, the network architecture based on [12] is explained in detail by dividing it into modules. It continues with how hyperparameters are chosen and finishes with how the network is evaluated. Section 5 presents all results divided into the overall performance of the networks and the grouping mechanism and discusses why wrong predictions occur. In the second part, in particular, a theory is set up for explaining why certain views are more discriminative than others. This work finishes with Section 6 that summarizes all results and gives an outlook on improving them.

# Chapter 2

# Fundamentals

This chapter covers the fundamentals for the methods presented and their application. It is divided into a section on neural networks and one on the used software and frameworks. The neural network section starts with the principle of neural networks. It continues with an explanation of the network architectures multilayer perceptrons and convolutional neural networks. The first serves as an example of how networks work in general. The second focuses on image processing and outperforms the first one in this task. Furthermore, it is explained how a network is trained for achieving a desired objective like object classification. 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 can help to improve the overall performance of a neural network. The second section explains which software and frameworks support building and training a model.

## 2.1 Artificial Neural Networks

This section investigates the types of neural networks that are important for this work. Furthermore, it explains how these types are structured and trained in order to fulfill an objective like a certain object classification. General knowledge for this is taken from [14] and [30].

### 2.1.1 Overview

Artificial neural networks are inspired by biological neural networks that constitute animal brains for recognizing patterns. Its task can be interpreted as being a universal approximator for any unknown function $f(x) = y$ where $x$ is the input and $y$ the output. Naturally, those variables are numeric. So every physical data like images, text or time series that is going to be used as an input or output 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 classes by recognizing patterns. The correct class of each input is given as an additional label. Therefore,

the network learns the correlation between input data and labels. A downside here with respect to effort is that every input must be labeled beforehand, usually by human knowledge. This type of learning is called supervised learning because each predicted class by the network of an input sample is compared with its given ground truth label. Use cases, for example, are the classification of cars or pedestrians in images or even the type of car in an image or whether an email is spam. In this work a classification is performed on images of objects with color marks.

Clustering divides data into clusters or groups, respectively, but without requiring labels. Therefore, this learning type is called unsupervised learning. So it is a classification task with dynamic class creation. Use cases are comparing data samples to each other and to find similarities or anomalies. Because unlabeled data occurs way more often than labeled data in real-world examples, a network could train on a broader range of related data independently of humans or given ground-truth labels, respectively, and probably gets more generalized than a classification one.

Regression is the prediction of an event, either current or in the future, by establishing correlations between past events and if existing additionally future events. A simple use case is the prediction of the 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.

### 2.1.2 Multilayer Perceptron

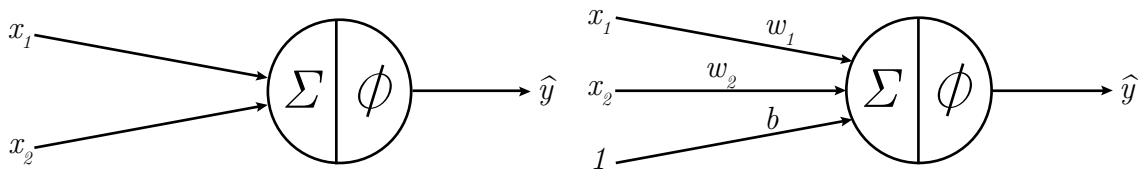
This section starts with an explanation of a single computational neuron and its development to become a perceptron and ends with an overview of the multilayer perceptron architecture.

#### Perceptron

McCulloch and Pitts [32] 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 can be the positive or correct class, respectively. This is called a binary classification. A schematic of this model can be seen in Fig. 2.1a. 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

$$\hat{y} = \phi\left(\sum_{i}^{n_x} x_i\right) \tag{2.1}$$

describes this operation where $n_x$ is the number of inputs, $x_i$ the $i$-th input and $\phi$ the used activation function, in this case, the threshold activation function. Plots of common




(a) Model of a McCulloch-Pitts-Neuron [32]. The inputs $x_i$ are summed up and put into the threshold activation function $\phi$ whose result is the neuron's output $\hat{y}$.

(b) Model of a perceptron [36]. The inputs $x_i$ are weighted by $w_i$ and summed up with the bias $b$. This sum is the argument of an activation function $\phi$ whose result is the perceptron's output $\hat{y}$.

Figure 2.1: Comparison of McCulloch-Pitts-Neuron and perceptron. The latter is based on the first but includes weights $w_i$ and a bias $b$ for weighting the inputs.

activation functions including their equations are shown in Fig. 2.21. 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.

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." [18] 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". Adapting this to the McCulloch-Pitts-Neuron means that some connections should be more or less important than others.

Frank Rosenblatt developed the first perceptron [36]. 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, i.e. to strengthen or weaken a connection. Thus, (2.1) changes to

$$\hat{y} = \phi\left(\sum_{i}^{n_x} x_i \cdot w_i\right) \tag{2.2}$$

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. 2.21b the Heaviside function outputs 0 if its argument 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. 2.1a to this results in Fig. 2.1b. There is one 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 (2.2) yields

$$\hat{y} = \phi\left(\left(\sum_{i}^{n_x} x_i \cdot w_i\right) + b\right) \tag{2.3}$$

where $x_i$ is the $i$-th input, $w_i$ the $i$-th weight, $b$ the bias and $\phi$ the Heaviside activation function. By changing the weights and biases, while keeping the inputs constant, a different behavior can be enforced. In general, the inputs and weights are written as vectors of

$$\boldsymbol{x} = \begin{pmatrix} x_1 & x_2 & \cdots & x_{n_x} \end{pmatrix}^T \tag{2.4}$$

$$\boldsymbol{w} = \begin{pmatrix} w_1 & w_2 & \cdots & w_{n_x} \end{pmatrix} \tag{2.5}$$

for simplicity. Inserting this in (2.3) results in

$$\hat{y} = \phi\left(\boldsymbol{w} \cdot \boldsymbol{x} + b\right) \tag{2.6}$$

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 neural network.

### Multilayer Perceptron

A multilayer perceptron consists of multiple perceptrons arranged into layers and solves complex tasks [3]. It is a universal approximator for every function [9] regardless of the activation functions used [21]. 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.

There are at least $L = 3$ 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 a fully-connected network. Because the data flow within the network is directed, i.e. only in one direction, and acyclic, the architecture is called feedforward neural network. A visualization of this is shown in Fig. 2.2. For clarity the weights and biases of each connection and perceptron are not displayed. Like the single perceptrons every perceptron in the network computes its own activation, a single numerical value, depending on its inputs, weights, and bias. A specific perceptron is referred to by its layer $l$ and position $j$ within this layer. Hence, its activation is denoted as $a_j^{[l]}$. Its weight of each connection, called an edge, is denoted as $w_{jk}^{[l]}$ where $k$ is the position of the preceding perceptron in layer $l-1$ and its bias as $b_j^{[l]}$. All weights of layer $l$ are stored in a compact matrix $\boldsymbol{W}^{[l]}$ with each perceptron's weights as a row vector. In this type of network architecture perceptrons are often referred to as nodes or units.

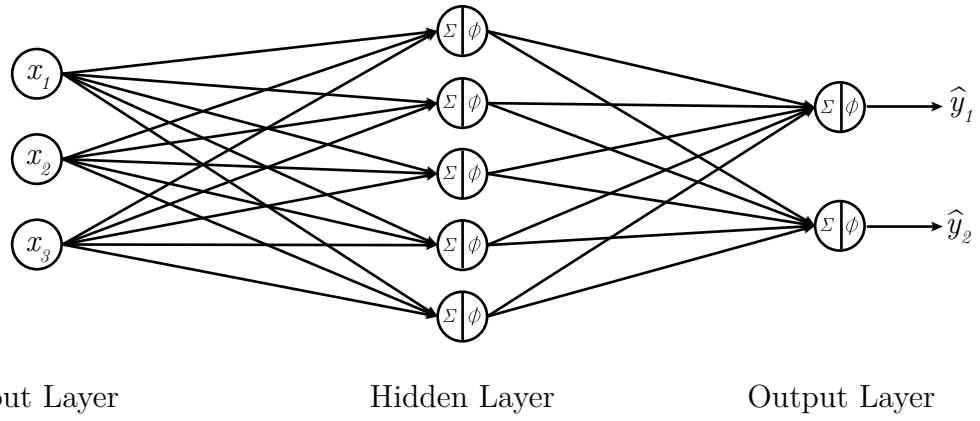



Figure 2.2: Multilayer perceptron with three layers. Each layer consists of multiple perceptrons. The input layer transfers numeric data into the network. The output layer provides the result of the network for interpretation and further processing. The layers in between, the hidden layers, perform calculations and forward the network data. Each connection between perceptrons has a weight, that is not displayed for clarity. Also, every perceptron 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 perceptrons in this layer depends on the data and how it is prepared. If the input data is an image, for example, the number of perceptrons should be equal to the number of pixels, so that every perceptron can hold the intensity value of one pixel. The output layer is responsible for providing the network's computation result so that it can be interpreted and further processed. The number of perceptrons in this layer depends on the desired number of values. If types of animals need to be classified in an image, every output perceptron would represent a single type or class, respectively. Assuming there are three types of animals possible, then there need to be three output perceptrons. In theory, the perceptron representing the correct class of an animal holds a one and every other a zero if the values are normalized to this range. Every layer between the input and the output layer is a hidden layer. Their name comes from the fact that they are not directly accessible from the outside. Their task is propagating the input data to the output layer while weighting known correlation features more than irrelevant ones. With at least one hidden layer every continuous function can be approximated. So, the network models the function

$$f(\boldsymbol{x}) = \boldsymbol{y} \tag{2.7}$$

where

$$\boldsymbol{x} = (x_0, x_1, \cdots, x_{n_x})^T \tag{2.8}$$

$$\boldsymbol{y} = (y_0, y_1, \cdots, y_{n_y})^T \tag{2.9}$$

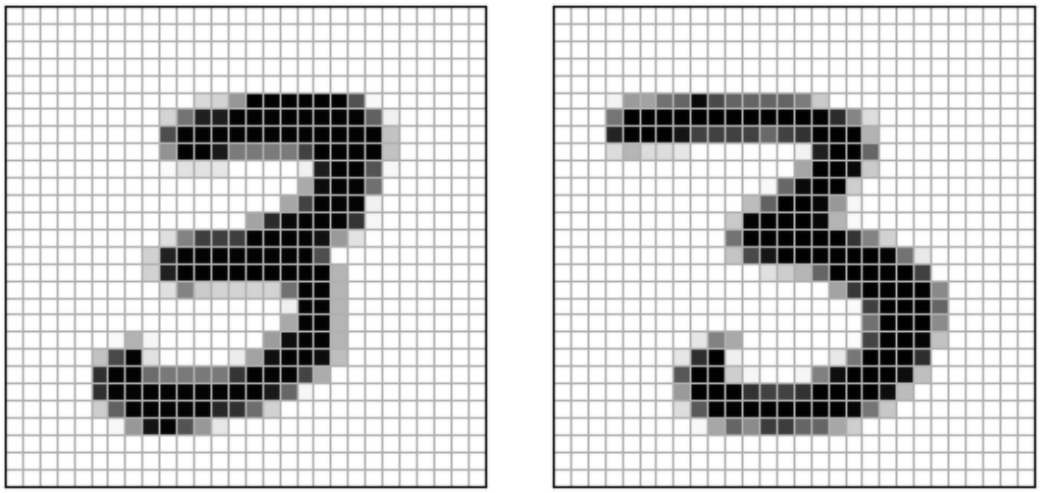


Figure 2.3: Handwritten digits from the MNIST digit dataset [29]. Represented as a $28 \times 28$ pixel matrix. Each cell represents a pixel.

are the input vector with $n_x$ elements and output vector with $n_y$ elements, respectively.

For the next example an image is used again. The task is to classify a handwritten digit from the MNIST dataset [29]. The digit can be seen in Fig. 2.3. Each grid cell represents a pixel. It is evident that the intensity of every pixel is relevant for classifying the digit. Thus, every pixel needs an associated perceptron 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 equal the number of input perceptrons. Assuming well-suited weights and biases, that must first be found in practice, the network knows which perceptrons are active for a particular digit. This means, that if in another image the same or similar pixels or perceptrons, respectively, have high intensities or activations, the same number needs to be classified. The downside of the flattening is, that the relationship of pixels like their 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 know, the classification fails. If, for example, a digit the network classified correctly is not centered anymore and downscaled to take up only half its original size in a $28 \times 28$ image, completely different perceptrons are active. 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 perceptrons naturally needs to be adapted. Due to this, additional weights and biases are introduced to the network, because every input perceptron is connected to every perceptron 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 2.1.3.

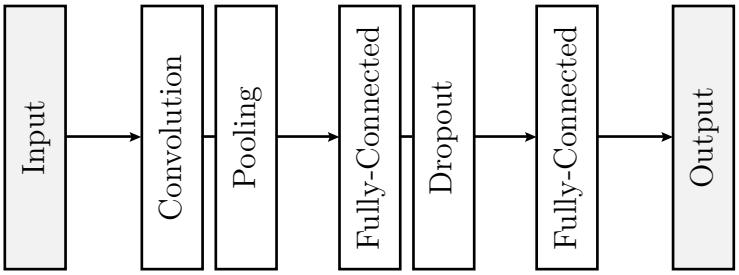



Figure 2.4: Layers of a convolutional neural network. Each combination of convolutional layer and pooling layer or fully-connected and dropout can be arbitrarily repeated. Moreover, pooling layers and dropout regularization layers are optional. The last fully-connected layer is not combined with a dropout layer.

### 2.1.3 Convolutional Neural Networks

Convolutional neural networks are suited for image processing tasks because they are invariant regarding the position of an object within an image. Moreover, they need fewer parameters than multilayer perceptrons for finding features because the weights of local features can be shared and applied to different regions of an image [28, 29]. Hence, they yield higher accuracies on generalized images in less training time. The latter refers to the process of finding well-suited parameters. Convolutional neural networks do not have an as strict separation in multiple layers as multilayer perceptrons do. They rather have a pool of several layers which can be arbitrarily connected, repeated, and tuned with respect to their parameters to fulfill one's needs as illustrated in Fig. 2.4. Commonly, convolutional layers are combined with pooling layers and fully-connected layers with dropout regularization layers. However, the latter combinations are optional. Each of them is explained in the following sections. Combinations and repetitions of those layers with their own hyperparameters are called architecture. Usually, convolutional and pooling layers manipulate the input and the resulting activations before fully-connected layers appear. There are different proposed architectures with their weights and biases available. The most common ones are AlexNet [27], VGG [41], GoogLeNet [47] and, ResNet [17].

#### Convolutional Layer

The first convolutional layer in a network usually works directly on the prepared input data for finding features. As the name suggests it performs a convolution with a filter matrix of arbitrary size on an input matrix of arbitrary size. The input matrix corresponds to spatial input neurons and is, for example, an image $\boldsymbol{I} \in \mathbb{R}^{u \times v}$. The filter or kernel matrix $\boldsymbol{K} \in \mathbb{R}^{i \times j}$ contains $i \cdot j$ weights of the network. The filter is moved across the neurons and performs a dot product within its window. Hence, its weights are reused for different areas in the image. Fig. 2.5 illustrates the following op-

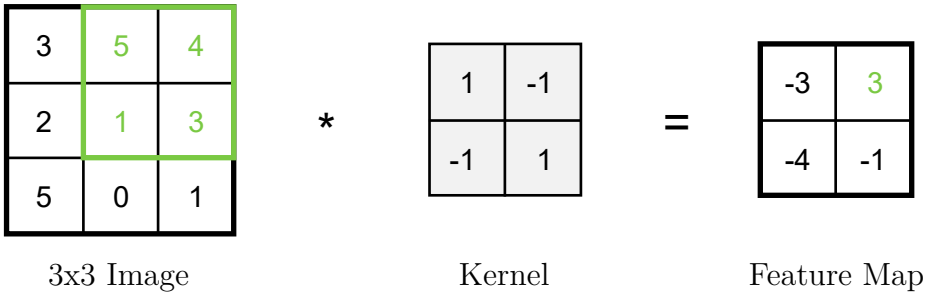



Figure 2.5: Convolution of an image $\boldsymbol{I}$ with a kernel $\boldsymbol{K}$. The $2 \times 2$ kernel is moved across the $3 \times 3$ image and performs a dot product multiplication within its window each time. Here, the kernel moves with a stride of $s = 1$, which results in the shown image on the right, the so-called feature map $\boldsymbol{F}$.

eration. 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 $\boldsymbol{F}$ at its corresponding place. In 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 $s = 1$ moves the filter by one pixel or neuron, respectively. There can be a different stride along the row and height dimension. This resulting matrix is called a feature map because it stores the features extracted from the input. With

$$\dim(\boldsymbol{F})_1 = \text{floor}\left(\frac{u - i}{s} + 1\right) \tag{2.10a}$$

$$\dim(\boldsymbol{F})_2 = \text{floor}\left(\frac{v - j}{s} + 1\right) \tag{2.10b}$$

its shape can be calculated. In this operation, the feature map is always smaller than the input, because only convolutions are performed with the filter inside the input. Sometimes this is not desirable, as it can lead to loss in information in the edge regions. This is because they are less frequent inside the filter window. Moreover, if multiple convolutions are performed consecutively, the feature map constantly shrinks, because each convolution operates on the latest feature map, until no feature can be extracted anymore or only few detailed ones. In practical terms, there are two common conventions for convolutions, which are called valid and same in the following. The former defines that no padding is applied and therefore a valid convolution is performed because only the real input and full kernel windows are taken into account. This means that the feature map has a size of $\boldsymbol{F} \in \mathbb{R}^{\dim(\boldsymbol{F})_1 \times \dim(\boldsymbol{F})_2}$. The latter means, that the size of the feature map equals the one of the input. Thus, a padding $p$ can be applied to the input. This means surrounding the input with zeros to create a larger input. The convolution

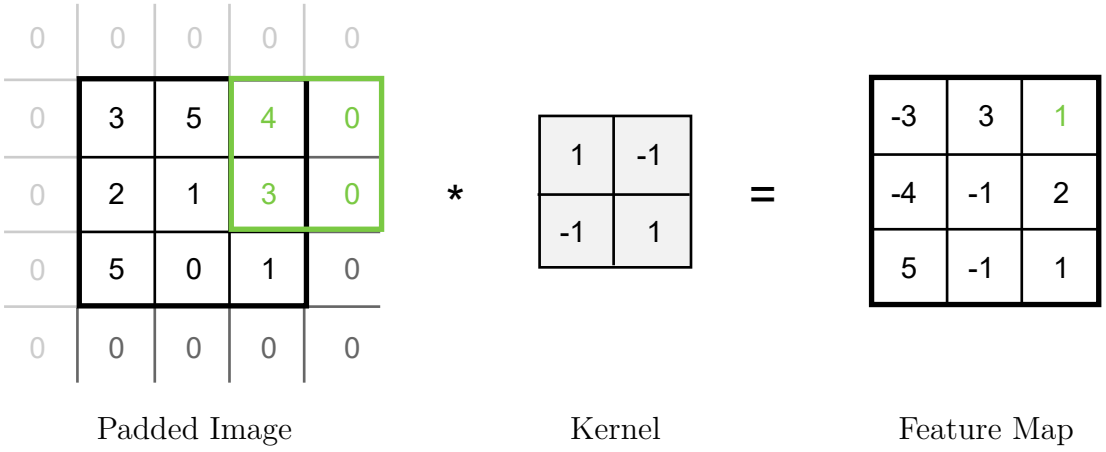



Figure 2.6: Convolution of a padded image with a $2 \times 2$ kernel $\boldsymbol{K}$. The $3 \times 3$ image $\boldsymbol{I}$ is surrounded with $p = 0.5$ rounds of zeros for yielding a feature map $\boldsymbol{F}$ of same size. If the amount of padding is odd, two contiguous sides are preferred.

operates like usual, just on a larger input. How much padding $p$ needs to be applied can be calculated by comparing both matrix shapes. Therefore, the expression

$$u := \frac{u - i + 2p}{s} + 1 \tag{2.11a}$$

needs to be valid, yielding

$$p = \frac{u(s-1) + i - s}{2} \tag{2.11b}$$

for calculating the amount of padding on each related side. However, in general, this only covers the padding height. If the image or filter are not symmetric, the padding along the width needs to be calculated as well by replacing $u$ with $v$ and $i$ with $j$. If the amount of padding is odd, it is performed in half rounds around the image where two contiguous sides of the image are preferred like it is shown in Fig. 2.6 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 three-dimensional inputs, where each matrix along the depth dimension is called channel, a convolution is performed almost identically. Instead of a kernel with a depth of one as for a two-dimensional input, it is extended to a depth that matches the input yielding $\dim(\boldsymbol{K})_3 = \dim(\boldsymbol{I})_3$. 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, i.e. $\dim(\boldsymbol{F})_3 = 1$. 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. This example is illustrated in Fig. 2.7. Usually, at the end of a

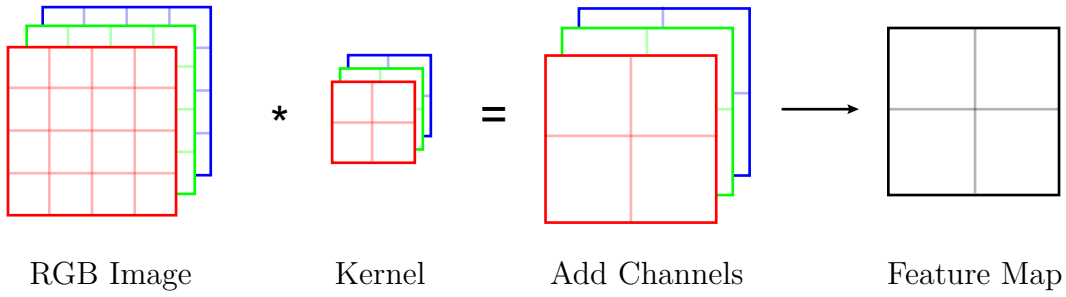



Figure 2.7: Convolving an RGB input $\boldsymbol{I}$ with $\dim(\boldsymbol{I})_3 = 3$ channels needs a kernel $\boldsymbol{K}$ with $\dim(\boldsymbol{K})_3 = \dim(\boldsymbol{I})_3$. This results in one convolution result per corresponding channels. Those are summed element-wise for yielding a two-dimensional feature map $\boldsymbol{F}$.

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. 2.21. Both the methods and their purposes are analog to the ones known from multilayer perceptron networks.

The kernel in Fig. 2.5 would find top-left to bottom-right diagonal lines because its convolution result is higher if the pixels in its window represent such a shape. For example, there is a black image with white shapes. If the filter is on a plain surface, where all pixel values have the same intensity, the convolution result is 0 due to the positive and negative weights. However, if it is over such a line, the result is higher because the pixels representing the line are fully taken into account. If the line points into the other direction the representing pixels are not taken into account, while the other two are weighted negatively. Like this but with slightly larger kernels and different weights more complex features can be found. Finding discriminative features depends on a satisfiable set of weights and biases, though. It is also possible to perform multiple different convolutions on the same input to find different features. They are stored as a matrix, where the number of different features represents the depth of the feature map $\boldsymbol{F}$. This whole process solves the limitation to a fixed position of features of the multilayer perceptrons architecture. Even if, for example, a digit is not centered anymore in the image, all features are found, because the kernel is moved over the image for checking the presence of a certain feature. Hence, this keeps the spatial relationship of pixels that is lost if an input is flattened as with multilayer-perceptrons networks, where each perceptron is responsible for one single pixel. Furthermore, because each feature is found by a moving filter, convolutional neural networks need way fewer weights and biases due to the possibility of reusing them for different image areas. The accuracy compared to multilayer perceptron networks is improved by concatenating several convolutions. That means a convolution is performed on the activation of an earlier convolution or more generally on the activation of its preceding layer. This way, first, rough features like edges are found for narrowing down possible classes, and the deeper it gets into the

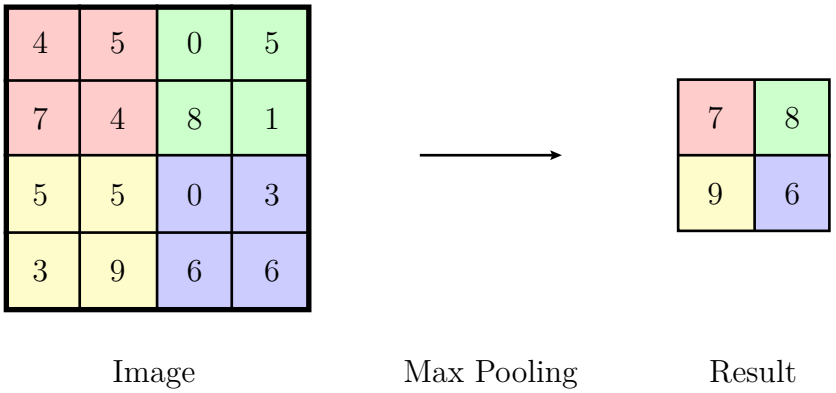



Figure 2.8: Max pooling with $2 \times 2$ filter $\boldsymbol{K}_{\max}$ across a $4 \times 4$ input $\boldsymbol{I}$ with a stride of $s = 2$. Within each window, the maximum of its values is computed. Finally, this yields a matrix with each maximum at its corresponding position.

network, the finer the features get.

### Pooling Layer

After obtaining features using a convolutional layer a pooling layer can be inserted working on their activations. Pooling serves as a spatial dimension reduction. This is done by moving a filter $\boldsymbol{K} \in \mathbb{R}^{i \times j}$ with a given stride $s$ over an input $\boldsymbol{I} \in \mathbb{R}^{u \times v}$ that compresses the information or values, respectively, within its window. However, the depth is usually not compressed and stays the same. The objective of this process is to remove unnecessary information while keeping important features and improving computational power as less spatial information is available. Hence, fewer weights and biases are needed which in turn improves training time. Fig. 2.8 illustrates the pooling process for a max pooling operation in practical terms. A max pooling filter $\boldsymbol{K}_{\max}$ yields the maximum within its window as the result. Moving such a $2 \times 2$ filter over an $4 \times 4$ input $\boldsymbol{I}$ with a stride of $s = 2$ yields a matrix with each maximum at its corresponding position. The maximum of the red colored $2 \times 2$ 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 can be calculated with (2.10) and $\dim(\boldsymbol{F})_3 = \dim(\boldsymbol{I})_3$. Another pooling type is mean pooling. Hereby, the result of each window is the mean of all its values. In many architectures max pooling outperforms mean pooling [39], however, in general, the type of pooling is problem-specific. As it can be seen, pooling layers do not have learnable parameters only hyperparameters.

### Fully-Connected Layer

The last part of a convolutional neural network mostly consists of at least one fully connected layer. Such a layer is identical to a perceptron layer in Fig. 2.2 but uses the

outputs or activations, respectively, of the previous convolutional or pooling layer. 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 snout 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 snout 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 [43]. This drops out nodes randomly during training with a given probability, i. e. changing their incoming and outgoing weights temporarily to zero. Hence, their weights are not adapted. The interpretation of the activations of the last fully-connected layer in the architecture is simplified by applying an additional softmax function that squashes them into a range of 0 and 1, whereas the sum of all equals 1, to represent percentages of confidence or a probability distribution, respectively [4]. The prediction $\hat{y}$ of a class $c$ manipulated with the softmax function can be written as

$$\hat{y}_c = \frac{\exp(a_c^{[L]})}{\sum_j^{n_y} \exp(a_j^{[L]})} \tag{2.12}$$

where $\boldsymbol{a}^{[L]}$ are the activations of the last layer. Because the softmax manipulation outputs a probability distribution, it can only be performed when the classes are mutually exclusive, i. e. when 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. The softmax or sigmoid manipulation corresponds to the output block in Fig. 2.4. Combining the last fully-connected layer with a dropout layer is not desirable because this would remove some predictions of classes.

### 2.1.4 Train a Neural Network

So far optimal weights and biases were assumed in all examples, that exactly model a desired function $f(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{B}) = \boldsymbol{y}$ where $\boldsymbol{W}$ and $\boldsymbol{B}$ store all parameters of a network. But in practical terms, they need to be found first for generating the desired output. This starts by collecting or generating and preparing a dataset from which the network can find correlations by changing the weights and biases. Then, those parameters are randomly initialized. Furthermore, the data samples of the dataset are used as input and are feed-forwarded through the network yielding a classification $\hat{\boldsymbol{y}}$. This classification is evaluated by a cost function. That result is back-propagated through the network by computing its gradients for changing the weights and biases. The forward pass and backward pass are repeated with different samples until an termination condition is satisfied. Fig. 2.9 illustrates this process. Each of these steps is covered in the following sections.

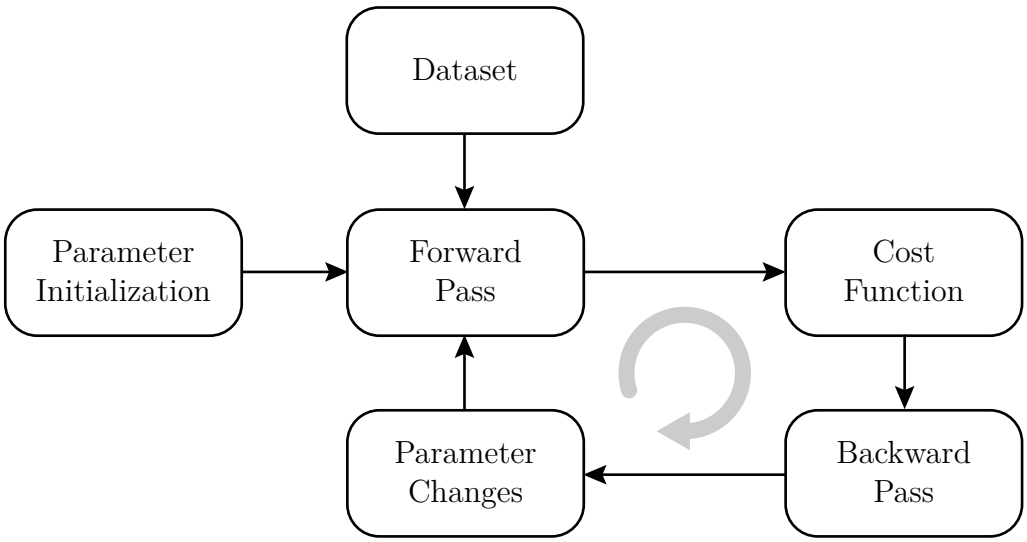
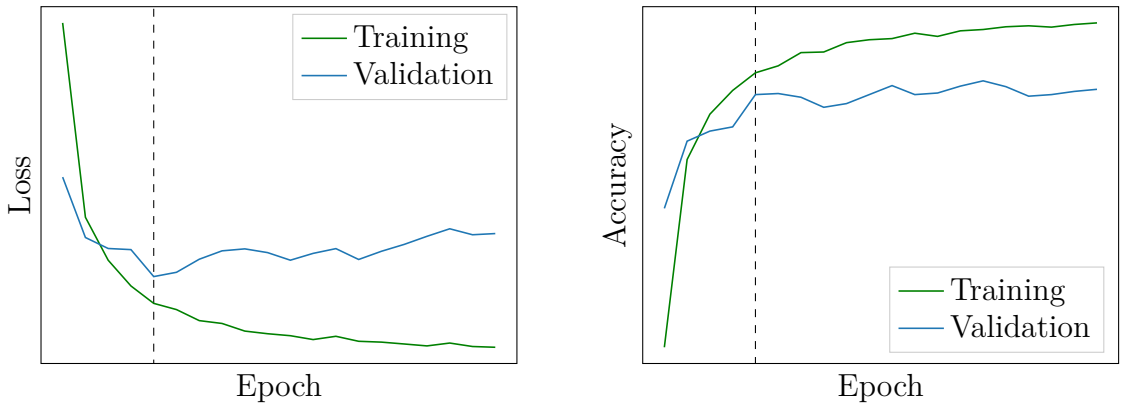



Figure 2.9: Training process. The parameters are initialized once at the beginning. The training process includes the forward pass, the cost function evaluation,the backward pass and the changing of parameters. This is repeated with different data samples.

### Dataset

The whole training process is based on the dataset from which the network is supposed to learn correlations of each input to 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 of its class. In the case of a classification task, the label represents the class. However, there is no general rule for the amount of data. It can be said, that more data is better for generalization, but processing too many samples can lead to overfitting the network to the shown data. The latter means, that the network is trained too 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 generalize to unknown data, because it slightly differs. 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, because this could be classified with the help of few filters, hence, few parameters that need to be adapted. 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. There are several datasets available, that are already sorted and labeled, like the MNIST handwritten digits or the ImageNet dataset [38]. Available datasets are not limited to images but may include CAD models like the ModelNet dataset [49] which is used for the network architecture going to be presented.




(a) Loss (b) Accuracy

Figure 2.10: Indicator of overfitting. If loss or accuracy of the validation set do not follow the direction of the one of the training set the network does not generalize. The dashed line represents when training should have been stopped.

Usually, samples of a dataset are split into a training set, test set, and a validation set [22]. The first contains data, the network is trained on. From this data correlations of each input to label are found. After an arbitrary number of training steps where parameter changes are executed, the performance of the network is tested on the validation set. This is data, the network is not trained on. The objective here is to check if overfitting occurs. If the loss of the training set decreases, the loss of the validation set has to decrease as well. This shows that the network still learns and gets better. If the loss of the training set decreases, but the loss of the validation set stays the same or increases, it is an indicator for overfitting. This concept is also valid for the accuracy but reversed. Both cases are visualized in Fig. 2.10. The dotted line indicates when training should have been stopped.

The test 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. It is bad practice but 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, an equal distribution of samples in terms of samples per class in each set should be minded. 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 [15] of the labels can be performed. Usually, the labels are categorical data. This means, they contain label or string values, respectively, instead of numeric values, that the networks needs. For example, there is a fashion variable with the values "boot", "sweatshirt" and "pants". The network would not know how to interpret these. Thus, these values need to be converted to numeric values. Furthermore, if these label values are outputs of the network, it should be easily possible

Table 2.1: One-hot encoding of categorical data. First, categorical label values are transformed to numeric values representing a class 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 classes, where every element is 0 except for the corresponding class which is 1.

| Categorical | Integer | One-Hot |
|---|---|---|
| "Boot" | 0 | $\boldsymbol{y}^{(1)} = (1, 0, 0)^T$ |
| "Sweatshirt" | 1 | $\boldsymbol{y}^{(2)} = (0, 1, 0)^T$ |
| "Pants" | 2 | $\boldsymbol{y}^{(3)} = (0, 0, 1)^T$ |

to convert them back from numeric values. Hence, they are converted to integers that represent a class. 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 classes 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 label vector $\boldsymbol{y}^{(2)} = (0, 1, 0)^T$ for the "sweatshirt" label. This vector has a length of the number of different classes available, where every element is 0 except the one of the corresponding class which is 1. Table 2.1 summarizes this approach. This encoding needs to be performed on each label in the dataset.

### 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 less time. In the other case, a converging to optimal values can be impossible. Reasons for this can be the exploding or vanishing of gradients during backpropagation [20]. 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. 2.11 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 true for other activation functions like the tanh as well. For the ReLU the accumulated gradients can become very large 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 2.1.4 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-

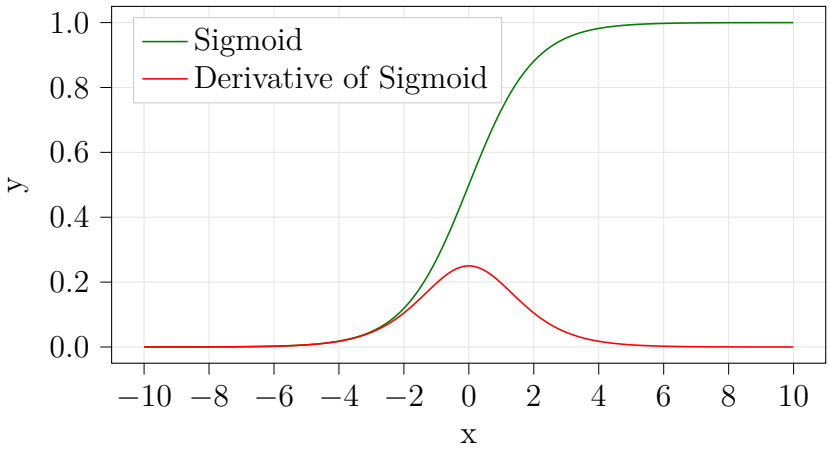



Figure 2.11: Sigmoid function and its derivative

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 the distribution of each neuron's output grows with the number of its inputs, because their variances are accumulated. 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 its $n_{\text{in}}$ inputs $\boldsymbol{a}^{[l-1]}$ and weights $\boldsymbol{w}^{[l]}$ by

$$
\begin{aligned}
\operatorname{Var}(z_j^{[l]}) &= \operatorname{Var}\left(\sum_k^{n_{\text{in}}} w_{jk}^{[l]} a_k^{[l-1]}\right) \\
&= \sum_k^{n_{\text{in}}} \operatorname{Var}\left(w_{jk}^{[l]} a_k^{[l-1]}\right) \\
&= \sum_k^{n_{\text{in}}} \left[\operatorname{E}(w_{jk}^{[l]})^2\right] \operatorname{Var}(a_k^{[l-1]}) + \operatorname{E}\left[(a_k^{[l-1]})\right]^2 \operatorname{Var}(w_{jk}^{[l]}) + \operatorname{Var}(a_k^{[l-1]}) \operatorname{Var}(w_{jk}^{[l]}) \\
&= \sum_k^{n_{\text{in}}} \operatorname{Var}(a_k^{[l-1]}) \operatorname{Var}(w_{jk}^{[l]}) \\
&= (n_{\text{in}} \operatorname{Var}(\boldsymbol{w}_j^{[l]})) \operatorname{Var}(\boldsymbol{a}^{[l-1]})
\end{aligned}
$$

where zero mean inputs and weights are assumed and an identical distribution of all $\boldsymbol{w}_j^{[l]}$ and $\boldsymbol{a}^{[l-1]}$. Now, $z_j^{[l]}$ needs to have the same variance as all of its inputs $\boldsymbol{a}^{[l-1]}$, which yields $\operatorname{Var}(\boldsymbol{W}^{[l]}) = 1/n_{\text{in}}$ as every weights variance. Hence,

$$
\boldsymbol{W}^{[l]} = \frac{N(0,1)}{\sqrt{n_{\text{in}}}} \tag{2.13}
$$

initializes the weights. This is mostly universal, but must be used for tanh activation functions. A similar analysis is done by *Glorot and Bengio* [13] whose recommendation is

$$\mathrm{Var}(\boldsymbol{W}^{[l]}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

where $n_{\text{in}}$ and $n_{\text{out}}$ 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

$$\mathrm{Var}(\boldsymbol{W}^{[l]}) = \frac{1}{n_{\text{out}}} \tag{2.14}$$

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

$$\boldsymbol{W}^{[l]} = N(0,1) \cdot \sqrt{\frac{2}{n_{\text{in}}}} \tag{2.15}$$

especially for ReLU neurons.

### Feed-Forward Pass

The actual training step, i.e. the finding of optimal weights and biases, starts with propagating samples through the network. Therefore, a dataset $\mathbb{D}$ containing $m$ pairs of inputs and corresponding labels is needed. Performing a one-hot encoding on the labels and assuming in general flattened input matrices yields

$$\mathbb{D} = (\boldsymbol{X}, \boldsymbol{Y}) \tag{2.16}$$

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 image and $n_y$ the number of classes. This set is then divided into a training, validation and test set. Each of them contains $m_{\text{train}}$, $m_{\text{valid}}$ or $m_{\text{test}}$ samples, respectively. Furthermore, there is a neural network with $L$ layers each containing an arbitrary number of neurons. Expressing the activation of every perceptron with (2.3) would get very confusing for a whole network. Hence, a matrix notation is preferred. First, for every $j$-th perceptron in the $l$-th layer its weights are summarized in a vector

$$\boldsymbol{w}_j^{[l]} = \begin{pmatrix} w_{j,1}^{[l]} & w_{j,2}^{[l]} & \cdots & w_{j,n_h^{[l-1]}}^{[l]} \end{pmatrix}^T \tag{2.17}$$

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 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

$$\boldsymbol{W}^{[l]} = \begin{pmatrix} \boldsymbol{w}_1^{[l]} & \boldsymbol{w}_2^{[l]} & \cdots & \boldsymbol{w}_{n_h^{[l]}}^{[l]} \end{pmatrix}^T \tag{2.18a}$$

$$\boldsymbol{b}^{[l]} = \begin{pmatrix} b_1^{[l]} & b_2^{[l]} & \cdots & b_{n_h^{[l]}}^{[l]} \end{pmatrix}^T \tag{2.18b}$$

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 (2.6). The weighted sum of all neurons of the $l$-th layer is computed as

$$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]} \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]} \tag{2.19}$$

with the activations vector $\boldsymbol{a}^{[l-1]}$ of the $l-1$-th layer. Putting this in an activation function yields

$$\boldsymbol{a}^{[l]} = \phi\left(\boldsymbol{z}^{[l]}\right) \tag{2.20}$$

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

$$\hat{\boldsymbol{y}}^{(i)} = f(\boldsymbol{x}^{(i)}, \boldsymbol{W}, \boldsymbol{B}) \tag{2.21}$$

as the network's prediction for the $i$-th data sample $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n_x}$ where $\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^{n_y}$ and $\boldsymbol{W}$ and $\boldsymbol{B}$ are concatenated matrices storing the parameters. This superscript in round brackets is part of the used convention. Feeding this result into a sigmoid function forces the values to be in a range between 0 and 1 representing class probabilities. 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. Those vectors can look, for example, like

$$\boldsymbol{y}^{(i)} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \end{pmatrix}^T \tag{2.22a}$$

$$\hat{\boldsymbol{y}}^{(i)} = \begin{pmatrix} 0.54 & 0.28 & 0.2 & 0.63 & 0.96 \end{pmatrix}^T \tag{2.22b}$$

where the first is the ground-truth label and the second the prediction. It can be clearly seen, that the prediction is completely wrong. The actual ground truth class has the second smallest probability in the prediction. Hence, the parameters need to be changed. In theory, an identical representation is desired. Because finding optimal parameters is an optimization problem, a metric for the performance of the network is necessary. This is served by a loss function that maps 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

$$H(y^{(i)}, \hat{y}^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})) \tag{2.23}$$

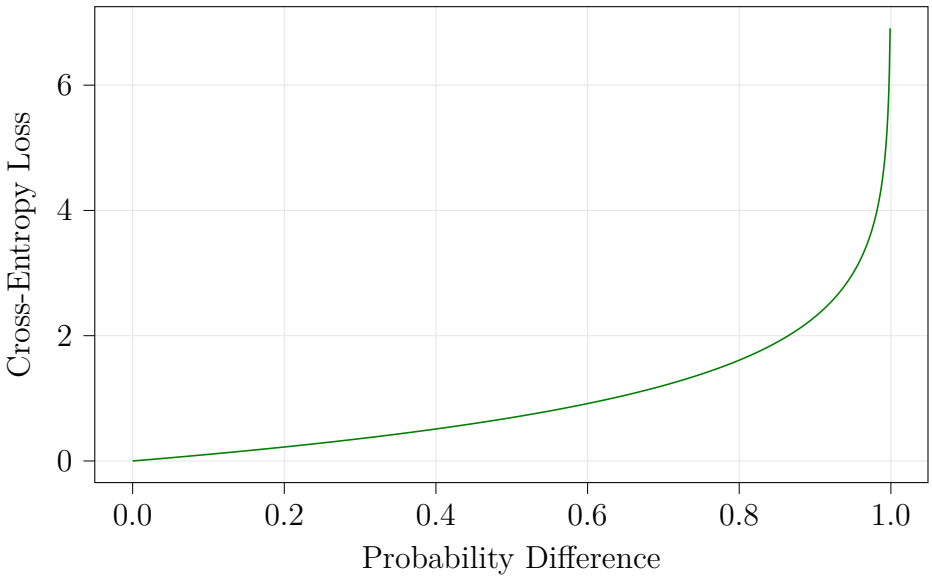



Figure 2.12: Cross-entropy loss. Large deviations in probability are strongly penalized.

for a single output representing two classes and as

$$H(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) = -\sum_{j}^{n_y} y_j^{(i)} \log(\hat{y}_j^{(i)}) \tag{2.24}$$

for multi-class classification [33], where $n_y$ is the number of classes, $\boldsymbol{y}^{(i)}$ the ground truth label and $\hat{\boldsymbol{y}}^{(i)}$ the predicted probabilities of the $i$-th sample. Due to the one-hot encoding of the labels, only the positive class $y_p^{(i)}$ is taken into account in the loss computation. Hence, (2.24) is reduced to

$$H(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) = -y_p^{(i)} \log(\hat{y}_p^{(i)}) \tag{2.25}$$

where $y_p^{(i)}$ and $\hat{y}_p^{(i)}$ denote the probability of the positive class and its corresponding prediction, respectively. A visualization of this expression is shown in Fig. 2.12. 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. Using each sample's loss value for computing an averaged loss yields

$$J(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{B}, \boldsymbol{y}) = J(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} H(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) \tag{2.26}$$

as the cost function for the training process. Computing the cost of a different set is possible by using the corresponding samples. 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 for a given training set.

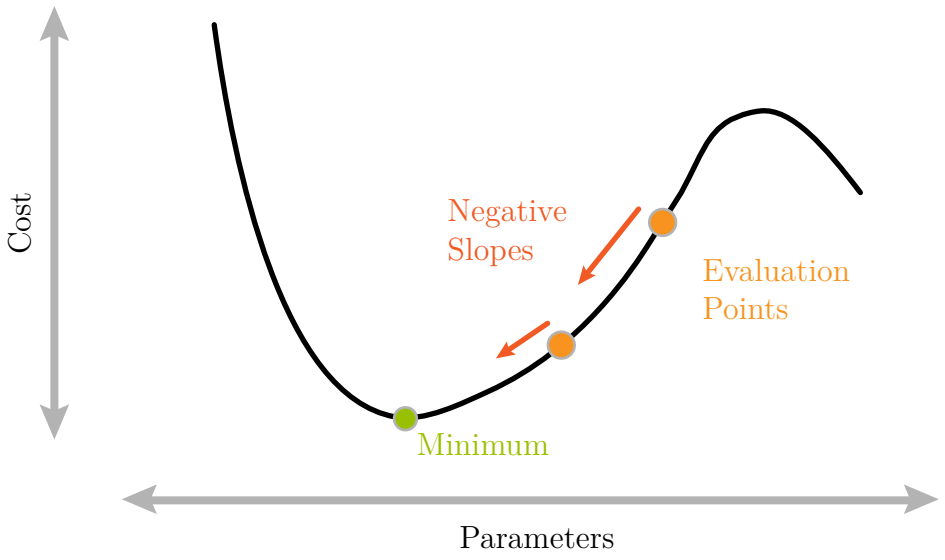



Figure 2.13: Schematic of gradient descent. Cost is evaluated, its negative gradients are computed and the parameters are moved along this direction until the minimum is reached.

### Backpropagation with Gradient Descent

Due to the non-linearities in the network induced by the activation functions, the minimum of the cost function cannot be computed analytically but numerically. Using backpropagation [37, 14] 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 [24, 35]. Fig. 2.13 illustrates this approach. The gradients point into the direction of steepest ascent. To reach the minimum, 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 by 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.

The goal of backpropagation is to compute the partial derivatives

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \frac{1}{m_{\text{train}}} \sum_{i}^{m_{\text{train}}} \frac{\partial J^{(i)}}{\partial w_{jk}^{[l]}} \tag{2.27a}$$

$$\frac{\partial J}{\partial b_{j}^{[l]}} = \frac{1}{m_{\text{train}}} \sum_{i}^{m_{\text{train}}} \frac{\partial J^{(i)}}{\partial b_{j}^{[l]}} \tag{2.27b}$$

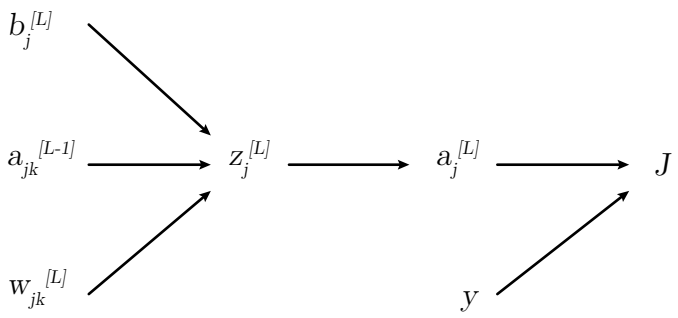



Figure 2.14: Data flow in a last-layer neuron for backpropagation

of the cost function w.r.t. the parameters by averaging the partial derivatives of cost functions of $m_{\text{train}}$ samples from the training set that were passed through the network. These $m_{\text{train}}$ samples build a so-called batch in this context. Fig. 2.14 recaps the data flow in a neuron in the last layer. By checking stepwise how a parameter directly influences a subsequently one, (2.27) can be written as

$$\frac{\partial J}{\partial w_{jk}^{[L]}} = \frac{\partial z_j^{[L]}}{\partial w_{jk}^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \frac{\partial J}{\partial a_j^{[L]}} \tag{2.28a}$$

$$\frac{\partial J}{\partial b_j^{[L]}} = \frac{\partial z_j^{[L]}}{\partial b_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \frac{\partial J}{\partial a_j^{[L]}} \tag{2.28b}$$

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

$$\frac{\partial J}{\partial a_k^{[L-1]}} = \sum_{j=1}^{n_y} \frac{\partial z_j^{[L]}}{\partial a_k^{[L-1]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \frac{\partial J}{\partial a_j^{[L]}} \tag{2.29}$$

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 (2.29) 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,

$$\delta_j^{[l]} = \frac{\partial J}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \tag{2.30}$$

defines the error of the $j$-th neuron in the $l$-th layer. Combining (2.28) and (2.30) yields

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial J}{\partial a_j^{[l]}} = \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} \delta_j^{[l]} \tag{2.31a}$$

$$\frac{\partial J}{\partial b_j^{[l]}} = \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial J}{\partial a_j^{[l]}} = \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} \delta_j^{[l]} \tag{2.31b}$$

as the general expressions for backpropagation where

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_j^{[l-1]} \tag{2.32a}$$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1 \tag{2.32b}$$

$$\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \phi'(z_j^{[l]}) \tag{2.32c}$$

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

$$\nabla \boldsymbol{J} = \begin{pmatrix} \frac{\partial J}{\boldsymbol{W}^{[1]}} & \frac{\partial J}{\boldsymbol{b}^{[1]}} & \frac{\partial J}{\boldsymbol{W}^{[2]}} & \frac{\partial J}{\boldsymbol{b}^{[2]}} & \cdots & \frac{\partial J}{\boldsymbol{W}^{[L]}} & \frac{\partial J}{\boldsymbol{b}^{[L]}} \end{pmatrix}^T \tag{2.33}$$

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

$$w_{jk}^{[l]}(\tau + 1) = w_{jk}^{[l]}(\tau) - \gamma \nabla \boldsymbol{J}(w_{jk}^{[l]}(\tau)) \tag{2.34a}$$

$$b_j^{[l]}(\tau + 1) = b_j^{[l]}(\tau) - \gamma \nabla \boldsymbol{J}(b_j^{[l]}(\tau)) \tag{2.34b}$$

where the hyperparameter $\gamma$ is the learning rate and $\tau$ 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 roughly exemplified in Fig. 2.15. Thus, a trade-off must be found or an adaption of the learning rate to certain circumstances like the

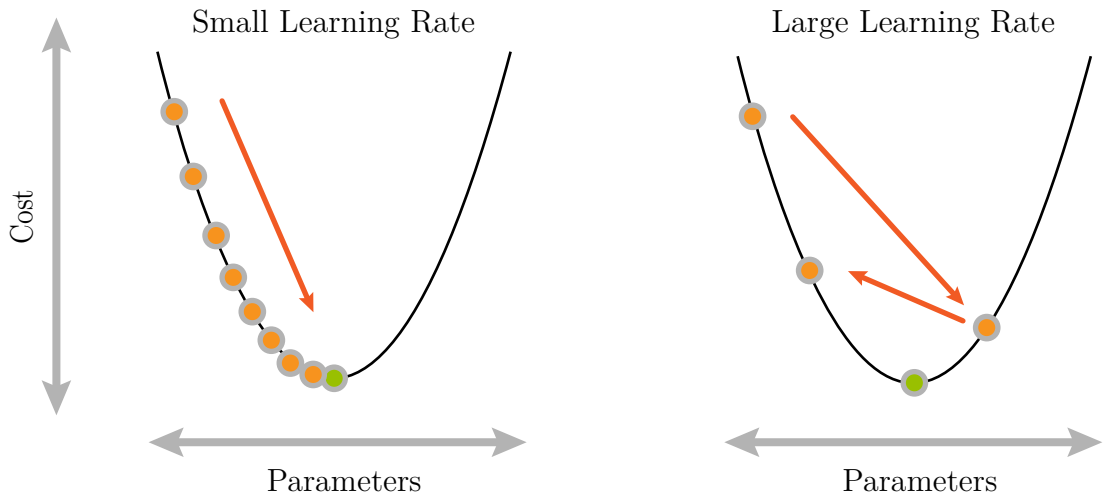



Figure 2.15: Comparison of learning rates. A small learning rate finds the minimum slowly but more exactly than a high learning rate. However, the latter tends to overshoot the minimum. This is roughly exemplified.

magnitude of the slope is necessary. According to *Bengio* [2] 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 remains a hyperparameter that is problem-specific, hence, only guidelines can be provided. One of them is, that the learning rate should be greater than $10^{-6}$ and less than 1.0. Another approach is decaying the initial learning rate either linearly or exponentially until iteration $\tau$ and then leaving it constant [14]. 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

$$\gamma(\tau) = \gamma_0 \exp(-\lambda\tau) \tag{2.35}$$

where $\gamma_0$ is the initial learning rate, $\lambda$ a factor and $\tau$ the iteration step. The concept of changing the learning rate over time is called learning rate schedule. Its values are 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 $\tau$, 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.* [8] 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. 2.16. With this gradient the

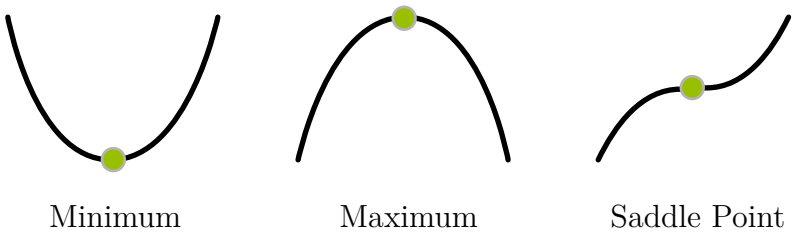



Figure 2.16: Types of critical points

algorithm would get stuck. Hence, adding some noise to (2.34) yields

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \gamma \nabla \boldsymbol{J}(w_{jk}^{[l]}) + \boldsymbol{\varepsilon}(w_{jk}^{[l]}) \tag{2.36a}$$

$$b_{j}^{[l]} := b_{j}^{[l]} - \gamma \nabla \boldsymbol{J}(b_{j}^{[l]}) + \boldsymbol{\varepsilon}(b_{j}^{[l]}) \tag{2.36b}$$

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

### 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[48]. Its objective is illustrated in Fig. 2.17. 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 accelerating the other one. In this example, the vertical parameter would be damped while the horizontal one is accelerated. Hence, it can use a larger learning rate and reaches the minimum more quickly. RMSProp adapts the 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 running average is calculated by

$$v(w_{jk}^{[l]}, \tau) = \beta_2 v(w_{jk}^{[l]}, \tau - 1) + (1 - \beta_2)(\nabla \boldsymbol{J}(w_{jk}^{[l]}))^2 \tag{2.37a}$$

$$v(b_{j}^{[l]}, \tau) = \beta_2 v(b_{j}^{[l]}, \tau - 1) + (1 - \beta_2)(\nabla \boldsymbol{J}(b_{j}^{[l]}))^2 \tag{2.37b}$$

where hyperparameter $\beta_2$ 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 inertia to the update procedure dampening oscillations and building up speed on flat surfaces. Finally, the parameters

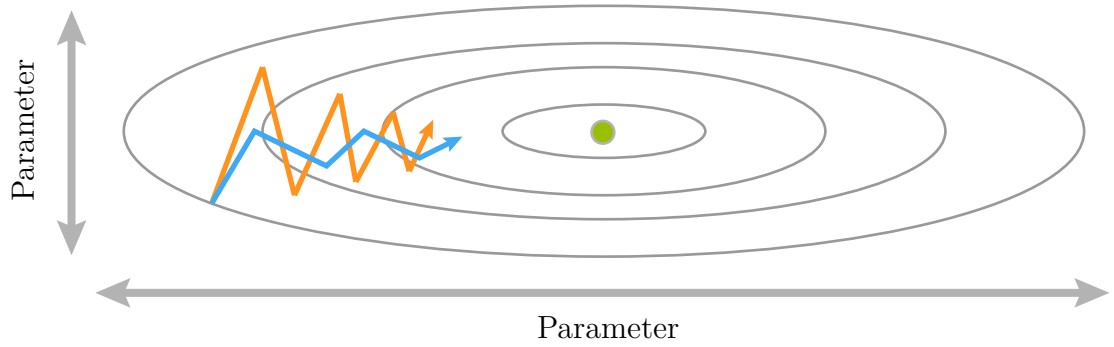



Figure 2.17: Process of gradient descent and RMSProp using contour lines. The orange line illustrates the process of gradient descent. It oscillates and moves slowly to the minimum. The blue line shows the process of the RMSProp algorithm. While the vertical parameter changes slower the horizontal one changes faster. Hence, RMSProp dampens oscillations and moves faster to the minimum.

are updated by

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \frac{\gamma}{\sqrt{v(w_{jk}^{[l]}, \tau)}} \nabla \boldsymbol{J}(w_{jk}^{[l]}) \tag{2.38a}$$

$$b_{j}^{[l]} := b_{j}^{[l]} - \frac{\gamma}{\sqrt{v(b_{j}^{[l]}, \tau)}} \nabla \boldsymbol{J}(b_{j}^{[l]}) \tag{2.38b}$$

using the weighted moving average just calculated. In reference to Fig. 2.17 assume $w_1$ as the horizontal parameter and $w_2$ as the vertical one. 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 simplify the choice of 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

$$m(w_{jk}^{[l]}, \tau) = \beta_1 m(\tau - 1) + (1 - \beta_1) \frac{\partial J}{\partial w_{jk}^{[l]}} \tag{2.39a}$$

$$m(b_{j}^{[l]}, \tau) = \beta_1 m(\tau - 1) + (1 - \beta_1) \frac{\partial J}{\partial b_{j}^{[l]}} \tag{2.39b}$$

for each parameter using a weighted average of the latest gradients[25] where $\beta_1$ 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 $\beta_1$ 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

$$\mathrm{E}\left[m\right] = \mathrm{E}\left[g\right] \tag{2.40a}$$

$$\mathrm{E}\left[v\right] = \mathrm{E}\left[g^2\right] \tag{2.40b}$$

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

$$\hat{m}(w_{jk}^{[l]}, \tau) = \frac{m(w_{jk}^{[l]}, \tau)}{1 - \beta_1^\tau} \tag{2.41a}$$

$$\hat{m}(b_j^{[l]}, \tau) = \frac{m(b_j^{[l]}, \tau)}{1 - \beta_1^\tau} \tag{2.41b}$$

$$\hat{v}(w_{jk}^{[l]}, \tau) = \frac{v(w_{jk}^{[l]}, \tau)}{1 - \beta_2^\tau} \tag{2.41c}$$

$$\hat{v}(b_j^{[l]}, \tau) = \frac{v(b_j^{[l]}, \tau)}{1 - \beta_2^\tau} \tag{2.41d}$$

using the expression from before. Finally,

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \gamma \frac{\hat{m}(w_{jk}^{[l]})}{\sqrt{\hat{v}(w_{jk}^{[l]})} + \varepsilon} \tag{2.42a}$$

$$b_j^{[l]} := b_j^{[l]} - \gamma \frac{\hat{m}(b_j^{[l]})}{\sqrt{\hat{v}(b_j^{[l]})} + \varepsilon} \tag{2.42b}$$

updates the parameters, where $\varepsilon$ is a small constant for preventing a division by zero. As for other optimization algorithms the learning rate $\gamma$ needs to be tuned. The remaining hyperparameters are recommended by *Kingma et al.* to be $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

### 2.1.5 Choice of Hyperparameters and Activations

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

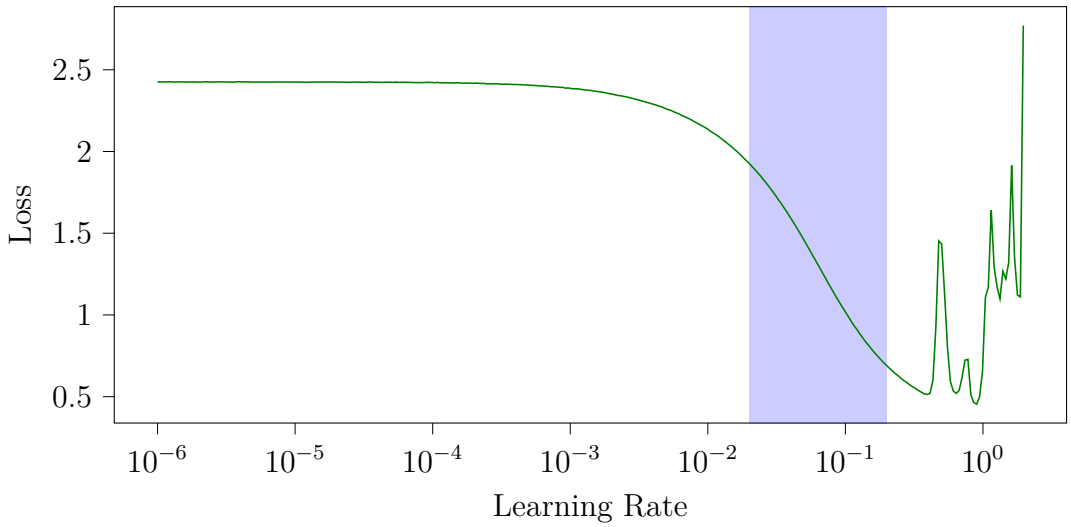



Figure 2.18: Optimal range of learning rates highlighted in blue. The learning rate is increased exponentially.

### Optimal Learning Rate

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

*Smith* introduces the cyclical learning rate [42]. For being cyclical, the learning rate needs a lower and upper bound. This builds a range of optimal learning rates. With the values in between, the cost function is 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. 2.18 shows a typical plot of the cost function against the learning rate on a logarithmic $x$-scale. It can be seen that the cost function does not noticeably improve with a low learning rate. 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. However, the actual range of optimal learning rates is only estimated based on a plot like Fig. 2.18 or its derivative. For the derivative plot, the optimal range is where the gradients are smallest and not in the oscillating region. *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 iteration step.

Another approach is doing warm restarts of the learning rate after several iterations, hence, it is called Stochastic Gradient Descent with Warm Restarts [31]. The idea behind

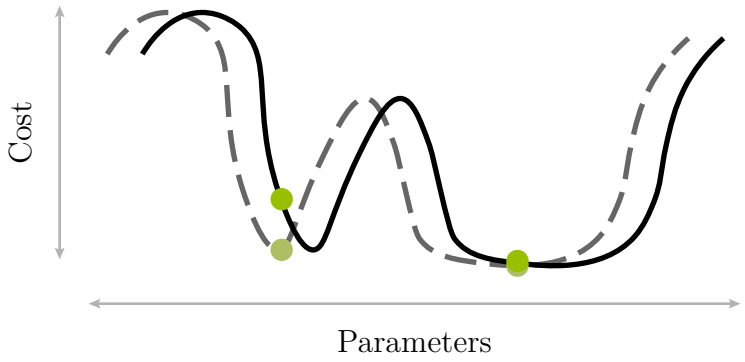
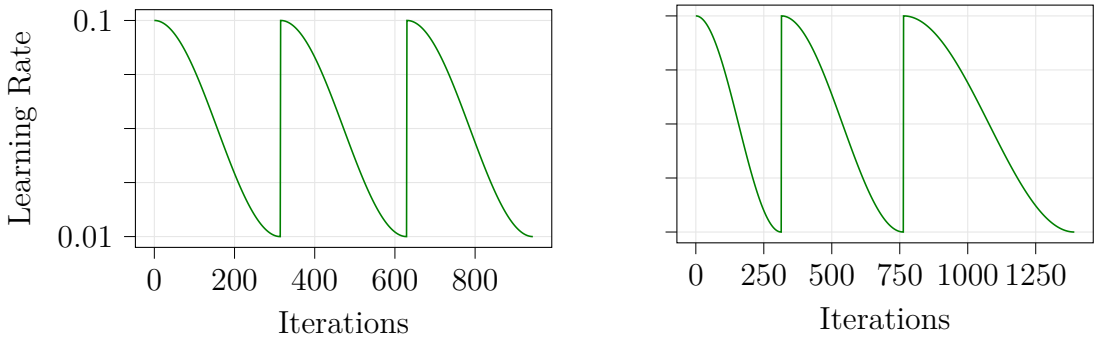



Figure 2.19: Cost function of different datasets. If the parameters represent a minimum of a dataset's cost function, the same parameters do not inevitably represent the minimum of a different dataset's cost function.




(a) Cosine Annealing

(b) Extended Cosine Annealing

Figure 2.20: Annealing of stochastic gradient descent with warm restarts

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 solution that is more generalized across datasets. Fig. 2.19 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. 2.20a. First, the cost is 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. 2.20b. It is assumed, that the more iterations are done, the flatter the found region gets. Thus, the longer its

minimum is searched.

### Optimal Batch Size and Number of Epochs

The batch size defines how many samples are propagated through the network at once. Moreover, the adaption of parameters due to backpropagation depends only on the current batch. This means, $m_{\text{train}}$ in the cost function in (2.26) and in the partial derivatives in (2.27) can be replaced with a given batch size as long as it is smaller than the actual number of samples in the corresponding set. For each sample within a batch the gradients of the cost function with respect to all parameters are calculated and finally averaged over all batch samples for yielding how the parameters need to be changed. That process is repeated for different batches until the whole dataset is propagated forward and backwards. Then it can be repeated with a shuffled dataset, hence, different batches.

For finding the optimal batch size, the most obvious choices are examined first. 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 method 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 in relation to the dataset result in a worse performance of the model in terms of generalization [23]. 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 tight minimum, similar to Fig. 2.19, due to their frequency. 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 computing cost of 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 converges 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 to choose 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.

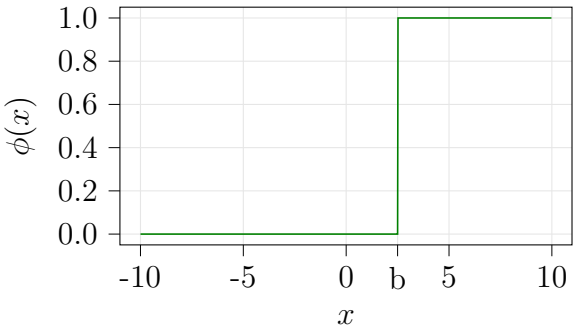
When using mini-batches the number of epochs is a hyperparameter, that can be tuned, as well. Important is the generalization of the network, while preventing under-

fitting or overfitting. Thus, the number of epochs cannot be 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.

### Activation Functions

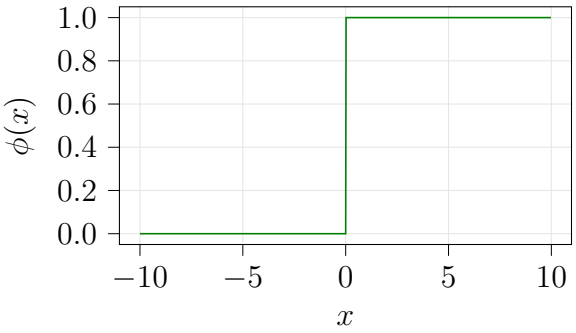
The activation functions of a neural network affect its performance and convergence as well. Common activation functions are shown in Fig. 2.21. The sigmoid function was the state-of-the-art, but has recently fallen out of favor. One disadvantage is its saturation which leads to a very small gradient. 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 gradient descent compared to sigmoid and tanh. According to *Krizhevsky et al.*, it is six times faster on the CIFAR10 dataset [26] compared to tanh activation functions [27]. It is assumed that this is mainly 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 $\lambda = 0.01$ 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 [30].

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.



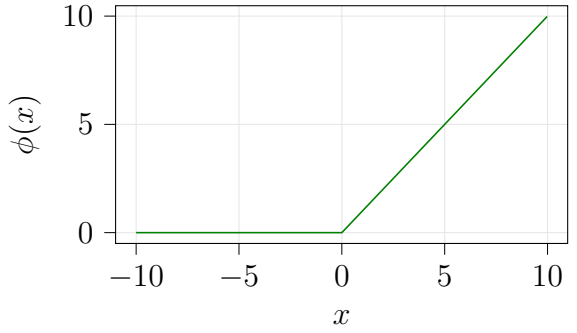

$$\phi(x) = \begin{cases} 1 & x \geq \text{Bias } b \\ 0 & x < \text{Bias } b \end{cases}$$

(a) Threshold



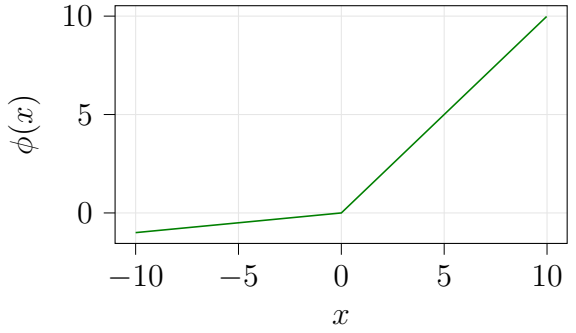

$$\phi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

(b) Heaviside



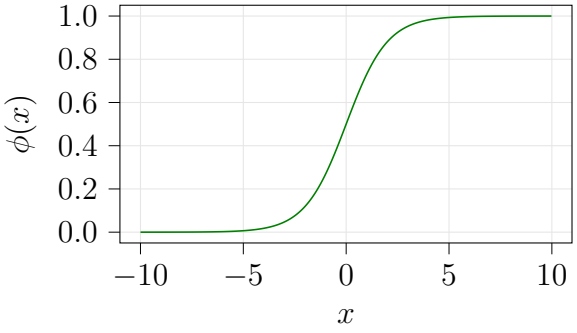

$$\phi(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

(c) Rectified linear unit



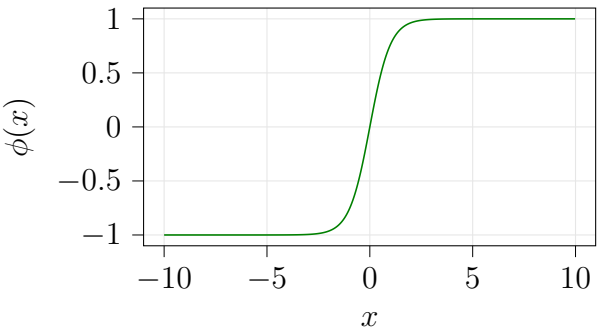

$$\phi(x) = \begin{cases} x & x \geq 0 \\ x \cdot \text{Slope } \lambda & x < 0 \end{cases}$$

(d) Leaky rectified linear unit




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

(e) Sigmoid




$$\phi(x) = \tanh(x)$$

(f) Tanh

Figure 2.21: Plots and equations of common used activation functions. Where the Bias $b$ is the threshold value and $\lambda$ adds a small slope. Usually, the latter is very small like $\lambda = 0.01$.

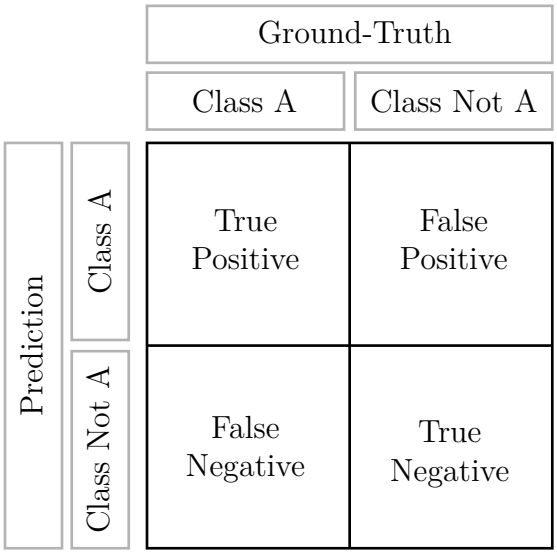



Figure 2.22: Confusion matrix for class A.

### 2.1.6 Metrics for Performance Evaluation

There are several common metrics available that, for example, measure the classification accuracy or its precision. However, they depend on some definitions that will be introduced first. The positive class of a data sample represents its ground-truth class, while the negative one represents any of the remaining ones, i. e. not the positive class. 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 of a sample, i. e. the network classifies this sample correctly as not the positive class. Furthermore, a false positive $FP$ is a wrong prediction of a sample as the positive class. The last definition is a false negative $FN$. This means the network incorrectly predicts a sample as a negative class. Their connection is visualized in Fig. 2.22 showing the so-called confusion matrix [11] for the example class Class A. This needs to be repeated for each class in a dataset. One metric is the accuracy

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.43}$$

of a class. Averaging all class accuracies yields the network accuracy. 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 and its balance, among others. If it is unbalanced there are, for example, more samples in a well-classifiable class than in a bad one, hence, the accuracy is shifted. The precision or positive predicted value of a class measures how accurate the related predictions are and is calculated by

$$PPV = \frac{TP}{TP + FP} \tag{2.44}$$

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

$$TPR = \frac{TP}{TP + FN} \tag{2.45}$$

where the denominator refers to the actual positives of a class. Both (2.44) and (2.45) must be calculated for each class to get a complete overview of the dataset. A simplified representation of the confusion matrix can be prepared by entering only the number of predictions per class. This is beneficial for a quick overview of the prediction distribution of multi-class classifications. Similar to Fig. 2.22 each row represents a predicted class, while the columns represent the ground-truth classes. Each prediction of the samples of a ground-truth class are summed per class and entered at the corresponding cell.

## 2.2 Software

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

### 2.2.1 Tensorflow

Tensorflow [1] is a free framework in particular for machine learning tasks. 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 network with an input and output layer and steadily feed and compare different tensor values.

### 2.2.2 Blender

Blender [5] is a free and open source 3D creation suite to model, texture and animate objects. 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.

# Chapter 3

# 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 based on the available 3D representation of objects that have been modeled using polygons, voxels, or point clouds, among others. *Osada et al.* [34] describe the signature of 3D objects by their shape distribution sampled from geometric shape functions. Those functions use angles, distances, areas and volumes of an object's polygon mesh for forming a shape distribution. By comparing the shape distribution of an object with the ones from similar and dissimilar objects, the class of the current object is found. On the other hand, there are view-based 3D shape descriptors. Those are created using multiple 2D views of an object instead of the raw 3D representation. *Chen et al.* [7] introduced LightField Descriptors, the first typical view-based descriptor. Such a descriptor is created by capturing images from 20 cameras positioned at the vertices of a dodecahedron with the solid 3D object inside. By assuming that the views remain in this spatial order, the descriptor of another object can be compared by rotating it, until both sets are aligned. There are 60 different variants of aligning them. For each variant the distance of both sets is calculated by comparing each view with its aligned one and adding the result up. The comparison is based on the light field representing the radiance in each view. The minimum distance of all variants is used for the classification. *Shu et al.* [40] presented the Principal Thickness Images Descriptor that describes contour and volumetric information of an object. First, they voxelize a 3D mesh object and perform a principle component analysis for yielding the three principle directions of the object. Then, they measure the thickness along those axes by counting the number of voxels along each. The thickness is finally encoded into a gray-scale image for each axis where the intensity of a pixel represents the number of voxels along the related axis. With the histogram of oriented gradients a feature descriptor for each image is extracted. In general, the advantages of view-based shape descriptors are their low dimensionality compared to model-based ones and, hence, the efficiency for processing. In general, view-based descriptors have desirable properties like low dimensionality and efficient evaluation. Moreover, they are more robust to noisy or holey 3D representations.

With the introduction of convolutional neural networks like AlexNet [27] and their

improvement in image classification, they could be used for creating descriptors. This performance was further improved with famous architectures like VGG-Net [41], ResNet [17] and Inception-v4 [46]. According to the ImageNet challenge [38] results, the last two architectures outperform humans regarding the top-5 classification error in 2D object classification. Using a variation of the VGG architecture, the so-called VGG-M [6], *Su et al.* [44] make their eight-layer multi-view convolutional neural network, short MVCNN, 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. The MVCNN architecture, however, performs a maximum pooling operation across all views for collecting discriminative features in a single shape descriptor. This leads to an accuracy of 89.9 %, which outperforms state-of-the-art descriptors, for a network trained on the ImageNet1k dataset [27] and fine-tuned with the ModelNet40 one [49] with 12 views per object. That means, the network is first trained with single views from ImageNet1k and further trained with multi-views from ModelNet40. In contrast, learning a single-view classification with an identical training and fine-tuning yields an accuracy of 88.6 %. In this case, the accuracies of the corresponding 12 views are averaged before the overall accuracy is calculated. Moreover, MVCNN outperforms 3D ShapeNets [49] that reaches an accuracy of 77.3 % and is using a convolutional network on raw CAD data. Hence, it is trained with the ModelNet40 dataset. In the comparison of 2D- and 3D-representations from *Su et al.* [45] the MVCNN architecture outperforms architectures working on point clouds and voxels. Furthermore, they could improve the performance of the vanilla MVCNN to 95.0 % per instance by using a deeper network and better object centering. However, *Hegde et al.* introduce their FusionNet [19] that combines a multi-view architecture, in particular [44], with a volumetric convolutional neural network for achieving 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.* [12], view-to-shape descriptor methods like the one from *Su et al.* are a milestone for 3D shape recognition and reflect the state-of-the-art. Since in [44] 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 with a higher weight than groups with less discriminative views. Finally, a single weighted group descriptor is computed representing the shape descriptor of the object. Because the grouping mechanism sounds promising for helping evaluate the information content of views, in particular, the one of views of the same object but with different material color marks, this work is based on it. Their network yields an accuracy of 92.6 % with an identical training and configuration as before and the GoogLeNet or Inception-v1

architecture [47], respectively. Using only 8 views results in an accuracy of 93.1 %. With transferring the MVCNN concept to the GoogLeNet architecture, an accuracy of 92.2 % instead of 89.9 % is achieved. Another view grouping approach is presented by *Cyr et al.* [10] 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. 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 between an aspect. Hence, it is supposed, that the first describes the object satisfiable.

# Chapter 4

# Methods

This chapter explains the implementation of the neural network architecture for classifying objects and the generation and preparation of the dataset. Each data sample used as an input for the network consists of multiple viewing perspectives of an object, a so-called multi-view. Each object has duplicates, where a different color mark is applied to each copy. The intention is to examine how the multi-view approach known from [44] and, in particular, [12] relates to multi-views where the shape of objects is the same but only differs in color marks. Furthermore, it is analyzed how each view of a data sample contributes to the classification result. In this context, the class of each object is referred to by a combination of its type of object and color mark. Hence, the terminology category describes the first and color the latter. The creation of the dataset is performed with 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 viewing perspectives 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 classes are preferably discriminative to each other to focus more on a classification of the color than on the category. Thus, supporting the objective of this work. This means the dataset should not contain only flowers, for example, but flowers and birds. For this work an existing dataset is chosen for being competitive to other researches that probably used the same as a benchmark for their architecture. The most popular dataset containing CAD objects is ModelNet [49]. It contains 127,915 CAD models divided into 662 object classes for now. For convenience, there are a 10-class and 40-class subset containing 10 or 40 popular classes, respectively. Both are cleaned in respect to a wrong class sorting and then split into a training and test set. Furthermore, the orientations of the models of the first one are aligned as well. Hence, the four classes bathtub, dresser, monitor, and sofa are

extracted from ModelNet10. For this work, this subset is sufficient because it provides enough information for the execution of its task.

Each object in this dataset is modeled using triangles, so-called faces, where each one is defined by the position of its vertices. A vertex is a three-dimensional point defined by a $x$-, $y$- and $z$-coordinate. This object representation is called polygon mesh. Additionally, each face has a normal vector and other information like a color and a texture. All properties of an object in this representation can be manipulated as desired. An alternative representation for three-dimensional objects in general is a point cloud, where 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. However, this yields not an as smooth surface as polygon meshes do, because between samples an interpolation is necessary. Moreover, due to measuring errors, holes or artifacts can be introduced.

### 4.1.2 Rendering Views of CAD Models

The following explains how multiple viewing perspectives of a single model are generated. The properties of each CAD model of ModelNet are stored in a file that is loaded and interpreted in Blender. To be able to refer to faces they are all part of the same basic coordinate system and are placed according to their defined vertices. Because all models are oriented beforehand, their top points along the $z$-axis. The origin of this coordinate system is set to the center of mass depending on the face areas. For adding lighting to the object a lamp needs to be placed inside that coordinate system. Blender offers several lamp types, but the Hemi lamp was chosen because it emits light radially from a plane. Therefore, a homogeneous illumination is ensured. Because all models have different heights and this type of lamp has no decay of intensity, it is placed above the objects far away from their origins. By assigning the lamp a direction $\boldsymbol{r}_l = (0, 0, 0)^T$ it emits light along the negative $z$-axis directly onto the model.

For rendering views, a camera object is required. Its parameters are left to the default values, except its view distance, which is set very high to work with all models. Hence, only its location and rotation needs to be set. Following the approach from [44, 45] the camera is elevated 30 degrees from the ground plane and points towards the origin. This results in the first rotation vector

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

in radians, where $r_{x,\text{deg}}$ is the rotation around the $x$-axis in degrees. The first, second and third element define a rotation around the $x$-, $y$- and $z$-axis, respectively. Because the camera points along the negative $z$-axis in its own coordinate system by default, $r_{x,\text{deg}} = 60$ corresponds to the mentioned setup. The next step is fitting the camera view to the model just by changing the location of the camera. Because Blender fits

the camera view exactly to the object, an image padding is necessary for having empty border regions for later convolutions. This is achieved by moving the camera away from the object along the line of their centers, i. e. along the negative view direction. It is moved by

$$\Delta d = \frac{d}{10} \tag{4.2}$$

where $d$ is the distance from the mesh origin to the camera. 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 $224 \times 224$ pixel Furthermore, it needs to be coped with aliasing. Because every pixel can only have a single color, diagonal lines 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 $\boldsymbol{c} = (64, 64, 64)$ resembling a dark gray for adding some noise to the views. A black background would yield pixel intensities of 0 and resembles, in general, no real-world views. Finally, this view is saved as a PNG file. 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

$$\boldsymbol{r}_c(v_i) = \left( \frac{60 \cdot \pi}{180}, 0, \frac{v_i \cdot \varphi \cdot \pi}{180} \right)^T \tag{4.3}$$

where $v_i$ is the view index, originally starting at 0, and $\varphi$ the moving interval in degrees. The latter is set to

$$\varphi = \frac{360}{n_v} = \frac{360}{12} = 30 \tag{4.4}$$

where $n_v$ equals the number of views. For the ability to compare this work to related researches, $n_v = 12$ is defined. The number of objects per set corresponds to [44]. That means, 90 objects per category class for the training set and 30 for the test set. This process is repeated for each CAD object.

### 4.1.3 Applying Color Marks

This work focuses on analyzing how views of the same objects but with different color marks contribute to the classification. Hence, their information content and significance is evaluated. This section covers how those color marks are applied. First, duplicates of every object are created. This is a trivial task because this can be done by rendering the native object first and then the colored one. In CAD modeling, objects or faces, in particular, are assigned a material with a color that reacts to light which produces shadows. Hence, coloring single faces differently becomes the chosen approach for applying

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

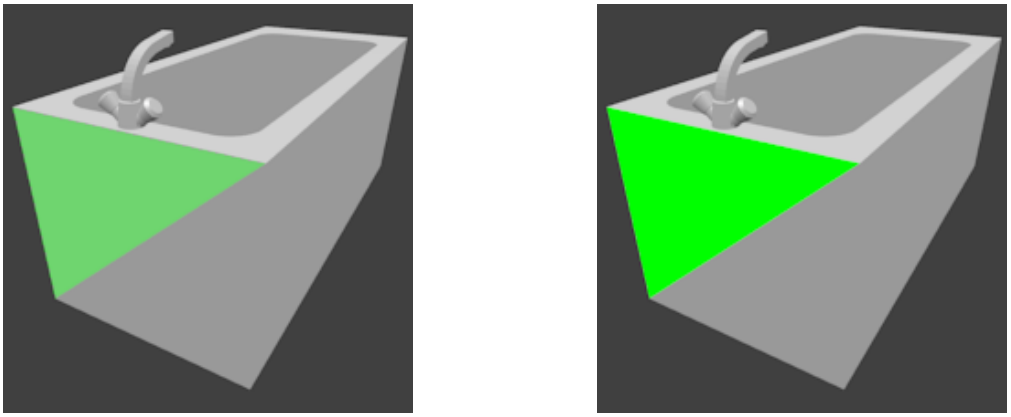
Figure 4.1: Threshold setup for filtering faces by their area size

color marks due to its simplicity. However, a well-suited face for being colored needs to be found. In the following those faces are referred to as an optimal face. It is important to filter all faces of an object by their area size. Fig. 4.1 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 decent size and can be seen and recognized by the network easily. This assumption was later verified by a not changing loss value during training. Like in Fig. 4.1a 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. 4.1b. 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. 4.1c. Additionally, larger faces should be preferred. Thus, the area of the optimal face have to satisfy

$$\alpha_{\text{opt}} > (\alpha_{\text{mean}} + \alpha_{\text{max}}) \cdot \lambda \tag{4.5}$$

where $\alpha$ are the related areas and $\lambda$ a scalar. With $\lambda = 0.3$ 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



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

Figure 4.2: Material manipulation on duplicated optimal faces

$\boldsymbol{p}_i$ is moved slightly to the center of the face $\mathbb{F} = (v_{f1}, v_{f2}, v_{f3})$ by

$$\boldsymbol{p}_{fi} = \boldsymbol{v}_{fi} + 0.05 \cdot (\boldsymbol{f}_o - \boldsymbol{v}_{fi}) \tag{4.6}$$

where $\boldsymbol{v}_{fi}$ is the related vertex and $\boldsymbol{f}_o$ 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, 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. 4.2, because the rendering engine does not know, which material should be rendered on the surface. In Fig. 4.2a 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 these 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. 4.2b the materials of all identical faces are changed, which leads to realistic color representations.

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 descending order. The intention is that choosing faces with the largest distances result in either an 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 need 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 matrix $\boldsymbol{X}_{set}$ 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, flattened to a vector $\boldsymbol{x}^{(i)}$ and then stored in one of the just mentioned view matrices. Simultaneously to the view gathering each one-hot encoded label needs to be created as well and put into a matrix $\boldsymbol{Y}_{set}$ 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 built 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 $n_y = n_c \cdot n_m$ classes, 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 classes, is either $n_y = n_c$ or $n_y = n_m$. In the multi-label classification case, both labels are used independently. Hence, the number of classes equals $n_y = n_c + n_m$. An example of those configurations is shown in Table 4.1.

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<br>bathtub_1<br>sofa_0<br>sofa_1 | bathtub<br>sofa<br>0<br>1 |
| Category | bathtub<br>sofa | n/a |
| Material | 0<br>1 | n/a |

When the number of classes is known, a one-hot encoding is performed on each label. Dividing each input and output matrix into datasets yields

$$\mathbb{D}_{\text{train}} = (\boldsymbol{X}_{\text{train}}, \boldsymbol{Y}_{\text{train}}) \tag{4.7a}$$
$$\mathbb{D}_{\text{test}} = (\boldsymbol{X}_{\text{test}}, \boldsymbol{Y}_{\text{test}}) \tag{4.7b}$$

where each column of $\boldsymbol{X}$ and $\boldsymbol{Y}$ 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 yielding $\tilde{\boldsymbol{X}}_{\text{set}}$ and $\tilde{\boldsymbol{Y}}_{\text{set}}$, respectively. 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 (4.7a) and (4.7b) 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 tasks, this approach is pursued. Furthermore, in [44] the VGG-M [6] architecture is used and in [12] GoogLeNet [47]. 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 [27] is chosen. It is very similar to the VGG-M

one but uses larger filters for convolutions and fewer nodes in the fully connected layers. However, the lesser parameters are a trade-off for accuracy.

In [12] it is shown that the performance of the network from [44] 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. 4.3. 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 $\boldsymbol{V}^{[l]}$ in the following. Fig. 4.4 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 $\boldsymbol{K}$ of size $7 \times 7$ and a stride of $s = 2$ on each $224 \times 224 \times 3$ input. Every filter extracts different features. In comparison, the original AlexNet uses filters of size $11 \times 11$ and a stride of $s = 4$. 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 of shape $Batch \times Height\_Out \times Width\_Out \times Filter\_Height \cdot Filter\_Width \cdot Channels\_In$ are extracted from the input tensor. 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 $\boldsymbol{V}^{[1]}$

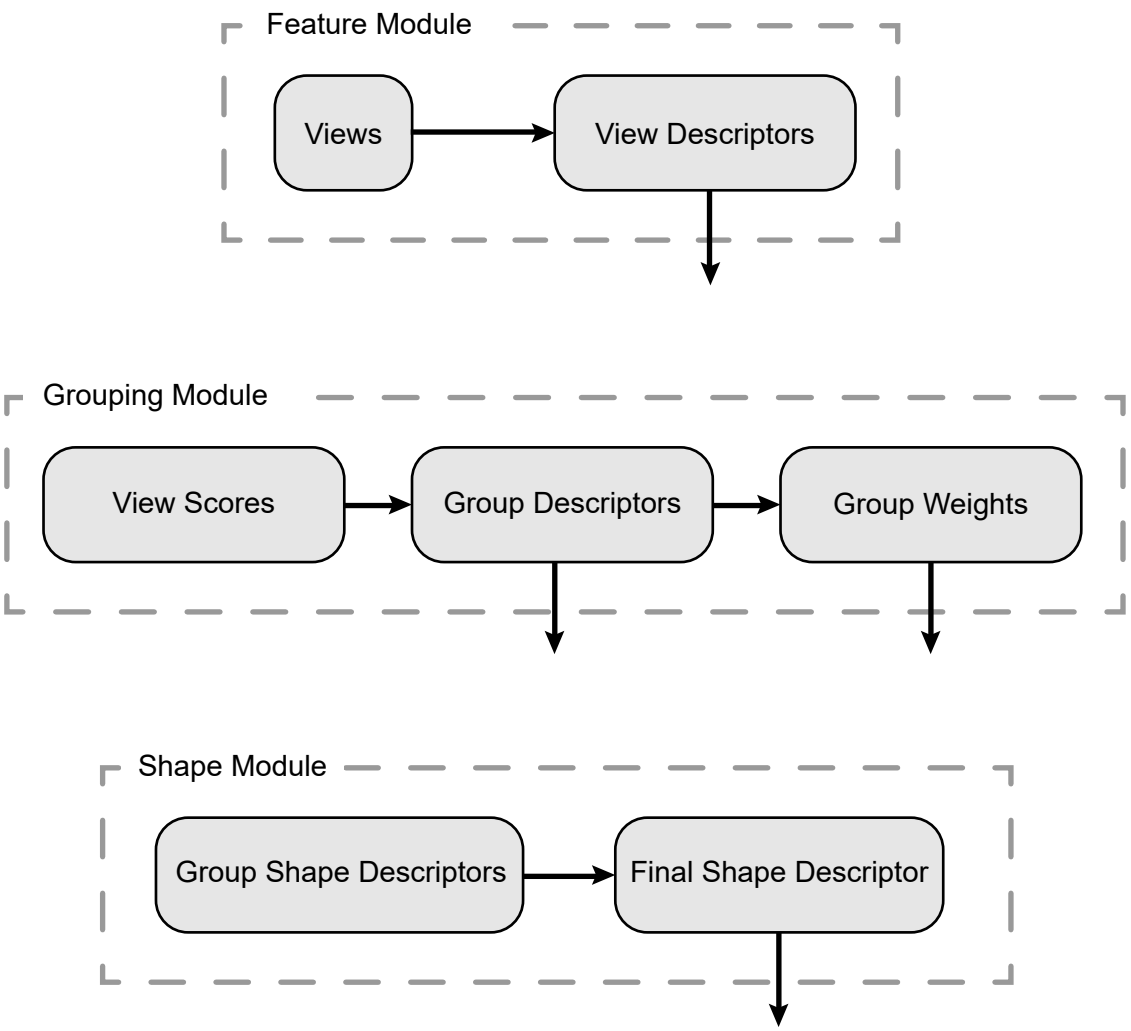
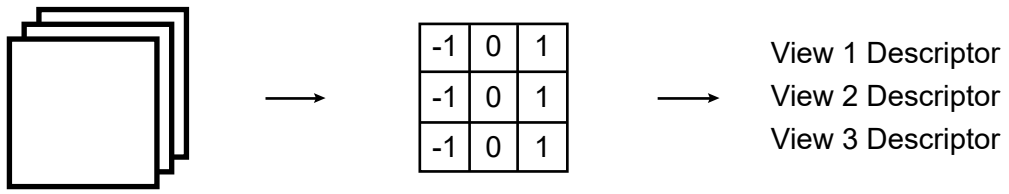



Figure 4.3: Modules of the multi-view architecture




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

Figure 4.4: Basic concept of the feature module

with a size of $1 \times 109 \times 109 \times 96$ in this case. 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 $\boldsymbol{K}_{\max}$ of size $3 \times 3$ and a stride of $s = 2$. 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. This yields a matrix of shape $1 \times 54 \times 54 \times 96$ for each pooling in the first main convolutional layer. For simplicity, it is defined that within each main layer the view descriptor is reused. Hence, the output of the $l$-th main layer is the view descriptor $\boldsymbol{V}^{[l]}$. 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 $\boldsymbol{K}$ of size $5 \times 5$ and a stride of $s = 2$. However, this time the input is padded in a way, that the output has the original input's size. This yields a convolution result $\boldsymbol{V}^{[2]}$ of shape $1 \times 27 \times 27 \times 256$. The max-pooling uses again a window $\boldsymbol{K}_{\max}$ of $3 \times 3$ and a stride of $s = 2$. The valid padding technique is applied resulting in a shape of $1 \times 13 \times 13 \times 256$. The third and fourth main layer use 384 filters $\boldsymbol{K}$ of size $3 \times 3$ for the convolution task with a stride of $s = 1$ each and the padding technique same. However, no pooling is performed. Hence, this yields a matrix $\boldsymbol{V}^{[3]}$ and $\boldsymbol{V}^{[4]}$ of shape $1 \times 13 \times 13 \times 384$ both the times. For the last main layer, the fifth one, a convolution with 256 filters $\boldsymbol{K}$ of size $3 \times 3$ is performed. The stride is $s = 1$ and the padding technique same, hence, the result $\boldsymbol{V}^{[5]}$ has a size of $1 \times 13 \times 13 \times 256$. The output's dimension is reduced with a valid max-pooling $\boldsymbol{K}_{\max}$ of size $3 \times 3$ and stride $s = 2$ to $1 \times 6 \times 6 \times 256$. In the end, this whole process results in a tensor $\bar{\boldsymbol{V}}^{[5]}$ containing each view's view descriptor $\boldsymbol{V}^{[5]}$ 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 $\boldsymbol{V}_v^{[5]}$ of each group $\mathbb{G}_g$ are then combined to a group descriptor $\boldsymbol{G}_g$. 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,

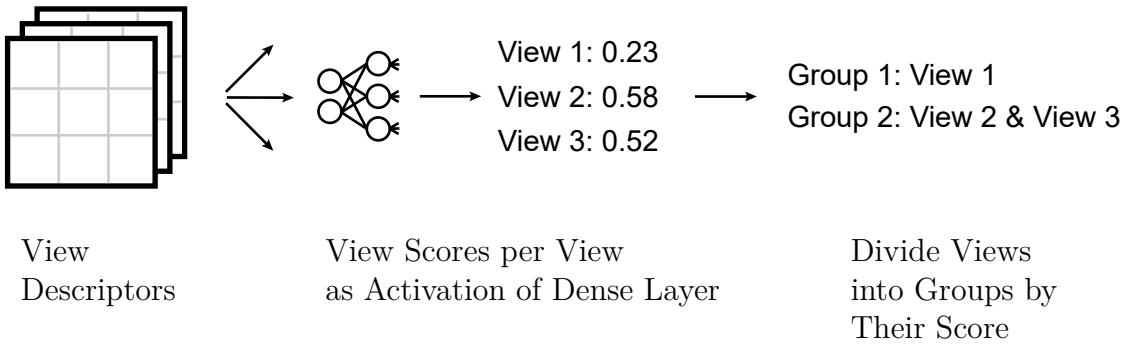



Figure 4.5: Group creation and view sorting in grouping module

that in [12] 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 some 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 used for training. 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 [12] representing a probability of discrimination. Because the sigmoid function saturates at values higher than around $|5|$, but the activation of the fully-connected layer is assumed to be larger, the natural logarithm of the activa-

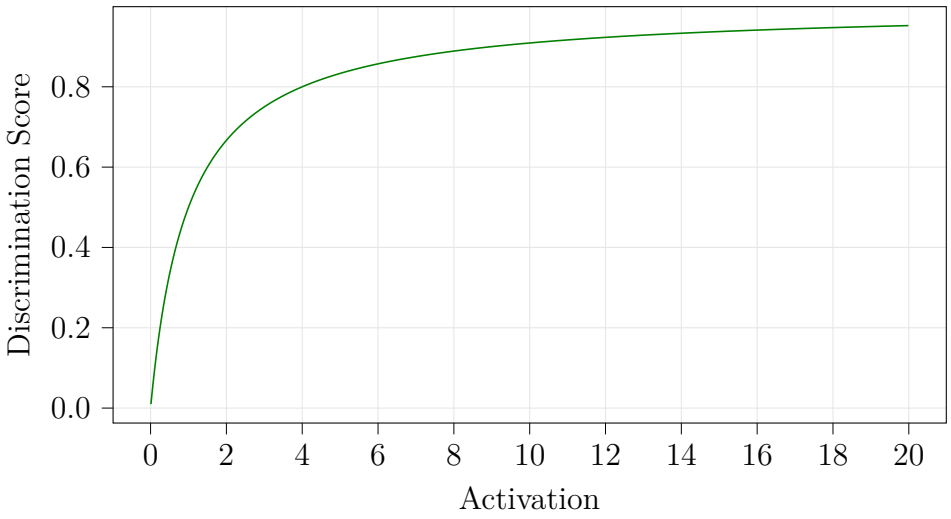



Figure 4.6: View discrimination score function plot

tion is computed first. This shifts the saturation to values that are presumably not in the range of the activation. For having a continuous function the absolute value of the activation is taken beforehand. This yields the expression for a view's score

$$\theta = \text{sigmoid}\left(\log\left(\left|a^{[d4]}\right| + \varepsilon\right)\right) \tag{4.8}$$

where $a^{[d4]}$ 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. 4.6. All view discrimination scores for a batch are stored in a tensor $\bar{\boldsymbol{\Theta}}$ with shape $Batch \times Views$.

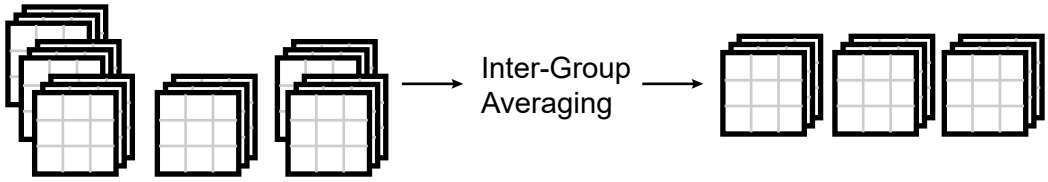
Dependent on each view score, the related views are divided into groups. For maximum flexibility, the number of groups $n_g$ equals the number of views $n_v$. Hence, the size of each group $r_g$ is related to the number of views $n_v$ per object and the range of possible view scores $r_\theta$. With the limit of (4.8)

$$\lim_{x\to\infty} \text{sigmoid}\left(\log\left(\left|a^{[d4]}\right| + \varepsilon\right)\right) = 1 \tag{4.9}$$

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

$$r_g = \frac{r_\theta}{n_v} = \frac{1}{12} \approx 0.083 \tag{4.10}$$

as each group's size. Hence, a group $\mathbb{G}_g$ contains views with scores of $(g-1)\cdot r_g \leq \theta < g\cdot r_\theta$ where $g = 1, 2, \ldots, n_v$. Fig. 4.5 illustrates the basic view score calculation and group




View Descriptors per Group

Single Group Descriptor per Group

Figure 4.7: Generating group descriptors by calculating the average of related features.

dividing. This example assumes a group size of 0.33. The group to which each view $\tilde{\boldsymbol{X}}_v^{(i)}$ belongs is calculated by

$$g = \text{floor}\left(\frac{\boldsymbol{\theta}_v}{r_g}\right) \tag{4.11}$$

yielding the group's index, where $\boldsymbol{\theta}_v$ is the view score of the $v$-th view of the corresponding multi-view sample. Those are stored for every view resulting in the group index vector $\boldsymbol{g}_{\text{idx}} = (g_1, g_2, \cdots, g_{n_v})^T$. The subscripts correspond to the view indices, e.g. $g_2$ belongs to $\tilde{\boldsymbol{X}}_2^{(i)}$. 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. 4.7. 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

$$\boldsymbol{G}_g = \frac{\sum_{\boldsymbol{D} \in \mathbb{G}_g} \boldsymbol{D}}{|\mathbb{G}_g|} \tag{4.12}$$

with the same size as a view descriptor, where $\boldsymbol{D}$ is a view descriptor $\boldsymbol{V}^{[5]}$ of group $\mathbb{G}_g$. The addition and division are 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 $\boldsymbol{G}_g$ gets a weight $w_g$ 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

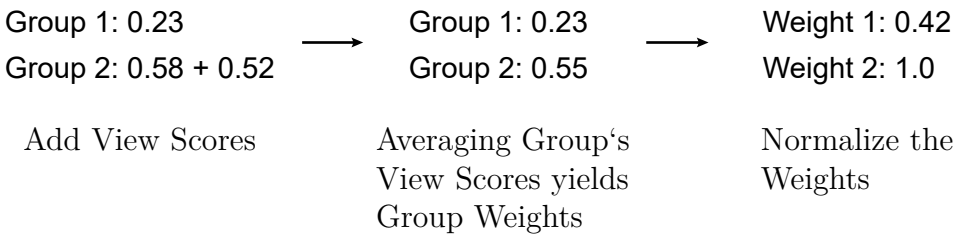



Figure 4.8: Calculation of group weights in grouping module

and divided by their number for calculating a mean. In mathematical terms,

$$w_g = \frac{\sum_{\boldsymbol{V} \in \mathbb{G}_g} \theta(\boldsymbol{V})}{|\mathbb{G}_g|} \tag{4.13}$$

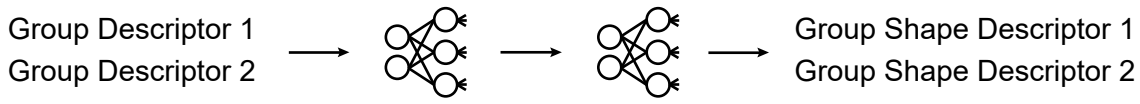
calculates the weight of the $g$-th group, where $\theta(\boldsymbol{V})$ is the score of a view $\boldsymbol{V}$ in $\mathbb{G}_g$. Furthermore, the weight of a group is normalized by

$$w_g = \frac{\hat{w}_g}{\max(\theta(\mathbb{G}_g))} \tag{4.14}$$

to a range of 0 and 1, where $\max(\theta(\mathbb{G}_g))$ yields the maximum score of the given group, for being able to compare it with the ones in [12]. The calculation of the group weights is illustrated in Fig. 4.8 referring to the example in Fig. 4.5. The weights of the padded group descriptors equal 0 for being ignored in a later matrix multiplication. Hence, the shape of the tensor $\bar{\boldsymbol{W}}_g$ 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 of 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 $\boldsymbol{G}_g$ is flattened into a column-vector $\boldsymbol{g}_g$. 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,




Group Descriptors From Grouping Module

Dense Layer Operation Results in a Shape Descriptor per Group

Figure 4.9: 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.

the two fully-connected operations

$$\boldsymbol{a}_g^{[6]} = \phi(\boldsymbol{W}^{[6]}\boldsymbol{g}_g + \boldsymbol{b}^{[6]}) \tag{4.15a}$$

$$\boldsymbol{a}_g^{[7]} = \phi(\boldsymbol{W}^{[7]}\boldsymbol{a}_g^{[7]} + \boldsymbol{b}^{[7]}) \tag{4.15b}$$

are performed as illustrated in Fig. 4.9. 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 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 $\bar{\boldsymbol{W}}_g$ 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, $\bar{\boldsymbol{W}}_{g,\text{sum}}$ 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 $\bar{\boldsymbol{W}}_{g,\text{sum}}$ of the shape $Batch \times 1 \times 1$ with weight sums. Now the weighted average can be computed by

$$\boldsymbol{S} = \frac{\boldsymbol{S}_g \boldsymbol{W}_g}{\boldsymbol{W}_{g,\text{sum}}} \tag{4.16}$$

where the division is performed element-wise. Due to the matrix multiplication, the padded entries have no impact. The result of the assembled tensor $\bar{\boldsymbol{S}}$ has a shape of

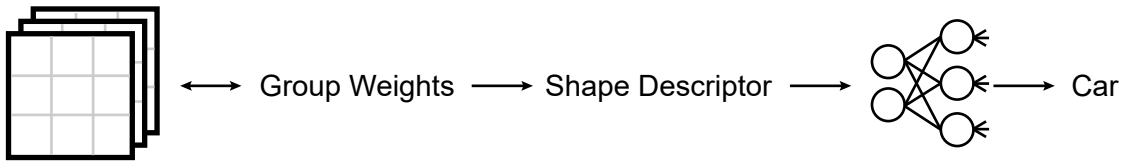



Figure 4.10: Basic concept of the shape module combined with a classification. A weighted average of the group shape descriptors with the related group weights is calculated yielding a single shape descriptor. It is fed into the last fully-connected layer resulting in the prediction of the network.

$Batch \times 4096 \times 1$ and represents the final single shape descriptor. This can be made more compact by changing the shape to $Batch \times 4096$ without losing 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 to the ones earlier and are described by

$$\boldsymbol{z}^{[8]} = \boldsymbol{W}^{[8]}\boldsymbol{S} + \boldsymbol{b}^{[8]} \tag{4.17}$$

using the related weights and biases. However, no activation function is applied here. For making the predictions $\boldsymbol{z}^{[8]}$ interpretable, they are fed into a softmax layer. This outputs a valid probability distribution $\hat{\boldsymbol{y}}$ depending on all its inputs $\boldsymbol{z}^{[8]}$. 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. 4.10.

## 4.4 Training the Architecture

The multi-view network architecture is trained by inputting the generated multi-view images $\tilde{\boldsymbol{X}}^{(i)}_{train}$ and $\tilde{\boldsymbol{X}}^{(i)}_{test}$ and comparing the prediction $\hat{\boldsymbol{y}}^{(i)}$ of the network with the corresponding one-hot encoded labels $\tilde{\boldsymbol{y}}^{(i)}_{train}$ and $\tilde{\boldsymbol{y}}^{(i)}_{test}$. 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 $m_b = floor(128/12) = 10$. 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 the 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

$$\tilde{\boldsymbol{L}} = \left( \left[ \tilde{\boldsymbol{X}}_{test}^{(1)}, \tilde{\boldsymbol{y}}_{test}^{(1)} \right], \left[ \tilde{\boldsymbol{X}}_{test}^{(2)}, \tilde{\boldsymbol{y}}_{test}^{(2)} \right], \cdots, \left[ \tilde{\boldsymbol{X}}_{test}^{(m_{test})}, \tilde{\boldsymbol{y}}_{test}^{(m_{test})} \right] \right) \tag{4.18}$$

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, 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 $\hat{\bar{\tilde{\boldsymbol{Y}}}}^{(i)}$ of the network needs to be compared to the ground-truth labels $\bar{\tilde{\boldsymbol{Y}}}^{(i)}$ of the $i$-th batch 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 $\hat{\tilde{\boldsymbol{y}}}^{(i)}$ is located. The same operation is performed on the corresponding ground-truth label $\tilde{\boldsymbol{y}}^{(i)}$. 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 $\bar{\boldsymbol{e}}$. Finally, the accuracy $\bar{\alpha}$ is calculated by

$$\bar{\alpha}^{(i)} = \frac{\sum_j \bar{e}_j^{(i)}}{|\bar{\boldsymbol{e}}^{(i)}|} \tag{4.19}$$

for the current batch $i$. In the case of multi-label classification, a probability threshold needs to be defined when a predicted feature is actually considered predicted. In this case, the threshold is $p_{\text{thres}} = 0.5$, 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 $\bar{\boldsymbol{e}}^{(i)}$ as well. The accuracy is calculated with (4.19).

Furthermore, a starting learning rate is necessary. Because finding it by trial-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

$$\gamma(\tau + 1) = \lambda\gamma(\tau) \tag{4.20}$$

where $\lambda = 1.1$ is the scaling factor and $\tau$ 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 fewer elements than the ones before. Hence, a weighted average is performed with the batch sizes as the weights. This yields the averaged cost

$$J\left(\hat{\bar{\boldsymbol{Y}}}, \bar{\bar{\boldsymbol{Y}}}\right) = \frac{\sum_i^{n_b} m_{b_i} \cdot J\left(\hat{\bar{\boldsymbol{Y}}}^{(i)}, \bar{\bar{\boldsymbol{Y}}}^{(i)}\right)}{|\boldsymbol{m_b}|} \tag{4.21}$$

and the averaged accuracy

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

where $\boldsymbol{m_b}$ 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 the 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

$$q_i = \frac{\sum_{j=\max(1,i-f+1)}^{i} p_j}{\max(0, i-f)} \tag{4.23}$$

where $f$ 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 $f = \text{floor}\,(0.1\,|\boldsymbol{p}|)$ 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 an 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

$$\alpha = \frac{TP}{TP + FP} \tag{4.24}$$

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 $\tilde{\tilde{\boldsymbol{X}}}^{(i)}$ representing the $i$-th 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, each view's activations are split across the last dimension that represents the features. Each feature's values $\boldsymbol{F}_i$ are normalized by

$$\boldsymbol{F}_{i,norm} = \frac{\boldsymbol{F}_i - \min(\boldsymbol{F}_i)}{\max(\boldsymbol{F}_i) - \min(\boldsymbol{F}_i)} \tag{4.25}$$

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 $\boldsymbol{S}_v^{(i)}$ is computed for each view $\tilde{\boldsymbol{X}}_v^{(i)}$ 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

$$\boldsymbol{S}_j^{(i)} = \frac{\partial \boldsymbol{z}^{(i)}}{\partial \tilde{\boldsymbol{X}}_v^{(i)}} \tag{4.26}$$

as a general expression. For plotting each saliency map is normalized with (4.25) and visualized in gray-scale.

# Chapter 5

# Results

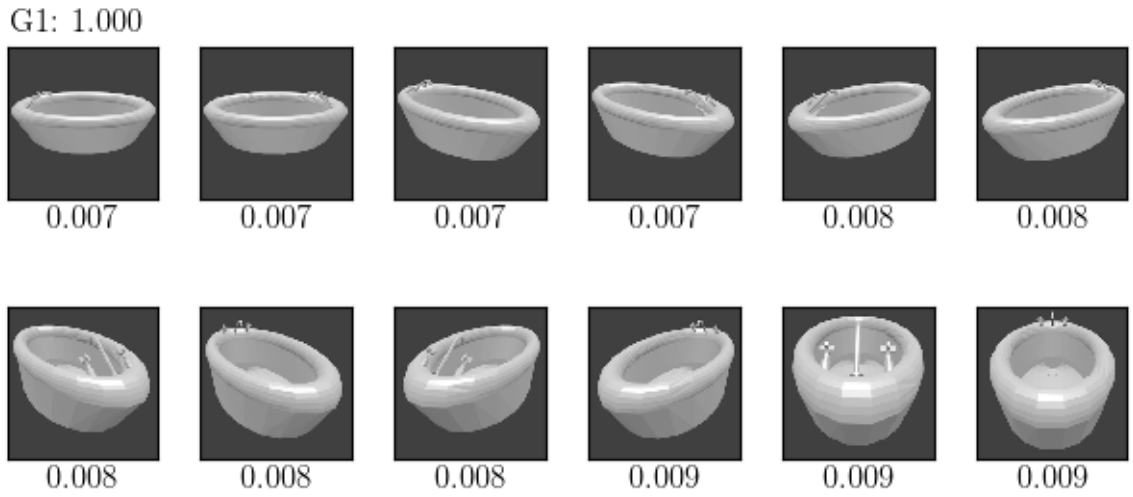
In this chapter, the results of the network and its components are going to be presented. First, the grouping mechanism is evaluated due to being the core of the architecture. Then the overall performance of the network is discussed, followed by an examination of misclassifications. There are several networks training with an increasing number of category and material classes for being able to compare and dedicate possible occurring effects. It starts with only a single category, in particular, bathtubs, with first 3 different materials. Those materials embrace the raw object, a green feature and a red one. This is further increased to 6 material features containing a green and red material, two green materials, and two red materials. Finally, four categories in total are classified including additionally dressers, monitors and sofas. Here an identical process of adding materials is performed in the same order as before. This leads to a total of 9 networks. However, for testing different hyperparameters, more networks have been trained on the dataset with bathtubs and 3 material. In the following the syntax *#categories-#materials* refers to the corresponding trained network. Every model is trained for 20 epochs with a batch size of 8. Each batch element contains 12 rendered views of an object. The initial learning rate is set to 0.0001. Furthermore, the dropout probability of layer 6 and 7 is specified as 0.5, which matches the AlexNet configuration. Any dataset samples presented or predicted belong to the test set.

## 5.1 View to Group Classification

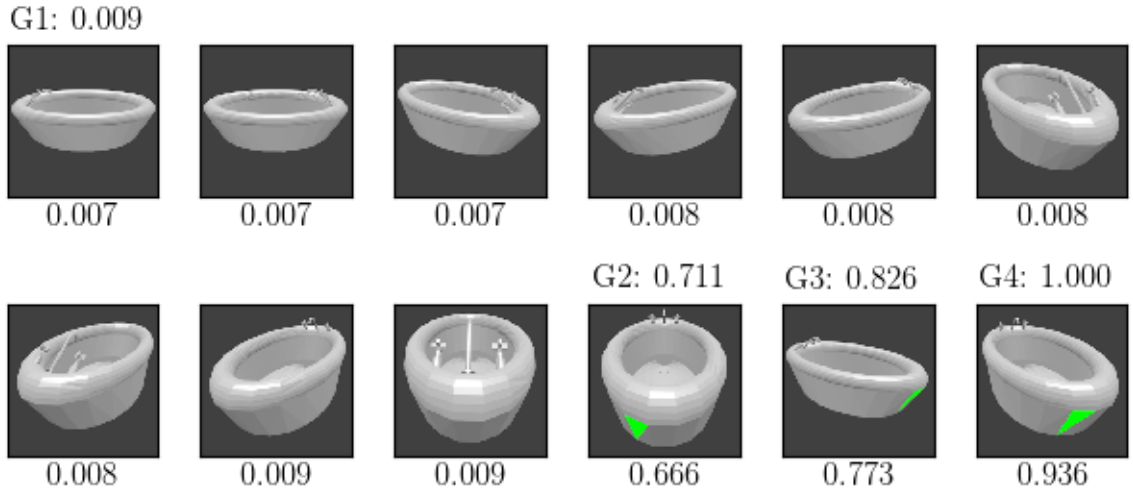
Because the grouping mechanism supplies the core functionality of the network architecture it is evaluated first. Even if the overall performance would yield satisfiable results, but the grouping mechanism would fail the original intention, it would need to be revised. In contrast, if the results of the network are not satisfiable, the grouping algorithm could be the cause.

The easiest case for evaluation is a single object category with three material features. Here the network only needs to find views with a material for assigning them a high discrimination score. It is supposed, that those views have the highest scores of all views and, hence, are members of a group with a high weight. Views where no material is seen, should have a score close to zero because the final prediction cannot rely on them

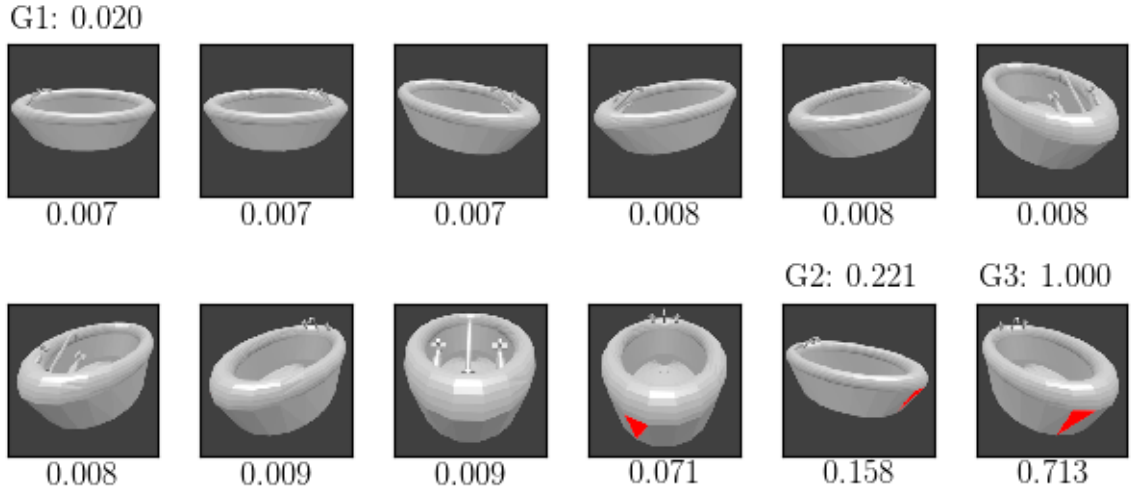
at all. Thus, this configuration is the most interpretable one. The group dividing for the 0-3 network is shown in Fig. 5.1. Each number below a view refers to its score. The text above views shows the group index with its corresponding weight. All views appear in ascending order by their score. Hence, all subsequent views are part of a group until another group is mentioned. In Fig. 5.1a a blank object is classified. Hence, every views is similar discriminative, due to no available colored material. Thus, the view scores are almost identical. Those little changes presumably depend on a different weight initialization and would even out after more training epochs. Although the scores are very low, the views are fully taken into account because they all belong to the same group with a weight of 1. However, in this particular case, a normalization of the group weight is not necessary. Without one the group weight would be $w_g = 0.0079$, thus, decreasing the shape descriptor enormously, but the network would learn that a descriptor close to 0 represents a blank object. The decision rule would be, if the descriptor represents no feature, the objects show no feature. With the classification of more categories, though, this is not possible anymore, because a very small descriptor cannot just represent any blank object, but the object category class. If there are two objects, for example, and only one view each shows a different feature with a small discrimination score, they would be divided into two categories. Without a normalization, all views would pretty much account to the same amount to the shape descriptor. Hence, the weights in the fully-connected layers for each specific present feature needs to be very large. With normalization, however, the group with one view is much higher weighted than the not discriminative views. Hence, normalization removes noise, i.e. not discriminative views, that could influence the prediction unfavorably. Fig. 5.1b and Fig. 5.1c show the expected result. The views showing the material feature are by far the top-rated views referring to their score. It looks like, that the network prefers the slightly tilted vertical edge with a feature to its right for recognizing material features. This exact edge is not visible in the first view showing a feature, due to the change in perspective. Due to the mesh representation of objects, all material features are triangles. Perhaps the dataset contains more features following this shape than in rotated ones, hence, the network focuses on that correlation. Moreover, both figures show exactly the same order. This shows, that the weights for each color channel are optimized in the same direction. However, the views with the green material are in a closer range compared to the ones with the red material. The latter differs extremely. The least discriminative view containing a material is closer to the not discriminative views than the discriminative ones. Considering several different training sessions of this configuration, it is observed, that those scores of each material usually are similar to each other. Hence, it is assumed, that a bad weight initialization caused this problem and a longer training would have evened it out. Nevertheless, the group metric is at 1 for every material. It should be noted, that always only a metric for the non-blank materials is given. In Fig. 5.2 the grouping of the 0-4 network is shown, which additionally classifies green-red material features. The divide of the blank object is omitted but all views have a score of 0.229. Compared to the 0-3 network, this network finds different correlations, because the order of views is




(a) Blank

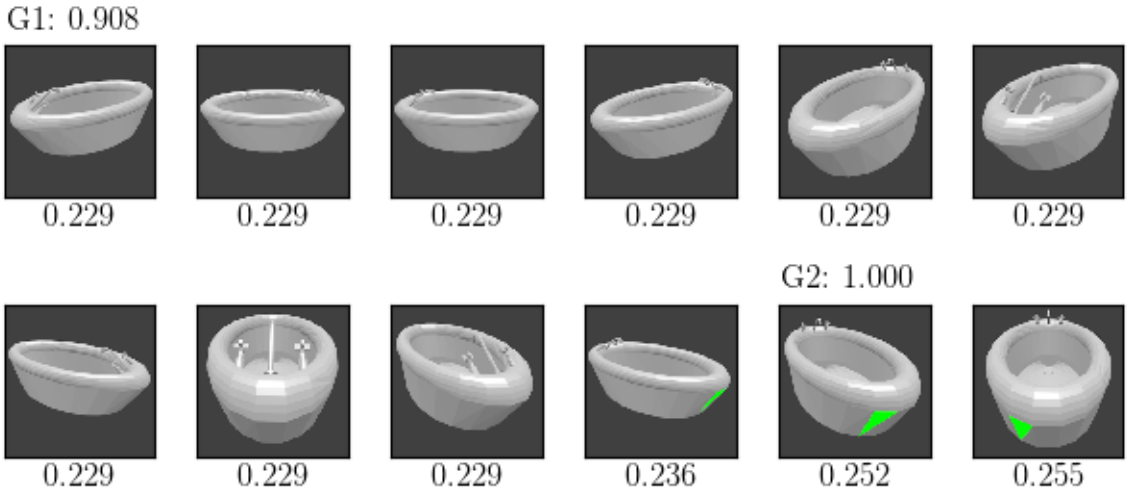



(b) Green material

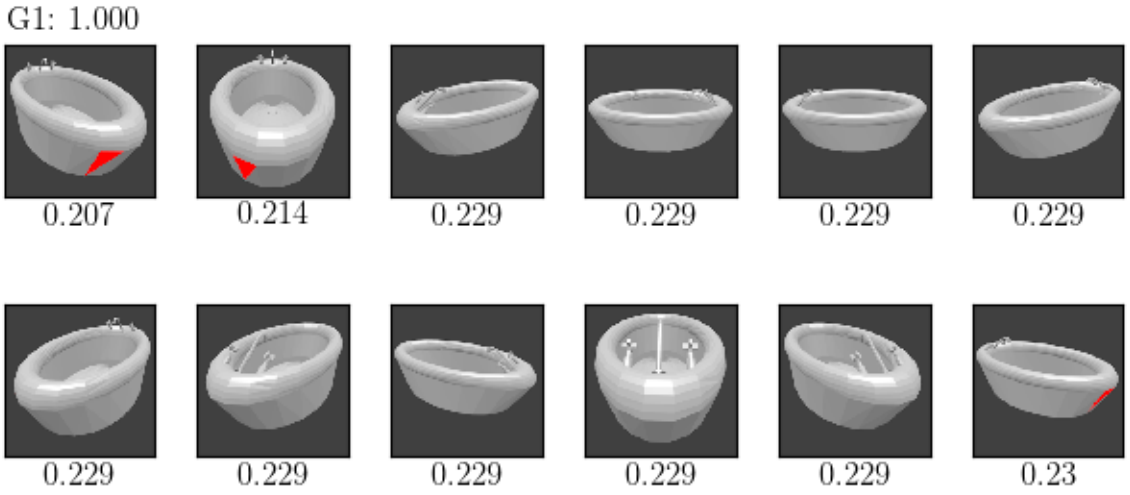



(c) Red material

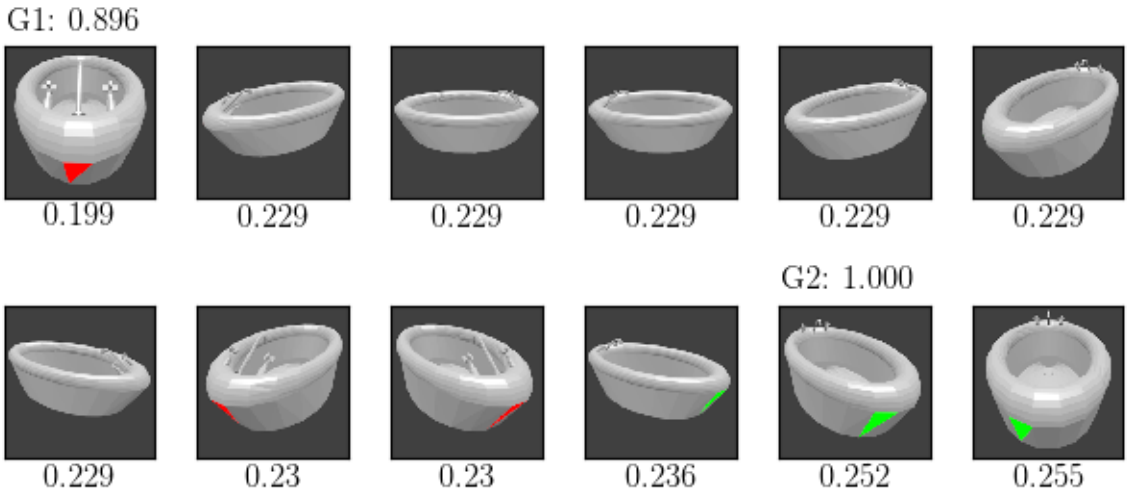
Figure 5.1: Grouping in 0-3 network




(a) Green



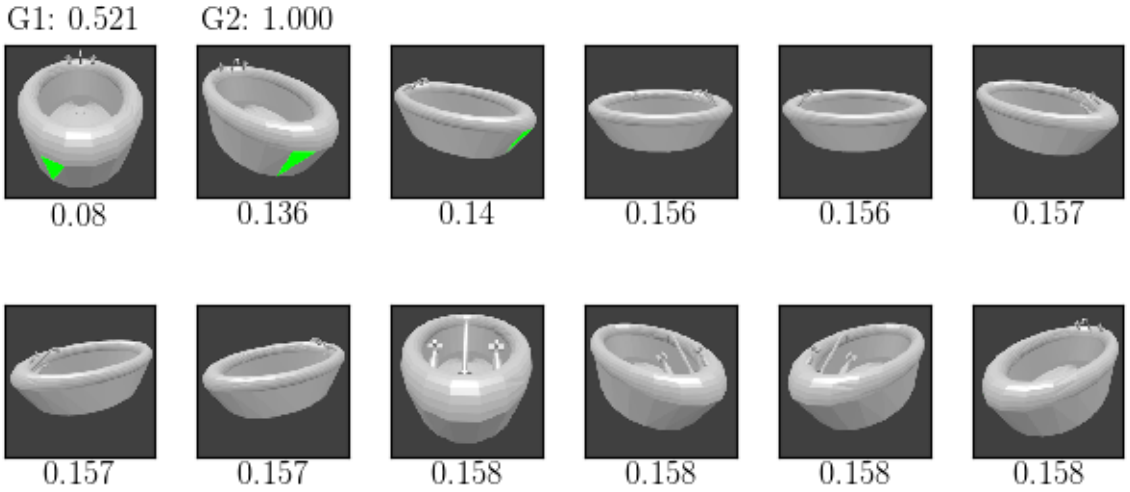

(b) Red



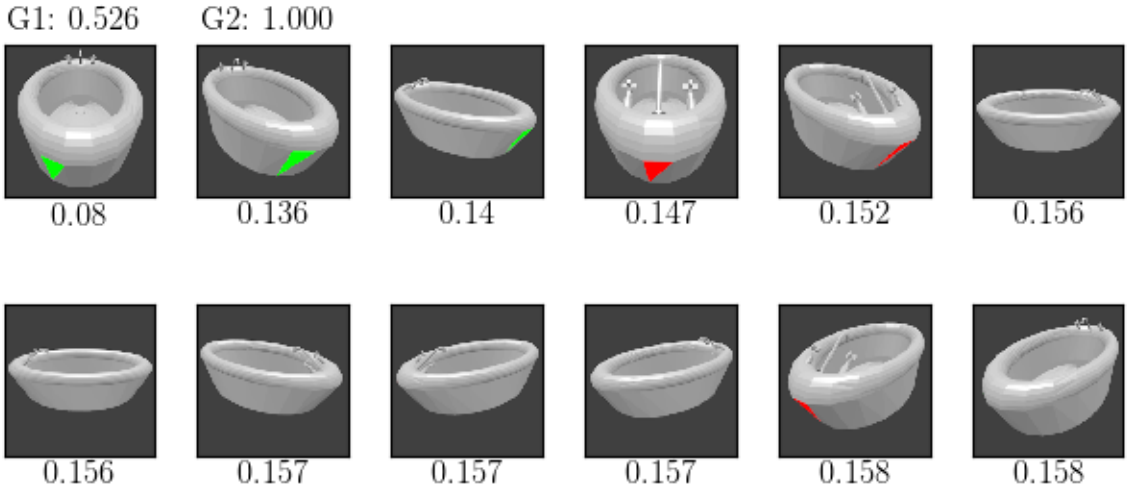

(c) Green-red

Figure 5.2: Grouping in 0-4 network

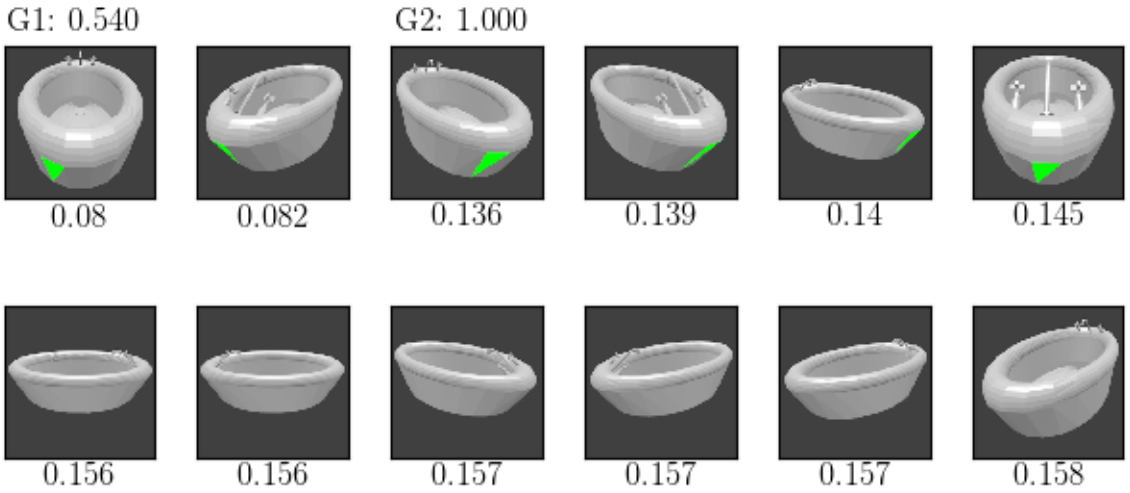
different. Furthermore, not only the views with an available material are rated high like in Fig. 5.2b, but also the normal views like in Fig. 5.2a. The reason is, that there could be two material features on a single object. Hence, a view where a feature is supposed to be present is rated high, although it is missing. This means, the discriminative information is the missing feature, thus, the view is more discriminative than it first seemed. Therefore, a worse group metric than before is expected. This is reflected by the values 0.735, 0.475 and 0.800 for each class. The averaged group metric is 0.670. This shows that the network prefers the green material as well because it is more often in the top group than the red one. This is also seen in the red divide, where two of three material views are the least discriminative ones. That suggests, that the red feature is treated as a sufficient criterion, while the green one is treated as the necessary criterion. Moreover, it is not surprising, that the green and red features are mostly present in the top group. Next, the grouping for the 0-5 network is examined, that additionally classifies green-green features. The important results can be seen in Fig. 5.3. The divide of the blank object is similar to the ones before. There is only one group and the view discriminative scores are in a range of 0.156 and 0.158. Not much changes for the divide of the red material. The second view from earlier changes its position with the top view. Moreover, the score of every view is decreased by about 0.07. However, that the top view shows a red material is possibly for adding certainty for the classification with avoiding large weights. For the green material, the material views are the least discriminative ones according to their score. This is shown in Fig. 5.3a. That effect must be due to the additional green-green feature. An explanation could be, that the network now rates views by missing material features. Hence, the shape descriptor becomes smaller the more features are available. This theory is almost supported by the divide of green-red, that is shown in Fig. 5.3b. Moreover, referring to the view scores, the views showing a green material have the exact same score than in the single green case. However, the theory is changed in so far, that not the number of features is crucial, but rather that every material feature has a range in which it is classified. Green materials are less discriminative than red ones and red ones are less discriminative than blank ones. Using the green-green divide from Fig. 5.3c confirms this more. The views with a green feature are less discriminative than the blank ones and also have a lesser score than the red ones in the earlier cases. Thus, the shape descriptor is large, if no feature is visible, neutral if red ones are visible, slightly smaller if green ones are visible, small if both features are visible and very small if two green features are visible. For this network, the group metrics are 0.419, 0.469, 0.768, 0.750 with an average of 0.601. Those of each single feature and of each double feature are similar and the average is smaller than in the 0-4 case. This supports the grouping theory even more in so far, that non-feature views are mainly members of the top group. For having balanced evaluation results, red-red material features are added, yielding the 0-6 network. Grouping blank views is almost identical to the earlier cases with the exception of view scores in a range of 0.46 to 0.472 The order of views for the green material has not changed notably. However, an additional group is created and the scores of the views with a visible feature changed to 0.336, 0.048 and 0.388. The




(a) Green

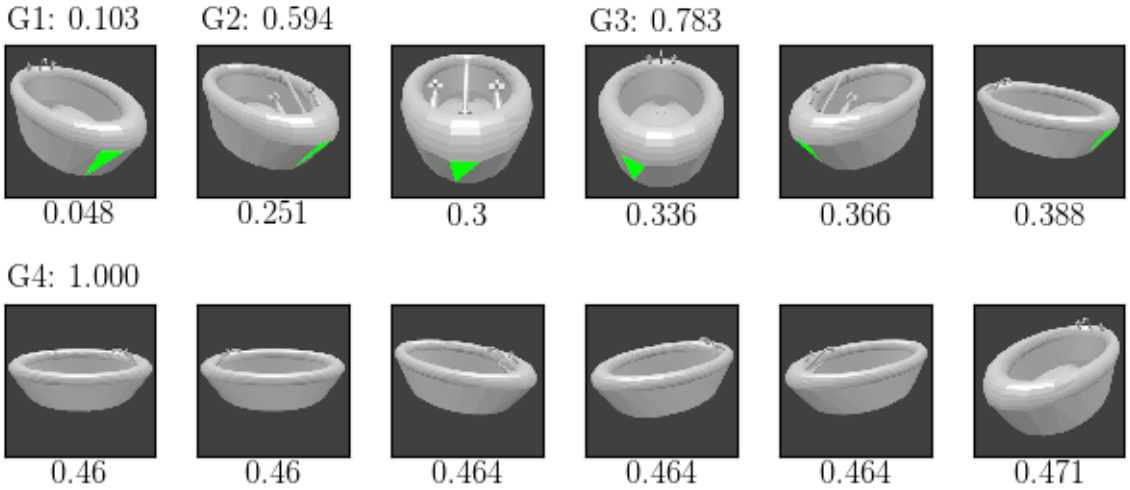



(b) Green-red

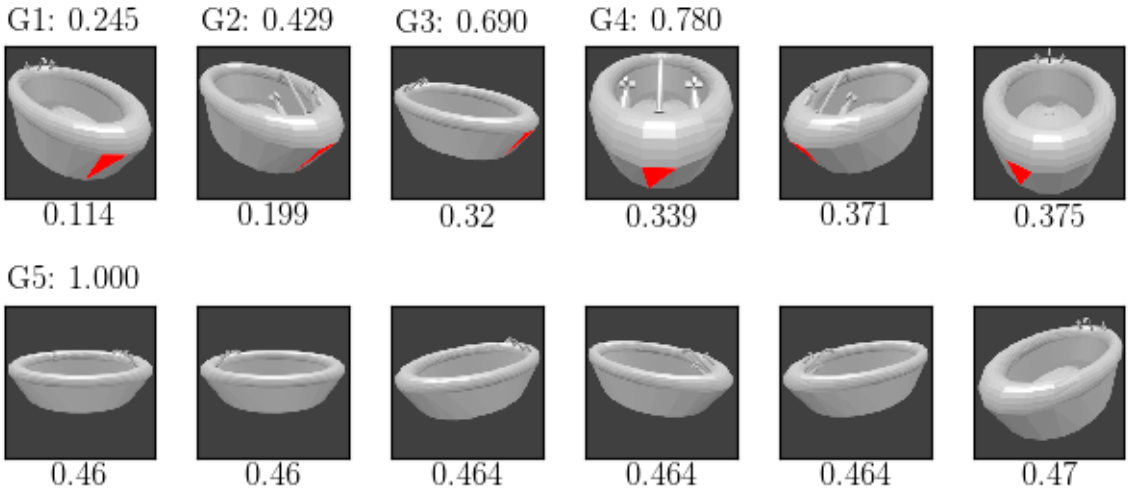



(c) Green-green

Figure 5.3: Grouping in 0-5 network




(a) Green-green




(b) Red-red

Figure 5.4: Grouping in 0-6 network

remaining scores are in a range of 0.46 to 0.472 which matches the blank case. In the red material case, the 0-5 top view is now the third-to-last. Hence, all views showing materials are the least discriminative ones with scores of 0.106, 0.313 and 0.358. The remaining views have identical scores to the other material cases. Moreover, all of those views are divided into four groups. In comparison, the green material and the red one have similar view scores, where the scores of the blank views are identical. For the green-red case, all views not showing a material are more discriminative than the ones showing one. This matches almost the grouping in the 0-5 network, but with five groups The top group only contains non-feature views. Furthermore, the scores of views with a green feature have the same score than in the single green feature case. However, this is applicable to the views with red features. This is due to the fact, that in this case, the second optimal face is colored red, hence, there are no comparable red faces. For the green-green and red-red case, that are shown in Fig. 5.4, the divide is almost identical. The first one uses four groups the second five. Though the view scores are similar, it is noticeable, that compared to each different colored feature in the green-red case views with a now green feature are less discriminative than views with a now red feature. Although, both times the general score is higher. Fortunately, this does not debunk the assumed theory of the group mechanism. The corresponding group metric scores are 0.217, 0.253, 0.420, 0.381, 0.504 with an average of 0.355. Their distribution is as expected, but with smaller values than in earlier networks. This is due to the larger number of groups and the higher diversity of non-feature views to feature-views.

For a full understanding of the grouping mechanism, it needs to be examined, if the assumed theory is still valid if additional categories are classified. This is done in the same order of number of classifications as with only material features. First, the 4-0 network is evaluated. It is supposed, that similar views like opposite perspectives of a symmetrical object are divided into the same group because they contain similar features. For example, the views of the left and right side of a car look almost identical, due to having the same contours but in a mirrored direction. A grouping divide is shown in Fig. 5.5. It can be seen that the assumption is wrong. Instead, the network prefers objects or features, respectively, that are diagonal from top left to bottom right. This is validated by more predictions. That such preferences are learned is already assumed from the 0-3 grouping. The scores, however, are difficult to interpret. For doing this it is necessary to compare all weights and activations, without achieving a real benefit, because the functionality of the grouping algorithm is already validated by using only materials. Hence, the grouping for the remaining networks is briefly summarized. Their divides are shown in Fig. 5.6, Fig. 5.7, Fig. 5.8 and Fig. 5.9. It can be seen, that each top group always contains at least one view with a visible feature. Furthermore, the remaining ones are not necessarily discriminative, because depending on the number of materials it is more important to find features classifying the category. Moreover, the more classes exist, the more groups are created. That suggests, that the assumption from before, that shape descriptors of different classes are in different ranges, could be valid, because this way the network is more flexible in weighting views.

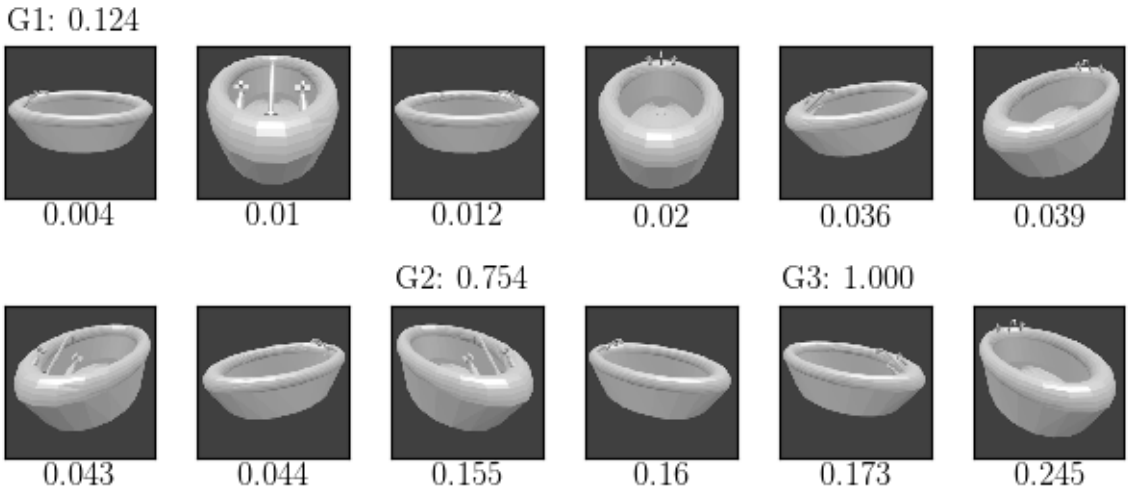



Figure 5.5: Grouping in 4-0 network

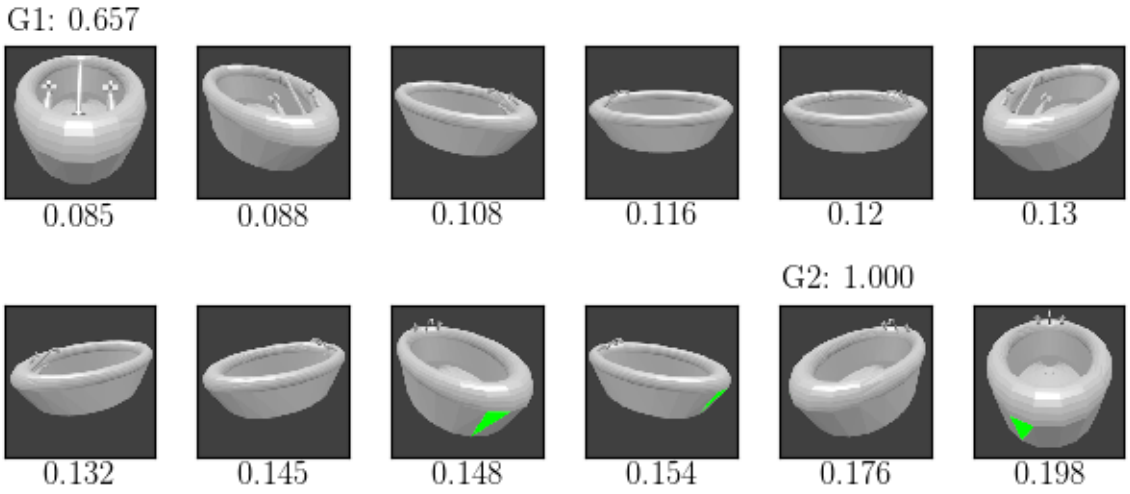



Figure 5.6: Grouping in 4-3 network

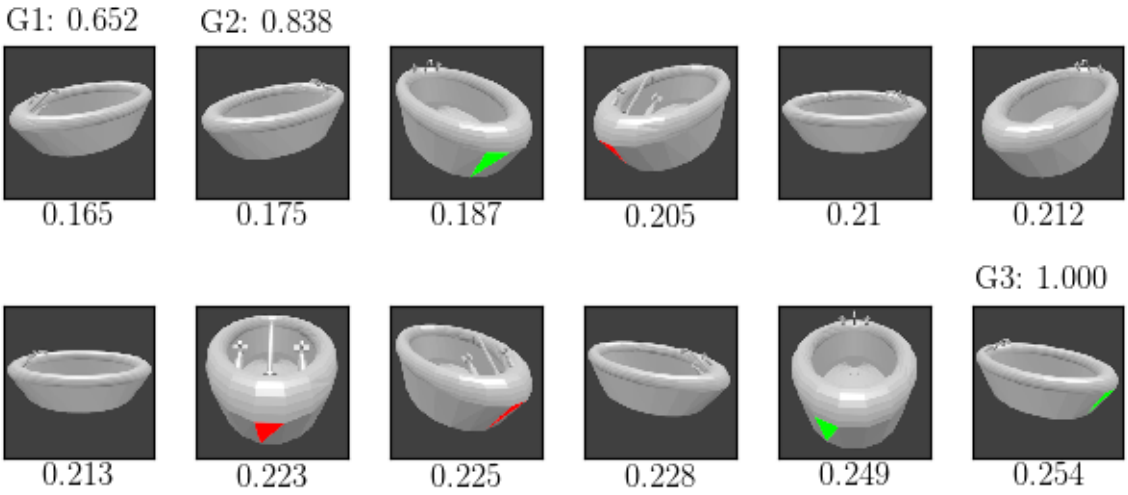



Figure 5.7: Grouping in 4-4 network

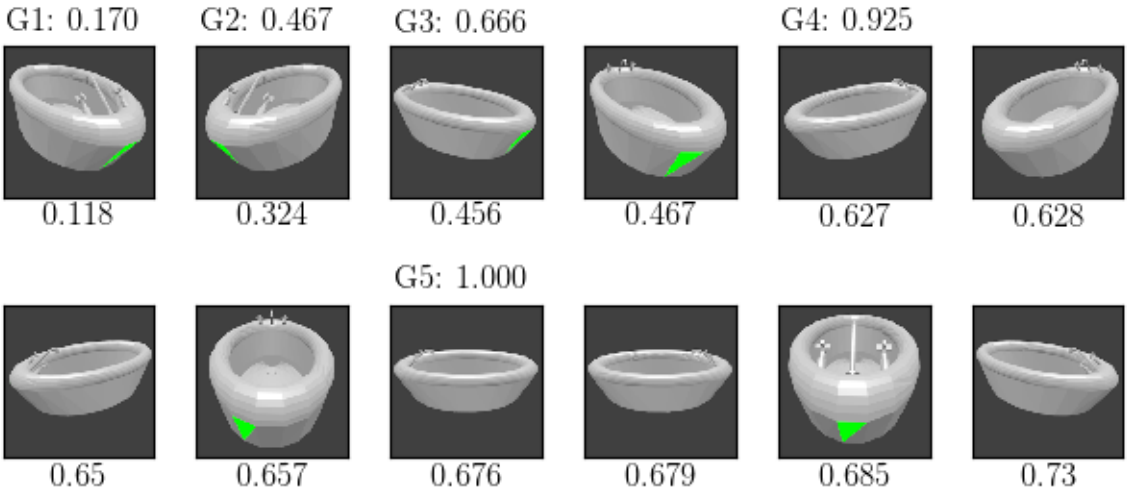



Figure 5.8: Grouping in 4-5 network

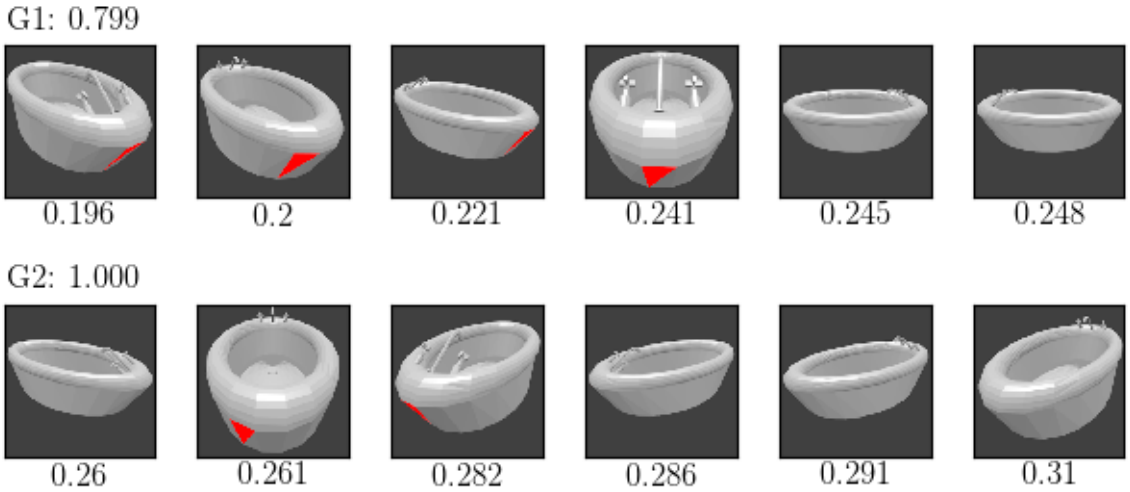



Figure 5.9: Grouping in 4-6 network

## 5.2 Overall Performance

For evaluating the overall performances of the network architecture, the choice of hyperparameters is explained. In Fig. 5.10 the increasing of the learning rate against the related loss is shown for finding the range of optimal learning rates. Additionally, the change in loss is outlined. This is performed on the 0-3 network. At around $10^{-4}$ the loss starts to decrease slightly. At around 0.004 its decrease gets strikingly faster until at around 0.01 it starts to increase drastically. Although for this network, in particular, a learning rate of $10^{-2}$ seems to be suited well, a general initial learning rate of $10^{-3}$ is chosen. On one hand, this is the most basic network, hence, it is supposed, that for more complicated ones, a smaller learning rate is better suited due to the more complex cost function. Furthermore, interpreting those graphs is time-consuming and for more complicated networks not that easy anymore, because the loss changes more rapidly. On the other hand, a learning rate of $10^{-2}$ is close to the increase. Hence, if the learning rate is shifted, the parameters of the network would be changed tremendously. It was actually verified, that a learning rate of $10^{-3}$ is a satisfiable choice for more complex networks because it lies close to the upper bound of the optimal learning rates range. As a default value for all networks, it works as well, though.

Furthermore, the decreased filter size in the first convolutional layer from $11 \times 11$ to $7 \times 7$ compared to the original AlexNet configuration is evaluated. In Fig. 5.11 the losses of the training process of both configurations are shown. It can be seen, that with the smaller filter the loss decreases over time, while for the other filter the loss saturates after 13 epochs. The latter presumably got stuck on a saddle point before and would decrease further with more training epochs. This could have been an unfavorable weight initialization, but based on all cost evaluations, the loss with the $7 \times 7$ filter decreases

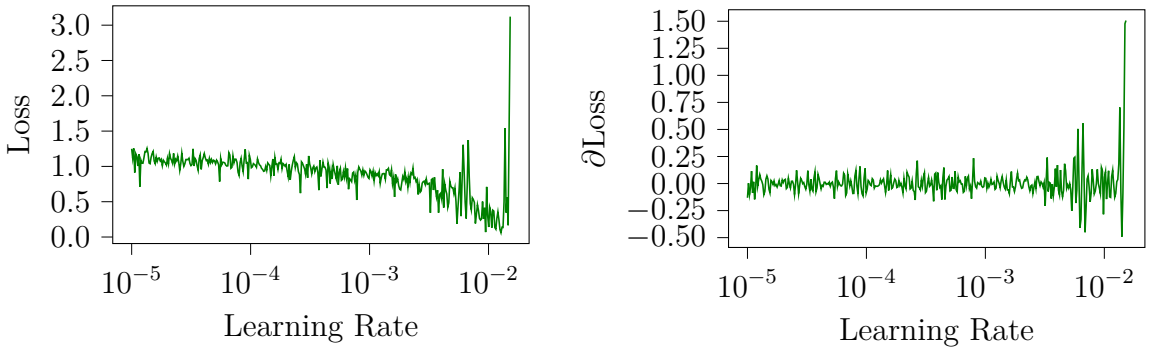



Figure 5.10: Optimal learning rate for the 0-3 network. Learning rate is initialized with 0.00001 and multiplied by 1.02 every iteration.

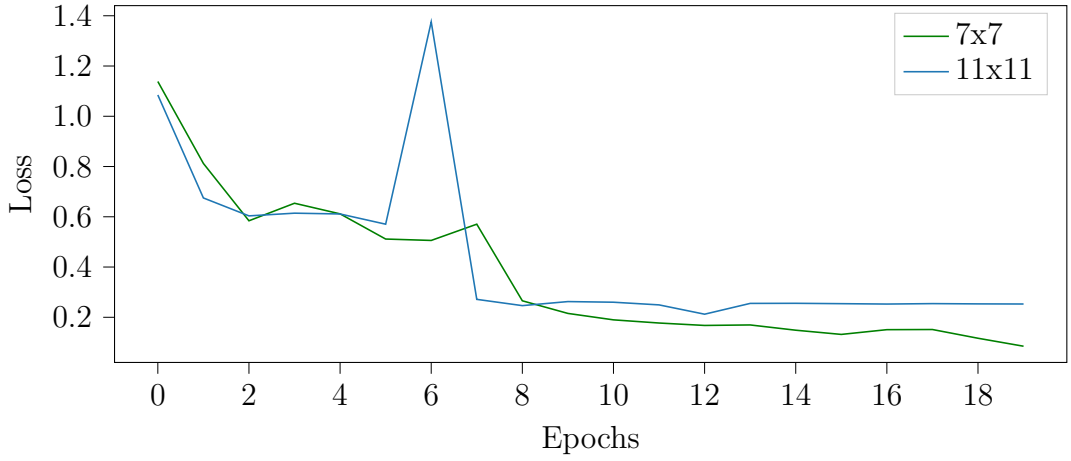
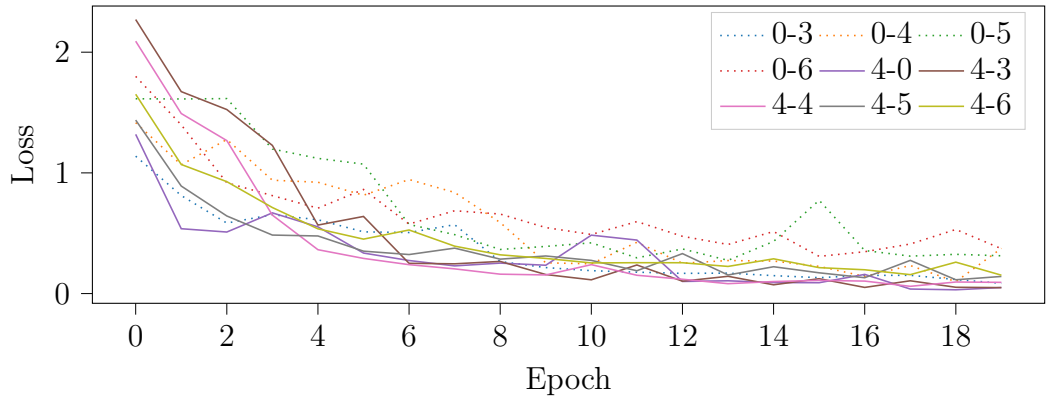


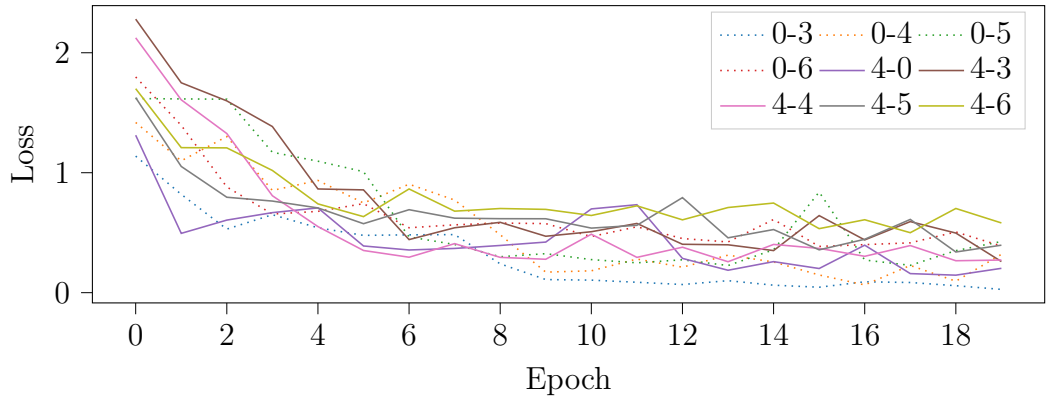

Figure 5.11: Comparison of filter sizes of first convolutional layer based on loss

much more and faster. That means, there were either many saddle points very close to each other, or the performance of the $11 \times 11$ filter is actually worse. Because more recent convolutional networks tend to use smaller filters, the latter theory is assumed. Hence, a filter with a size of $7 \times 7$ is chosen for the first convolution.

The overall training losses for all networks are shown in Fig. 5.12a and the testing losses in Fig. 5.12a. For a more compact visualization of training and testing, they are only split into two separate graphs. As expected, the 0-3 network starts with the smallest loss and proceeds the most smoothly compared to all other networks, due to its simplicity. During training most closely to this comes the 0-4 network, however, with more rapid changes. This is expected as well because it is just slightly more complicated. The 0-5 and 0-6 network have since the 12th epoch the highest losses of all single category networks. This is not surprising, because they are challenged with the double material features. However, as the training proceeds, their cost function is noticeable going to be minimized, it just takes longer due to their complexity compared to the other single category networks. It is surprising, though, that the remaining networks are part of the ones with the smallest losses. Based on those, they can stick with the 0-3 model. Moreover, since the 13th epoch, they change considerably small in loss compared to the single category ones. However, this is difficult to explain, because the cost function is unknown. It could be, that they are on a plateau with only a small slope. Though it is unlikely that this happens to all of them in the same epoch when every network has different initialized weights, hence are located on different spots at the cost function. If there is only small progress, because the parameters are close to a very small local minimum or the actual global one of each cost function, an indicator of overfitting could be noticeable in the testing losses. However, there is no obvious increase in loss visible. Not even in the direct comparison of each network's training and testing losses. The only visible increase is for the 4-3 network after the 14th epoch, but it decreases after the 18th again. So it was presumably only on a bad location for generalization. Hence, the training of the networks can be continued for more epochs for trying to achieve a smaller loss and better generalization. It is not surprising, that the 0-3 network has the smallest loss again. The other networks are similar to each other. However, it is noticeable, that the single-category networks have more rapid changes at the beginning and the remaining networks later in the training process. This is presumably because of the different number of iterations per epoch. The more complex networks have larger datasets, due to the additional number of material features and categories, hence, more batches with a parameter adaption after each. This way the less complex networks need more epochs for having processed the same number of batches than a complex network. That also explains why the four-categories networks have less noticeable changes after several epochs than the single-category ones. For the sake of completeness, the related accuracies of the training processes are shown in Fig. 5.13. Here the same effects can be seen as with the losses.

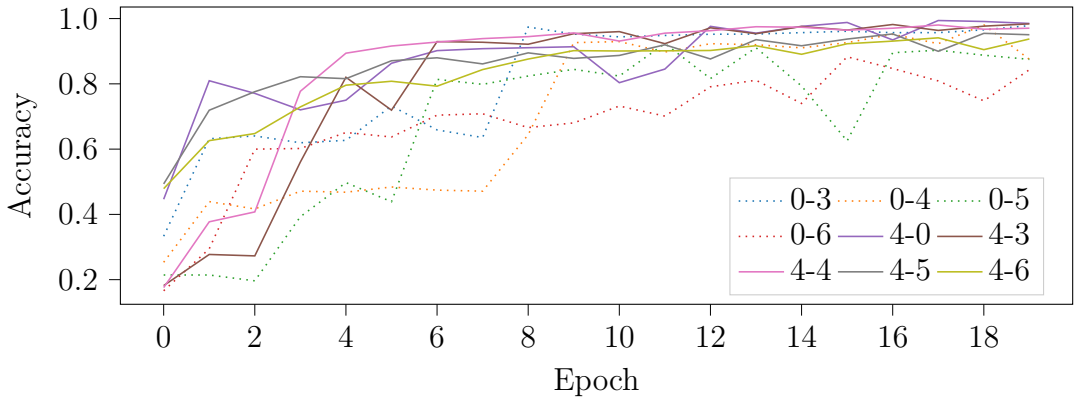



(a) Training losses of all networks

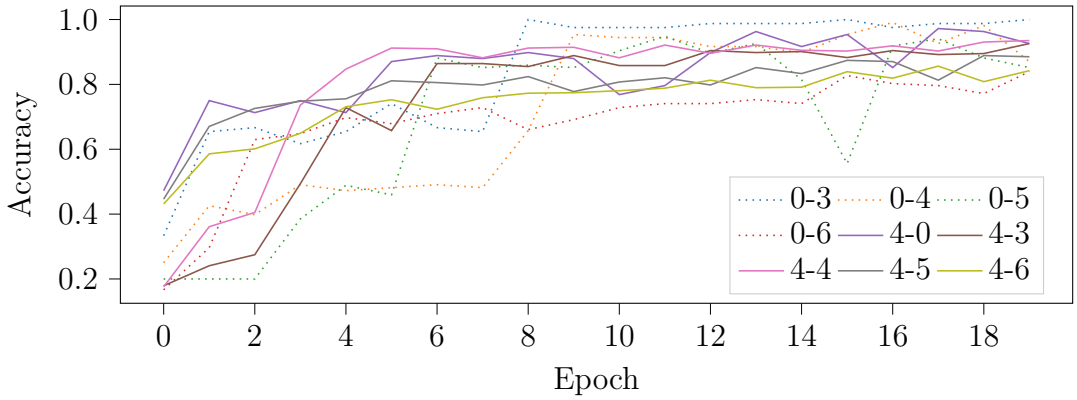



(b) Test losses of all networks

Figure 5.12: Training and test losses of networks




(a) Training accuracies of all networks




(b) Test accuracies of all networks

Figure 5.13: Training and test accuracies of networks

## 5.3 Prediction Accuracy

Because Fig. 5.13 reveals no detail of the accuracy per class or data sample this is evaluated separately. First, several metrics per class are examined. Then, depending on those results misclassifications are discussed.

### 5.3.1 Class Accuracies

The class accuracies of the single-category networks are presented in Table 5.1. Every material consists of 27 multi-views. Supported by its small loss, the 0-3 network has an overall accuracy of 100%. The more materials are added, the worse the accuracies of the related models get. This decrease is only slightly, though. Furthermore, it can be seen, that the accuracies of the blank models are always exact. Due to being the most simple case, it is not surprising. The networks just need to realize, that there is no specific material color present. For the 0-4 network, something strange happens. Although the accuracies seem ordinary, the misclassifications are interesting. It classifies 5 images with a red material feature as blank objects. Something similar happens for the green-red material features. There are 5 ones classified as green. However, this is more understandable than the blank misclassifications, due to the actual existence of that color. How this could happen is discussed for all networks based on the actual views, but for reasons of overview just in the following subsection. The 0-5 network classifies 14 objects as green-green when they only have one green feature, hence the bad accuracy of this class. This is understandable, though. Perhaps those features are placed next to each other and seem like a single one. For the green-green class, only two multi-views are predicted wrong. In fact as the single-green class. This shows, that the network generally prefers a green-green prediction. Those misclassifications would even out perhaps with more training epochs or data samples for learning better correlations. The results of the 0-5 network are similar to the 0-4 one. 14 actual green material multi-views are predicated as green-green ones, while the other way round only 2 are misclassified as single green material objects. The 0-6 network predicts similar as well. However, the single-feature as double-feature misclassifications are much better for the red material. Only 4 red ones are misclassified while 12 green ones are misclassified. The predictions vice versa are better for the green material, though. Only 2 are wrongly predicted as a single-material feature for the green ones and 8 for the red ones. It seems like the networks prefer double green features and single red features. Because this is valid for all of them, taking the weight initialization as the reason is unlikely. Moreover, for each object, the same optimal faces are colored, so an unbalanced learning due to the dataset prevented. But why exactly the networks behave like that remains unknown for now. The per class type accuracies of the four-category networks are shown in Table 5.2. In general, they are slightly better than the ones of the single-category networks, though they are more complicated. This is presumably due to the larger training sets. The 4-0 network predicts 5 bathtub objects as sofas, 1 dresser object as a monitor and 2

Table 5.1: Accuracy per class of single-category networks

| | 0-3 | 0-4 | 0-5 | 0-6 |
|---|---|---|---|---|
| Blank | 1.0 | 1.0 | 1.0 | 1.0 |
| Green | 1.0 | 0.852 | 0.481 | 0.556 |
| Red | 1.0 | 0.815 | 0.963 | 0.852 |
| Green-Red | | 0.815 | 0.889 | 1.0 |
| Green-Green | | | 0.926 | 0.926 |
| Red-Red | | | | 0.704 |
| Overall | 1.0 | 0.870 | 0.852 | 0.840 |

sofa objects as dressers. That seems like all object categories share some features for being able to misclassify them as different categories. For example, if only bathtubs are misclassified as sofas and sofas are misclassified as bathtubs, it is certain that only those categories share features. However, that are not many wrong predictions, so again a longer training should increase the accuracies. The 4-3 network predicts some bathtubs as sofas as well, but each predicted material feature is correct. In numbers, it is five, four and one wrong predictions per related material feature. This shows, that at least for bathtubs green-red materials can be classified almost easily. Furthermore, some dressers are mistaken for monitors and some sofas for dressers. Those wrong predictions match the earlier network. Moreover, a few colored objects are predicted as the same blank object. This happens four times for sofas and once for monitors and dressers. For the 4-4 network almost the same prediction characteristics are valid as for the earlier four-category networks. Surprisingly is that every monitor object is classified correctly. Maybe this is due to a well-suited weight initialization because the actual network architecture is unchanged. The 4-5 and 4-6 networks experience a sudden drop in overall accuracy. This was also the case with the single-category networks, but not that drastically. Apparently, the double material features are more challenging for multiple object categories due to more features. Furthermore, both networks share some prediction characteristics of the earlier networks. The favorite misclassification categories of the actual categories persist. However, this is mostly for bathtubs. The remaining categories mostly predict wrong within their category. In particular the 4-5 network is really good with the dresser classes, however, its green-green class is the worst within that category. For bathtubs, the single-green class is the worst within, because it has the most misclassifications in this category with 6 objects as double-green bathtubs. Vice versa only two multi-views are predicted wrongly. For dressers, monitors, and sofas the green-green classification has by far the worst accuracy within each category. The differences are 0.134, 0.36 and 0.154, respectively, to the second worst class within each object category. In conclusion, this network shows, that the critical material classes are similar to the single-object networks, but it is more challenging for four-object networks to classify double-features consistently. This is almost completely supported by the

Table 5.2: Accuracy per types of classes of four-category networks

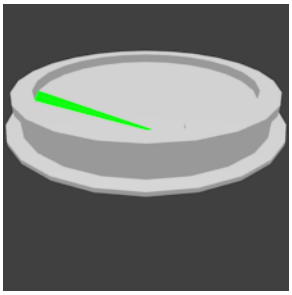
| | 4-0 | 4-3 | 4-4 | 4-5 | 4-6 |
|---|---|---|---|---|---|
| Bathtub (27) | 0.812 | 0.877 | 0.917 | 0.830 | 0.778 |
| Dresser (30) | 0.967 | 0.967 | 0.942 | 0.960 | 0.911 |
| Monitor (25) | 1.0 | 0.987 | 1.0 | 0.904 | 0.920 |
| Sofa (26) | 0.923 | 0.872 | 0.885 | 0.838 | 0.755 |
| Blank | | 0.926 | 0.935 | 0.954 | 0.944 |
| Green | | 0.926 | 0.944 | 0.870 | 0.722 |
| Red | | 0.926 | 0.935 | 0.981 | 0.815 |
| Green-Red | | | 0.926 | 0.870 | 0.907 |
| Green-Green | | | | 0.750 | 0.843 |
| Red-Red | | | | | 0.822 |
| Overall | 0.926 | 0.926 | 0.935 | 0.885 | 0.842 |

4-6 network, because it shares those characteristics of worst material classes within somehow. This time, though, the worst material classes for bathtubs and sofas are the single-green ones. That confirms their correlation even more. The single-material classes for the remaining categories are balanced. For the double material features the red-red ones are the worst for bathtubs and monitors and the green-green ones for dressers and sofas. Here it can be seen again, that usually one material color is preferred over the other. Nonetheless, usually, if a material is misclassified in double-material networks, its prediction is either the single-material or double-material class of the same object. Presumably, all the errors presented in this section can be reduced by a longer training process, so that each network learns more and better correlations for predicting similar classes. The possibility of this is supported by the loss graphs, that show no overfitting yet. If this does not achieve the desired result, more data samples could be added for supplying more correlations.

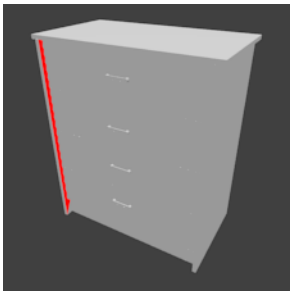
### 5.3.2 Misclassifications

By comparing all misclassifications of all networks several similarities are found. It is noticed that an object with a wrong prediction is likely to reappear across different networks. Not always with the exact same material, though, but this could be due to different weight initializations. The most common reason for predicting the wrong class within the same object category are small features. Some examples are shown in Fig. 5.14. The first multi-view is often predicted as blank and the second one as green-red. Why the material color is predicted wrong, however, remains unknown. Similar objects are found in the dataset which are predicted wrong that have features of a similar size like Fig. 5.14c. For the fourth one the red face is never seen completely, hence, it is never seen in a triangular shape like usual material faces except for one view. There

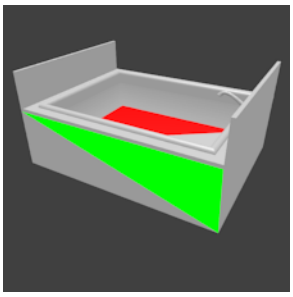
(a) Predicted as blank



(b) Predicted as green-red

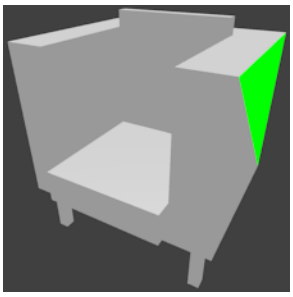


(c) Predicted as blank



(d) Predicted as green

(e) Predicted as blank sofa



(f) Predicted as green-green dresser

Figure 5.14: Reasons for incorrect predictions are small features, hidden features, and missing shadows.

it is truncated to a small triangular, though, and experiences the same problems as the other two multi-views presumably. Two optimal faces next to each other forming a rectangle result in a correct classification in general, however. The case of two faces building a larger triangle is not present in the dataset. Although it is assumed that such a multi-view would be predicted wrong. The reason for predicting bathtubs incorrectly as sofas are mostly missing shadows like in Fig. 5.14e. Because mainly the long left side transitions into the plane due to a missing sharp shadow at the edge the actual bathtub is more likely to be predicted as a sofa. This object is indeed likely to reappear in different networks as a wrong prediction. However, the actual and predicted material differs. Fig. 5.14f is classified as a dresser with a green-green material feature. Though, it is a sofa and has only one green material feature. This shows that many multi-views are just predicted wrong without an interpretable or obvious cause.

# Chapter 6

# Discussion

## 6.1 Conclusions

The general objective is to classify objects and materials, where a version with each material feature of each object exists. Those objects are visualized with 12 views from different perspectives. Grouping those views regarding their discrimination is supposed to increase the performance, hence, it is building the core functionality of the presented network architecture. Each discrimination is represented by a view discrimination score that is a single number calculated by a fully-connected layer with a view descriptor as input. Here the last view descriptor from the fifth convolutional layer is chosen, that is the last one in the network because it resembles the most accurate features of a view. Depending on their scores, the views are divided in 12 uniformly distributed groups for having the most flexibility. For each group, a view descriptor is generated by averaging its contained view descriptors. This is possible because the views of each group have probably similar features, thus, they are combined equally. Each group gets a weight associated that is the average of its included view discrimination scores. Then each single group descriptor is propagated through two fully-connected layers generating a shape descriptor for the particular group. Hence, the output of the seventh layer are shape descriptors. With those group shape descriptors, a weighted average is calculated using each related group weight. This generates a single shape descriptor describing the actual object. Propagating this through the last fully-connected layer, the eighth one, yields the prediction of the network. Combining this with a softmax layer yields the prediction probabilities for the classes. A summary of the layers is given in Table 6.1.

For the grouping mechanism, the following results are observed. The 0-3 network treats views with a visible material feature the most discriminative. However, those discriminations are reduced drastically when more material features are added and mostly views showing no material feature are preferred for discrimination. Moreover, by comparing equivalent material classes but with a different color like green to red or green-green to red-red it is shown, that each score differs. Hence, the assumption is postulated that for each material class a range of values for the final shape descriptor is defined. Depending on how large it is a particular material class is predicted.

The minimum losses and maximum accuracies during the training process of all net-

Table 6.1: Network layer summary

| Operation | Window | Output Size | Stride | Padding |
|---|---|---|---|---|
| Conv1 | $7 \times 7$ | $109 \times 109 \times 96$ | 2 | Valid |
| Pool1 | $3 \times 3$ | $54 \times 54 \times 96$ | 2 | Valid |
| Conv2 | $5 \times 5$ | $27 \times 27 \times 256$ | 2 | Same |
| Pool2 | $3 \times 3$ | $13 \times 13 \times 256$ | 2 | Valid |
| Conv3 | $3 \times 3$ | $13 \times 13 \times 384$ | 1 | Same |
| Conv4 | $3 \times 3$ | $13 \times 13 \times 384$ | 1 | Same |
| Conv5 | $3 \times 3$ | $13 \times 12 \times 256$ | 1 | Same |
| Pool5 | $3 \times 3$ | $6 \times 6 \times 256$ | 2 | Valid |
| Fc1 | | 4096 | | |
| Fc2 | | 4096 | | |
| Fc3 | | # of Classes | | |
| Softmax | | | | |

works are visualized in Fig. 6.1 and Fig. 6.2. The corresponding values are noted in Table 6.2. With the addition of material features the networks get more complicated, hence their loss increases and the accuracy decreases. It is noticeable, that using 5 or 6 material classes, wrong predictions are very likely to be within the same category but with the related single or double feature of the same color. This can presumably be coped by a larger dataset or a longer training due to no indicator of overfitting. The first simulates the latter by having more batches, thus more updates are performed on the parameters. For the losses it can be seen, that for the single-category networks they are similar for training and testing set. However, the ones for the four-category losses differ extremely. Those training losses are much smaller than the ones from the single-category networks, though. The smaller losses are presumably due to the larger datasets. The difference between training losses and testing losses is supposedly due to the different cost functions of both sets. Because the cost function of the four-category networks is way more complex than the one of the single-category networks by having more extrema. If now well-suited parameters for the training set are found, they do not necessarily represent a well-suited point in the testing cost function as illustrated in Fig. 2.19. This can be overcome by finding a broader minimum. It is difficult to compare the networks to recent researches because their objective is different. However, the closest one to the MVCNN and GVCNN results is the 4-0 network, because it uses classes from the ModelNet10 dataset and no material features. The first one reaches an accuracy of 89.9% taking the network with the closest configuration as a reference. This was improved to 95.0% by *Su et al.* The GVCNN reaches an accuracy of 92.6%. However, all of them use the ModelNet40 as a benchmark, so the results are not fully comparable but indicate the right direction.

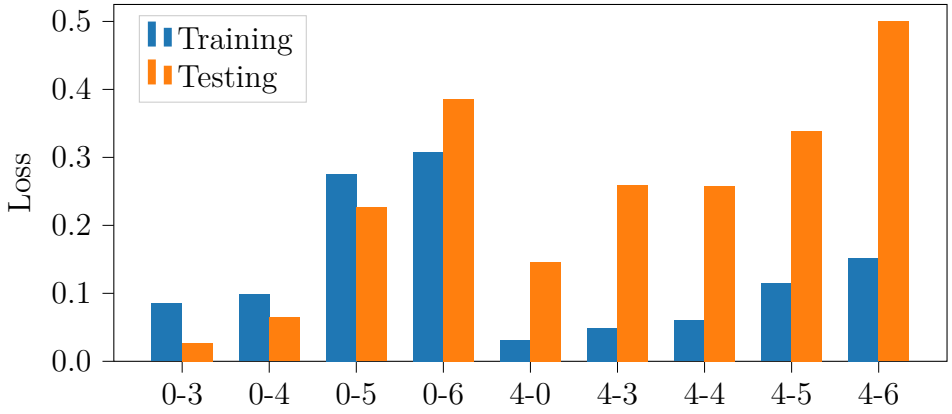



Figure 6.1: Losses of all networks

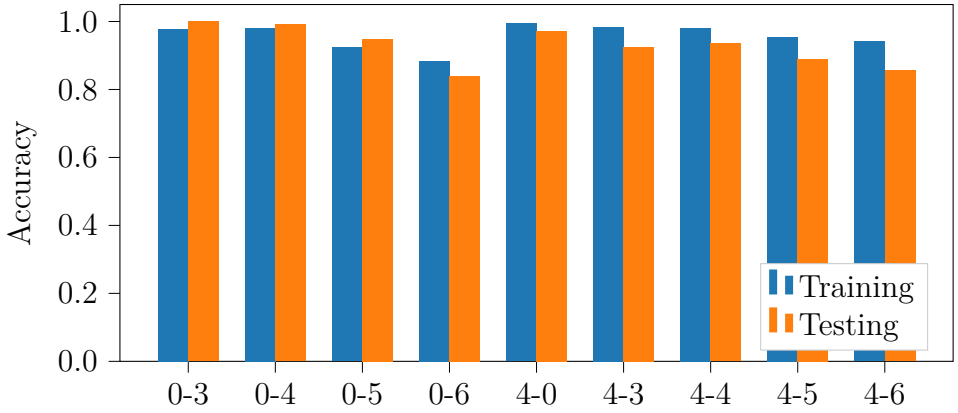



Figure 6.2: Accuracies of all networks

Table 6.2: Performance of all networks

| | 0-3 | 0-4 | 0-5 | 0-6 | 4-0 | 4-3 | 4-4 | 4-5 | 4-6 |
|---|---|---|---|---|---|---|---|---|---|
| Train Loss | 0.085 | 0.099 | 0.275 | 0.308 | 0.031 | 0.049 | 0.06 | 0.114 | 0.152 |
| Test Loss | 0.027 | 0.065 | 0.226 | 0.386 | 0.146 | 0.26 | 0.258 | 0.339 | 0.5 |
| Train Accuracy | 0.978 | 0.981 | 0.923 | 0.883 | 0.994 | 0.984 | 0.98 | 0.955 | 0.941 |
| Test Accuracy | 1.0 | 0.991 | 0.948 | 0.84 | 0.972 | 0.926 | 0.935 | 0.889 | 0.856 |

## 6.2 Outlook

The current networks can be improved by tuning the dataset among others. More lights could be added to the scene or the current light is placed at the position of the camera and points along its view axis directly at the object. This way more shadows are created in a view presumably leading to detection of more edge features. For creating optimal faces that are not occluded by others, ray casts for as many vertices in the face as possible need to be performed and checked if the ray hits the actual face. However, this gets very computationally expensive and would take a huge amount of time. The advantage of this would be, that those results could be stored an used for finding the second optimal face. If resources and time are no constraints, every other face could be examined for being the second optimal face. Its results are then compared with the ones from the first one under the restriction of a given number of views where one should be visible. Otherwise, choosing a valid face with the maximum distance from the first is a working approach. In this case, though, a manual deleting of some bad samples needs to be performed. Furthermore, the trained networks could be tested on real-world samples. Real-world objects could be digitalized with a 3D scanner and propagated through the network. This is likely to give somehow different results as true 3D models. Moreover, real-world colors and materials can be assigned to each model, either CAD or digitalized, for supplying more features.

The networks could be improved by a longer training because no indication of overfitting exists. Moreover, restarts for the learning rate could be added to avoid the differences in the loss for the four-category networks by stepping over small minima. Furthermore, a change in a number of groups and their bin size could be evaluated. However, it cannot be rated if this improves performance. The minimum of groups should be related to the current number of groups used for the predictions, though. Probably the largest boost in performance would be a change of the underlying network architecture. There are many networks that achieve higher accuracies according to the ImageNet challenge in object detection in images like Inception-v4 and ResNet. If they are combined with the grouping mechanism higher accuracies than with the current implementation are expected. However, the position of the grouping module and the input of the fully-connected layer for calculating the view discrimination scores are different if the new architecture is nested. Their properties need to be examined first.

# Bibliography


[1] Abadi, Martín; Agarwal, Ashish; Barham, Paul; Brevdo, Eugene; Chen, Zhifeng; Citro, Craig; Corrado, Greg S.; Davis, Andy; Dean, Jeffrey; Devin, Matthieu; Ghemawat, Sanjay; Goodfellow, Ian; Harp, Andrew; Irving, Geoffrey; Isard, Michael; Jia, Yangqing; Jozefowicz, Rafal; Kaiser, Lukasz; Kudlur, Manjunath; Levenberg, Josh; Mané, Dan; Monga, Rajat; Moore, Sherry; Murray, Derek; Olah, Chris; Schuster, Mike; Shlens, Jonathon; Steiner, Benoit; Sutskever, Ilya; Talwar, Kunal; Tucker, Paul; Vanhoucke, Vincent; Vasudevan, Vijay; Viégas, Fernanda; Vinyals, Oriol; Warden, Pete; Wattenberg, Martin; Wicke, Martin; Yu, Yuan; Zheng, Xiaoqiang: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015

[2] Bengio, Yoshua: Practical recommendations for gradient-based training of deep architectures. In: *Computing Research Repository*, 2012

[3] Bishop, Christopher M.: Neural Networks for Pattern Recognition, Oxford University Press, Inc., 1995

[4] Bishop, Christopher M.: Pattern Recognition and Machine Learning (Information Science and Statistics), Springer-Verlag, 2006

[5] Blender Foundation: Blender 2.80 Reference Manual, 2019

[6] Chatfield, Ken; Simonyan, Karen; Vedaldi, Andrea; Zisserman, Andrew: Return of the Devil in the Details: Delving Deep into Convolutional Nets. In: *Computing Research Repository*, 2014

[7] Chen, Ding-Yun; Tian, Xiao-Pei; Shen, Yu-Te; Ouhyoung, Ming: On Visual Similarity Based 3D Model Retrieval. In: *Computer Graphics Forum*, Blackwell Publishers, Inc and the Eurographics Association, 2003, pp. 223–232

[8] Choromanska, Anna; Henaff, Mikael; Mathieu, Michaël; Arous, Gérard Ben; LeCun, Yann: The Loss Surface of Multilayer Networks. In: *Computing Research Repository*, 2014

[9] Cybenko, George: Approximation by superpositions of a sigmoidal function. In: *Mathematics of Control, Signals and Systems* Vol. 2, 1989, pp. 303–314

[10] Cyr, Christopher M.; Kimia, Benjamin B.: A Similarity-Based Aspect-Graph Approach to 3D Object Recognition. In: *International Journal of Computer Vision* Vol. 22, Kluwer Academic Publishers, 2004, pp. 5–22

[11] Fawcett, Tom: An Introduction to ROC Analysis. In: *Pattern Recognition Letters* Vol. 27, Elsevier, 2006, pp. 861–874

[12] Feng, Yifan; Zhang, Zizhao; Zhao, Xibin; Ji, Rongrong; Gao, Yue: GVCNN: Group-View Convolutional Neural Networks for 3D Shape Recognition. In: *Computer Vision and Pattern Recognition*, Institute of Electrical and Electronics Engineers, 2018

[13] Glorot, Xavier; Bengio, Yoshua: Understanding the difficulty of training deep feedforward neural networks. In: *International Conference on Artificial Intelligence and Statistics*, 2010

[14] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron: Deep Learning, MIT Press, 2016

[15] Harris, David; Harris, Sarah: Digital Design and Computer Architecture, Second Edition, Morgan Kaufmann Publishers Inc., 2012

[16] He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In: *Computing Research Repository*, 2015

[17] He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian: Deep Residual Learning for Image Recognition. In: *Computer Vision and Pattern Recognition*, Institute of Electrical and Electronics Engineers, 2016, pp. 770–778

[18] Hebb, Donald O.: The Organization of Behavior: A Neuropsychological Theory, Wiley, 1949

[19] Hegde, Vishakh; Zadeh, Reza B.: FusionNet: 3D Object Classification Using Multiple Data Representations. In: *Computing Research Repository*, 2016

[20] Hochreiter, Sepp: Untersuchungen zu dynamischen neuronalen Netzen, Universität München, 1991

[21] Hornik, Kurt: Approximation capabilities of multilayer feedforward networks. In: *Neural Networks* Vol. 4, Elsevier, 1991, pp. 251–257

[22] James, Gareth; Witten, Daniela; Hastie, Trevor; Tibshirani, Robert: An Introduction to Statistical Learning: With Applications in R, Springer-Verlag, 2014

[23] 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: *Computing Research Repository*, 2016

[24] Kiefer, J.; Wolfowitz, J.: Stochastic Estimation of the Maximum of a Regression Function. In: *The Annals of Mathematical Statistics* Vol. 23, The Institute of Mathematical Statistics, 1952, pp. 462–466

[25] Kingma, Diederik P.; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *International Conference on Learning Representations*, 2015

[26] Krizhevsky, Alex: Learning Multiple Layers of Features from Tiny Images, 2009

[27] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Neural Information Processing Systems*, Curran Associates Inc., 2012, pp. 1097–1105

[28] LeCun, Yann; Bengio, Yoshua: The Handbook of Brain Theory and Neural Networks. In: Arbib, Michael A. (Hrsg.): *Convolutional Networks for Images, Speech, and Time Series*, MIT Press, 1998, pp. 255–258

[29] Lecun, Yann; Bottou, Léon; Bengio, Yoshua; Haffner, Patrick: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE*, Institute of Electrical and Electronics Engineers, 1998, pp. 2278–2324

[30] Li, Fei-Fei; Karpathy, Andrej; Johnson, Justin: CS231n: Convolutional Neural Networks for Visual Recognition, Spring 2019

[31] Loshchilov, Ilya; Hutter, Frank: SGDR: Stochastic Gradient Descent with Warm Restarts. In: *International Conference on Learning Representations*, 2017

[32] McCulloch, Warren S.; Pitts, Walter: A Logical Calculus of the Ideas Immanent in Nervous Activity. In: *Neurocomputing: Foundations of Research*. MIT Press, 1988, pp. 15–27

[33] Murphy, Kevin P.: Machine learning: a probabilistic perspective, MIT Press, 2013

[34] Osada, Robert; Funkhouser, Thomas; Chazelle, Bernard; Dobkin, David: Matching 3D Models with Shape Distributions. In: *International Conference on Shape Modeling & Applications*, IEEE Computer Society, 2001, pp. 154–166

[35] Robbins, Herbert; Monro, Sutton: A Stochastic Approximation Method. In: *The Annals of Mathematical Statistics* Vol. 22, The Institute of Mathematical Statistics, 1951, pp. 400–407

[36] Rosenblatt, Frank: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. In: *Psychological Review*, American Psychological Association, 1958, pp. 65–386

[37] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J.: Learning representations by back-propagating errors. In: *Nature* Vol. 323, Nature Publishing Group, 1986, pp. 533–536

[38] 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: *International Journal of Computer Vision* Vol. 115, Kluwer Academic Publishers, 2015, pp. 211–252

[39] Scherer, Dominik; Müller, Andreas; Behnke, Sven: Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In: *International Conference on Artificial Neural Networks: Part III*, Springer-Verlag, 2010, pp. 92–101

[40] Shu, Zhenyu; Xin, Shiqing; Xu, Huixia; Kavan, Ladislav; Wang, Pengfei; Liu, Ligang: 3D Model Classification via Principal Thickness Images. In: *Computer Aided Design* Vol. 78, Butterworth-Heinemann, 2016, pp. 199–208

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

[42] Smith, Leslie N.: No More Pesky Learning Rate Guessing Games. In: *Computing Research Repository*, 2015

[43] Srivastava, Nitish; Hinton, Geoffrey; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In: *Journal of Machine Learning Research* Vol. 15, MIT Press, 2014

[44] Su, Hang; Maji, Subhransu; Kalogerakis, Evangelos; Learned-Miller, Erik: Multi-view Convolutional Neural Networks for 3D Shape Recognition. In: *International Conference on Computer Vision*, IEEE Computer Society, 2015, pp. 945–953

[45] Su, Jong-Chyi; Gadelha, Matheus; Wang, Rui; Maji, Subhransu: A Deeper Look at 3D Shape Classifiers. In: *Computing Research Repository*, 2018

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

[47] 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*, Institute of Electrical and Electronics Engineers, 2015

[48] Tieleman, Tijmen; Hinton, Geoffrey: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. In: *COURSERA: Neural Networks for Machine Learning*, 2012

[49] Wu, Zhirong; Song, Shuran; Khosla, Aditya; Yu, Fisher; Zhang, Linguang; Tang, Xiaoou; Xiao, Jianxiong: 3D ShapeNets: A deep representation for volumetric shapes. In: *Computer Vision and Pattern Recognition*, Institute of Electrical and Electronics Engineers, 2015, pp. 1912–1920