Hochschule
Augsburg University of
Applied Sciences

BACHELORARBEIT

Bildklassifikation mit neuronalen Netzen für die Lebensmittelproduktion

Lukas Degle
Matrikelnummer: 950226
lukas.degle@hs-augsburg.de
1. Juli 2019

Elektrotechnik

Informations- und Kommunikationstechnik

Betreuung an der Hochschule Augsburg

Herr Prof. Dr.-Ing. Friedrich Beckmann

Betreuung bei opdi-tex GmbH

Herr Andreas Gareis

Verfasser:

Lukas Degle

Untere Ringstraße 5b
86899 Landsberg am Lech
Tel.: +49 176 303 42442

Hochschule:

Hochschule Augsburg
Fakultät Elektrotechnik

An der Hochschule 1
86161 Augsburg
Tel.: +49 821 55 86-3350
www.hs-augsburg.de

Unternehmen:

opdi-tex GmbH

Gewerbering 9
86922 Eresing
Tel.: +49 8193 937103
www.opdi-tex.de

Zusammenfassung

Die Firma opdi-tex stellt Kamerasysteme zur Qualitätskontrolle in der Lebensmittelindustrie her. Unter anderem werden die Kamerasysteme bei der Produktion von Backwaren wie etwa Keksen eingesetzt. Derzeit wird mit Techniken der industriellen Bildverarbeitung wie beispielsweise einer Kantenerkennung eine Klassifikation der Kekse in *gute* und *schlechte* Kekse durchgeführt. Diese Zuordnung könnte in Zukunft eine künstliche Intelligenz übernehmen.

In dieser Arbeit soll dazu ein künstliches neuronales Netzwerk, mithilfe von maschinellem Lernen, die Unterschiede zwischen *guten* und *schlechten* Keksen erkennen und sie anschließend richtig klassifizieren. Die für dieses Supervised Learning der künstlichen Intelligenz benötigten Bilder werden mit einem Kamerasystem der Firma opdi-tex erstellt.

Zum Maschinenlernen wird das Framework *pytorch* verwendet. Dieses läuft auf dem GPU-Server *breakout* im Rechenzentrum der Hochschule Augsburg. Hier werden die Berechnungen zum Lernen der künstlichen Intelligenz durchgeführt.

Des Weiteren wird im Rahmen dieser Arbeit das Verhalten des künstlichen, neuronalen Netzwerkes hinsichtlich verschiedener Trainingskonzepte untersucht. Es werden beispielsweise die Auswirkungen vom Bearbeiten der Trainingsbilder (Augmentation), Änderung der Größe des Trainingsdatensatzes und die Verwendung von Transfer-Learning genauer betrachtet.

Danksagung

An dieser Stelle gilt mein ganz besonderer Dank Herrn Prof. Dr.-Ing. Friedrich Beckmann und Herrn Andreas Gareis für die hervorragende Betreuung dieser Bachelorarbeit. Sie haben sowohl fachlich als auch persönlich enorm zum Gelingen dieser Arbeit beigetragen.

Des Weiteren bedanke ich mich bei Herrn Karl-Ludwig Schinner, Dipl.-Ing. sowie der gesamten Firma opdi-tex GmbH für ihre Unterstützung.

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit selbstständig, ohne Zuhilfenahme unzulässiger Hilfsmittel und unter Angabe aller verwendeten Quellen, verfasst wurde. Die Arbeit wurde in dieser oder ähnlicher Form noch für keine anderen Prüfungszwecke genutzt. Wörtliche und sinngemäße Zitate wurden als solche kenntlich gemacht.

Landsberg am Lech, 1. Juli 2019

_____

Lukas Degle

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADAM adaptive moment estimation

CNN Convolutional Neural Network

CPU central processing unit

CUDA Compute Unified Device Architecture

GPU graphics processing unit

KNN Künstliches Neuronales Netzwerk

MNIST Modified National Institute of Standards and Technology

1 Aufgabenstellung

Die Firma opdi-tex GmbH entwickelt und produziert Kamerasysteme für automatisierte Fertigungsstraßen in der Lebensmittel- und Textilindustrie. Diese telezentrischen Farbzeilenkameras erstellen detailgenaue Aufnahmen von Förderbändern mit Breiten von über 2 m. Anhand dieser Bilder ist es möglich die Qualität der sich auf den Förderband befindenden Produkte zu beurteilen. Dafür werden die Objekte, bei denen es sich häufig um Backwaren wie Keksen handelt, auf dem Band lokalisiert und klassifiziert. Dies geschieht mittels Bildbearbeitungswerkzeugen. Diese Klassifizierung muss für jedes Produkt und jede Sorte neu programmiert und eingestellt werden.

Ziel dieser Arbeit ist es, mit Hilfe von maschinellem Lernen, künstliche neuronale Netzwerke zu trainieren. Diese sollen Keksbilder verschiedenen Klassen zuordnen. Ein solches System kann auf dem Förderband lokalisierte Kekse selbst klassifizieren. Dieses alternative System kann die Zuordnung von Objekten dem Förderband vereinfachen.

Die Arbeit lässt sich in mehrere Unterpunkte aufteilen:

- Einarbeitung in die Thematik Maschinen-Lernen und künstliche neuronale Netze
- Einarbeitung in die Programmiersprache *python* und das Framework *pytorch*
- Erstellung der Bilddaten mit einem Kamerasystem der Firma opdi-tex
- Bilddaten in Trainings- und Validierungsdaten aufteilen
- Geeignete neuronale Netze auswählen
- Netz an die Problemstellung anpassen
- Trainieren der Netze
- Ergebnisse analysieren

2 Grundlagen

2.1 Natürliche und Künstliche Neuronale Netze

2.1.1 Natürliche neuronale Netze

Bei dem menschlichen Gehirn handelt es sich um ein natürliches neuronales Netzwerk. Es besteht aus etwa 100 Milliarden Nervenzellen. Diese, auch als Neuronen bekannten Nervenzellen, bestehen unter anderem aus einem Axon, den Dendriten sowie einem Nukleus, dem Zellkern. Die Dendriten dienen zum Empfang von Informationen von anderen Neuronen. Diese werden im Nukleus verarbeitet und über das Axon an andere Nervenzellen weitergegeben. [Ert08, S. 242] Die Informationsübertragung funktioniert mittels chemischer und elektrischer Signale. Beim Lernen eines natürlichen neuronalen Netzwerks werden neue Verbindungen zwischen den Nervenzellen ausgebildet. [GSS14, S. 361]

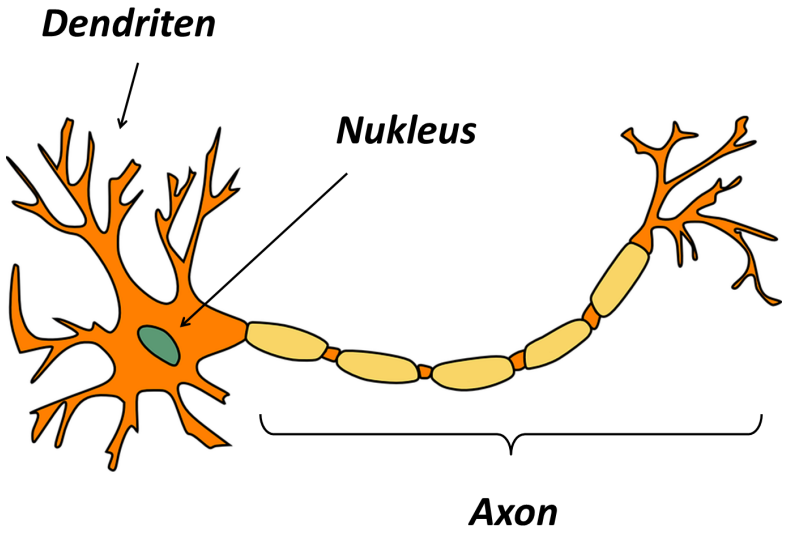

Abbildung 2.1: Vereinfachter Aufbau eines natürlichen Neurons

2.1.2 Künstliche neuronale Netze

Auch künstliche Neuronen nehmen, wie ihre natürlichen Äquivalente Informationen auf, verarbeiten diese und geben sie an andere Neuronen weiter. Bei diesen verarbeiteten Informationen handelt es sich um Zahlen die auch als die Aktivierungen **a** der Neuronen bezeichnet wird. Ist die Aktivierung sehr groß ist das Neuron aktiv. [GSS14, S. 366] Die Anordnung der Neuronen in künstlichen neuronalen Netzen erfolgt in Ebenen, welche in entsprechender Fachliteratur auch als Layer bekannt sind. Die Verbindungen zwischen den künstlichen Neuronen werden durch Zahlenwerte (Gewichte) repräsentiert. Die Lernphase dient dazu, die Gewichte des künstlichen neuronalen Netzwerks so zu verändern, das die geforderte Funktion erfüllt wird.

Ein solches künstliches neuronales Netz wird meistens mit einer entsprechenden Software auf einem Rechner erstellt und gespeichert. [Hea13, S. 2] Die Verarbeitung der Informationen übernimmt eine Recheneinheit des Computers (z.B. CPU bzw. GPU).

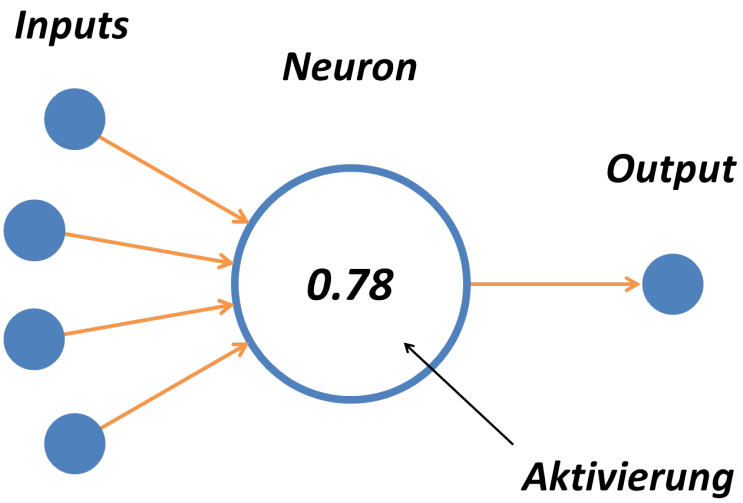

Abbildung 2.2: Vereinfachter Aufbau des künstlichen Neurons

2.2 Aufbau eines künstlichen neuronalen Netzwerks

2.2.1 Einstieg

Bei der Bildklassifikation soll einem Bild eine Klasse, aus einer festgelegten Menge von Klassen, zugeordnet werden. Dafür eignen sich künstliche neuronale Netze. Im Folgenden soll der Aufbau dieser Netze näher betrachtet werden.

Zum Einstieg in diese Thematik wird der MNIST-Datensatz zur Veranschaulichung verwendet. Dieser enthält tausende von Bildern, auf denen handgeschriebene Ziffern dargestellt sind. Diese Bilder der Größe 28 x 28 Pixel sollen durch ein neuronales Netz klassifiziert werden. In der Abbildung 2.3 sind Beispielbilder des Datensatzes zu sehen. [Yan19]

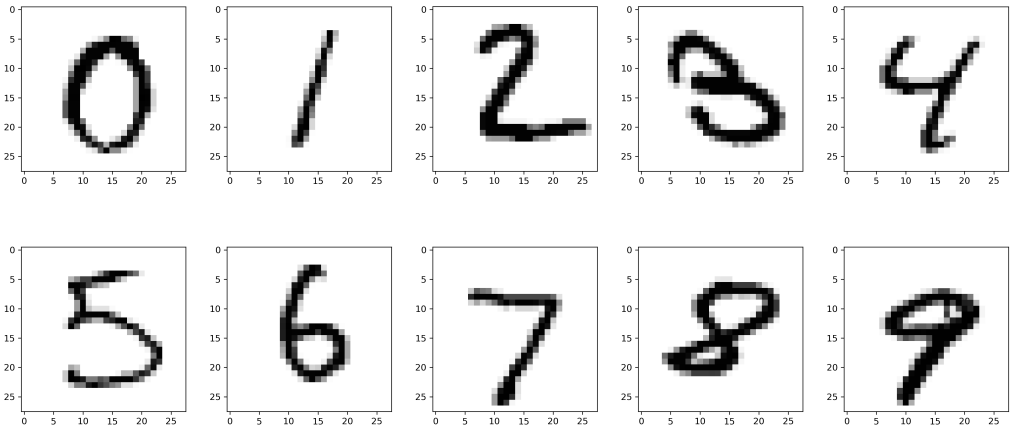

Abbildung 2.3: Beispielbilder der Ziffern 0 bis 9 des MNIST-Datensatzes

2.2.2 Eingangs- und Ausgangsebene

Die Neuronen eines künstlichen neuronalen Netzwerks (KNN) werden in Ebenen angeordnet. Ein KNN besteht dabei mindestens aus einem Eingangs- und einem Ausgangslayer.

Die Anzahl der Neuronen in der Eingangsebene entspricht der Anzahl von Eingangswerten. Diese ergeben sich aus Anzahl der Pixel eines zu klassifizierenden Bildes. Mit den Daten des MNIST-Datensatzes entspricht das 784 Neuronen.

$$28\ Pixel \cdot 28\ Pixel \cdot 1\ Farbkanal = 784\ Neuronen$$

Die Farb- bzw. Graustufenwerte aller Pixel des zu klassifizierenden Bildes werden den Input-Neuronen als deren Aktivierung zugewiesen.

Die Neuronen des Output-Layers repräsentieren die Klassen, denen die Eingangsdaten zugeordnet werden können. Um Bilder mit den Ziffern von 0 bis 9 klassifizieren zu können sind 10 Neuronen in der Ausgangsebene. Jedes Neuron repräsentiert eine Ziffer. Die Aktivierung dieser Neuronen beschreibt wie stark, laut KNN das Input Bild mit der jeweiligen Ziffer übereinstimmt. Somit ist das Ausgangsneuron mit der größten Aktivierung gleichbedeutend mit der Entscheidung des Netzwerks. [Mic19]

2.2.3 Hidden-Layer

Für die meisten Anwendungen sind zwischen dem Input- und Outputlayer zusätzliche Ebenen hilfreich. Durch diese sogenannten Hidden-Layer kann die Leistungsfähigkeit eines KNN gesteigert werden. [Yan19]

Die Anzahl der Hidden-Layer sowie die darin enthaltenen Neuronen sind im Gegensatz zu den Eingangs- und Ausgangsebenen nicht an die Eingangsdaten oder Klassen gebunden. Die in Abbildung 2.4 verwendeten 14 Neuronen pro Hidden-Layer dienen lediglich der Illustration und stellen somit keine allgemeingültige Anzahl dar. In der Realität kann die Neuronenanzahl je Hidden-Layer über 1000 ansteigen.

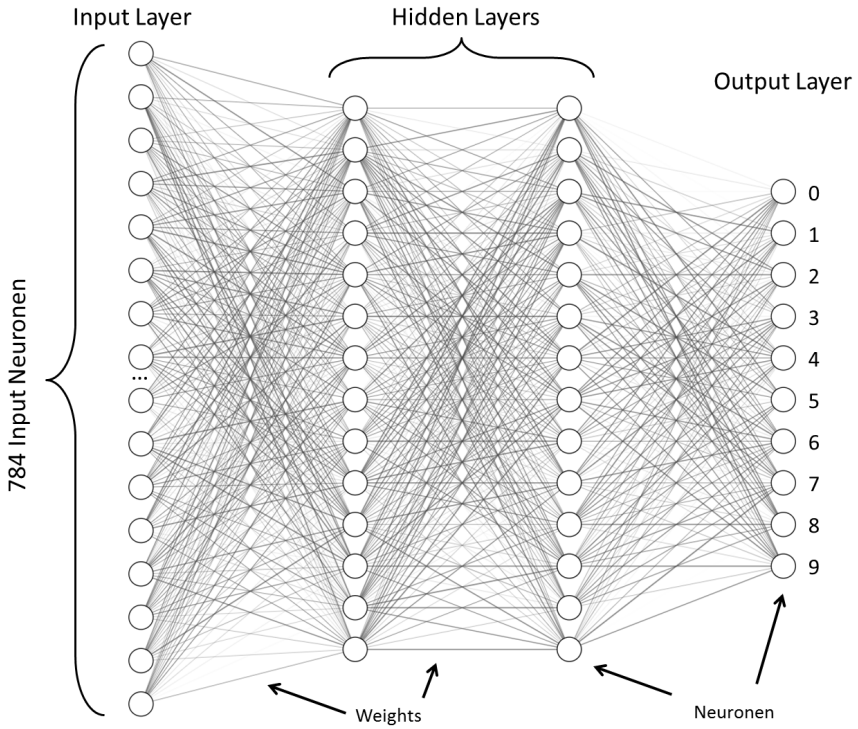

Abbildung 2.4: Aufbau eines einfachen KNN zur Klassifizierung der MNIST-Daten [Neu19]

2.2.4 Gewichte/Weights

Die Verbindungen zwischen den Neuronen verschiedener Ebenen heißen Gewichte (englisch: Weights). Bei den in Abbildung 2.4 dargestellten linearen Layern ist jedes Neuron eines Layers mit allen Neuronen der benachbarten Ebenen verbunden. Es gibt keine Verbindungen zwischen Neuronen desselben Layers. Jedes Gewicht besitzt einen Zahlenwert **w**. Zur Berechnung einer Aktivierung a_x^L eines Neurons im Layers **L** an der Position **x**, werden die Aktivierungen **a** des vorangestellten Layers **L-1** mit den Gewichten $w_{i,x}$, welche zum dem gesuchten Neuron führen, multipliziert. Danach werden alle Produkte addiert. Dadurch bildet ein Neuron die gewichtete Summe aller Vorgängerneuronen. [Ert08, S. 245]

$$a_x^L = a_0^{L-1} \cdot w_{0,x} + a_1^{L-1} \cdot w_{1,x} + ... + a_n^{L-1} \cdot w_{n,x} = \sum_{i=1}^{n} a_i^{L-1} \cdot w_{i,x} \tag{2.1}$$

Die Gewichte werden im Lernprozess/Training durch Algorithmen verändert um die Leistungsfähigkeit, also die Erkennungsgenauigkeit, des Netzwerkes zu erhöhen.

2.2.5 Aktivierungsfunktion

Die hintereinander geschalteten Ebenen (auch als Komposition bezeichnet) enthalten bis zu diesem Abschnitt nur lineare Anteile. Diese Komposition besitzt wenig Komplexität, da sie mithilfe arithmetischer Umformung in ein einfaches Netzwerk mit nur zwei Ebenen (In- und Output) gewandelt werden kann. Um dem vorzubeugen werden nichtlineare Aktivierungsfunktionen verwendet. Dadurch bleibt die durch Komposition entstandene Komplexität des Netzwerks erhalten. [Dev19]

Eine Aktivierungsfunktion **f** wandelt die gewichtete Summe (x-Achse) in die Aktivierung des Neuron (y-Achse) um. In Abbildung 2.5 sind die Verläufe von zwei Aktivierungsfunktionen abgebildet.

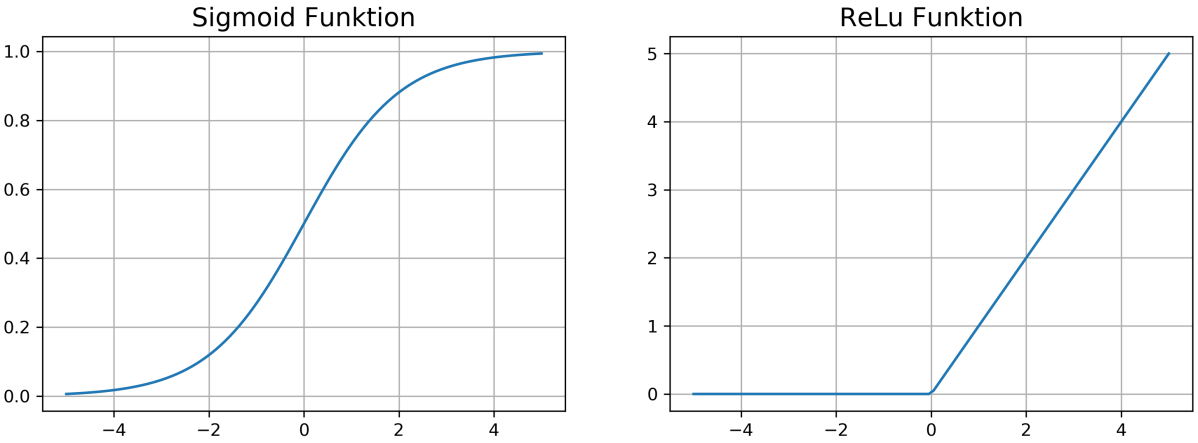

Abbildung 2.5: Verläufe von Sigmoid- und ReLu-Funktionen

Durch die Verwendung einer Aktivierungsfunkion **f** wird die Formel zur Berechnung der Aktivierung folgendermaßen ausgedrückt:

$$a_x^L = f\left\{\sum_{i=1}^{n} a_i^{L-1} \cdot w_{i,x}\right\} \tag{2.2}$$

2.2.6 Bias Wert

Alle Neuronen der Hidden-Layer und Output-Layer besitzen einen Bias-Wert **b**. Dieser dient der zusätzlichen Gewichtung. Er ermöglicht es die Aktivierungsfunktion f entlang der x-Achse zu verschieben. So werden Neuronen erst ab bestimmten Werten der gewichteten Summe aktiv. Es ergibt sich die Formel 2.3. Die Bias-Werte **b** werden genau wie die Gewichte **w** im Training des Netzwerks optimiert. [Hea13, S. 9]

$$a_x^L = f\left\{\sum_{i=1}^{n} a_i^{L-1} \cdot w_{i,x} + b_x^L\right\} \tag{2.3}$$

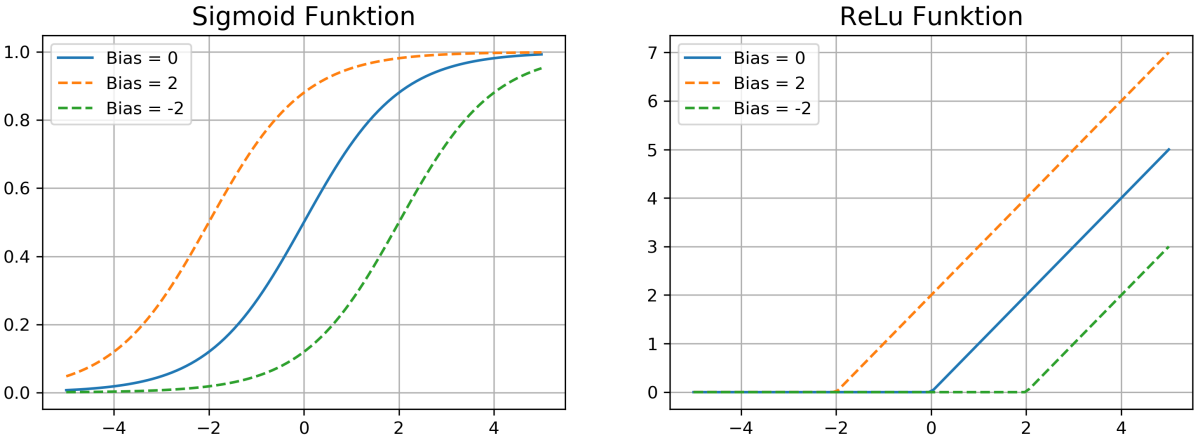

Abbildung 2.6: Sigmoid- und ReLu-Funktionen mit verschiedenen Bias Werten

2.2.7 Klassifikation / Forward-Path

Zur Klassifikation eines Bildes werden ausgehend von der Eingangsebene in Richtung des Ausgangs Ebene für Ebene die Aktivierungen aller Neuronen berechnet. Sind alle Aktivierungen des KNN bestimmt, kann anhand der höchsten Aktivierung im Output-Layer die Entscheidung des Netzwerks ermittelt werden. Somit müssen für jedes neue Eingangsbild auch alle Aktivierungen neu berechnet werden.

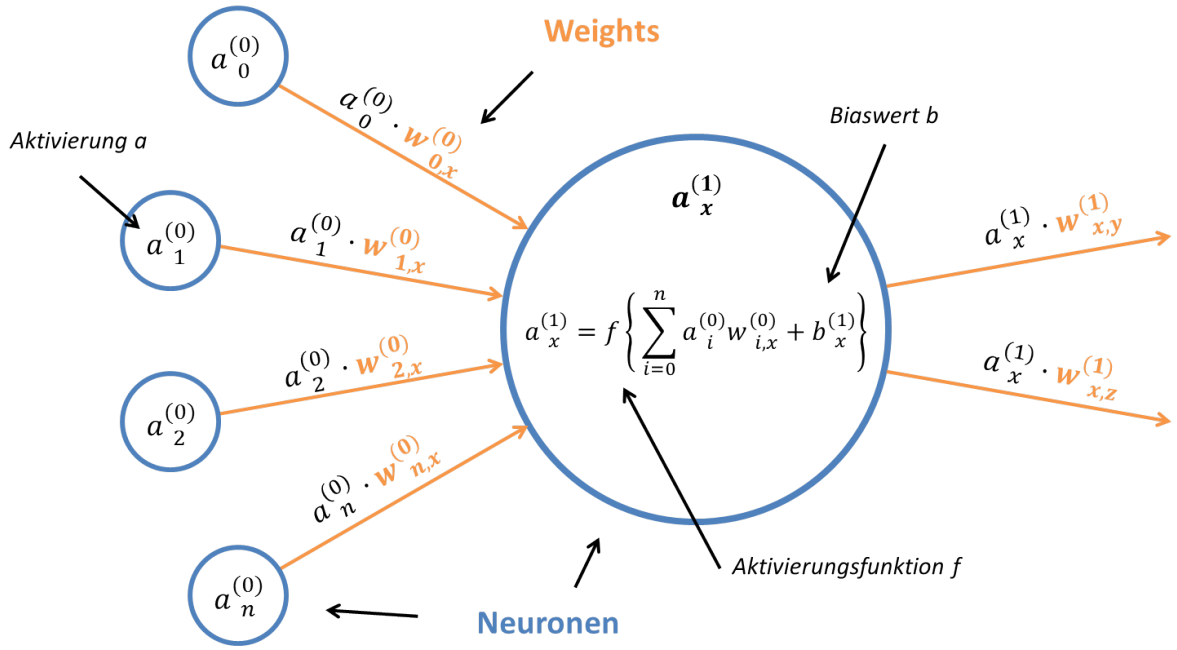

Abbildung 2.7: Detaillierte Darstellung eines künstlichen Neurons $a_x^{(1)}$

Dass sich die Eingangsdaten in das Netz auch bei Bildern gleicher Klasse stark unterscheiden können, erkennt man in Abbildung 2.8 der MNIST-Bilder. Trotzdem soll die Klasse richtig erkannt werden. Das funktioniert nur mit richtig abgestimmten Gewichten w und Bias Werten b. Im Netzwerk aus Abbildung 2.4, befinden sich insgesamt 11.312 Gewichte und 38 Bias Werte. In diesen Zahlen steckt die 'Intelligenz' des neuronales Netzes.

$$784 \cdot 14 + 14 \cdot 14 + 14 \cdot 10 = 11.312 \; \textit{Gewichte}, \; 14 + 14 + 10 = 38 \; \textit{Bias Werte}$$

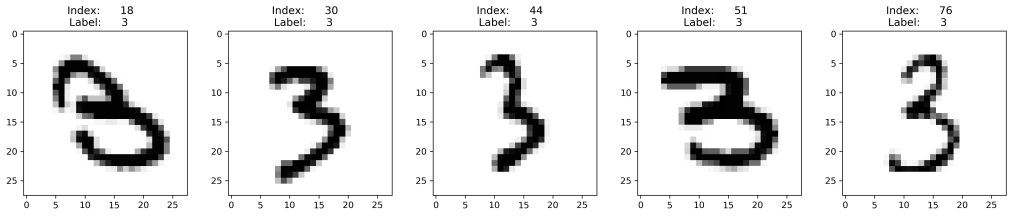

Abbildung 2.8: Beispielbilder der Zahl drei des MNIST-Datensatz

2.3 Funktionsweise eines neuronalen Netzwerks

Zum besseren Verständnis der Funktionsweise eines neuronalen Netzwerks, werden ein paar Bilder aus dem Datensatz genauer betrachtet. Dabei wurden die Ziffern in einzelne Komponenten wie Striche und Kreise aufgeteilt. Dadurch wird ersichtlich dass sich einige dieser Komponenten für unterschiedliche Klassen wiederholen.

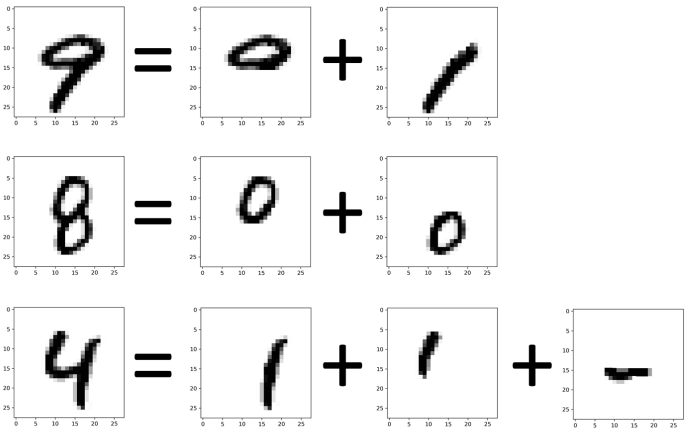

Abbildung 2.9: MNIST-Daten werden in einzelne Komponenten aufgeteilt

Zum Beispiel besteht eine neun aus einem Kreis und einem langen Strich. Ein Kreis ist in ähnlicher Form auch in einer acht enthalten während der lange Strich auch in einer vier verwendet wird. Diese Komponenten lassen sich wieder in Einzelteile unterteilen. Kreise werden beispielsweise in mehrere gekrümmte Kanten geteilt. So lässt sich das komplexere Problem, Erkennen einer Ziffer, in mehrere einfachere Probleme, z.B. Erkennen einer Kante aufteilen.

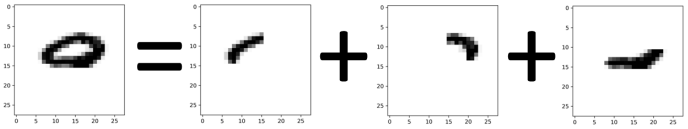

Abbildung 2.10: Einzelteile verschiedener Komponenten aus dem MNIST-Datensatz

Zur Veranschaulichung wird davon ausgegangen, dass ein Neuron im ersten Hidden-Layer ein Einzelteil, also eine kurze Kante an bestimmten Position der Bildfläche erkennt. Jedes Neuron sucht dabei an einer anderen Position nach einer bestimmten Kante. Ist es erfolgreich wird das Neuron aktiv.

Die Neuronen im zweiten Hidden-Layer können, anhand der gefundenen Einzelteile im ersten Hidden-Layer ganze Komponenten, wie Kreise oder lange Striche, erkennen. Auch hier werden die entsprechenden Neuronen aktiv, wenn eine Komponente gefunden wurde.

Die Aktivierung der Neuronen im Output-Layer (jedes Neuron repräsentiert eine Ziffer) setzt sich wiederum aus der jeweiligen Gewichtung der Komponenten-Aktivierung zusammen, wodurch das Neuron mit der größten Aktivierung die Ziffer des Eingangsbildes widerspiegelt. In der Abbildung 2.11 wird beispielhaft gezeigt wie eine Bildklassifikation in einem neuronalen Netz abläuft. Die aktiven Neuronen sind gelb markiert. [Gra19]

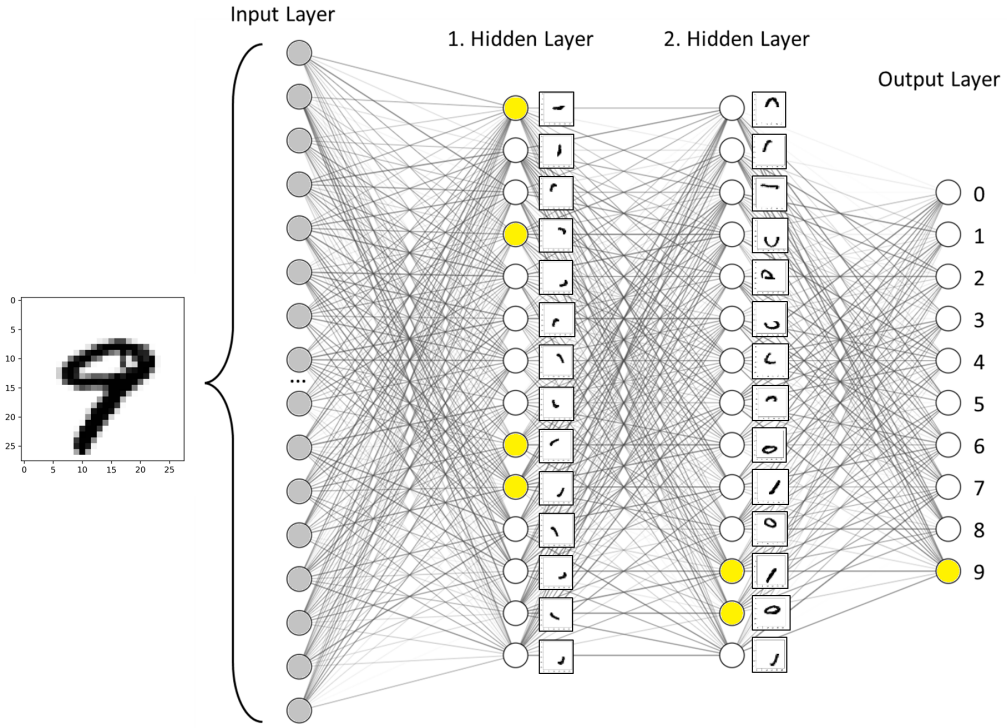

Abbildung 2.11: Schematische und stark vereinfachte Darstellung einer Klassifizierung

2.4 Beschreibung verschiedener Ebenen

2.4.1 Lineare-Layer

Die in Abschnitt 2.2 vorgestellten Ebenen eines KNN werden als Lineare-Layer oder auch als Fully-Connected-Layer bezeichnet. Um die Berechnung aller Aktivierungen **a** einer Ebene **L** mit einer Formel zu formulieren, werden deren Elemente mit Vektoren und Matrizen ausgedrückt. In Klammern sind die Dimensionen der Matrizen angegeben. n_{out} entspricht der Anzahl der Neuronen in Ebene **L** während n_{in} die der Ebene **L-1** beschreibt.

$$a^L_{(1,n_{out})} = f\left\{a^{L-1}_{(1,n_{in})} \cdot w_{(n_{in},n_{out})} + b^L_{(1,n_{out})}\right\} \tag{2.4}$$

2.4.2 Convolutional-Layer

Bei den Convolutional-Layern befinden sich die Gewichte in einer festgelegten Anzahl von Filtern **w**. Diese haben eine definierte Größe (Kernelsize) wie z.B. 3 x 3. Jedes Filter wird dann wie ein Sichtfenster mit der Schrittweite **s** (Stride) über die Input-Matrix **x** geschoben. Dies geschieht von links nach rechts und Zeile für Zeile. Vor jedem Schritt des Filters wird mit dessen Gewichten und den Werten aus dem Sichtfenster der Input-Matrix ein Ergebnis berechnet. Dazu werden die übereinander liegenden Zahlenwerte der Input-Matrix und des Filters jeweils multipliziert und die entstehenden Produkte addiert. Bei dieser Faltung entsteht für jedes Filter eine Ergebnismatrix **y**. [Jus19]

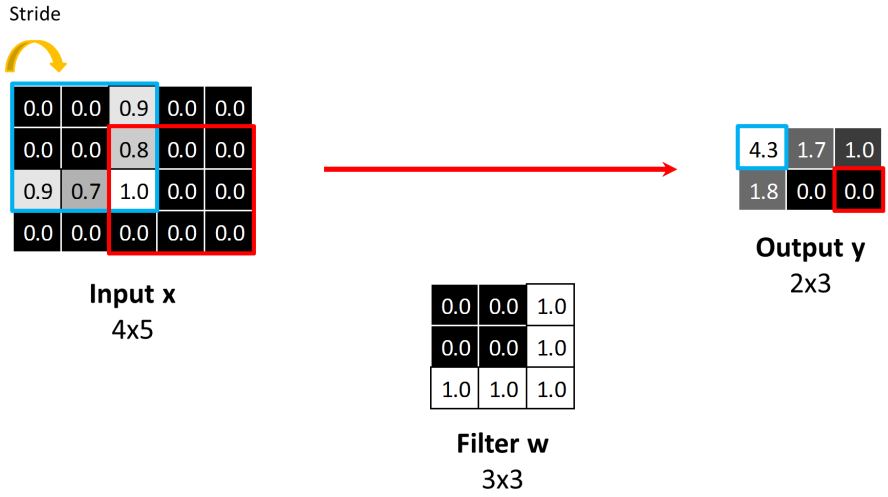

Abbildung 2.12: Visualisierung einer Faltung der Input-Matrix mit einem Filter

Filtergröße / Kernelsize

Gibt die Höhe und Breite des Filters **w** an. Im Beispiel wurde ein Filter der Größe 3 x 3 verwendet.

Stride / Sprungweite s

Diese gibt an um wie viele Pixel sich der Ausschnitt der Input-Matrix für jeden Schritt ändert. Im Beispiel wurde eine Sprungweite von eins verwendet.

Padding p

Das Padding beschreibt um wie viele Nullen die Input-Matrix in jede Richtung erweitert werden muss, bevor die Faltung durchgeführt werden kann. Verwendet man beispielsweise ein Padding von eins für die 4 x 5 Input Matrix so erhält man eine 6 x 7 Matrix (die 4 x 5 Matrix von Nullen umgeben). Diese größere Matrix wird dann gefaltet. Im Beispiel ist ein Padding von null verwendet worden.

Output

Mit der Formel 2.5 kann die Höhe und Breite des Ergebnisses **y** einer Faltung der Eingangsdaten **x** mit dem Filter **w** berechnet werden. **x** und **w** beschreiben die Kantenlängen der Eingangs- und der Filtermatrizen. [PyT19]

$$y = \left\lfloor \frac{x + 2 \cdot p - w}{s} + 1 \right\rfloor \tag{2.5}$$

Bias

Zusätzlich existiert für jedes Filter ein Bias-Wert, welcher zu der Faltungsmatrix addiert wird. Die resultierenden Summen werden wie bei den linearen Layern mit der Aktivierungsfunktion verrechnet.

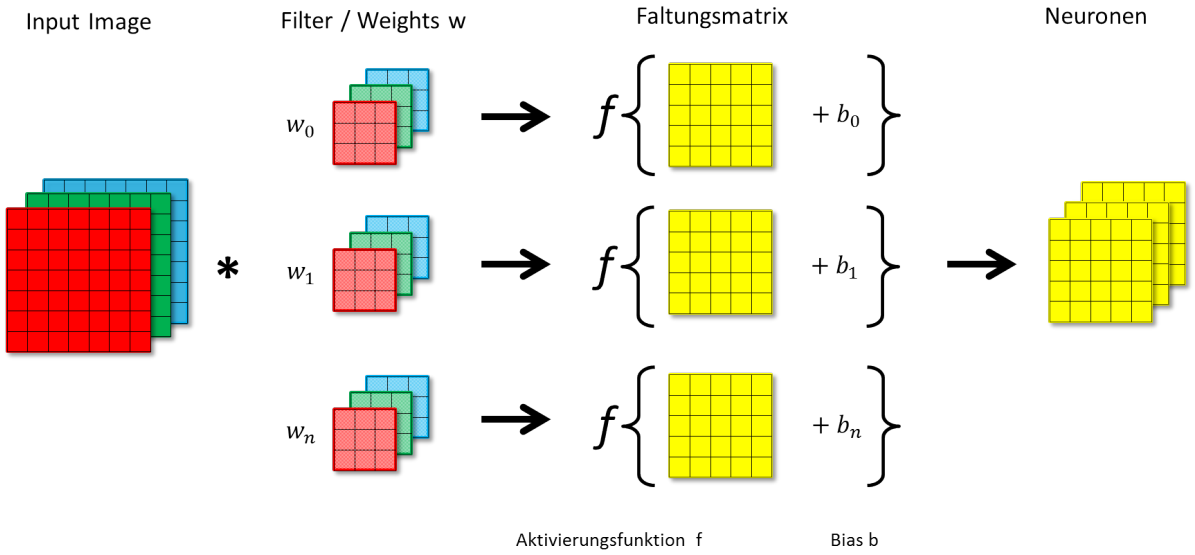

Abbildung 2.13: Übersicht zum Convolutional-Layer

Die Abbildung 2.13 stellt den gesamt Ablauf in einem Convolutional-Layer dar. Dabei wird eine Eingangsmatrix der Tiefe 3 für die 3 Farben des eines RGB-Bildes verwendet. Die Filter besitzen die gleiche Tiefe wie die Inputs der Ebene. So entsteht für jede Faltung mit einem Filter eine Ergebnismatrix der Tiefe 1. Diese werden danach übereinander angeordnet. Werden **n** Filter in einer Ebene verwendet, so ergibt sich daraus eine Ergebnismatrix der Ebene der Tiefe n.

2.4.3 MaxPooling-Layer

MaxPooling-Layer werden häufig nach einem Convolutional-Layer verwendet um die Größe einer Matrix zu reduzieren und nur die relevanten Aktivierungen an die nächste Ebene weiterzugeben. Ein MaxPooling-Layer funktioniert ähnlich wie ein Convolutional-Layer. Ein Sichtfenster (Größe durch Kernelsize festgelegt) wird von links nach rechts und Zeile für Zeile über die Input-Matrix geschoben. Vor jedem Verschiebungs-Schritt wird der größte Wert aus dem Fenster in die Ergebnismatrix geschrieben. Die exakte Vorgehensweise wird durch die Parameter: Kernelsize, Stride und Padding vorgegeben. Somit kann zur Größenberechnung der Ergebnismatrix abermals Formel 2.5 verwendet werden. [Jus19]

Abbildung 2.14: Beispiel eines MaxPooling-Layer mit Kernel Size 2 x 2, Stride 2 und Padding 0

2.4.4 DropOut-Layer

In einem Dropout Layer werden mit einer festgelegten Wahrscheinlichkeit zufällige Neuronen während des Trainingsprozesses kurzzeitig abgeschaltet, sodass deren Ergebnisse in den nachfolgenden Berechnungen nicht berücksichtigt werden. Diese Methode ist effektiv um Overfitting (Auswendiglernen der Trainingsdaten) zu vermeiden. DropOut-Layer sind nur in der Trainingsphase aktiv. Sobald das KNN voll umfänglich und mit bestmöglicher Genauigkeit eingesetzt wird, werden diese Ebenen deaktiviert. [PyT19]

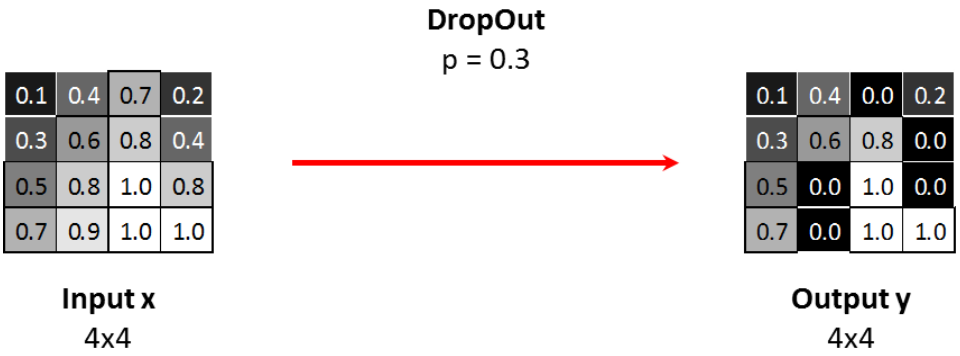

Abbildung 2.15: Beispiel eines DropOut Layer mit einer DropOut-Wahrscheinlichkeit von 0.3

2.5 Lernen / Backpropagation

2.5.1 Einstieg

Ziel des Lernalgorithmus ist es für die Variablen des Modells (Gewichte, Filter und Biases) Werte ist zu finden, welche die Eingangsdaten den richtigen Klassen zuordnen. Dieser Prozess wird auch als Backpropagation bezeichnet.

Dabei wird im Rahmen dieser Arbeit ausschließlich das sogenannte Supervised Learning behandelt. Bei diesem Lernverfahren ist jeder Eingabedatensatz (Bild) fest mit einer Klasse verknüpft, wodurch sich das Netzwerk selbst überprüfen kann. Die zum Eingangsdatensatz gehörende Klasse wird als Target bezeichnet, da es das Ziel des Modells ist, diese zu erkennen. Zur Erstellung des Targets wird üblicherweise die One-Hot-Kodierung verwendet.

2.5.2 Kettenregel

Für die Backpropagation wird die Kettenregel der Differentiation benötigt. Dabei wird sie in folgender Form angewandt.

$$\frac{\partial a}{\partial x} = \frac{\partial a}{\partial b} \cdot \frac{\partial b}{\partial x} \tag{2.6}$$

2.5.3 Loss-Funktion

Die Loss Funktion berechnet aus dem Output-Vektor y des KNN und dem zum Input gehörigen Target-Vektor T eine reale Zahl. Diese wird als der Fehler bzw. Error E des Netzes bezeichnet. Der Error E kann mit verschiedenen Loss Funktionen ermittelt werden. Bei dieser Bachelorarbeit wird die Mean-Squared-Error Loss Funktion angewandt.

$$f_{Loss}(y, T) = \frac{1}{m} \cdot \sum_{i=1}^{m} (y_i - T_i)^2 = E \tag{2.7}$$

2.5.4 Ablauf Backpropagation

Um ein Gewicht w_x anzupassen, muss bestimmt werden wie sich der Fehler **E**, bei Änderungen des Gewichts w_x, verhält. Dieses Verhalten kann durch die Ableitung der Loss Funktion nach dem anzupassenden Gewicht beschrieben werden.

$$dw_x = \frac{\partial f_{Loss}}{\partial w_x} \cdot E \tag{2.8}$$

Der Ablauf des Lernprozesses lässt sich in mehrere Abschnitte unterteilen. Dabei werden schrittweise die Werte dw_x berechnet. Im Anschluss werden damit die neuen Werte der Gewichte bestimmt.

Forward-Path

Beim Forward-Path werden, genau wie bei der Klassifikation eines Bildes, Ebene für Ebene die Aktivierungen **a** der Neuronen über die Gewichte, Biases, Filter und Aktivierungsfunktionen bestimmt. Bis man den Output erhält. Die Loss Funktion $f_{Loss}(y, T)$ berechnet anschließend mit dem Output **y** und dem Target **T** den Error **E**.

Die dabei benutzten Funktionen $f_{Lin}(a, w, b)$ für Lineare-Layer, $f_{Sig}(a)$ für die Sigmoid-Aktivierungsfunktion und $f_{Loss}(y, T)$ der Loss-Funktion werden als Forward-Path eines jeweiligen Layers bezeichnet.

Backward-Path

Neben dem Forward Paths besitzt jede Ebene des Netzwerks sowie der Loss auch einen Backward-Path. Dieser berechnet die partiellen Ableitungen des Forward-Paths. Dabei entstehen für Lineare Layer die Ableitungen $\frac{\partial f_{Lin}}{\partial a}$, $\frac{\partial f_{Lin}}{\partial w}$ und $\frac{\partial f_{Lin}}{\partial b}$, für die Sigmoid-Aktivierungsfunktion die Ableitung $\frac{\partial f_{Sig}}{\partial a}$ sowie für den Loss $\frac{\partial f_{Loss}}{\partial y}$

Beginnend mit dem Loss Layer werden im Backward Path die einzelnen Ebenen nacheinander rückwärts durchlaufen. Dabei wird jeweils der Input dy, da_x oder E (beim Loss) des Backward Paths mit den partiellen Ableitungen multipliziert. (siehe Abbildung 2.16) [May19], [Syl19], [Ert08, S. 268]

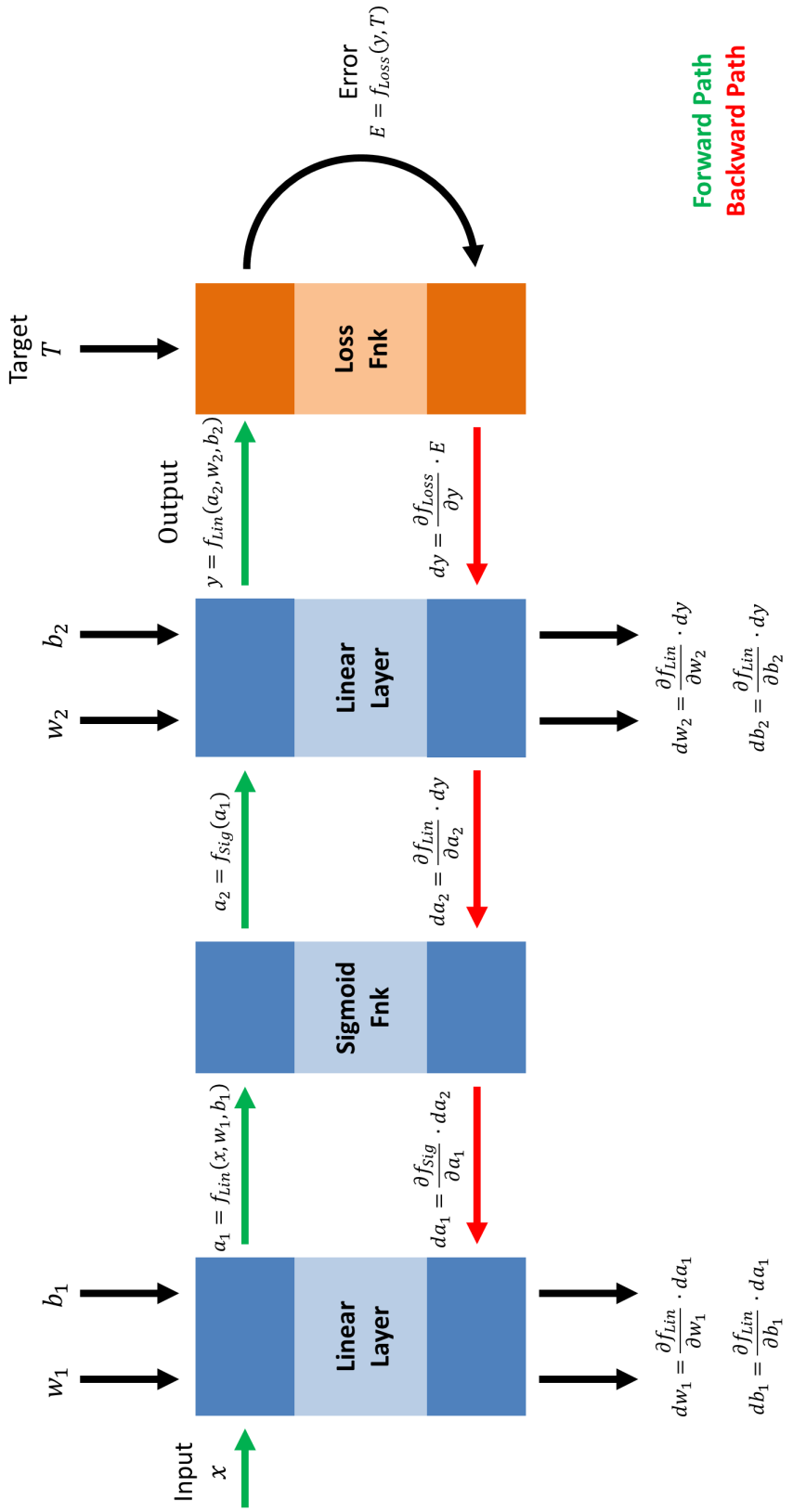

Abbildung 2.16: Beispiel für den Ablauf der Backpropagation
Bei allen Größen handelt es sind um Matrizen (Ausnahme: Error E)

So ergeben sich Ausdrücke dw_x und db_x für alle Gewichte und Bias-Werte. Betrachtet man beispielsweise den Ausdruck dw_2 und ersetzt dy und y (Formeln aus Abbildung 2.16), erhält man:

$$dw_2 = \frac{\partial f_{Lin}}{\partial w_2} \cdot dy = \frac{\partial f_{Lin}}{\partial w_2} \cdot \frac{\partial f_{Loss}}{\partial y} \cdot E = \frac{\partial f_{Lin}}{\partial w_2} \cdot \frac{\partial f_{Loss}}{\partial f_{Lin}} \cdot E \tag{2.9}$$

Mit Anwendung der Kettenregel, ergibt sich, bezogen auf das Gewicht w_2, die gewünschte Formel 2.8.

Gewichte anpassen

Der Wert dw_2 gibt die Änderung des Fehlers E an wenn das Gewicht w_2 um dw vergrößert wird. Ist dw_2 positiv wird der Fehler für positive Änderungen von w_2 größer. Um den Fehler zu reduzieren, fließt dw_2 mit negativem Vorzeichen in die Rechnung ein.

Zusätzlich wird eine Konstante namens Learning-Rate **LR** eingeführt. Durch sie kann die das Ausmaß der Gewichtsänderung bestimmt werden. Die Learning-Rate bewegt sich üblicherweise im Wertebereich zwischen 10^{-2} und 10^{-6}. Daraus ergibt sich die Formel 2.10, welche die einfachste Variante der Gewichtsanpassung beschreibt [Syl19], [Ert08, S. 268] :

$$w_{neu} = w_{alt} - LR \cdot dw \tag{2.10}$$

Funktionen des Backward-Paths

Zur besseren Verständlichkeit des Backward-Paths sind nachfolgend für verschiedene Ebenen die Funktionen des Forward- und Backward-Paths dargestellt.

Loss (Mean-Squared-Error)

Forward Funktion	$f_{Loss} = \frac{1}{m} \cdot \sum_{i=1}^{m} (y_i - T_i)^2$
Backward Funktion	$\frac{\partial f_{Loss}}{\partial Y_i} = \frac{2}{m} \cdot (y_i - T_i)$

Sigmoid

Forward Funktion	$f_{Sig} = \frac{1}{1+e^{-x}}$
Backward Funktion	$\frac{\partial f_{Sig}}{\partial x} = \frac{e^{-x}}{(1+e^{-x})^2}$

Linear Layer

Forward Funktion	$f_{Lin} = x \cdot w + b$
Backward Funktion	$\frac{\partial f_{Lin}}{\partial x} = w \quad \frac{\partial f_{Lin}}{\partial w} = x \quad \frac{\partial f_{Lin}}{\partial b} = 1$

3 Aufbau

3.1 Daten

Zum Trainieren eines neuronales Netzwerks und der Klassifikation von Daten wird ein entsprechender Datensatz benötigt. Die Kamerasysteme von Opdi-Tex werden häufig bei der Produktion von Backwaren wie Keksen oder Croissants eingesetzt. [ot19b] Um an viele Daten zu kommen bietet es sich an einen Datensatz mit handelsüblichen Keksen zu erstellen. Kekse entsprechen genau dem Einsatzbereich der Kamerasysteme und können zudem verhältnismäßig günstig erworben werden.

Sodass die Erkenntnisse nicht von einer Sorte abhängig sind, werden zwei unterschiedliche Keks-Sorten verwendet (Milka Schoko Minis und Nutella Biscuits).

Die Abbildung 3.2 zeigt den zur Aufnahme der Rohdaten verwendeten Scanner inklusive eines Förderbands und die Abbildung 3.1 einige der sortierten Milka Kekse.

Abbildung 3.1: Sortierte Kekse

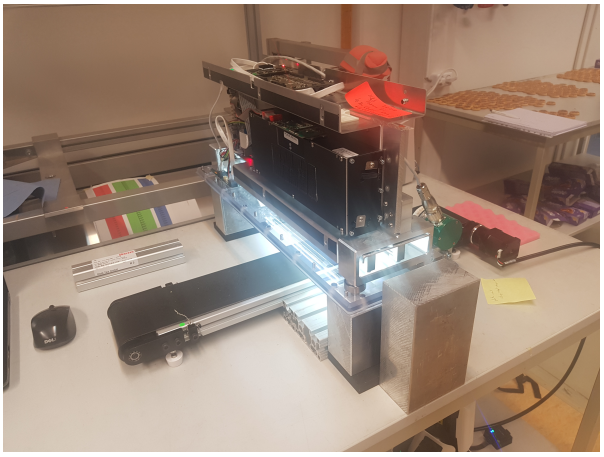

Abbildung 3.2: Scanner mit Förderband

3.1.1 Aufnahme der Rohdaten

Die Rohdaten werden mit Hilfe des Förderbandes und des Kamerasystems der Firma Opdi-Tex erstellt. Dazu wird die Aufnahme der Scankamera gestartet und alle Kekse einer Klasse nacheinander auf das Förderband gelegt. Mit einem Geber am Förderband erstellt die Kamera ein 10000 Zeilen langes Bild der durchlaufenden Kekse. Pro Klasse entstanden bis zu 13 dieser Aufnahmen.

Die Kamerasysteme sind über 2 m breit und besitzen deshalb mehrere Sensoren. [ot19a] Die Daten der einzelnen Sensoren werden im Kamerasystem zu einem Bild über die gesamte Breite zusammengefügt. Dabei können leichte Verschiebungen bzw. Fragmente entstehen. Um einen möglichst realitätsnahen Datensatz zu generieren, sollen darin auch Aufnahmen dieser Verschiebungen inkludiert sein. Dazu wird bei einem Drittel der Aufnahmen das Förderband genau im Schnittbereich von zwei Kamerasensoren positioniert. In Abbildung 3.3 sind Ausschnitte der Rohdaten an zu sehen. Der obere Ausschnitt zeigt Aufnahmen ohne Verschiebungen und der untere Aufnahmen mit Verschiebungen. Die Verschiebungen sind an der Farbe der Nutella Biscuits (rechte Seite) erkennbar.

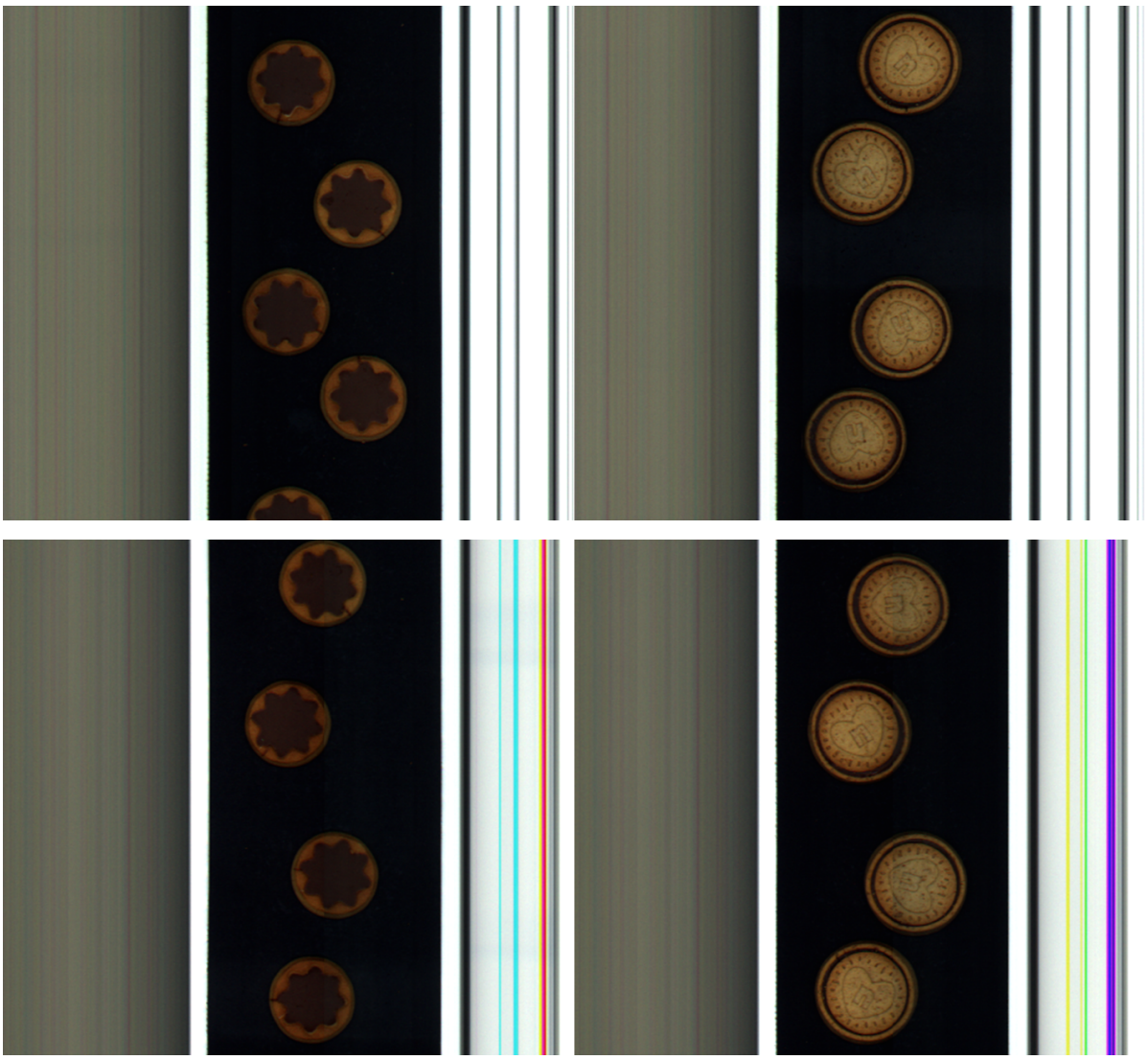

Abbildung 3.3: Ausschnitte der Rohdaten ohne Verschiebungen (oben) und mit Verschiebungen (unten)

3.1.2 Aufteilung in Klassen

Da der Herstellungsprozess Schwankungen unterliegt, sind selbst bei Keksen gleicher Sorte optische Unterschiede zu erkennen. So variiert bei den Milka Schoko Minis beispielsweise die Füllmenge der Schokomasse während bei den Nutella Biscuits die Position des Keksdeckels von der Mitte abweichen kann.

Anhand dieser optischen Differenzen, werden für jede Sorte separate Klassen definiert. Zwar könnten die Kekse auch einfach in *gut* und *schlecht* eingeteilt werden, was allerdings den Nachteil mit sich bringt, dass die Klassifikation keine Rückschlüsse auf die Fehlerart zulässt. Das heißt, wird ein Keks in der Produktion als fehlerhaft klassifiziert ist es hilfreich, wenn der Anlagenführer Details zum Fehler bekommt (z.b zu viel oder zu wenig Schokolade) und nicht nur erfährt, dass der Keks als *schlecht* eingestuft wurde. Zum anderen ist es für das Netzwerk leichter, wenn die Merkmale eines bestimmten Fehlers genau einer Klasse zugeordnet werden und nicht alle auftretenden Fehlermerkmale in einer gemeinsamen Klasse *schlecht* vereint werden. Außerdem ist es dadurch einfacher nachzuvollziehen was das Netzwerk berechnet und welche Merkmale im Eingangsbild welchen Klassen zugeordnet werden. Die nicht fehlerhaften Klassen können zudem im Nachhinein, falls gewünscht, in einer Klasse *schlecht* vereint werden.

Die Milka Schoko Minis werden in die Klassen: *gut*, $fraglich$, *schlecht* oder *kaputt* eingeteilt. Die ersten drei Klassen unterscheiden sich hauptsächlich in der Menge der dunklen Schokolade die den Keks bedeckt. Darunter liegt eine Schickt weißer Schokolade die bei zu wenig dunkler Schokolade erkennbar wird.

Milka-Gut:

Gute Kekse sind sauber mit dunkler Schokolade befüllt und es sind keine weiße Ränder zu sehen.

Milka-Fragliche:

Fragliche Kekse besitzen etwas weniger dunkle Schokoladenmasse, sodass sich leichte weiße Ränder der weißen Schokolade abzeichnen.

Milka-Schlecht:

Die Klasse der schlechten Kekse hat zu wenig dunkle Schokolade. Die nicht bedeckte weiße Schokolade ist deutlich erkennbar.

Milka-Kaputt:

Die kaputten Kekse sind anhand abgebrochener Kanten erkennbar. Da dieser Fehler in den zur Verfügung stehenden Keksen selten auftritt, werden hierfür abgebrochene Kanten nachträglich, per Hand hinzugefügt.

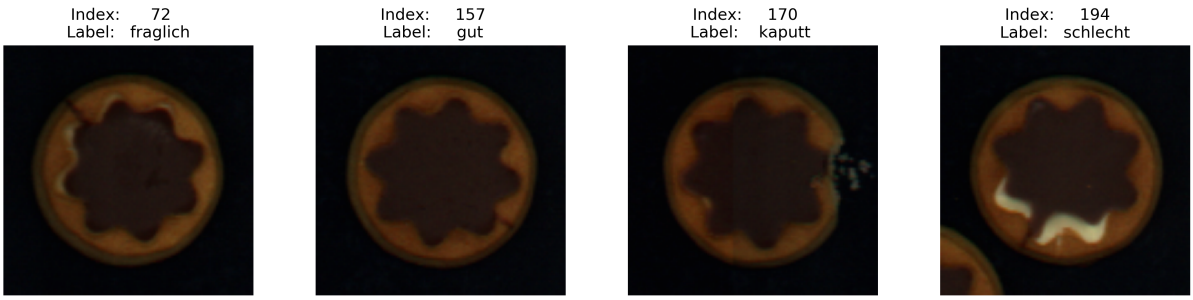

Abbildung 3.4: Beispiele der verschiedenen Klassen der Milka Kekse

Die Nutella Biscuits erhielten die Klassen: *gut*, *dick*, *fraglich*, *kaputt*, *leer*, *kein Deckel*, *schief* und *unsauber*.

Nutella-Gut:

Kekse dieser Klasse zeichnen sich durch einen besonders zentrierten Keksdeckel, einen gleichmäßigen Schokoladenrand und eine saubere Keksfläche aus.

Nutella-Dick:

Der Schokoladenrand ist deutlich breiter als bei den restlichen Keksen.

Nutella-Schief:

Der Keksdeckel befindet sich nicht in der Mitte.

Nutella-Fraglich:

In dieser Klasse befinden sich Kekse, die sich bezüglich Sauberkeit und Symmetrie an der Toleranzgrenze befinden. Kekse dieser Klasse können den der Klassen dick und schief durchaus ähnlich sehen.

Nutella-kein-Deckel:

Dieser Fehler dient der Simulation von Fehlern der Produktion. Hier kommt es vor dass nicht jeder Keks einen Deckel erhält. In einer handelsüblichen Kekspackung befinden sich keine Kekse mit diesem Fehlerbild, weswegen für die Generierung eines Datensatzes von guten Keksen nachträglich der Deckel entfernt wird.

Nutella-Leer:

In der Produktion kann es vorkommen, dass die Kekse nicht mit Schokolade befüllt werden. Um einen Datensatz für diese Fehlerart generieren zu können, werden gute Kekse umgedreht aufgenommen, da die Rückseite optisch ähnlich zu unbefüllten Keksen ist.

Nutella-Unsauber:

Der äußere Rand des Kekses ist mit Schokolade verschmiert.

Nutella-Kaputt:

Kekse mit abgebrochenen Kanten. Dafür wurden Kekse anderer Klassen modifiziert.

Abbildung 3.4 und Abbildung 3.5 zeigen Beispielbilder für alle beschriebenen Klassen.

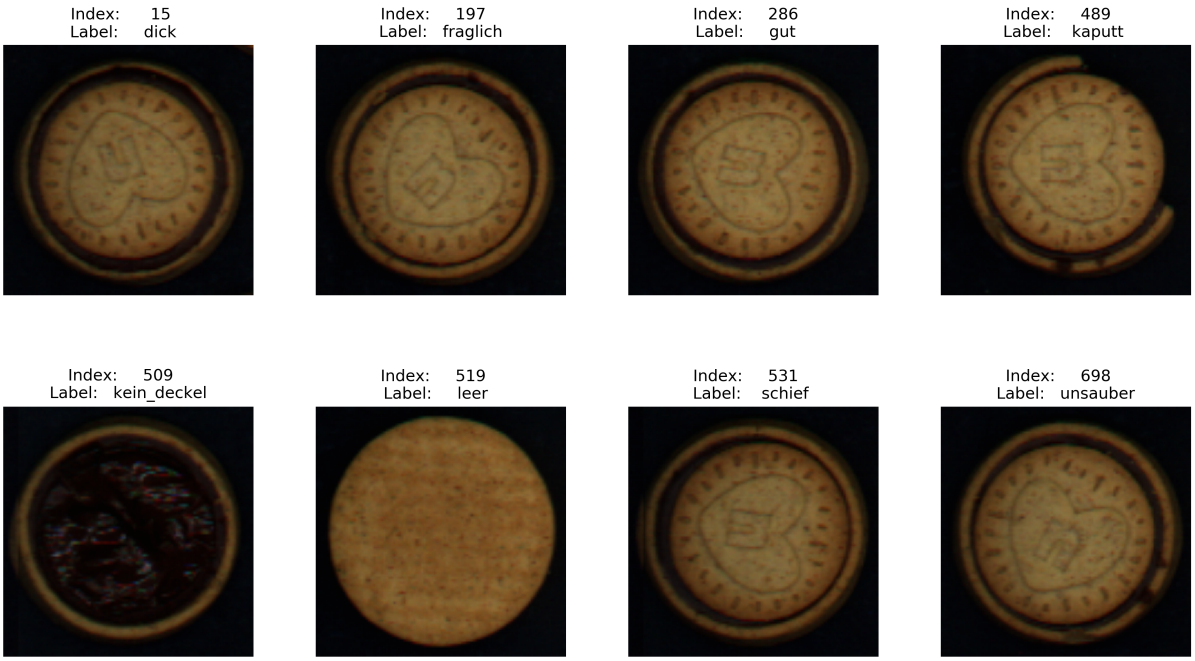

Abbildung 3.5: Beispiele der verschiedenen Klassen der Nutella Kekse

3.1.3 Ausschneiden der Kekse

Durch Methoden der klassischen Bildbearbeitung werden die einzelnen Kekse auf den Rohdaten detektiert, deren Position bestimmt und diese in einer json-Datei abgelegt. Mit diesen Informationen können unter Zuhilfenahme eines Skriptes alle Kekse aus den Rohdaten ausgeschnitten werden. Dabei beträgt die Größe aller Bilder 100 x 100 Pixel, so werden die Kekse beider Sorten komplett dargestellt. Zusätzlich wird für jede Klasse ein Ordner erstellt, in welchen die einzelnen Keksbilder einsortiert werden. Die Abbildungen 3.4 und 3.5 zeigen ein paar dieser ausgeschnittenen Bilder.

3.1.4 Aufteilung in Evaluierungs- und Trainingsdaten

Um eine realitätsnahe Aussage über die Leistungsfähigkeit eines neuronalen Netzes treffen zu können, ist es üblich die Evaluierung des Modells mit anderen, dem Netz unbekannten, Daten durchzuführen. Deshalb wird der Gesamtdatensatz in Evaluierungs- und Trainingsdaten aufgeteilt. Die Trainingsdaten werden nur zum Lernen (Änderung der Gewichte des Modells) verwendet, während die Evaluierungsdaten zur Bestimmung der Erkennungsgenauigkeit benutzt werden. Das kommt dem Anwendungsfall sehr nahe,

auch dort unbekannte Daten klassifiziert werden müssen. Im Rahmen dieser Arbeit wird der Anteil der Evaluierungsdaten auf 15 % der Bilder einer Klasse festgelegt. Abbildung 3.6 zeigt den Gesamtdatensatz mit den Anteilen der Evaluierungs- und Trainingsdaten aller Klassen.

Milka	fraglich	gut	kaputt	schlecht	gesamt Milka
Training	543	399	44	675	1661
Evaluierung	96	71	8	120	295
gesamt	639	470	52	795	1956

Nutella	dick	fraglich	gut	kaputt	kein Deckel
Training	827	693	1218	74	74
Evaluierung	147	123	215	14	14
gesamt	974	816	1433	88	88

Nutella	leer	schief	unsauber	gesamt Nutella	Nutella & Milka
Training	56	944	44	3930	5591
Evaluierung	11	167	8	699	994
gesamt	67	1111	52	4629	6585

Abbildung 3.6: Anzahl der Kekse der Evaluierungs- und Trainingsdaten

3.2 Netzwerk

Als neuronales Netzwerk soll ein fertiges Modell aus dem Pytorch-Framework verwendet werden. Das hat den Vorteil, dass diese Modelle auch mit vortrainierten Gewichten und Bias-Werten geladen werden können. Das kleinste convolutional neural Network (CNN), welches dem Nutzer zur Verfügung steht, ist das AlexNet. [ASH12]

Da das AlexNet von Beginn an gute Leistungen zeigte und Genauigkeiten von über 90 % erreichte, ist es nicht notwendig ein leistungsfähigeres CNN zu verwenden. In Folge dessen wird wurde das AlexNet als Modell für diese Bachelorarbeit ausgewählt.

Aufbau

Das AlexNet kann in zwei Teile aufgeteilt werden. Der erste Teil besteht aus Convolutional- und MaxPool Layern, in denen die verschiedenen Merkmale der Bilder erkannt werden. Der zweite Teil besteht aus Linearen- und DropOut-Layern. Hier werden die Input-Bilder anhand ihrer Merkmale Klassen zugeordnet.

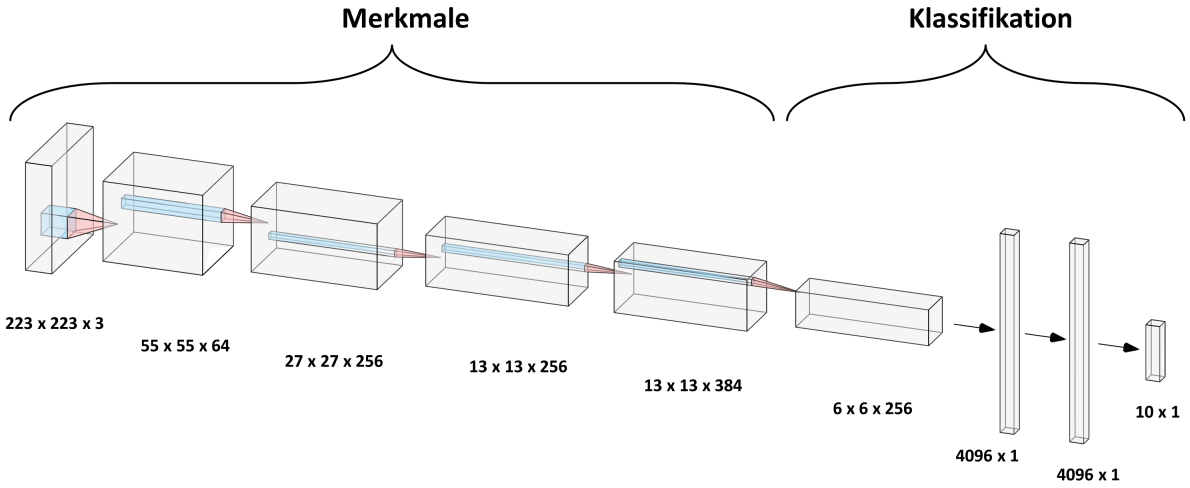

Abbildung 3.7: Graphische Darstellung vom Aufbau des AlexNets [Neu19]

Merkmale

Die erste Hälfte des Netzwerks besteht aus fünf Convolutional-Layern und drei MaxPooling-Layern. Dadurch wird ein Input-Image der Größe 223 x 223 x 3 umgewandelt in die Größe 6 x 6 x 256. Da die Daten aus dem Datensatz in den Dimensionen 100 x 100 x 3

vorliegen, werden die Bilder auf die Eingangsgröße des AlexNet gestreckt.

Klassifikation

Der hintere Bereich des Netzes besitzt drei Lineare-Layer und zwei DropOut-Layer. Die 9216 ($6 \cdot 6 \cdot 256 = 9216$) gefunden Merkmale bzw. Neuronen des ersten Abschnittes werden den richtigen Klassen zugeordnet.

Die letze Ebene, welche angibt welcher Klasse ein Bild vom Netz zugeordnet wird, besitzt standardmäßig 1000 Ausgangsneuronen. Das begründet sich darin, da das AlexNet zum Klassifizieren des ImageNet-Datensatzes, welcher 1000 Klassen besitzt, entworfen wurde. [ASH12], [Ima19] Dieses lineare-Layer wird durch ein neues ersetzt. Es hat dabei die gleiche Anzahl an Eingängen (4096) aber eine andere Anzahl an Ausgängen. Diese stimmt mit der Anzahl der zu erkennenden Klassen überein. Das entspricht vier Neuronen beim Milka Datensatz, acht Neuronen beim Nutella Datensatz bzw. zwölf Neuronen für die Kombination aus beiden Datensätzen.

Als Aktivierungsfunktion wird im gesamten Modell die ReLu-Funktion verwendet.

Vortrainiert / Pretrained

Wird ein Modell erstellt, erhalten alle Gewichte Zufallszahlen und die Bias-Werte auf null gesetzt werden. Pytorch bietet dem Nutzer die Möglichkeit das Modell vortrainiert zu laden. Dabei werden Gewichte und Bias-Werte geladen die sich bei dem Training mit dem ImageNet-Datensatz als geeignet herausgestellt haben. Wird im nachfolgenden von einem vortrainierten Modell gesprochen heißt das, dass bis auf die letzte Ebene alle Ebenen die vortrainierten Parameter verwenden. Für dieses geänderte Layer stehen keine vortrainierten Parameter zur Verfügung, da es verändert wurde.

Abbildung 3.8 gibt eine Übersicht aller Ebenen des AlexNets. Dabei handelt es sich um die Implementierung des AlexNets in Pytorch [PyT19]. Die Dimension gibt dabei jeweils die Output-Größe einer Ebene an.

Ebene	Typ	Filter	Dimension	Kernel	Stride	Padding	Aktiv. Fnk.
-	Input	1	223x223	-	-	-	-
Features							
1	Convolution	64	55x55	11x11	4	2	ReLu
2	MaxPool	64	27x27	3x3	2	0	-
3	Convolution	192	27x27	5x5	1	2	ReLu
4	MaxPool	192	13x13	3x3	2	0	-
5	Convolution	384	13x13	3x3	1	1	ReLu
6	Convolution	256	13x13	3x3	1	1	ReLu
7	Convolution	256	13x13	3x3	1	1	ReLu
8	MaxPool	256	6x6	3x3	2	0	-
Classifier							
9	DropOut	-	-	-	-	-	-
10	Linear	-	4096	-	-	-	ReLu
11	DropOut	-	-	-	-	-	-
12	Linear	-	4096	-	-	-	ReLu
13	Linear	-	10	-	-	-	-

Abbildung 3.8: Tabellarische Darstellung vom Aufbau des AlexNets

3.3 Ablauf

Hier soll der Programmablauf des Python-Skripts zum Trainieren und Testen des AlexNets erklärt werden. Das Training eines CNN erfolgt meist in mehreren Epochen. Eine Epoche beschreibt dabei das Training mit allen Trainingsdaten.

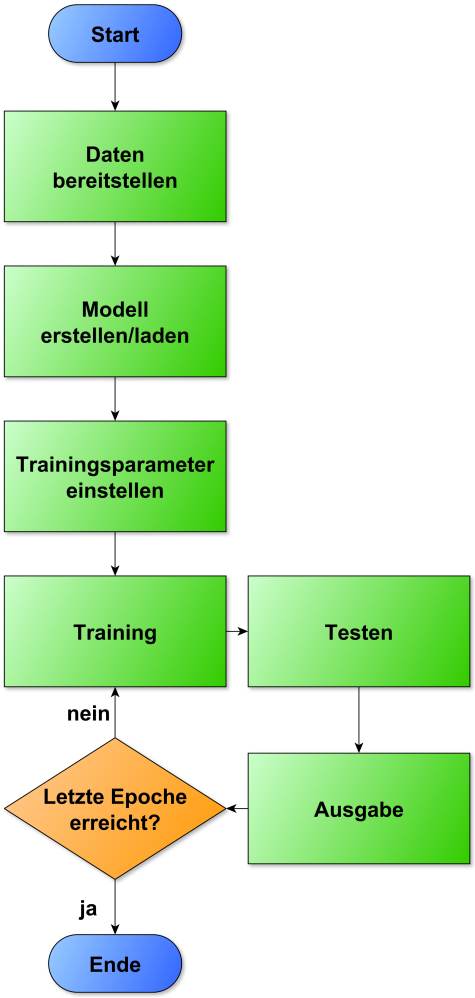

Abbildung 3.9: Graphische Darstellung des Gesamtablaufs

3.3.1 Daten bereitstellen

Datensatz

Um die Keksbilder, softwareseitig zu einem Datensatz zusammen zu führen wird eine Klasse *Dataset* erstellt. Bei dieser Klasse handelt es sich um eine Ableitung der vom

Pytorch-Framework bereitgestellten Klasse *torch.utils.data.Dataset*. Um die richtige Funktionalität des Datensatzes bereitzustellen, müssen einige Methoden überschrieben werden.

__init__ Es wird der Dateipfad zu den Keksbildern angegeben und festgelegt ob es sich um ein Testdatensatz oder ein Trainingsdatensatz handeln soll. Der Testdatensatz besteht aus den ersten 15 % (aufgerundet) einer Bilder jeder Klasse. Dem Trainingsdatensatz werden die restlichen Bilder zugewiesen. Für jeden Datensatz entsteht eine Liste mit Dateinamen und zugehörigen Klassen.

__len__ Diese Methode gibt die Anzahl der Daten in einem Datensatz zurück.

__getitem__(i) Anhand des Indexes i, welcher die Position eines Bildes im Datensatz angibt, kann dieses Bild (100 x 100 Pixel) geladen und in eine Matrix der Größe 3 x 100 x 100 konvertiert werden. i kann Werte von 0 bis $\_\_len\_\_ - 1$ annehmen. Soll während des Trainings Augmentation (Kapitel: 4.1) verwendet werden, wird die Matrix an dieser Stelle entsprechend bearbeitet. Eine Normierung nach *Pytorch* findet ebenfalls an dieser Stelle statt. [PyT19] Die Matrix wird dann auf die Eingangsgröße des CNN (3 x 223 x 223) gestreckt und zusammen mit dem, aus der zugehörigen Klasse geformten, Target-Vektor zurückgegeben.

Mit der Klasse *Dataset* können nun Objekte erstellt werden, die mit der Methode __getitem__ Daten für das Trainieren und Testen des AlexNet bereitstellen.

DataLoader

Die Klasse *torch.utils.data.DataLoader* ermöglicht es dem Nutzer, die Daten eines *Datasets* in Batches anzuordnen. Dabei werden die Daten und Targets aus dem *Dataset* gebündelt. Die verwendeten Batches haben eine Größe von 32 Daten-Matrizen und Target-Vektoren. So kann das Netzwerk 32 Bilder auf einmal lernen bzw. klassifizieren. Zudem kann der *DataLoader* so initialisiert werden, dass er die Daten in zufälliger Reihenfolge ($shuffle = True$) ausgibt. Diese Funktion wird für den Trainingsdatensatz verwendet um sicherzustellen, dass die Reihenfolge der Bilder keinen Einfluss auf das Training hat. Die beiden Objekte $Loader\_Train$ und $Loader\_Test$ stellen dem Modell somit die nötigen Daten und Targets zur Verfügung.

3.3.2 Modell erstellen/laden

Je nach Anforderung kann in diesem Programmabschnitt ein komplett neues AlexNet erstellt oder ein gespeichertes Modell geladen werden. Wird das AlexNet (*torchvision.models.alexnet*) neu generiert, so kann das mit vor trainieren Parametern ($pretrained = True$) oder Zufallswerten geschehen. Danach wird das Ausgangslayer des Modells an die Anzahl der zu klassifizierenden Klassen angepasst.

Um das Training zu beschleunigen wird das verwendete Modell auf die Grafikkarte (GPU) verschoben. Damit die erhöhte Grafikkartenperformance für die Berechnung des CNNs genutzt werden kann, müssen auch die Eingangsdaten für die GPU zugänglich sein. Das bringt einen enormen Geschwindigkeitsvorteil.

Sollen bestimmte Ebenen des Modells nicht trainiert werden, wird für die Parameter dieser Ebenen die Variable $requires\_grad$ auf $False$ gesetzt. Die Ebenen sind dadurch eingefroren.

3.3.3 Trainingsparameter einstellen

Nachdem ein CNN mit allen Parametern geladen wurde, kann ein Trainingsalgorithmus (bei Pytorch auch Optimizer genannt) gewählt werden. Der für seine Effizienz bekannte Optimierungsalgorithmus ADAM (Adaptive Moment Estimation) [KB] ist in den Pytorch-Bibliotheken bereits zu finden. Es handelt sich dabei um die Klasse *torch.optim.Adam*. Sie benötigt zum initialisieren die zu trainierenden Parameter des CNN und die Learning-Rate LR.

Eine Loss-Funktion wird ebenfalls definiert. Es wird der unter $torch.nn.MSELoss$ zu findende Mean-Squared-Error verwendet. Pytorch bietet eine große Anzahl von Loss Funktionen an. Pytorch bietet noch einige weitere Loss-Funktionen an. Da es sich bei dieser Arbeit um einen ersten Prototypen handelt, wird auf den trivialen MSELoss zurückgegriffen.

3.3.4 Trainieren

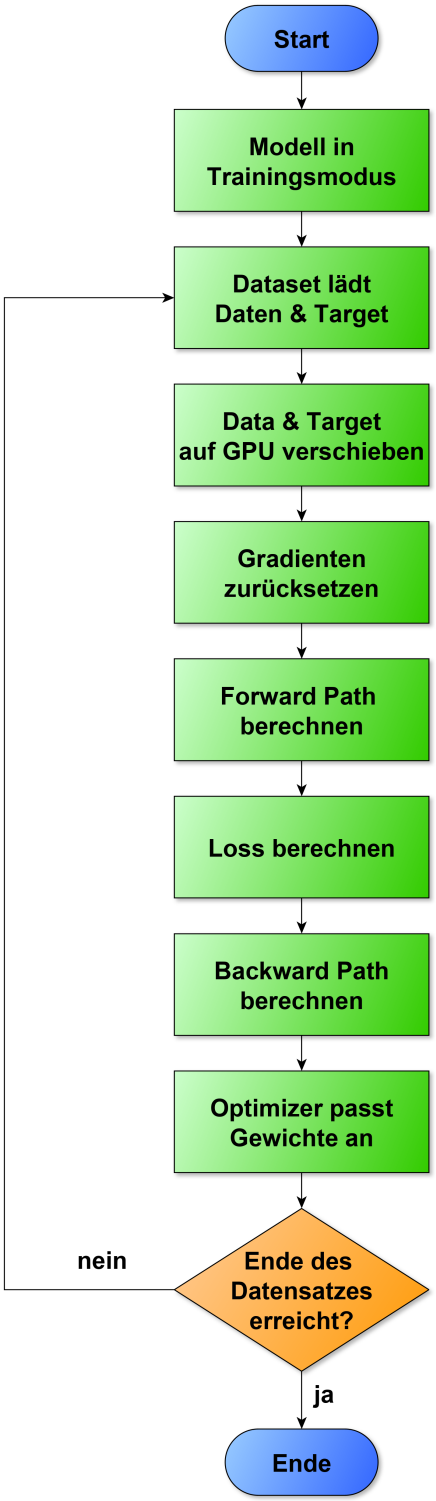

Abbildung 3.10: Trainingsablauf einer Epoche

In diesem Abschnitt wird das Training einer einzelnen Epoche beschrieben. Dazu wird das Modell zuerst in den Trainingsmodus versetzt, wodurch alle DropOut Layer aktiviert werden. ($model.train()$) (Kapitel 2.15)

Anschließend wird ein Batch (Datenpaket) des Trainingsdatensatzes geladen. Diese beinhaltet 32 Eingangsdaten und Target-Vektoren. Diese werden auf der GPU gespeichert.

Als nächstes setzt der Optimierungsalgorithmus die Gradienten der Parameter zurück. Das ist vergleichbar mit den Variablen dw_x aus Kapitel 2.5. Diese werden für das Training jeder Eingangsdaten neu berechnet. Da Pytorch diese Werte automatisch für alle Variablen sowie Matrizen und Vektoren berechnet, müssen sie aktiv auf Null gesetzt werden. [Sou19]

Mit Hilfe der Eingangsdaten wird über den Forward-Path des CNN der Output y bestimmt. Die Loss-Funktion berechnet anschließend den Error E. Damit lassen sich nun über den Backward-Path die Gradienten (wie dw_x) aller Netzwerkparameter bestimmen.

Zuletzt passt der Optimizer mit mit Hilfe der Gradienten die Parameter des Modells an.

Diese Schritte laufen solange ab, bis der Trainingsdatensatz komplett gelernt wurde.

3.3.5 Testen

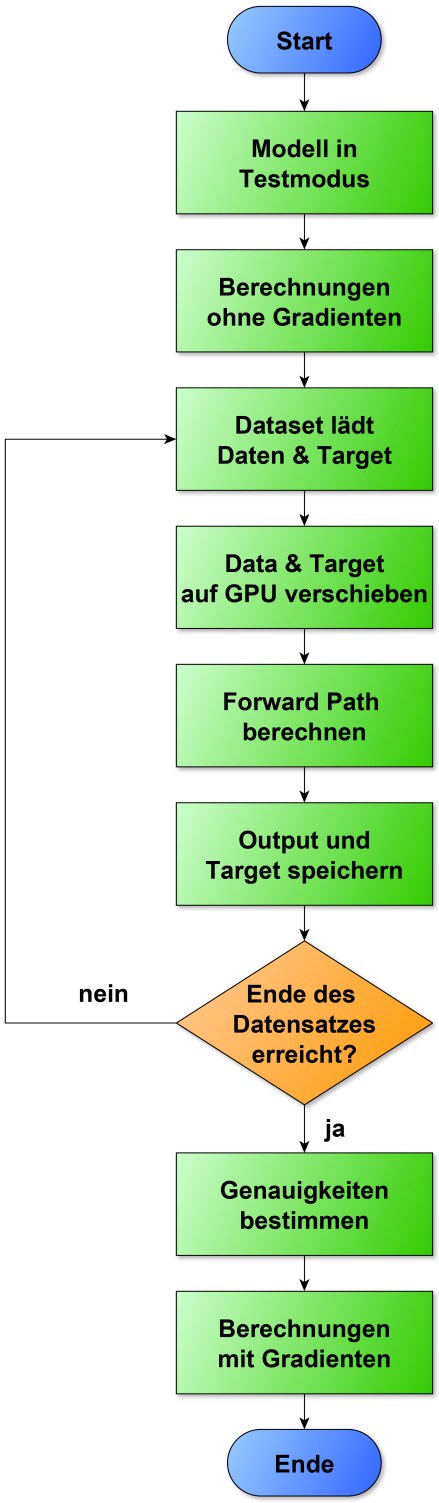

Abbildung 3.11: Testablauf einer Epoche

Dieses Kapitel beschreibt das Testen des CNNs mit den Testdaten beschrieben. Dazu wird das Modell zuerst in den Testmodus versetzt, wodurch alle DropOut-Layer deaktiviert werden. ($model.eval()$)

Der gesamte Testablaufs wird ohne die automatische Berechnung der Gradienten durchlaufen ($with\ torch.no\_grad()$). Das verhindert den Verbrauch von Ressourcen.

Zu Beginn werden die Eingangsdaten und Target-Vektoren aus dem Trainingsdatensatz auf die GPU geladen. (Siehe Anmerkungen 3.3.4)

Die Input Daten werden über den Forward Path des CNNs klassifiziert. Die Vorhersage des Modells gibt die Position des größten Werts im Output y an.

Die Vorhersage und das Target werden jeweils in einer Liste abgelegt.

Sobald alle Testdaten einmal klassifiziert sind, kann anhand der Vorhersagen des Modells und der Targets die Gesamtgenauigkeit sowie die Genauigkeit der jeweiligen Klasse bestimmt werden. Die Confusion-Matrix (Kapitel 4.3.3) wird ebenfalls durch diese Informationen erstellt.

3.3.6 Ausgabe

Unter dem Punkt Ausgabe werden mehrere Funktionen zusammen gefasst. Diese dienen der Speicherung des CNNs und der Genauigkeiten sowie der Ausgabe des Trainingsfortschritts in der Konsole.

Speicherung des Modells

Das Modell wird jede Epoche gespeichert. So geht bei einem Systemfehler nicht der gesamte Fortschritt mehrerer Stunden und vieler Epochen von Berechnungen verloren, sondern nur der der aktuellen Epoche. Außerdem wird zusätzlich das Modell mit der höchsten Erkennungsgenauigkeit gespeichert.

Speichern der Genauigkeiten

Nach jeder Epoche wird die Gesamtgenauigkeit sowie die Genauigkeit der einzelnen Klassen in einer CSV-Datei abgelegt. So kann im Nachhinein der Verlauf eines Trainings dargestellt werden. Am Anfang wird die Genauigkeit nach dem ersten Training aufgezeichnet, darum ist auf den Graphen im Kapitel 4 der Ausgangspunkt der Verläufe nicht gleich, obwohl das Training mit dem gleichen Ausgangsmodell stattfand.

Konsolenausgabe

In der Konsole wird die aktuell beendete Epoche zusammen mit der Genauigkeit und der dafür benötigten Berechnungsdauer angezeigt. So kann der Fortschritt beobachtet und die Gesamtdauer für mehrere Epochen abgeschätzt werden.

4 Ergebnisse

4.1 Augmentation

Augmentation heißt, dass mithilfe von Bildbearbeitung der Trainingsdaten neue Daten erstellt werden, um dem Datensatz zu vergrößern. Dazu gibt es verschiedene Transformationsmöglichkeiten, welche auch kombiniert werden können: [Raj19]

- Spiegeln (vertikal und/oder horizontal)
- Rotieren
- Skalieren (vertikal und/oder horizontal)
- Zuschneiden
- Verschieben (vertikal und/oder horizontal)
- Verrauschen

Welche Methoden sich eignen hängt stark von den Daten und den Anforderungen an das System ab. Sollen zum Beispiel Zahlen erkannt werden ist es unpraktisch diese zu Rotieren. Da zum Beispiel eine um 180° gedrehte Zahl '6' genau wie eine '9' aussieht.

Um Wiederholungen zu vermeiden und möglichst viele verschiedene Daten durch die Augmentation zu gewinnen, wird diese häufig bewusst mit Zufallswerten durchgeführt. So kann bei allen Methoden entweder ein Zahlenbereich oder eine Wahrscheinlichkeit angeben werden. Mit diesem Bereich wird dann beispielsweise die Winkeländerung des Originalbildes durch eine Rotation beschrieben. Ob eine Transformation wie eine Spiegelung durchgeführt wird, geben die Wahrscheinlichkeitswerte an. Würden alle augmentierten Bilder gespeichert werden, bräuchte man ein Vielfaches des ursprünglichen Datensatzes an Speicherplatz. Deshalb werden vor dem Training die Originalbilder aus dem entsprechenden Datensatz durch bestimmte Transformationen geändert und danach wird das

geänderte Bild wieder verworfen. So müssen nur die ursprünglichen Daten auf der Festplatte hinterlegt sein.

Die Trainingsdaten beider Kekssorten werden mit mehreren Methoden verändert. Zum einen werden diese um einen zufälligen Winkel zwischen -180° und +180° gedreht. Zusätzlich erfolgt eine Verschiebung in vertikaler und horizontaler Richtung im Bereich um bis zu 5 % der Bildbreite. Außerdem wird das ganze Bild in einem Bereich von 95 % bis 105 % der Bildgröße skaliert und mit jeweils 50 % Wahrscheinlichkeit vertikal und horizontal gespiegelt.

Auf der Vorderseite der Nutella Kekse befindet sich ein kleines 'n'. Wird das Keksbild gespiegelt, befindet sich der gerade Strich des Buchstabens auf der rechten Seite. Da es sich hierbei um eine sehr kleine Abweichung handelt, wird diese nicht berücksichtigt. Ein spiegelverkehrtes 'n' würde nicht als solches erkannt werden.

Die bei der Augmentation entstandenen freigewordenen Bereiche müssen mit einer Füllfarbe bedeckt werden. Dazu wurden mehrere Hintergrundfarben mit einem Bildbearbeitungsprogramm manuell ausgelesen, und verglichen. Die Farbe #06090e sieht dabei dem schwarzen Förderband am ähnlichsten. Diese wurde bei der Augmentation als Füllfarbe verwendet.

Die Abbildungen 4.1 und 4.2 zeigen für beide Kekssorten zuerst Aufnahmen ohne und darunter mit Augmentation. Besonders gut sichtbar sind die zufällige Rotation sowie die vertikale und horizontale Verschiebung. Bei einigen Keksen sind auch die Skalierungen zu erkennen.

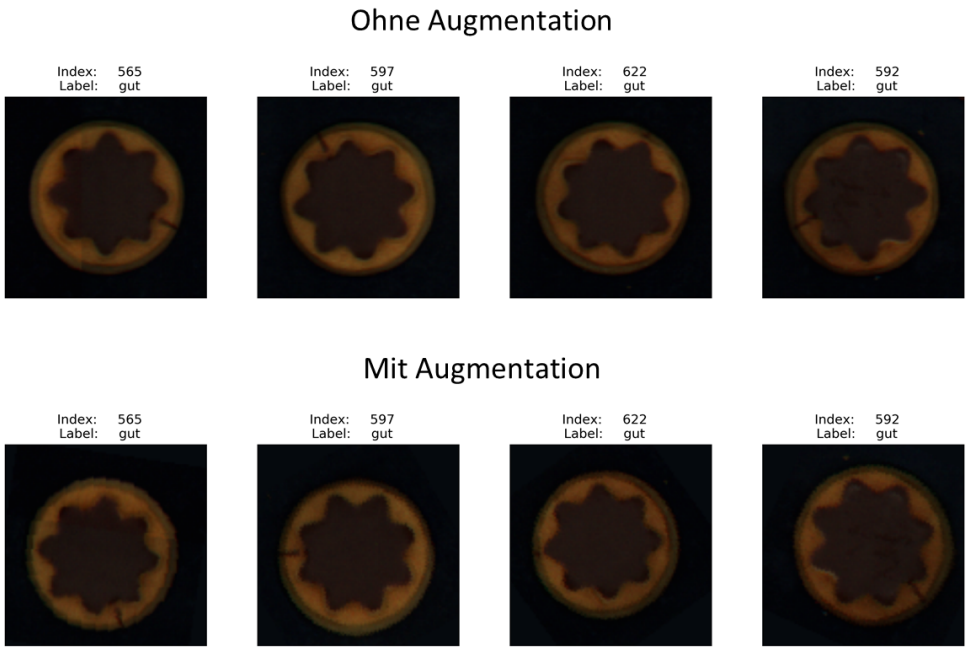

Abbildung 4.1: Vergleich von Milka Daten ohne und mit Augmentation

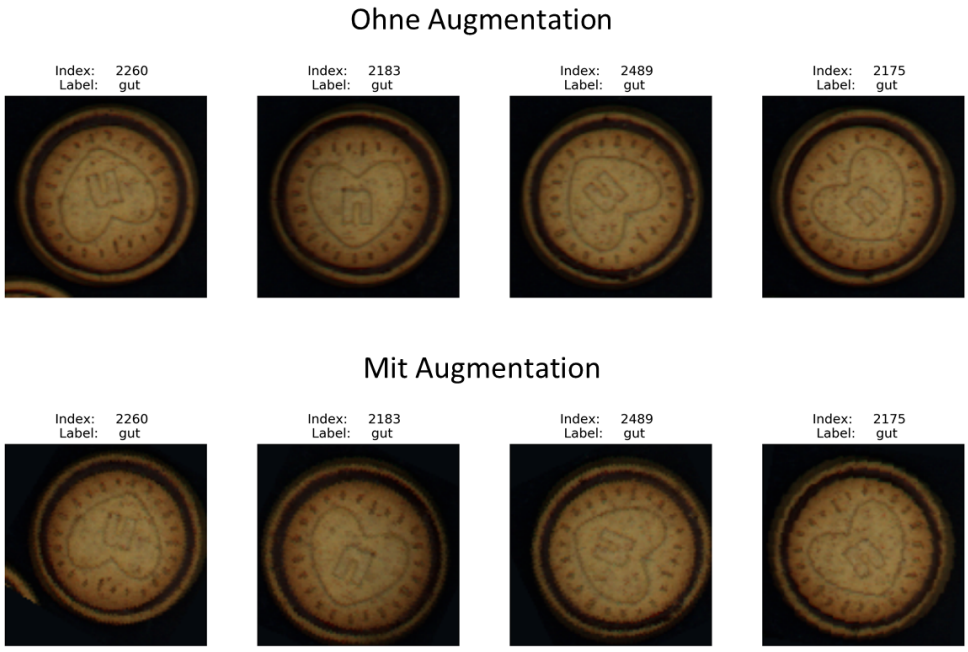

Abbildung 4.2: Vergleich von Nutella Daten ohne und mit Augmentation

Um die möglichen Vorteile der Augmentation darzustellen werden 2 neuronale Netze über 100 Epochen trainiert. Dabei handelt es sich jeweils um ein vortrainiertes AlexNet. Um für beide Verfahren gleiche Startbedingungen zu schaffen, wird ein Modell erstellt und kopiert. So können beide trotz der mit Zufallszahlen gefüllten Ausgangsebenen mit den gleichen Werten beginnen. Trainiert werden alle Ebenen. Dafür wird der Datensatz mit beiden Kekssorten verwendet. Bei einem Modell wird die Augmentation angewandt, während bei dem anderen keine Augmentation angewandt wird.

Graph	mit Augmentation	ohne Augmentation
Modell	AlexNet	AlexNet
Trainierte Layer	alle	alle
pretrained	ja	ja
Datensatz	combiniert	combiniert
Augmentation	ja	nein
Lernalgorithmus	Adam	Adam
Learningrate	$10^{-4.0}$	$10^{-4.0}$
Lossfunktion	MSELoss	MSELoss

Abbildung 4.3: Einstellungen beim Training des Augmentation-Vergleichs

Die Abbildung 4.4 zeigt den Verlauf der beiden Erkennungsgenauigkeiten für den Testdatensatz. Zu erkennen ist dabei dass sich das Modell ohne Augmentation schneller an die Daten anpasst, weil dem Modell immer die exakt gleichen Bilder gezeigt werden. Die Parameter können sich so besser auf die sich wiederholenden Aufnahmen einstellen. (Epoche 0 bis 20) Das Modell mit Augmentation ist anfangs langsamer, weil es jede Epoche veränderte Daten zum Trainieren bekommt. Dadurch dauert es länger bis die Parameter sich an die Daten angepasst haben.

Dafür entwickelt sich bei dem Netzwerk mit Augmentation eine größere Variabilität. So können bestimmte komplett unbekannte Bilder aus dem Testdatensatz ab Epoche 45 deutlich besser richtig zugeordnet werden als beim Modell ohne Augmentation. Das liegt daran, dass durch die Augmentation Daten erzeugt werden, die auch denen im Testdatensatz ähneln. So kann die Genauigkeit kontinuierlich auf bis zu 98.0 % (Epoche 95) gesteigert werden. Der Kontrahent dagegen kann ab Epoche 20 seine Erkennungsrate kaum noch steigern. Er erreicht ein Maximum von 93.5 %.

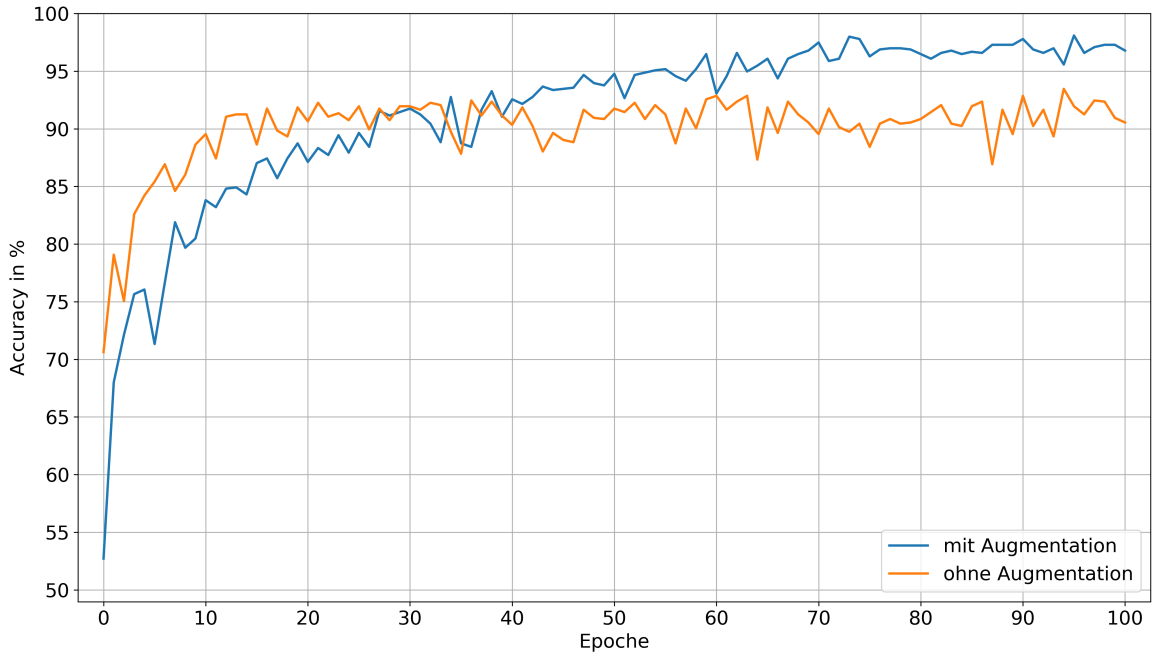

Abbildung 4.4: Verläufe der Genauigkeit mit und ohne Augmentation

Neben dem offensichtlichen Vorteil der gesteigerten Genauigkeit hat Data-Augmentation noch einen weiteren positiven Effekt. Nämlich das das so genannte Overfitting (Überanpassung) besser vermieden werden kann. Bei Overfitting spricht man von dem Effekt dass sich das Modell nach gewisser Trainingszeit so gut an die Trainingsdaten angepasst hat, dass es bei den Testdaten zu einer Verschlechterung der Erkennungsgenauigkeit kommt. Das Netz hat sich zu sehr auf die Trainingsdaten spezialisiert. Es hat diese sozusagen auswendig gelernt. Durch die sich ständig verändernden Daten der Augmentation kann diesem Effekt vorgebeugt werden.

4.2 Learning-Rate

Wie im Kapitel 2.5.4 beschrieben, spielt die Learning-Rate LR eine wichtige Rolle bei der Anpassung von Modellparametern. Zudem ist mit der Auswahl des Training-Algorithmus die Learning-Rate eine der wenigen Möglichkeiten wie der Nutzer das Lernverhalten beeinflussen kann.

Bei der Learning-Rate werden kleine Werte verwendet, um Erlerntes des bisherigen Trainings nicht bei dem Update der Parameter mit starken Änderungen zu überschreiben sondern mit kleinen Anpassungen zu verbessern.

$$w_{neu} = w_{alt} - LR \cdot dw \tag{4.1}$$

Zum Vergleich unterschiedlicher Learning-Rates wurden 5 Modelle mit jeweils verschiedenen Learning-Rates trainiert. Dabei erhält jedes eine der Learning-Rates:$10^{-3.5}$, 10^{-4}, $10^{-4,5}$, 10^{-5}, $10^{-5,5}$. Durch die Augmentation sind die Ergebnisse leicht von den zufälligen Bildtransformationen abhängig.

Graph	LR 3.5	LR 4.0	LR 4.5	LR 5.0	LR 5.5
Modell	AlexNet	AlexNet	AlexNet	AlexNet	AlexNet
Trainierte Layer	alle	alle	alle	alle	alle
pretrained	ja	ja	ja	ja	ja
Datensatz	Milka	Milka	Milka	Milka	Milka
Augmentation	ja	ja	ja	ja	ja
Lernalgorithmus	Adam	Adam	Adam	Adam	Adam
Learningrate	$10^{-3.5}$	$10^{-4.0}$	$10^{-4.5}$	$10^{-5.0}$	$10^{-5.5}$
Lossfunktion	MSELoss	MSELoss	MSELoss	MSELoss	MSELoss

Abbildung 4.5: Einstellungen beim Training des Learning-Rate-Vergleichs

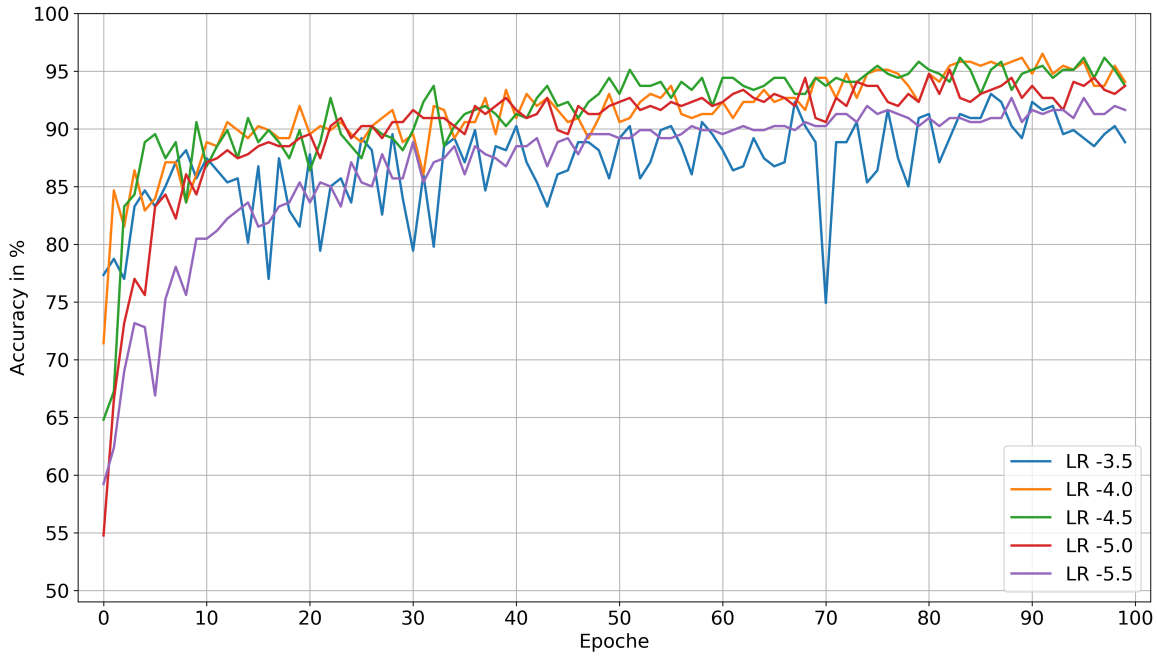

Abbildung 4.6: Verläufe der Genauigkeit beim Training mit verschiedenen Learning-Rates

Die Abbildung 4.6 zeigt den Verlauf der Erkennungsgenauigkeit über den Trainingsepochen für unterschiedliche Learning-Rates an. Die Legende des Diagramms gibt dabei jeweils nur den Exponenten x in $LR = 10^x$ an. Der Verlauf von LR $10^{-3.5}$ ist besonders sprunghaft. Das erklärt sich durch die großen Änderungen am Netzwerk die diese Learning-Rate verursacht. Diese bewegen sich auch oft über den optimalen Punkt der Parameter hinaus.

Die Kurve der LR $10^{-5.5}$ hat mit der kleinsten Learning-Rate im Vergleich genau das gegenteilige Problem wie LR $10^{-3.5}$. Die Änderungen der Parameter fallen in diesem Fall besonders gering aus. Deshalb dauert es länger bis die Genauigkeit dieses Systems das Niveau der anderen erreicht. Durch die kleinen Parameterupdates ist dieser Graph besonders glatt. Die Learning-Rates von $10^{-3.5}$ und $10^{-5.5}$ verhalten sich im Vergleich zu den restlichen am schwächsten.

Die Graphen von LR $10^{-4.0}$ und LR $10^{-4.5}$ steigen steil an und nähern sich dann langsam ihren Maxima von 96.5 % an. Die LR$10^{-5.0}$ verläuft leicht unterhalb dieser beiden. Allerdings auch etwas glatter. Auch dies ist in der geringeren Änderung der Netzwerkpa-

rameter begründet. Erst nach 100 Epochen erreicht die Kurve das Niveau von LR $10^{-4.0}$ und LR $10^{-4.5}$.

Die für dieses System optimale Learning-Rate befindet sich zwischen $10^{-4.0}$ und $10^{-4.5}$ und für Training mit mehr als 100 Epochen kann auch eine kleinere Learning-Rate zwischen $10^{-4.5}$ und $10^{-5.0}$ gewählt werden.

Außerdem ist erkennbar dass obwohl unterschiedliche Learning-Rates verwendet wurden, die Trainingsalgorithmen trotzdem Steigerungen der Genauigkeit ermöglichen. Das ist im adaptiven Moment des Adam-Algorithmen begründet. Dieses hat genau wie die Learning-Rate einen direkten Einfluss auf die Änderungen der Netzwerk Parameter. Im Verlauf des Trainings wird das adaptive Moment allerdings kleiner und damit auch die Änderungen am Modell. So kann das System immer näher an den optimalen Punkt bewegt werden. [KB]

4.3 Transfer Learning

4.3.1 Loose Layer

Transfer Learning heißt allgemein, Erkenntnisse die beim Lösen eines Problems entstanden sind auf ein neues Problem anzuwenden. Bei dem Maschinen Lernen spricht man von Transfer Lernen wenn man ein, auf einem komplett anderem Datensatz vortrainiertes, Modell auf einen neuen Datensatz trainiert. Im Bereich der Bildklassifikation hat das neuronale Netzwerk bereits die Fähigkeit erlangt, Kanten, Farben und Formen zu erkennen. Diese Fähigkeiten sollen nun für die Klassifikation anderer Daten genutzt werden.

Um herauszufinden ob das funktioniert soll zunächst ein vortrainiertes Modell mit einem nicht trainierten Modell verglichen werden. Das nicht trainierte Netz wird mit Zufallszahlen in den Gewichten und Nullen in den Bias Werten initialisiert. Deshalb wird das nicht trainierte Modell von zwei separat initialisierten Netzen repräsentiert um den Zufallsfaktor geringer zu halten. Auch das pretrained Modell wird durch zwei separat initialisierte Netze repräsentiert. Da durch die Modifikation im letzten Layer auch Zufallswerte vorkommen.

Bei dem pretraind Modell handelt es sich um ein mit dem ImageNet-Datensatz vortrainiertes AlexNet. Es werden alle Ebenen trainiert.

Graph	Pretraind 0	Pretraind 1	Random 0	Random 1
Modell	AlexNet	AlexNet	AlexNet	AlexNet
Trainierte Layer	alle	alle	alle	alle
pretrained	ja	ja	nein	nein
Datensatz	combiniert	combiniert	combiniert	combiniert
Augmentation	ja	ja	ja	ja
Lernalgorithmus	Adam	Adam	Adam	Adam
Learningrate	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$
Lossfunktion	MSELoss	MSELoss	MSELoss	MSELoss

Abbildung 4.7: Einstellungen beim Training mit und ohne vortrainierten Parametern

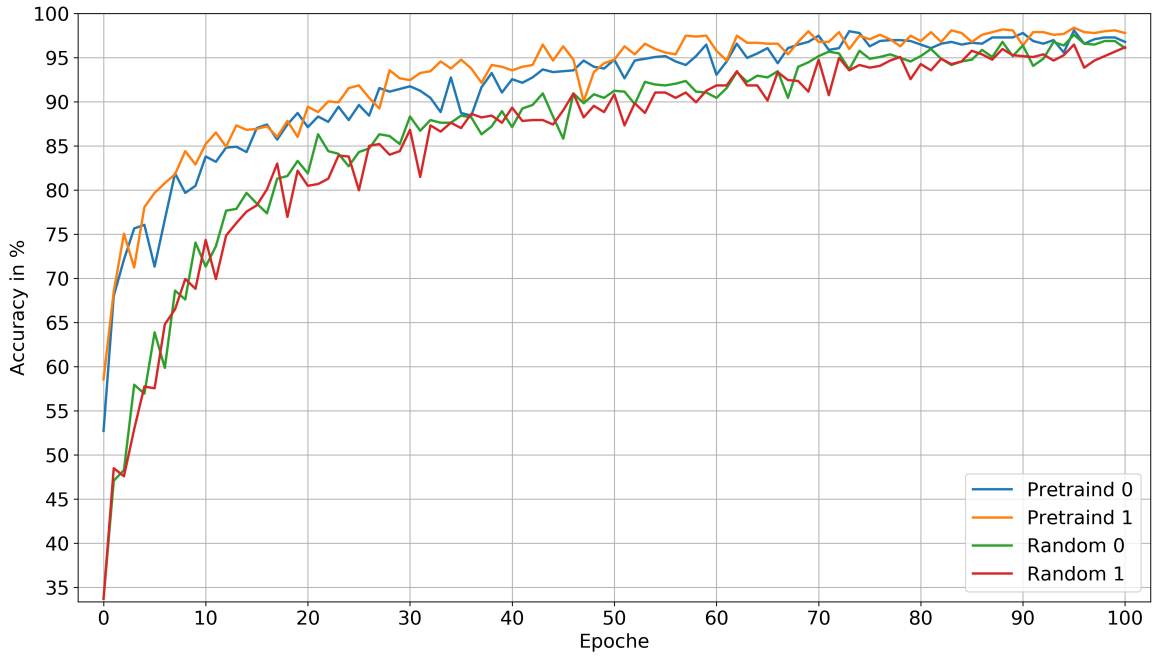

Abbildung 4.8: Verläufe der Genauigkeit mit und ohne vortrainierten Parametern

In Abbildung 4.8 zeigt sich, dass die vortrainierten Modelle sich deutlich schneller an das neue Problem anpassen können. Sie erreichen nach etwa 60 Epochen ihre Maxima. Danach stagnieren sie auf diesem Level. Bei den untrainierten Modellen dauert es 100 Epochen bis die Erkennungsgenauigkeit der vortrainierten Netze erreicht wird.

Obwohl die vortrainierten Modelle auf komplett anderen Daten trainiert wurden, sind sie besser auf neue Datensätze eingestellt. Durch die zufällige Initialisierung müssen viele Parameter stark angepasst werden, während die vortrainierten Gewichte bereits näher an der optimalen Position liegen. Durch die Verwendung von vortrainierten Netzen kann beim Training Zeit gespart werden.

4.3.2 Frozen Layer

Nachdem festgestellt wurde, dass die vortrainierten Netze sich schneller an einen anderen Datensatz anpassen können, soll festgestellt werden wie gut das funktioniert wenn nicht alle Ebenen trainiert werden. So sollen bei mehreren Netzen einige der Parameter in dem vortrainierten Zustand eingefroren werden und nur ein Teil der Layer Anpassungen durch das Lernen erfahren. Wahlweise werden die ersten drei, vier oder fünf Convolutional-Layer festgehalten (siehe Abbildung 4.10). Zum Vergleich wird zudem der Graph, bei dessen Modell alle Ebenen trainiert wurden, dargestellt.

Graph	All Layers loose	3 Conv frozen	4 Conv frozen	5 Conv frozen
Modell	AlexNet	AlexNet	AlexNet	AlexNet
Frozen Layer	keine	ersten 3 Conv	ersten 4 Conv	ersten 5 Conv
pretrained	ja	ja	ja	ja
Datensatz	combiniert	combiniert	combiniert	combiniert
Augmentation	ja	ja	ja	ja
Lernalgorithmus	Adam	Adam	Adam	Adam
Learningrate	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$
Lossfunktion	MSELoss	MSELoss	MSELoss	MSELoss

Abbildung 4.9: Einstellungen beim Training mit Frozen Layern

Schicht	Typ	All Layers loose	3 Conv frozen	4 Conv frozen	5 Conv frozen
-	Input				
Features					
1	Convolution	loose	frozen	frozen	frozen
2	MaxPool	-	-	-	-
3	Convolution	loose	frozen	frozen	frozen
4	MaxPool	-	-	-	-
5	Convolution	loose	frozen	frozen	frozen
6	Convolution	loose	loose	frozen	frozen
7	Convolution	loose	loose	loose	frozen
8	MaxPool	-	-	-	-
Classifier					
9	DropOut	-	-	-	-
10	Linear	loose	loose	loose	loose
11	DropOut	-	-	-	-
12	Linear	loose	loose	loose	loose
13	Linear	loose	loose	loose	loose

Abbildung 4.10: Trainierte und festgehaltene Ebenen des AlexNet

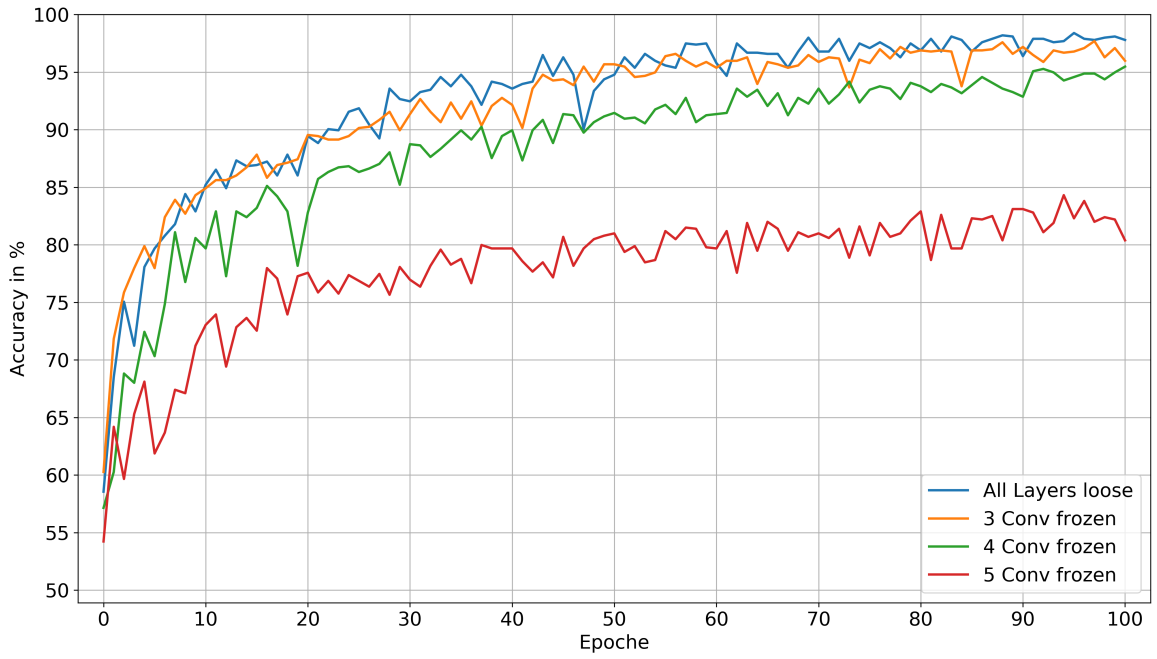

Abbildung 4.11: Verläufe der Genauigkeit mit Frozen Layern

Abbildung 4.11 zeigt die Genauigkeit bei der Erkennung der Testdaten bei dem Training verschiedener Netze mit den Trainingsdaten. Die Verläufe der unterschiedlichen Graphen zeigen, je mehr Ebenen des Modells trainiert werden, umso besser können diese sich an das Problem anpassen und umso besser wird die Erkennungsgenauigkeit.

Werden alle fünf Convolutional-Layer festgehalten, erreicht das System eine Genauigkeit von 84 %. Werden ein oder zwei Convolutional-Layer mehr trainiert, steigt die maximale Genauigkeit auf 95 % oder sogar 98 % an. So kann mit den ersten drei Convolutional-Layern mit vortrainierten Parametern eine mit einem komplett trainierten Modell vergleichbare Genauigkeit erreicht werden.

Die Verwendung von pretrained Netzen mit eingefrorenen Layern kann zudem Overfitting vermeiden. Diese Ebenen müssen sich dazu vorallem am Eingang befinden. So können die Eingangsfilter nicht Pixel für Pixel der Trainingsdaten auswendig lernen.

4.3.3 Auswertung mit Confusion-Matrix

Die Confusion-Matrix (Wahrheitsmatrix) ist ein Hilfsmittel mit dem abgelesen werden kann wie die Klassen bei der Klassifikation zugeordnet wurden. Das kann besonders praktisch sein um festzustellen ob beispielsweise ein kaputter Keks als gut eingeordnet wurde.

Die Matrix zeigt auf der linken Seite die tatsächlichen, festgelegten Klassen an. Auf der unteren Seite werden die vom Modell ausgegebenen Klassen dargestellt. Die Matrix ist dazu folgender Maßen zu lesen: Alle Testdaten einer Klasse sind in einer Zeile verteilt. In der Abbildung 4.12 sind zum Beispiel 96 Kekse der Klasse $milka\_fraglich$ $(79 + 9 + 8 = 96)$ vorhanden. Die Spalten geben dazu die Klassifizierung an. 79 Kekse der Klasse $milka\_fraglich$ wurden als solche richtig erkannt. Neun Kekse der Klasse $milka\_fraglich$ wurden allerdings als $milka\_gut$ und acht wurden als $milka\_schlecht$ klassifiziert.

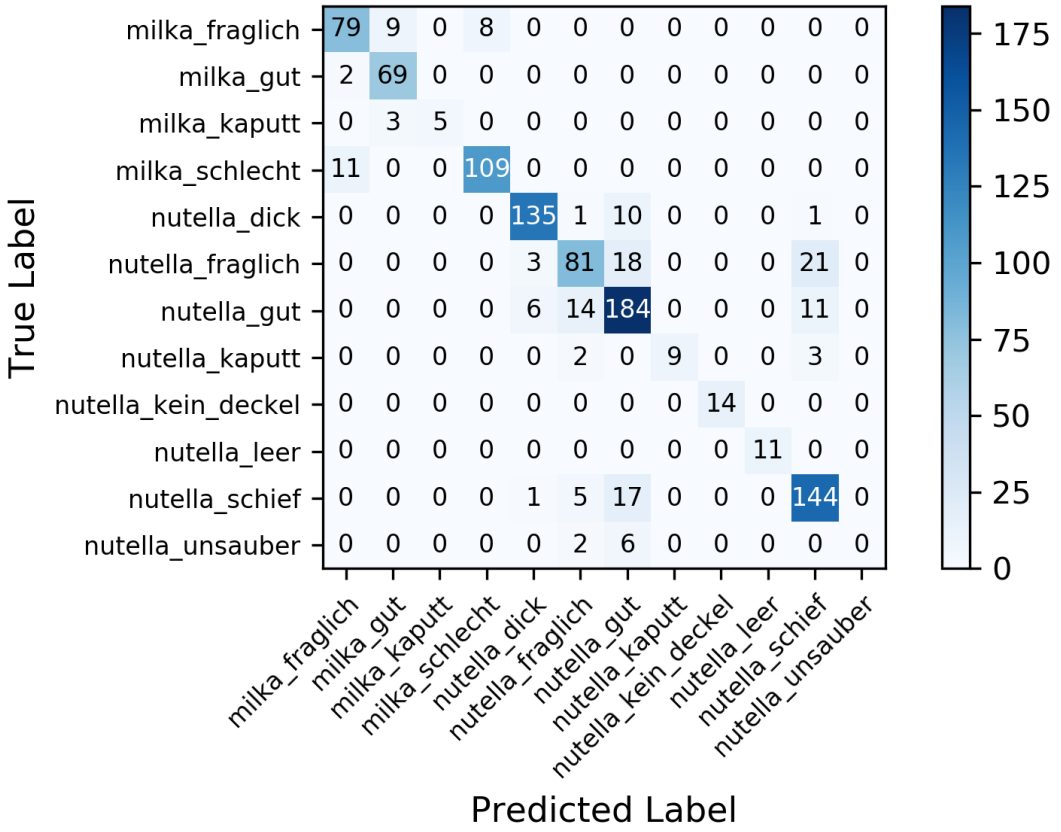

Abbildung 4.12: Confusion Matrix mit 84.5 % Genauigkeit

Die Abbildung 4.12 zeigt die Confusion Matrix des Modells mit fünf Frozen Layern und 84.5 % Genauigkeit aus dem Kapitel Transfer Learning 4.3.2. Sie zeigt, dass die beiden Sorten von Nutella und Milka zu 100 % richtig erkannt wurden. Allerdings wurden auch von acht Keksen der Klasse *milka_kaputt* drei Stück als *milka_gut* klassifiziert.

Zum Vergleich wird in Abbildung 4.13 die Confusion Matrix des Modells mit drei Frozen Layern und 98 % Genauigkeit aus dem Kapitel Transfer Learning 4.3.2 dargestellt. In diesem Fall werden die kaputten Kekse beider Klassen richtig klassifiziert.

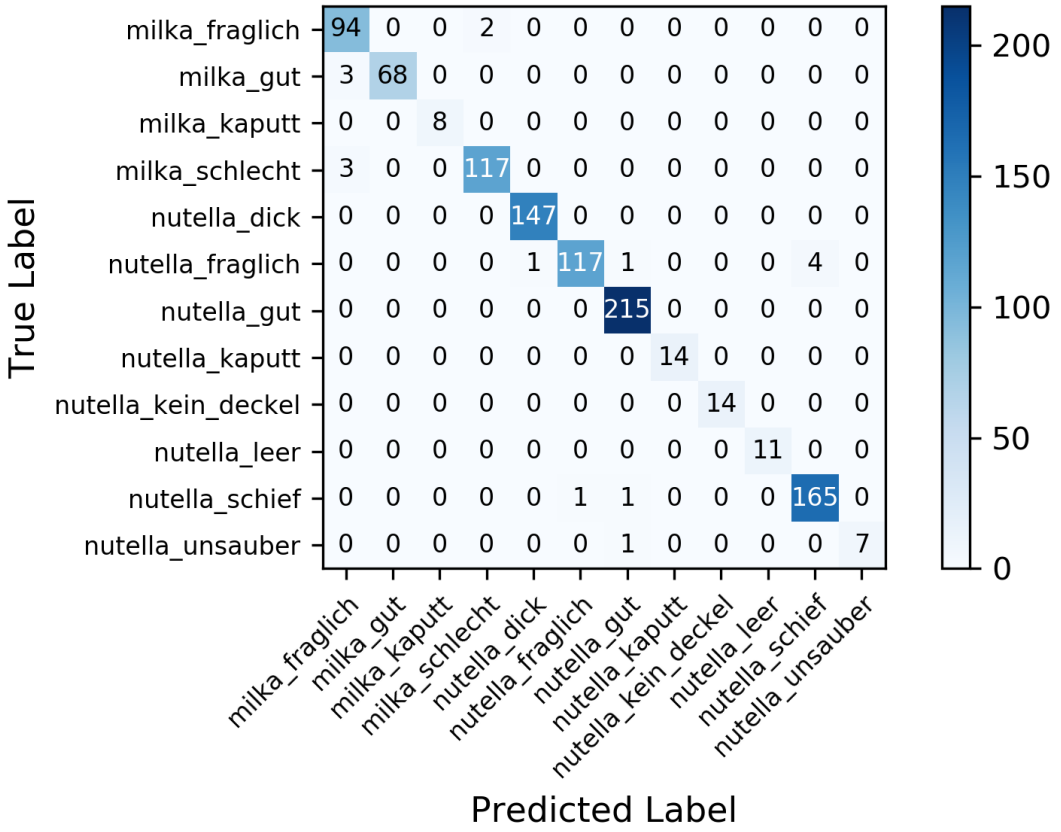

Abbildung 4.13: Confusion Matrix mit 98 % Genauigkeit

In Abbildung 4.14 werden alle Fehlerbilder des Modells mit 98 % Genauigkeit aus der Confusion Matrix 4.13 dargestellt. Dabei sind für jeden Keks die vom Modell vergebene Klasse sowie das festgelegte Klasse (Label) angegeben. Es fällt auf, dass einige vom CNN vergebenen Klassen durchaus nachvollziehbar sind (z.b. Kekse mit den Indices: 237, 250, 276, 477, 481, 515, 523). Andererseits wurden auch Klassen vergeben die, mit dem menschlichen Auge betrachtet, weniger Sinn ergebenen (z.b.: 66, 554). Dass der Datensatz nicht perfekt ist zeigt das Bild mit Index 89. Hier ist ein stark abgeschnittener Keks abgebildet. Die Anzahl solcher Daten liegt im gesamten Datensatz (Training- und Testdaten) im einstelligen Bereich.

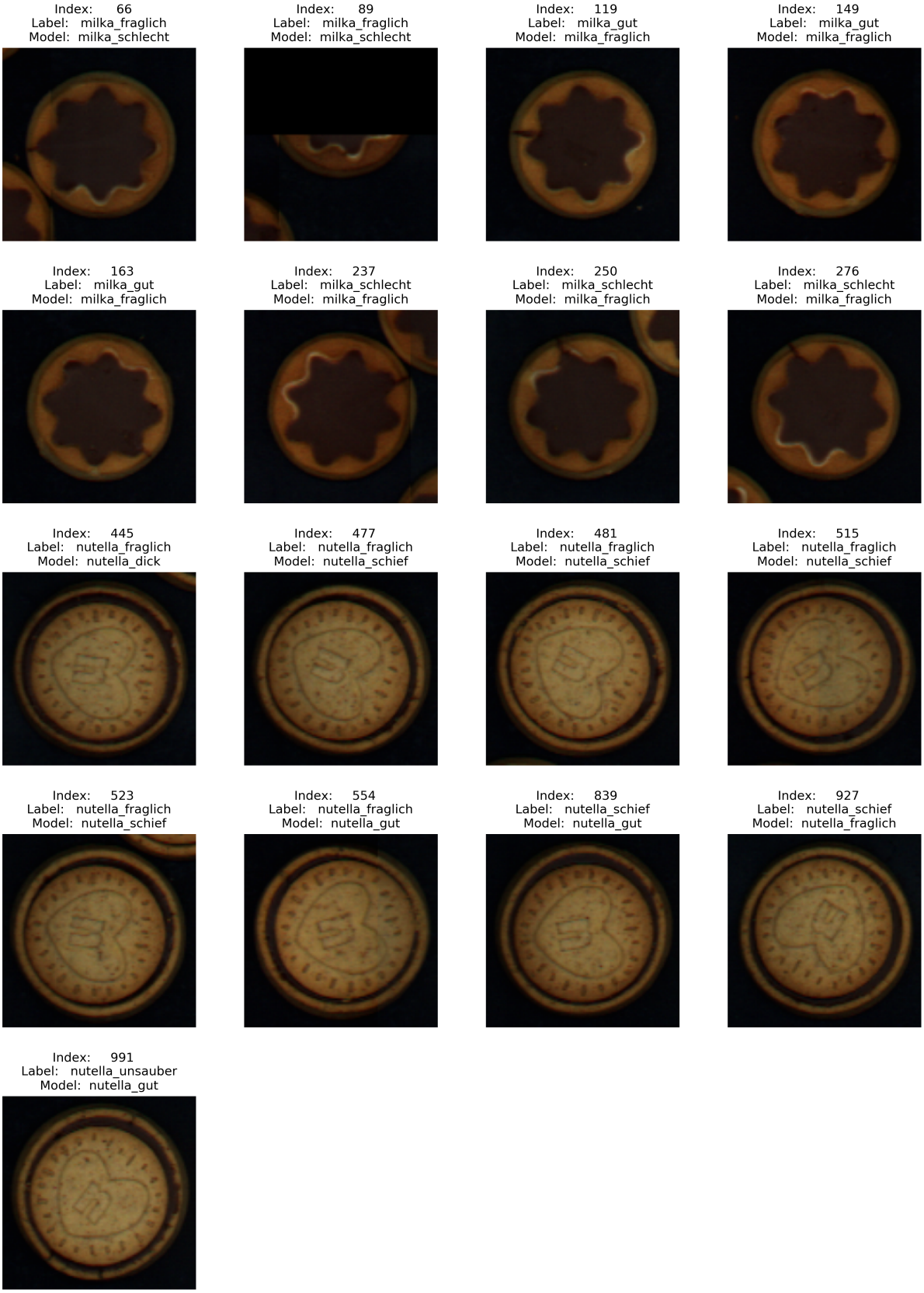

Abbildung 4.14: Alle falsch klassifizierten Bilder des Modells mit 98 % Genauigkeit

4.4 Größe des Datensatzes

Im Folgenden sollen die Auswirkungen eines kleineren Datensatzes untersucht werden. Dazu wird der kombinierte Datensatz aus Milka und Nutella Keksen verwendet. Als Testdaten werden immer die ersten 15 % jeder Klasse verwendet (994 Bilder). Die restlichen 85 % stehen als Trainingsdaten zur Verfügung. Der Trainingsdatensatz wird im Folgenden in unterschiedlichen Größen verwendet. Der kleinste Trainingsdatensatz erhält mit 982 Dateien etwa die Größe der Testdaten. Zusätzlich ergeben sich Trainingsdatensatzgrößen von 1642, 2957, 3616 und 5591 Daten. Die Aufteilung der gesamten Daten ist in Abbildung 4.15 dargestellt. Die genaue Verteilung der Trainingsdaten auf die Klassen zeigt die Tabelle 4.16.

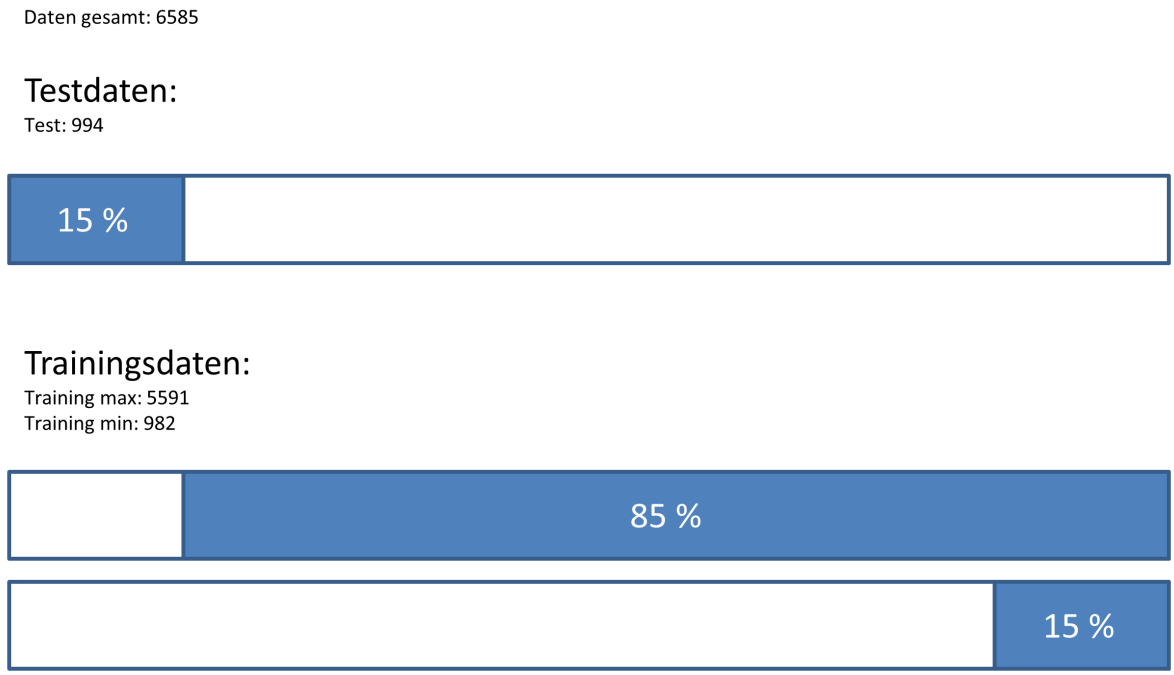

Abbildung 4.15: Sinnbildliche Aufteilung der Daten

	TD 982	TD 1642	TD 2957	TD 3616	TD 5591
m_fraglich	95	159	287	351	543
m_gut	70	117	211	258	399
m_kaputt	7	13	23	28	44
m_schlecht	119	198	357	437	675
n_dick	146	243	438	535	827
n_fraglich	122	204	367	448	693
n_gut	214	358	644	788	1218
n_kaputt	13	22	39	48	74
n_kein_deckel	13	22	39	48	74
n_leer	10	16	30	36	56
n_schief	166	277	499	611	944
n_unsauber	7	13	23	28	44
Summe	982	1642	2957	3616	5591

Abbildung 4.16: Verteilung der Daten bei unterschiedlich Trainingsdatensatzgrößen

Mit diesen veränderten Größen der Datensätze wurden verschiedene Modelle trainiert.

Graph	TD 982	TD 1642	TD 2957	TD 3616	TD 5591
Modell	AlexNet	AlexNet	AlexNet	AlexNet	AlexNet
Trainierte Layer	alle	alle	alle	alle	alle
pretrained	ja	ja	ja	ja	ja
Datensatz	combiniert	combiniert	combiniert	combiniert	combiniert
Größe	982	1642	2957	3616	5591
Augmentation	ja	ja	ja	ja	ja
Lernalgorithmus	Adam	Adam	Adam	Adam	Adam
Learningrate	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$
Lossfunktion	MSELoss	MSELoss	MSELoss	MSELoss	MSELoss

Abbildung 4.17: Einstellungen beim Training mit unterschiedlich Großen Datensätzen

Die Auswertung der Genauigkeit in Abhängigkeit der Epochen ist in diesem Fall unpraktisch. Während bei dem großen Datensatz eine Epoche aus 5591 erlernten Daten besteht, erlernt ein Modell mit dem kleinen Datensatz pro Epoche nur 982 Bilder. Deshalb ist der Vergleich nach Epochen nicht ausgeglichen.

Um einen Vergleich nach der Anzahl gelernter Daten zu schaffen müssen die verschiedenen Modelle unterschiedlich viele Epochen lernen. Das Modell mit 5591 Bildern lernt in

125 Epochen 698875 Bilder. Damit kann die Anzahl der Epochen anderer Datensätze so berechnet werden, dass alle Modelle gleich viele Daten lernen. Das Netz dass mit dem Datensatz mit 982 Bildern trainiert, muss 712 Epochen lernen. Auf der x-Achse wird die Anzahl der gelernten Trainingsbilder aufgetragen.

Da die Genauigkeit mit den Testdaten am Ende einer Epoche bestimmt wird, ergeben sich unterschiedlich viele Punkte für die Graphen. Deshalb ist der Abstand der einzelnen Messpunkte im Diagramm bei den verschiedenen Verläufen unterschiedlich.

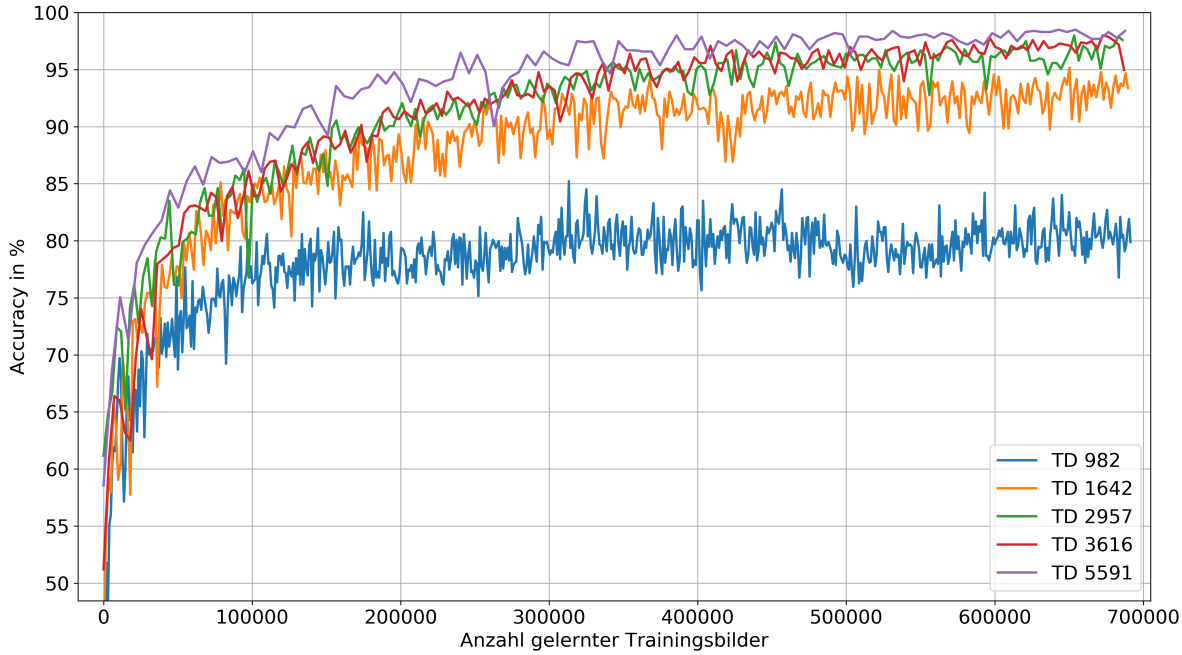

Abbildung 4.18: Verläufe der Genauigkeit mit unterschiedlich Datensatzgrößen

Die Abbildung 4.18 zeigt die Erkennungsgenauigkeit mit dem immer gleichen Testdatensatz für die mit unterschiedlich großen Trainingsdatensätzen trainierten Modelle über der Anzahl der trainierten Bilder an.

Generell lässt sich feststellen dass, je größer der Trainingsdatensatz ist, umso größer ist auch die maximal erreichbare Genauigkeit bei den Testdaten. Das Modell mit der geringsten Anzahl von Trainingsbildern erreicht ein Maximum von 85 %. Das mit 1642 Bildern trainierte System erreicht 95 %. Die Netze welche mit 2957 und 3616 Daten trainierten erreichen mit 97.5 % fast genau dieselbe Genauigkeit wie der komplette Datensatz von

5591 Bildern (98 %).

Zudem zeigt sich dass die Graphen der größeren Datensätze etwas schneller ansteigen als die der Kleinen. Das liegt daran dass, die kleinen Datensätze stärker von der Augmentation abhängig sind. Die größeren Datensätze haben immer eine große Auswahl an Daten die das Netz erlernt. Damit können schnell gute Ergebnisse erzielt werden. Die kleineren Datensätze besitzen nicht diese Auswahl und sind damit anfangs auch nicht so leistungsfähig.

4.5 Unbekannte Klassen

Im folgenden Abschnitt wird das Verhalten eines Modells bei der Klassifikation unbekannter Klassen analysiert. Dazu werden vor dem Training eines Modells die Klassen *milka_kaputt* und *nutella_kaputt* aus den Trainingsdaten entfernt. So das die Bilder dieser Klassen dem Netzwerk völlig unbekannt sind. Durch das Training erreicht das Modell mit den Testdaten der restlichen zehn Klassen eine Genauigkeit von 98 % (siehe Abbildung 4.19).

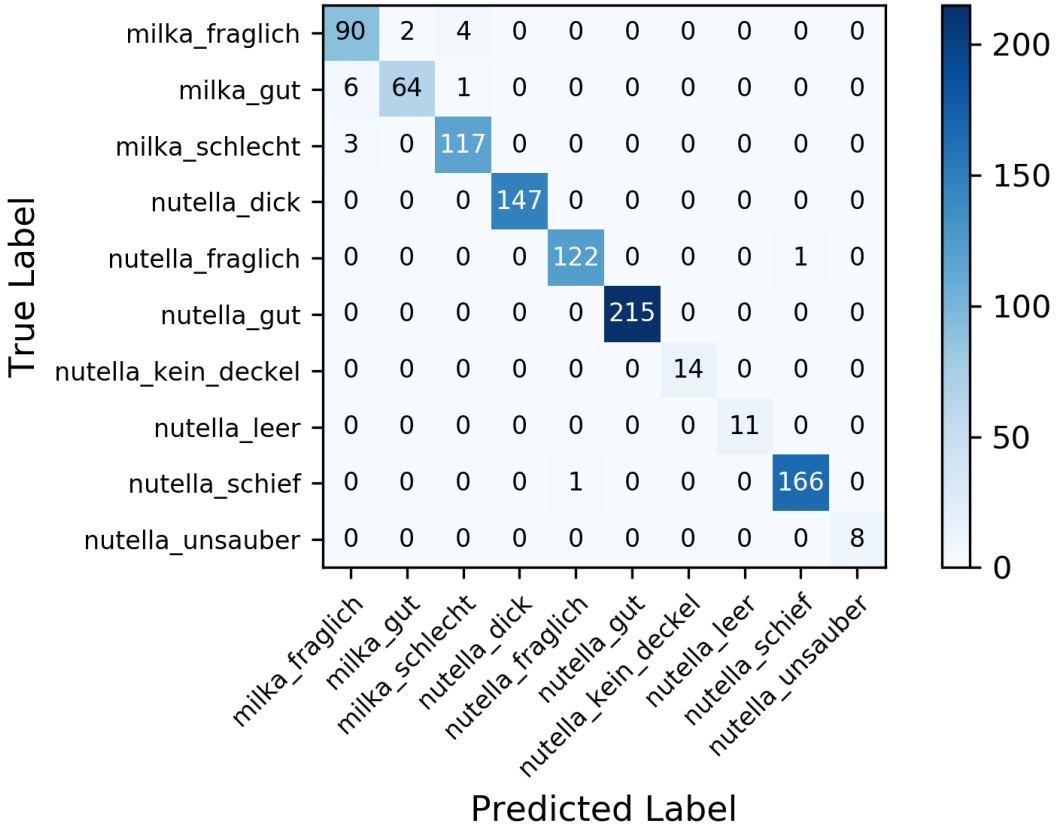

Abbildung 4.19: Confusion Matrix der 10 gelernten Klassen mit 98 % Genauigkeit

Das CNN macht mit Blick auf die Erkennungsgenauigkeit der Testdaten einen sehr guten Eindruck. Um herauszufinden wie dieses Modell auf unbekannte Klassen von Keksen reagiert, werden die Bilder der Klassen *milka_kaputt* und *nutella_kaputt* klassifiziert. Dieses Ergebnis ist in Abbildung 4.20 dargestellt.

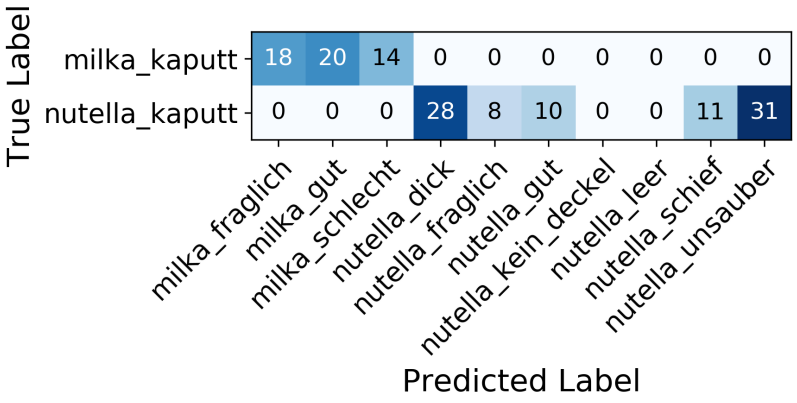

Abbildung 4.20: Confusion Matrix für Daten unbekannter Klassen

Es zeigt sich, das von 52 Keksbildern der dem Netzwerk unbekannten Klasse *milka_kaputt*, 20 als *milka_gut* klassifiziert wurden. Auch von den Keksen der Klasse *nutella_kaputt* wurden zehn Stück als *nutella_gut* erkannt.

Das heißt, weißt ein Keks einen Fehler auf, welcher dem Modell nicht bekannt ist, kann die Korrektheit der vom Netzwerk vergebenen Klasse nicht sichergestellt werden. Um dem Vorzubeugen gibt es verschiedene Möglichkeiten. Zum einen sollte das CNN möglichst viele Klassen und damit auch Fehlerfälle erlernen. Zum anderen kann eine zusätzliche Klasse *keine_Übereinstimmung* geschaffen werden. Wird bei der Klassifikation eines Bildes nicht eine festgelegte Wahrscheinlichkeit für die erkannte Klasse erreicht, wird es dieser Zusatz-Klasse zugeordnet.

Das Maximum im Output gibt die, laut CNN, wahrscheinlichste Klasse des Inputs an. Im Idealfall beträgt das Maximum eins. Ist der Wert des Maximums geringer, sinkt auch die Wahrscheinlichkeit, dass diese Klasse richtig erkannt wurde. In Abbildung 4.21 ist die Verteilung der Maxima des Outputs $\mathbf{y}$ für die Testdaten sowie für die unbekannten Daten (*milka_kaputt* und *nutella_kaputt*) abgebildet.

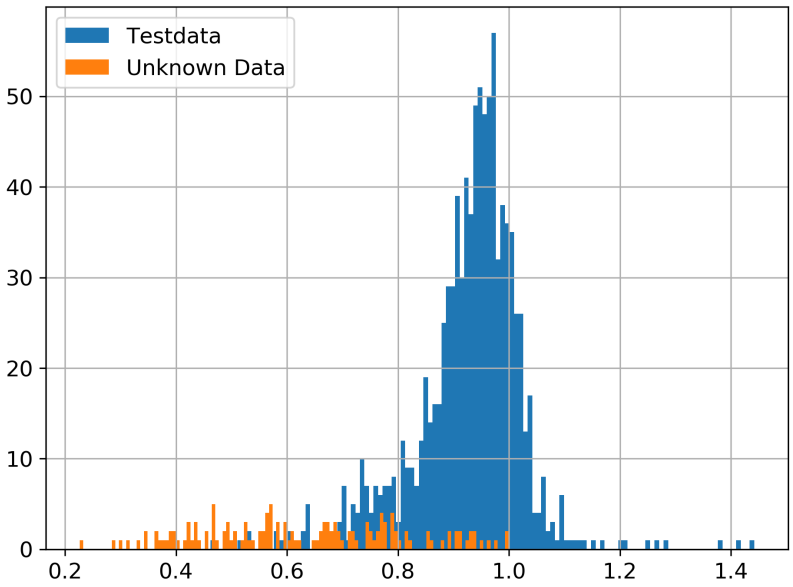

Abbildung 4.21: Verteilung der Maxima der Test- und der unbekannten Daten

Es ist zu erkennen dass, das Modell bei den bekannten Testdaten höhere Maxima berechnet als bei den unbekannten Daten. Das heißt, die Wahrscheinlichkeit das die Entscheidung des CNNs korrekt ist, ist bei unbekannten Daten nicht so groß wie bei den Testdaten. Bei den Testdaten wurden 972 Maxima und bei den unbekannten Daten 140 Maxima aufgetragen. Anhand des Histogramms wird ein Grenzwert von 0.6 festgelegt. Alle Bilder deren Maxima kleiner als der Grenzwert sind, werden als Klasse *keine_Übereinstimmung* klassifiziert. Dadurch ergeben sich neue Confusion-Matrizen (Abbildung 4.22 und Abbildung 4.23). Durch den Grenzwert und die neue Klasse können 48 % der Klassen *milka_kaputt* und *nutella_kaputt* ausgefiltert werden. Gleichzeitig werden wird dadurch auch 1.2 % der Testdaten als Klasse *keine_Übereinstimmung* klassifiziert. Das verschlechtert die Genauigkeit von 98 % auf 97.5 %.

Diese Ausführung soll nur als Beispiel dienen wie die Klassifikation erweitert werden kann. Soll eine Funktion dieser Art implementiert werden, bietet es sich an, dem AlexNet eine SoftMax-Aktivierungsfunktion am Ausgang hinzuzufügen. (In *Pytorch* besitzt das AlexNet in der ursprünglichen Form keine SoftMax-Funktion am Ausgang.) Diese Funktion berechnet aus dem Ausgangsvektor die Wahrscheinlichkeiten der einzelnen Klassen. So kann als Grenzwert direkt eine Wahrscheinlichkeit angegeben werden.

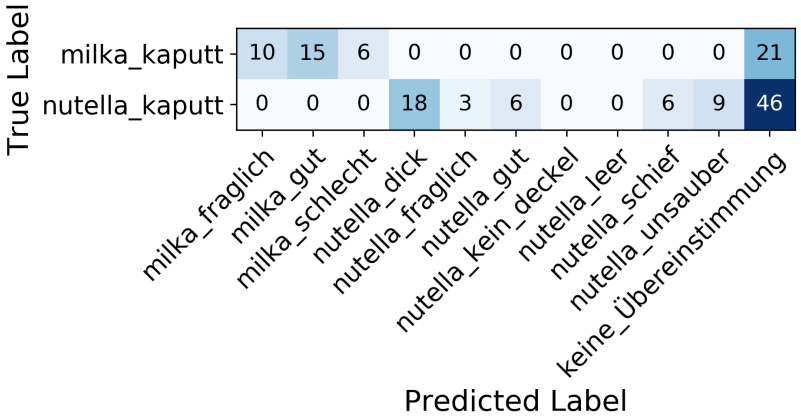

Abbildung 4.22: Confusion Matrix für Daten unbekannter Klassen nach Optimierung

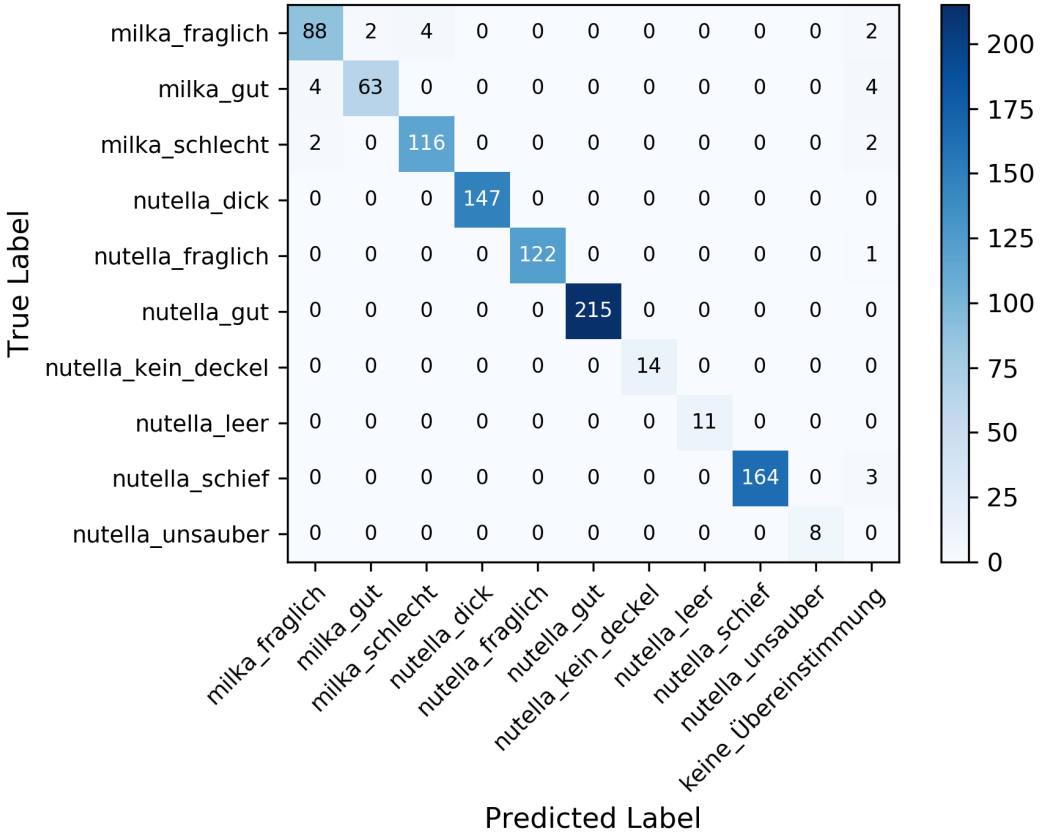

Abbildung 4.23: Confusion Matrix der 10 gelernten Klassen nach Optimierung

4.6 Dauer der Klassifikation

Da eine Farbzeilenkamera etwa 4000 Kekse pro Minute scannt und diese dann Klassifiziert werden sollen, wird in diesem Kapitel wird die Klassifikationsdauer der eines AlexNets genauer betrachtet. Dabei wird der im Kapitel 3.3.5 beschriebene Ablauf verwendet. Mit dem Python Module *time* wird die Dauer der einzelnen Schritte im Ablauf gemessen. Die Batch-Größe wird auf eins gesetzt. Der Testdatensatz wird komplett durchlaufen und die Berechnungszeiten der einzelnen Abschnitte werden summiert. Danach werden die gemessenen Zeiten für ein Bild bestimmt ($Zeit/AnzahlBilder$). Da auf den GPU-Server der Hochschule Augsburg mehrere Leute Zugriff besitzen, wurde zudem sichergestellt, dass zum Testzeitpunkt kein anderer Prozess eine Grafikkarte belegt. Zusätzlich wurden die Tests 10 mal wiederholt und für die einzelnen Zeiten die Mittelwerte bestimmt.

GPU

Zur Klassifikation wird eine Grafikkarte des Rechners Breakout im Rechenzentrum der Hochschule Augsburg verwendet. Dabei handelt es sich um eine Nvidia Geforce GTX 1080 mit 8 GB RAM. *Pytorch* kann auf diese GPU über die Softwareschnittstelle $CUDA$ zugreifen und damit können Berechnungen auslagert werden.

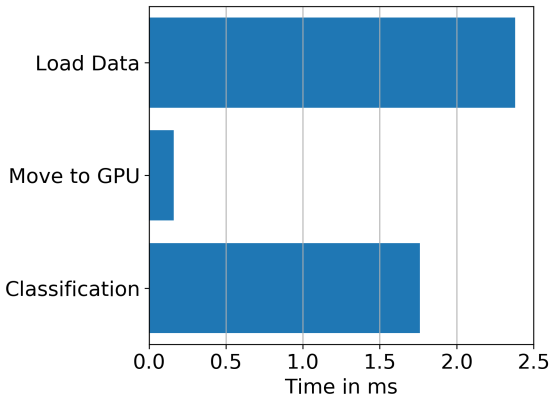

Abbildung 4.24: Klassifikationszeit eines Bildes auf einer GPU

Auffällig ist, dass das Laden der Daten einen großen Anteil am Ablauf hat. Um das zu verbessern wird im Dataset (Kapitel 3.3.1) nicht nur der Dateiname und Pfad in einer Liste gespeichert, sondern auch eine Liste mit den Bilddaten (Opjekte des Types $PIL.Image$) angelegt. So muss nicht jedes Bild einzeln aus dem entsprechenden

Verzeichnis geladen werden, sondern befindet sich bereits im Arbeitsspeicher und kann schneller verwendet werden. Dadurch wird die Gesamtdauer von 4.34 ms (Abbildung 4.24) auf 3.0 ms (Abbildung 4.25) verkürzt. Das ergibt 13 825 Bilder/min bzw. 20 000 Bilder/min.

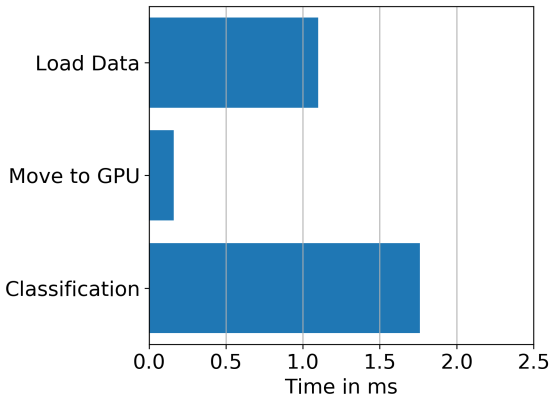

Abbildung 4.25: Klassifikationszeit eines Bildes auf einer GPU mit verbessertem Datenset

CPU

Ist keine *CUDA*-fähige Grafikkarte im System verbaut, müssen die Berechnungen auf der CPU durchgeführt werden. Die beiden Intel Xeon Broadwell E5-2690v4 des Hochschulrechners benötigen 13 ms pro Bild. Das sind 4615 Bilder/min.

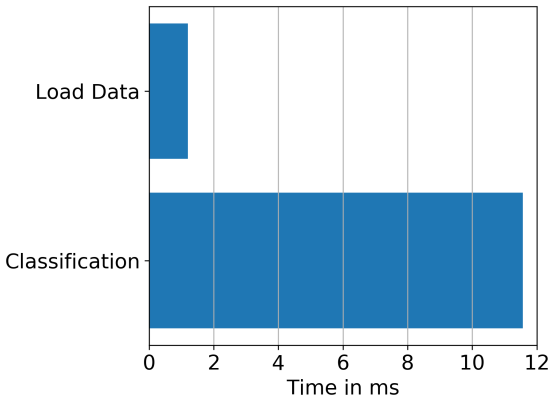

Abbildung 4.26: Klassifikationszeit eines Bildes auf einer CPU mit verbessertem Datenset

4.7 Ausblick

Ziel der vorliegenden Arbeit war es, künstliche neuronale Netzwerke (KNN) mit Hilfe von maschinellem Lernen zu trainieren. Die KNNs sollen Backwaren z.b. Kekse verschiedenen Klassen zuweisen. Der dafür aus zwei Kekssorten generierte Datensatz enthält 12 unterschiedliche Klassen, 5591 Trainingsdaten und 994 Testdaten. Damit wurde das AlexNet trainiert und getestet. Dabei ergeben sich Erkennungsgenauigkeiten bis zu 98.0 %.

Dabei spielen verschiedene Faktoren eine Rolle. Durch den Einsatz von Data-Augmentation während des Trainings wird die Genauigkeit von 93.5 % auf 98.0 % gesteigert. Die Größe des Trainingsdatensatzes beeinflusst ebenfalls die Ergebnisse. Grundsätzlich gilt je mehr verschiedene Bilder zum Lernen verwendet werden, desto größer wird die Leistungsfähigkeit des AlexNets. Ab einer Anzahl von 3000 Daten erreicht das KNN 97.5 % Erkennungsgenauigkeit für beide Kekssorten.

Mit der Wahl einer geeigneten Learning-Rate und der Verwendung von vortrainierten Netzwerken, wird die Trainingsdauer reduziert. Außerdem hat sich gezeigt, dass auch mit Hilfe von Transfer-Learning und nicht-trainierten Ebenen gute Leistungen erreicht werden. Trotzdem muss dem Anwender bewusst sein, dass Fehlerfälle (Klassen) welche nicht erlernt wurden jeder Klasse zugeordnet werden können.

Für die weitere Arbeit bietet es sich an den Cross-Entropy-Loss zu verwenden. Dabei handelt es sich um eine für Bildklassifikation typische Loss-Funktion. Der in dieser Arbeit genutzte Mean-Squared-Error dient mehr als Einstieg und findet in modernen CNNs kaum noch Anwendung. Ebenso besitzen viele Modelle eine SoftMax-Aktivierungsfunktion als Ausgangsebene, welche aus dem Output die zugehörigen Wahrscheinlichkeiten berechnet. Die Verwendung der SoftMax-Funktion empfiehlt sich auch für das AlexNet.

Für das Erlernen einer neuen Sorte in der Produktion kann folgendermaßen vorgegangen werden. Zuerst werden eine kleinere Menge von Keksen händisch unterschiedlichen Klassen zugeordnet und mit der Zeilenkamera digitalisiert. Damit wird das CNN zunächst trainiert und eingesetzt. Anhand der Menge an Daten die im Betrieb entstehen kann der Trainingsdatensatz vergrößert werden. Durch ein erneutes Training mit dem größeren Datensatz wird das Netzwerk robuster und genauer.

Abbildungsverzeichnis

Literaturverzeichnis

[ASH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc, 2012.

[Dev19] Deval Shah. Activation functions. `https://towardsdatascience.com/activation-functions-in-neural-networks-58115cda9c96`, Aufgerufen am 07.07.2019.

[Ert08] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Vieweg+Teubner Verlag / GWV Fachverlage GmbH Wiesbaden, Wiesbaden, 1. aufl. edition, 2008.

[Gra19] Grant Sanderson. But what is a neural network? | deep learning, chapter 1. `https://www.youtube.com/watch?v=aircAruvnKk&t=1s`, Aufgerufen am 07.07.2019.

[GSS14] Günther Görz, Josef Schneeberger, and Ute Schmid. *Handbuch der künstlichen Intelligenz*. Oldenbourg, München, 5., überarb. und aktualisierte aufl. edition, 2014.

[Hea13] Jeff Heaton. *Deep learning and neural networks*, volume / Jeff Heaton ; Vol. 3 of *Artificial intelligence for humans*. Heaton Research Inc, St. Louis, MO, February, 2013.

[Ima19] ImageNet. About imagenet. `http://www.image-net.org/about-overview`, Aufgerufen am 07.07.2019.

[Jus19] Justin Johnson. Convolutional neural networks. `http://cs231n.github.io/convolutional-networks/`, Aufgerufen am 07.07.2019.

[KB] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization.

[May19] Mayank Agarwal. Back propagation in convolutional neural networks. https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c3 Aufgerufen am 07.07.2019.

[Mic19] Michael Nielsen. Using neural nets to recognize handwritten digits. http://neuralnetworksanddeeplearning.com/chap1.html, Aufgerufen am 07.07.2019.

[Neu19] Neural Network Tool. http://alexlenail.me/NN-SVG/index.html, Aufgerufen am 07.07.2019.

[ot19a] opdi tex. Intelligente bildverarbeitungs-lösungen. https://www.opdi-tex.de/produkte/, Aufgerufen am 07.07.2019.

[ot19b] opdi tex. Lebensmittel-fertigung und -verpackung. https://www.opdi-tex.de/verwendungsgebiete/, Aufgerufen am 07.07.2019.

[PyT19] PyTorch. Dokumentation. https://pytorch.org/docs/stable/index.html, Aufgerufen am 07.07.2019.

[Raj19] Raj Bharath. Data augmentation. https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation Aufgerufen am 07.07.2019.

[Sou19] Soumith Chintala. Deep learning with pytorch: A 60 minute blitz. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html, Aufgerufen am 07.07.2019.

[Syl19] Sylvain Gugger. A simple neural net in numpy. https://sgugger.github.io/a-simple-neural-net-in-numpy.html#a-simple-neural-net-in-numpy, Aufgerufen am 07.07.2019.

[Yan19] Yann LeCun. The mnist database: The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, Aufgerufen am 07.07.2019.