HOME
HOCHSCHULE
MERSEBURG FH

University of
Applied Sciences

Masterarbeit

Bildklassifizierung mit Neuronalen Netzen

Zur Erlangung des akademischen Grades eines
Master of Engineering

Fachbereich: Ingenieur- und Naturwissenschaften
Referent: Prof. Dr. Michael Schenke
Korreferent: Prof. Dr. Karsten Hartmann

Eingereicht von:
David Ebert
Matrikelnummer: 22644
[redacted]

Vitzenburg, den 01.04.2019

Inhaltsverzeichnis

1. Einleitung

1.1 Abgrenzung

Auf den folgenden Seiten dieser Abschlussarbeit werden grundlegende Einblicke in die Entwicklung eines Neuronalen Netzes vermittelt. Dabei wird auf das besondere Umfeld sowie ausgewählte Eigenheiten eingegangen und ein Überblick über die verwendete Entwicklungsumgebung verschafft. Der Fokus der Arbeit liegt auf den entsprechenden Möglichkeiten unter Verwendung von Visual Studio Code, TensorFlow und Keras. Außerdem wird in dieser Abschlussarbeit untersucht, ob Unterschiede in der Größe der Bilder einen Einfluss auf die Genauigkeit und die Performance des zu entwickelnden Neuronalen Netzes aufweisen. Auch wird untersucht ob verschiedene Aktivierungsfunktionen und Optimizer die Genauigkeiten des Netzwerkes beeinflussen können.

1.2 Aufbau

Diese Arbeit besteht grundsätzlich aus einem theoretischen und einem Praktischen Teil, die jeweils übersichtlich in mehrere Abschnitte untergliedert sind. Zu Beginn dieser Arbeit wird der Begriff „Neuronales Netz“ definiert und es werden einige Anwendungsbeispiele genannt. Weiterhin wird in dem theoretischen Teil kurz auf Entstehung und Geschichte der Neuronalen Netzwerke eingegangen. Es wird an einem Beispiel, dem Perzeptron, die grundlegende Funktionsweise der Neuronalen Netze erklärt. Außerdem werden im Grundlagenteil der Arbeit die Grundlagen der Bildverarbeitung besprochen. Weiterhin wird in dem theoretischen Teil der Arbeit auf eine spezielle Form der Neuronalen Netze dem, Convolutional Neural Network, eingegangen. Auf diese Art von Netzwerken wird sich der größte Teil dieser Arbeit beziehen, auch wird mit diesem Netzwerktyp die Praktische Umsetzung erfolgen.
Der Praktische Teil der Arbeit wird in drei große Abschnitte unterteilt. Zunächst wird der zu verwendete Datensatz vorverarbeitet, um später das zu entwickelnde Convolutional Neural Network anlernen zu können. Hierbei werden die Bilder des Datensatzes in eine feste Größe konvertiert, da ein Neuronales Netz nur mit Bildern arbeiten kann, die dieselbe Größe aufweisen. Im Anschluss wird das Convolutional Neural Network entwickelt. Der letzte große Abschnitt des Praktischen Teils befasst sich mit der Optimierung des Convolutional Neural Network und der Manuellen Bildeingabe.
Am Ende dieser Arbeit wird ein Resümee gezogen und ein Ausblick gegeben.

2. Was ist ein Neuronales Netz

Ein neuronales Netz ist bis zu einem gewissen Grad dem Aufbau des biologischen Gehirns nachempfunden. Das neuronale Netz besteht aus einem abstrahierten Modell miteinander verbundener Neuronen, durch deren spezielle Anordnung und Verknüpfung sich Anwendungsprobleme aus verschiedenen Bereichen der Statistik, der Technik und der Wirtschaftswissenschaften computerbasierend lösen lassen. Ein neuronales Netz ist ein System, um Informationen zu verarbeiten. Wie das Gehirn eines Menschen muss das neuronale Netz seine Aufgaben erlernen können. Um Aufgaben erlernen zu können benötig das neuronale Netz Algorithmen. Für diese Algorithmen gelten die allgemeinen Regeln der Allgemeingültigkeit, Ausführbarkeit, Eindeutigkeit, Endlichkeit und der Terminiertheit. Die Neuronalen Netze fallen unter den Bereich der künstlichen Intelligenz. Wie Menschen können neuronale Netze aus Beispielen, Abstraktionen und Generalisierungen lernen. Der Vorteil eines neuronalen Netzes ist, dass ein neuronales Netzwerk durch Trainingsdaten selbstständig Lernen kann, ohne dies explizit zu programmieren. Ein neuronales Netzwerk besitzt ein hohes Maß an Parallelität bei der Verarbeitung von Informationen. Ein weiterer Vorteil ist, die hohe Fehlertoleranz und die verteilte Wissensrepräsentation, wodurch ein ausgefallenes Neuron nur einen kleinen Wissensausfall bedeutet.

2.1 Die Vernetzung von Neuronen im biologischen Gehirn

Das Nervensystem von Menschen und Tieren besteht aus Nervenzellen (Neuronen). Diese Neuronen sind mittels Synapsen miteinander verknüpft. Die Synapsen können als Verknüpfungsstellen oder Knoten eines interneuronalen Netzwerkes aufgefasst werden. Zwischen den Neuronen und Zellen der Neuroglia und Astroglia findet ein Austausch in chemischer und elektrischer Form statt, der die Gewichtung von Signalen verändern kann. Ein Neuron besitzt in der Regel mehrere Eingänge und einen Ausgang. Wenn die Summe der Eingangssignale einen bestimmten Schwellenwert überschreitet dann feuert das Neuron. Diese Aktionspotentiale sind in Serie die primären Ausgangssignale von Neuronen. Über Synapsen werden diese Signale zu anderen Zellen übertragen. An elektrischen Synapsen werden die Potenzialänderungen in unmittelbaren Kontakt weitergegeben. Die Weiterleitung an chemische Synapsen erfolgt als sekundäres Signal über Botenstoffe. Prägend für Nervenzellen sind ihre Zellfortsätze, mit denen die Nervenzelle Kontakte zu einzelnen anderen

Zellen herstellen kann. Diese Zellfortsätze (Dendriten) dienen vorrangig der Aufnahme von Signalen anderer Zellen. Während die Übertragung der Signale über das Axon geschieht. Ein Axon besteht aus mehreren Abzweigungen, Seiten- oder Nebenästen. Somit kann das Signal eins Neurons an mehrere Neuronen übermittelt werden. Dies geschieht einerseits efferent. Efferent werden die Fortsätze einer Nervenzelle genannt, über die aus einem Bestimmten Bereich Signale fort und an andere Zellen weitergeleitet werden. Andererseits können einem Neuron Signale afferent von verschiedenen anderen Neuronen zufließen. Afferent werden die Fortsätze von Nervenzellen genannt, über die einem bestimmten Bereich zufließen.

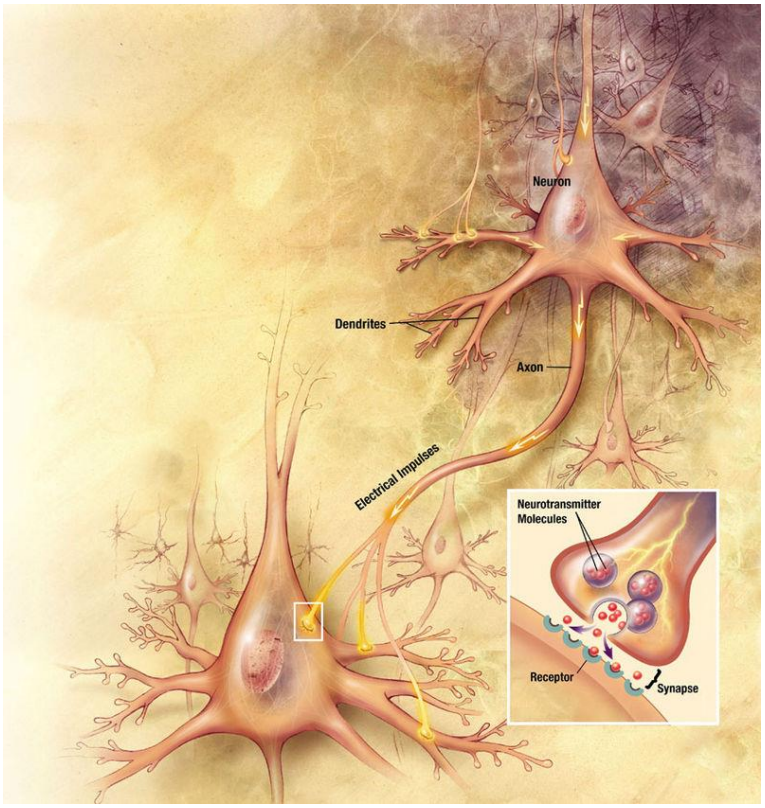

Abbildung 1: Vernetzung von Neuronen im biologischen Gehirn
https://de.wikipedia.org/wiki/Neuronales_Netz#/media/File:Neurons_big1.jpg

2.2 Die Vernetzung von Neuronen Im künstlichen neuronalen Netz

Künstliche Neuronale Netze sind der Versuch, die in der Gehirnforschung gewonnen Erkenntnisse über das Zusammenspiel von Neuronen und Synapsen zu modellieren. Das künstliche Neuronale Netz besteht aus 3 Schichten. Die erste Schicht dient dazu die Rohinformationen in das Netz zu speisen. Diese Eingabeneuronen sind wie die Sinnesorgane bei Menschen und Tieren wie die Haut, die Retina oder das Ohr zu verstehen. Zwischen den Eingangsneuronen und den Ausgangsneuronen liegen eine bis zig Schichten mit verknüpften Neuronen. Je komplexer die Aufgabenstellung ist je mehr verknüpfte Neuronenschichten

liegen dazwischen. Diese Zwischenschichten nennt man verdeckte Schichten. Die verdeckten Schichten lernen anhand von Tausenden bis Millionen von Beispielen. Aus diesen Rohinformationen werden zunächst simple Muster und Strukturen extrahiert, um aus diesen immer komplexer werdende Merkmale zu formen, mit deren Hilfe immer komplexere Aufgaben gelöst werden können.

Die letzte Schicht besteht aus den Ausgabeneuronen, welche sämtliche möglichen Ergebnisse repräsentieren. Im biologischen Gehirn beeinflussen nicht alle Neuronen einander gleich stark, unter anderem kommt es auf die Nähe der Synapse zum Zellkörper (Soma) an. Die künstlichen Neuronalen Netze bilden dies nach. Jede Verbindung zwischen zwei Neuronen besitzt ein individuelles Gewicht. Je höher das Gewicht ist, desto größer ist der Reiz, den das eine Neuron auf ein anderes ausübt. Wie das Neuron auf den Reiz reagiert, kann man auf verschiedene mathematische Arten lösen. Einerseits mit einer binären Schwellenwertfunktion. Diese Funktion kennt dabei jedoch nur zwei Aktivitätslevel. Übersteigen die ankommenden Reize den individuellen Schwellenwert des Neurons, so gibt das Neuron den Wert Eins aus und es feuert. Bleiben die Summe an Reizen unter dem individuellen Schwellenwert, gibt das Neuron den Wert Null aus. Mit dieser Schwellenwertfunktion würde das künstliche Neuronale Netz lernen indem man die individuellen Schwellenwerte bestimmt. Der Nachteil einer solchen Funktion besteht darin, dass kleinste Änderungen an den Schwellenwerten das gesamte künstliche Neuronale Netz zum Kippen bringen können. Um diesem Sprunghaften Anstieg des Aktivierungslevels entgegenzuwirken, wählt man stetige Aktivierungsfunktionen.

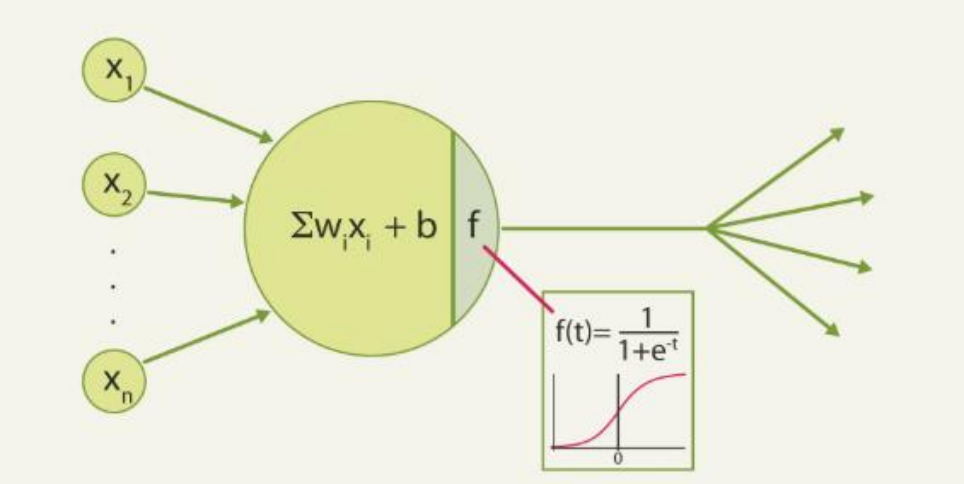

Abbildung 2: Neuronen und Synapsen im Künstlichen Neuronalen Netzwerk

2.3 Geschichte der Neuronalen Netzwerke

Die Anfänge der Neuronalen Netzwerke gehen auf das Jahr 1943 zurück. Warren McCulloch und Walter Pitts beschrieben in diesem Jahr die Vernetzung von elementaren Einheiten zu einem Netzwerk, womit praktisch alle logischen und arithmetischen Funktionen berechnen lassen könnten. Donald O. Hebb formulierte 1949 die Hebbsche Lernregel, die in ihrer allgemeinen Form die meisten der künstlichen Lernverfahren darstellt.

Die erste Blütezeit der Neuronalen Netzwerke begann 1951. In diesem Jahr gelang es Marvin Minsky mit seiner Dissertationsarbeit den Bau des Neuronencomputers Snarc. Dieser Computer war in der Lage seine Gewichte automatisch anzupassen, jedoch war Snarc praktisch nicht einsetzbar. Frank Rosenblatt und Charles Wightman entwickelten in den Jahren 1957 – 1958 den ersten Neurocomputer, mit dem Namen Mark I Perceptron. Mit diesem Neurocomputer konnten, mit Hilfe eines 20 x 20 Pixel Großen Sensors, einfache Ziffern erkannt werden. Im Jahr 1960 stellten Bernard Widrow und Marcian E. Hoff … ADALINE vor. Das war das erste Neuronale Netz, das eine weite kommerzielle Verbreitung fand. Zur Anwendung kam dieses Netz in der Analogtelefonie, um in Echtzeit Echofilterungen vorzunehmen. Dieses Neuronale Netz lernte mit der Deltaregel. 1969 veröffentlichten Marvin Minsky und Seymour Papert eine genaue mathematische Analyse des Perzeptrons … aufzeigte, dass wichtige Probleme mit dem Perzeptron nicht gelöst werden können. Mit einem Perzeptron ist es nicht möglich die XOR-Operation aufzulösen und weiterhin gab es Probleme mit der linearen Separierbarkeit. Daraufhin wurden die meisten Forschungsgelder für neuronale Netze gestrichen.

Erst im Jahr 1972 präsentierte Teuvo Kohonen den linearen Assoziator. Zwei Jahre später entwickelte Paul Werbos für seine Dissertation die Fehlerrückführung. Diese Fehlerrückführung bekam erst später eine sehr große Bedeutung. Ab dem Jahr 1976 widmenden sich Stephen Grossberg und Gail Carpenter dem Problem, ein neuronales Netz lernfähig zu halten, ohne dabei bereits erlerntes zu zerstören. Die Adaptive Resonanztheorie wurde von diesen beiden formuliert. 1982 beschreibt Teuvo Kohonen die nach ihm benannten selbstorganisierenden Karten. Diese selbstorganisierenden Karten sind eine Art von neuronalen Netzen. Sie sind als unüberwachtes Lernverfahren ein leistungsfähiges Werkzeug des Data-Mining. 1983 wurde ein weiteres Neuronales Modell Namens Neocognitron vorgestellt. Das Modell wurde von Kunihiko Fukushima, S. Miyake und T. Ito entworfen. Das

Neocognitron ist ein hierarchisches mehrschichtiges neuronales Netz. Es findet Anwendung bei der Erkennung handschriftlicher Zeichen und bei anderen Mustererkennungs-Aufgaben.
1985 wurde der Backpropagation of Error als Verallgemeinerung der Delta-Regel durch die Parallel-Distributed-Processing Gruppe separat entwickelt. Somit wurden nicht linear separierbare Probleme durch mehrschichtige Perzeptrons lösbar.
In der jüngsten Zeit befinden sich neuronale Netzwerke in einer Art Wiedergeburt, da sie bei herausfordernden Anwendungen bessere Ergebnisse als konkurrierende Lernverfahren liefern. Heute wieder richtig wichtig sind die tiefen vorwärtsgerichteten Netzwerke wie das Konvolutionsnetz von Kunihiko aus den 80ern. Das Team von Yann LeCuns von der New York University wandte auf das Konvolutionsnetz den 1989 schon gut bekannten Backpropagation-Algorithmus an. Schnelle GPU-Implementierungen dieser Kombination wurden 2011 durch Dan Ciresan und Kollegen in Schmidhubers Gruppe eingeführt. Durch diese Implementierungen gewannen sie zahlreiche Preise. Die GPU-basierenden Konvolutionsnetze waren auch die ersten künstlichen Mustererkenner die eine übermenschliche Performance aufwiesen.

2.4 Anwendungsgebiete von KI in denen Menschen heute schon schlechter sind

Die Convolutional Networks werden heute bereits in vielen Bereichen eingesetzt. In einigen Bereichen sind diese Netzwerke sogar in der Lage den Menschen zu übertreffen. Bereits 2011 gewann ein Convolutional Network des Schweizer Forschungsinstitutes für Künstliche Intelligenz den deutschen Wettbewerb für Verkehrszeichenerkennung. Das Netzwerk erkannte aus 50000 Verkehrszeichen 99,46% korrekt. Die Menschlichen Gegenspieler erkannten nur 98,84% der Verkehrszeichen korrekt. Somit war das künstliche Netzwerk mehr als doppelt so gut.
Ein weiterer Bereich der Convolutional Networks ist die Hausnummererkennung auf Millionen von Google-Street-View Bildern. Das Netzwerk von Google konnte diese Aufgabe innerhalb einer Stunde meistern.
Das Convolutional Network von der Firma Affectiva analysierte Menschen in 75 Ländern über Jahre hinweg, während diese Menschen Videos schauten. Die Künstliche Intelligenz lernte über diesen Zeitraum die Gefühlsregungen der Menschen wie Freude, Überraschung, Ekel und Traurigkeit zu deuten und richtig einzuordnen. Das Convolutional Network ist dabei

inzwischen präziser und schneller als die meisten Menschen. Weiterhin kann die Künstliche Intelligenz auch ein falsches von einem echten Lächeln unterscheiden.

Die US Forscher haben Neuronale Netze eingesetzt, um anhand von Gewebebildern die Überlebensrate von Menschen zu vorhersagen die an Krebs erkrankt waren. Die künstliche Intelligenz lernte verdächtige Merkmale zu identifizieren die Krebszellen von gesunden Zellen zu unterscheiden zu können. Die Künstliche Intelligenz entdeckte sogar mehr Merkmale, als die bis dahin in der medizinischen Literatur bekannt waren.

Außerdem schaffte die Künstliche Intelligenz von Geoffrey Hinton innerhalb von 2 Wochen unter tausenden von Molekülen diejenigen aufzuspüren, die sich für Medikamente einsetzten lassen.

Künstliche Intelligenzen lassen sich auch für die Wartung von Maschinen einsetzten, wobei diese eine Vielzahl an Daten von Sensoren von Windturbinen auswerten. Durch die Auswertung der Daten ist die Künstliche Intelligenz in der Lage ungewöhnliche Schwingungen oder einen unrunden Lauf festzustellen. Somit ist es möglich ein Serviceteam zu den betroffenen Anlagen zu schicken noch bevor eine Beschädigung kommt. Auch der Hochgeschwindigkeitszug Valaro aus Spanien nutzt eine Künstliche Intelligenz, um eine vorausschauende Wartung vorzunehmen und die Ausfallzeiten oder Verspätungen des Zuges zu gering wie möglich zu halten.

Die künstlichen Intelligenzen werden auch für Computerspiele eingesetzt. Im Jahr 2016 gewann die Künstliche Intelligenz AlphaGo ein Go Spiel gegen den weltbesten Go-Spieler. Die gleichen Entwickler stellten nur Monate zuvor eine Künstliche Intelligenz vor, die sich eigenständig 49 Atari Spiele beigebracht hat. Die Künstliche Intelligenz spielte die Spiele so lange und variierte für jedes Spiel die Strategie bis die Künstliche Intelligenz die zu erreichende Punktzahl maximieren konnte.

Die Abbildung zeigt wie sich die rasche Verbesserung der künstlichen Intelligenz in den letzten Jahren beschleunigt hat.

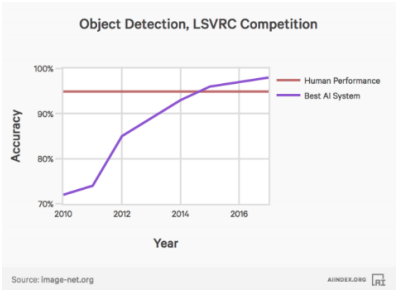

Abbildung 3: Performance von Künstlicher Intelligenz gegenüber dem Menschen
https://jaai.de/wp-content/uploads/2018/02/Bildschirmfoto-2018-02-05-um-18.12.59.png

3. Lernverfahren im künstlichen Neuronalen Netz

In diesem Kapitel der Abschlussarbeit geht es um die verschieden Lernverfahren in einem künstlichen Neuronalen Netzwerk. Die Lernverfahren dienen dazu, einem künstlichen Neuronalen Netzwerk bei zu bringen, für bestimmte Eingangsmuster dazugehörige Ausgangsmuster zu erzeugen. Dieses Verhalten geschieht grundsätzlich auf drei verschiedenen Wegen.

3.1 Überwachtes Lernen

Mit dem überwachten Lernen wird dem künstlichen Neuronalen Netz ein Eingangsmuster gegeben und die Ausgabe, die das künstliche Neuronale Netz in seinem aktuellen Trainingszustand produziert. Diese Ausgabe wird mit dem Wert verglichen, den das künstliche Neuronale Netz eigentlich ausgeben soll. Durch den Vergleich der Soll- und Ist-ausgabe können Änderungen an der Konfiguration des künstlichen Neuronalen Netzes sowie der Neuronen vorgenommen werden. Diese Methode richtet sich schon im Vorhinein an eine vorher festgelegte zu lernende Ausgabe. Die Ergebnisse dieses Lernverfahrens können mit den bekannten, richtigen Ergebnissen verglichen, also überwacht werden. Das Ziel dieses Trainings ist es dass, das neuronale Netz nach dem Training selbstständig Ein- und Ausgabe Muster assoziieren kann und bis dato ähnliche unbekannte Muster zu einem plausiblen Ergebnis zuführen kann.

Liegen die Ergebnisse des überwachten Lernens in einer kontinuierlichen Verteilung vor, deren Ergebnisse beliebig viele quantitative Werte in einem vorher festgelegtem Wertebereich sind, liegt ein Regressionsproblem vor. Bei einem Regressionsproblem geht es meist um Vorhersagen wie zum Beispiel die Preisentwicklung von Aktien auf Basis von bestimmten Variablen vorherzusagen.

Ein Klassifikationsproblem liegt vor, wenn die Ergebnisse in diskreter Form vorliegen. Ein Beispiel für ein Klassifikationsproblem ist die Unterscheidung ob es sich bei einer Email um Spam oder kein Spam handelt oder ob der Inhalt eines Bildes zu einer Klasse gehört oder nicht. Das in dieser Arbeit zu entwickelnde neuronalem Netz bezieht sich auf die überwachte Lernmethode.

Um ein Problem mit überwachtem Lernen zu lösen müssen die folgenden Schritte durchgeführt werden.

Als erstes muss die Art der Trainingsbeispiele bestimmt werden. Es muss bestimmt werden von welcher Art die Daten des Trainingsdatensatzes sind. Bei einer Textanalyse können die Daten zum Beispiel ein einzelnes Zeichen, ein komplettes Wort oder ein kompletter Satz sein. Der nächste Schritt in der Vorgehensweise ist eine genaue Datenerhebung des vorangegangenen Schrittes. Weiterhin müssen die zu erklärenden Variablen und die erklärten Variablen erhoben werden. Diese Datenerhebung kann von Menschlichen Experten, durch Messungen und andere Methoden erhoben werden.

Die Genauigkeit der gelernten Funktion hängt stark davon ab, wie die zu erklärenden Variablen dargestellt werden. Die zu erklärenden Variablen werden in einen ein Vektor transformiert, der die Merkmale enthält, um die zu erklärende Variable zu beschreiben. Die Anzahl von Merkmalen in dem Vektor sollte nicht zu groß sein, aber er sollte genügend Merkmale enthalten, um die Ausgabe genau vorhersagen zu können.

Daraufhin müssen der Lernalgorithmus und die Struktur der gelernten Funktion bestimmt werden. Sollte ein Regressionsproblem bestehen, sollte überprüft werden ob eine Funktion mit oder ohne Parameter besser geeignet ist.

Nun sollte der Lernalgorithmus auf den gesamten Trainingsdatensatz angewandt werden. Bei einigen Lernalgorithmen fordern diese vom Anwender bestimmte Regelparameter. Durch Optimierung einer Teilmenge des Datensatzes oder durch Kreuzvalidierung können diese Parameter spezifisch angepasst werden.

Als letzten Schritt muss die Genauigkeit der gelernten Funktion bestimmt werden. Durch die Erlernung der Parameter beziehungsweise die Parametrierung sollte die Funktion an einem Testdatensatz auf ihre Genauigkeit gemessen werden. Der Testdatensatz muss von dem Trainingsdatensatz strikt getrennt werden.

3.2 Unüberwachtes Lernen

Beim unüberwachten Lernen sind keine gekennzeichneten Daten vorhanden. Auf diese Daten kann kein Trainierbarere Algorithmus angewendet werden. Da keine Ergebnisse, auf die der Algorithmus kommen soll, vorliegen. Im unüberwachten Lernen werden Algorithmen verwendet, die die Struktur der Daten analysieren und daraus Sinnvolle Informationen aus den Daten abbilden. Das Unüberwachte Lernen dient dem Data-Mining. Mit unüberwachtem Lernen kann die automatische Segmentierung oder die Komprimierung von Daten zur Dimensionsreduktion erlernt werden.

3.2.1 Clusteranalyse

Bei der Clusteranalyse wird eine Menge von Daten in Klassen unterteilt. Mit der Clusteranalyse wird versucht, ohne die verschiedene Klassen zu kennen, die Daten so zu klassifizieren das daraus Cluster entstehen die ähnliche Eigenschaften aufweisen.

3.2.2 Komprimierung

Bei der Komprimierung wird versucht, viele Eingabewerte in einer kompakteren Form zu repräsentieren. Das Ziel dieser Komprimierung ist es, die Daten so zu komprimieren, dass so wenige Eigenschaften wie möglich verloren gehen. Ein Komprimierungsverfahren ist die Hauptkomponentenanalyse.

3.3 Bestärktes Lernen

Das bestärkte Lernen steht für eine Menge an Methoden, bei der ein Agent selbstständig eine Strategie zum Lösen eines Problems erlernt. Um dies zu gewährleisten erhält der Agent Belohnungen. Auch eine negative Belohnung ist möglich. Um eine Belohnung zu erhalten wird dem Agenten nicht vorgezeigt welche Aktion in welcher Situation die Beste ist. Durch dieses Belohnungssystems approximiert der Agent eine Nutzenfunktion, die beschreibt, welchen Wert ein bestimmter Zustand oder eine Aktion hat. Um die Strategien der Agenten zu Lernen gibt es verschiedene Lernverfahren. Die erfolgreichsten Lernverfahren sind die Monte-Carlo-Methoden und die Temporal Difference Lerning Methode. Bei diesen Verfahren handelt es sich um eine Reihe von Algorithmen, bei denen der Agent eine Nutzenfunktion besitzt. Durch diese Nutzenfunktion kann der Agent einen bestimmten Zustand oder eine bestimmte Aktion in einem Zustand bewerten.

4. Aktivierungsfunktionen

Die Aktivierungsfunktion stellt den Zusammenhang zwischen dem Netzeingang und dem Aktivierungslevel eines Neurons dar. Um die Aktivierungsfunktion grafisch darzustellen wird ein Zweidimensionales Diagramm verwendet. Die x-Achse zeigt in dem Diagramm den Netzeingang. Das Aktivierungslevel zeigt die y-Achse. Der Aktivierungslevel einer solchen Funktion wird durch eine Ausgabefunktion in die Ausgabe transformiert, den das Neuron an

das weitere Neuron der nächsten Schicht im Netzwerk weiterleitet. Für die Ausgabefunktion wird die Identitätsfunktion verwendet. Diese Funktion leitet immer den Ausgabewert der Funktion weiter. In der praktischen Umsetzung kommen die Sigmoid, die Tanh und die ReLu Aktivierungsfunktion zum Einsatz.

4.1 Binäre Schwellenwertfunktion

Überschreit die Eingabe einen vorher definierten Schwellenwert so liefert diese Funktion eine Eins als Ausgabe und das Neuron feuert. Ist das Ergebnis der Eingabe unter dem vorher festgelegten Schwellenwert, so gibt die Funktion eine Null weiter. Das Neuron feuert nicht.

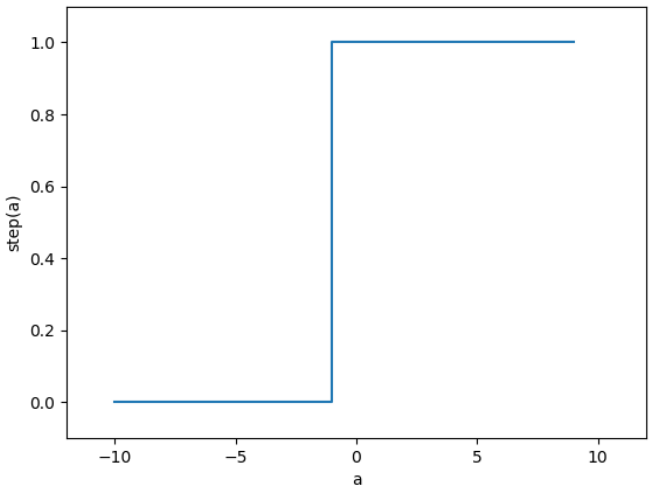

Abbildung 4: Diagramm Binäre Schwellenwertfunktion

$$f(x) = \begin{cases} 0: < 0 \\ 1: \geq 0 \end{cases}$$

Formel 1: Binäre Schwellenwertfunktion

4.2 Sigmoid Funktion

Mit der Sigmoid-Funktion gibt es keine sprunghafte Änderung des Aktivitätslevels, sondern immer eine von den gewichteten Summen der Eingangssignale abhängige Ausgabe. Diese Funktion simuliert die zwei Aktivitätslevel des diskreten Modells näherungsweise. In einem Bereich schwacher Eingangssignale erzeugt die Sigmoid-Funktion ebenfalls ein sehr niedriges, nah am Minimum liegendes Ausgangssignal. In einem Bereich starker Eingangssignale liefert die Funktion, ein nah am Maximum liegendes Ausgangssignal. In einem schmalen, mittleren Bereich der Eingangssignale steigt die Kurve relativ stark an. Der Bias bestimmt dabei wie stark

der Reiz sein muss, um das Neuron überhaupt anzuregen. Mit dem Bias steuert man die Empfindlichkeit des Neurons. Die Sigmoid Funktion hat den Nachteil das Sie weit weg von ihrem Schwellenwert kaum noch etwas erlernen kann.

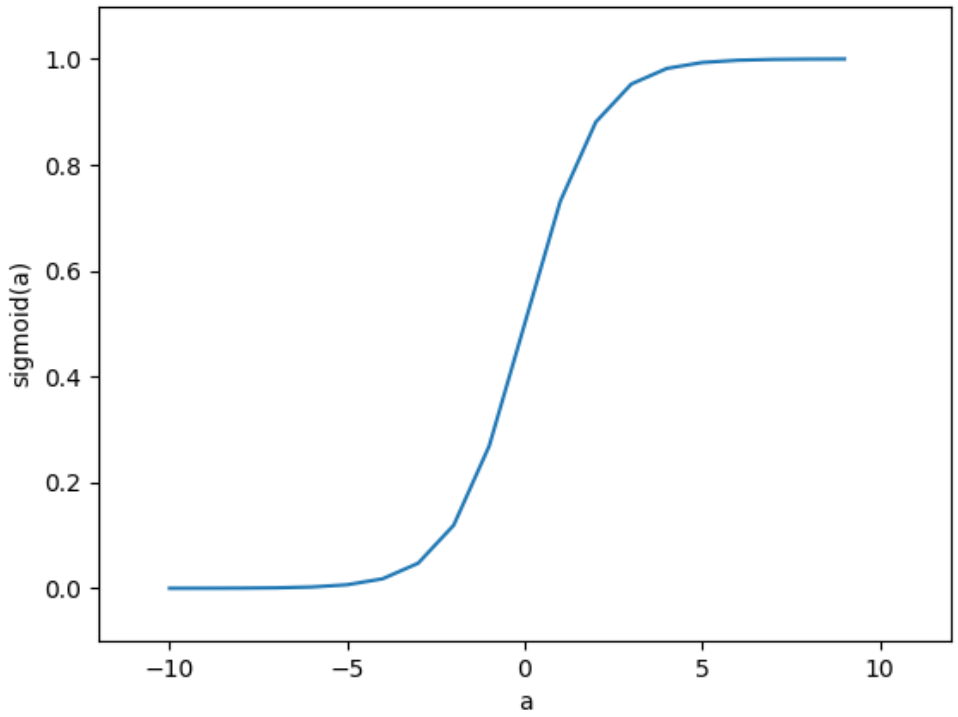

Abbildung 5:Sigmoid-Funktionsgraph

$$sig(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t} = \frac{1}{2} * (1 + tanh\frac{t}{2})$$

Formel 2: Sigmoid-Funktion

4.3 Tanh Funktion

In Neuronalen Netzen kann als Alternative zur Sigmoidfunktion die hyperbolische Tangensfunktion als Aktivierungsfunktion verwendet werden. Diese Funktion ist ähnlich der Sigmoid-Funktion. Auch bei dieser Funktion gibt es keinen sprunghaften Anstieg des Aktivitätslevels. Ein Nachteil dieser Funktion ist, dass die Tanh Funktion weit vor dem Schwellenwert wo das Ergebnis nahe Null ist, kaum Möglichkeiten hat etwas zu erlernen.

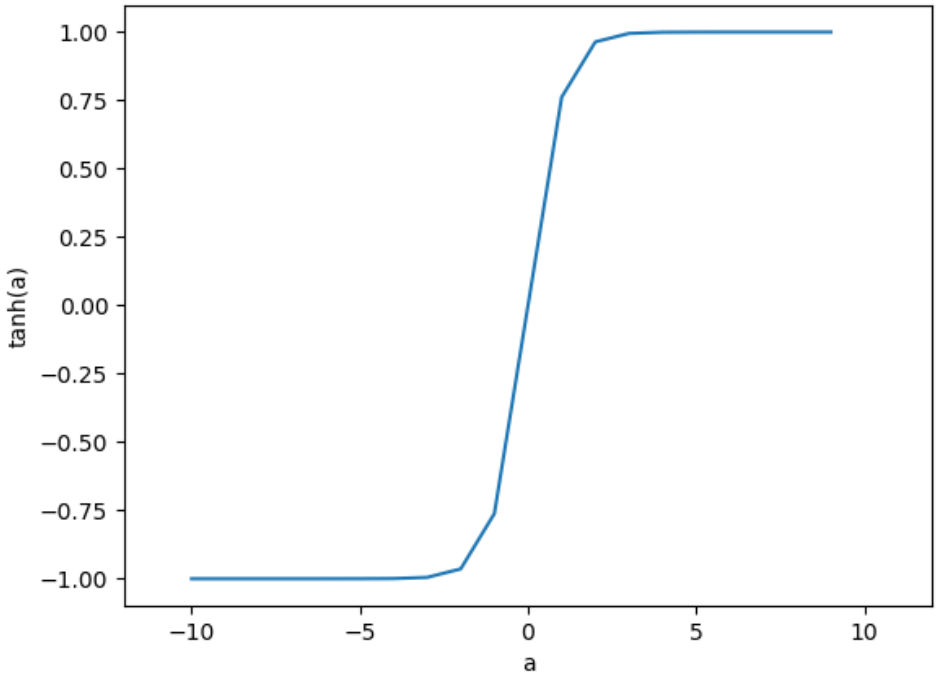

Abbildung 6: TanH-Funktionsgraph

$$f(x)\tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{+x})}$$

Formel 3: TanH-Funktion

4.4 ReLu Funktion

Die ReLu ist die am häufigsten verwendete Aktivierungsfunktion in Neuronalen Netzen mit Klassifikationsproblemen. Diese Funktion gibt eine Null zurück, wenn sie eine Negative Eingabe empfängt. Ist die Eingabe positiv so gibt die ReLu genau diesen Wert weiter. Ein Problem bei dieser Aktivierungsfunktion ist es, dass es keine Unterscheidung zwischen Negativen Werten und Null gibt. Ein weiteres Problem dieser Funktion ist, dass die Neuronen in Zustände versetzt werden können, in denen sie für alle Eingänge inaktiv werden. In so einem Zustand fließen keine Gradienten rückwärts durch das Neuron, somit bleibt das Neuron in einem ständigen Inaktiven Zustand stecken. Wenn von diesem Problem eine Vielzahl von Neuronen in diesem Zustand stecken, ist die Effektivität des Netzwerkes stark eingeschränkt. Diesem Zustand kann entgegengewirkt werden indem die Lernrate reduziert wird.

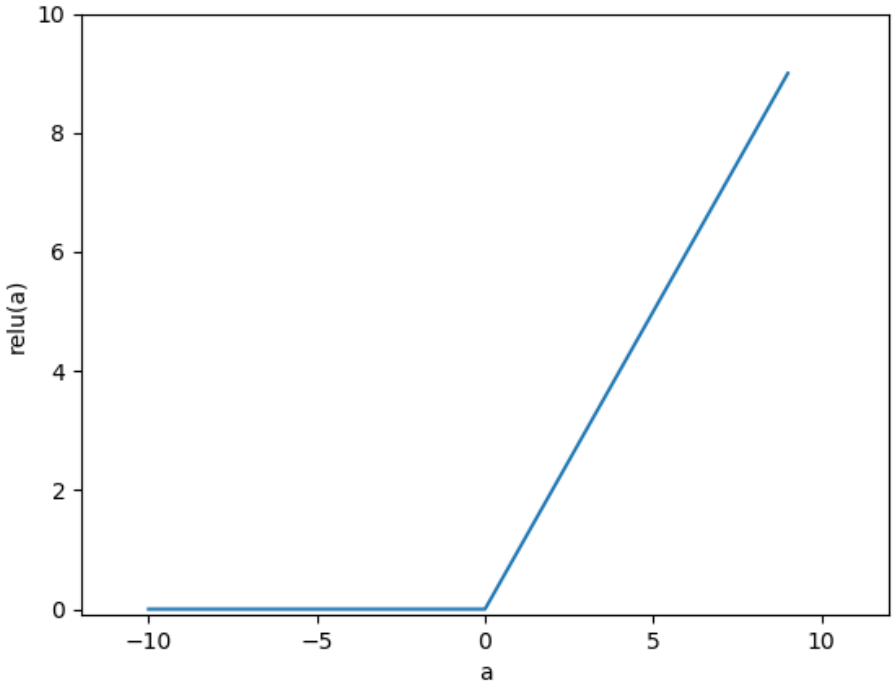

Abbildung 7: ReLu-Funktionsgraph

$$f(x) = \max(0, x)$$

Formel 4: ReLu-Funktion

5. Ein einfaches Neuronales Netzwerk das Perzeptron

Das Perzeptron ist ein vereinfachtes Neuronales Netzwerk. Die Grundversion des Perzeptrons besteht aus einem einzelnen Neuron mit anpassbaren Gewichten und einem Schwellenwert. Aus dieser Grundversion lassen sich verschiedene Kombinationen des ursprünglichen Modells ableiten, dabei wird zwischen einlagigen und mehrlagigen Perzeptren unterschieden.

5.1 Einlagiges Perzeptron

Das einlagige Perzeptron besteht aus einer Eingabe und einer Ausgabeschicht. In dieser Art von Perzeptren gibt es keine Versteckten Neuronenschichten. Mit einlagigen Perzeptren ist es möglich Probleme zu lösen die linear separierbar sind. Bei einem einlagigen Perzeptron gehen von der Eingabeschicht Verbindungen zu den trainierbaren Gewichten, die über eine Aktivierungsfunktion mit der Ausgabeschicht verbunden sind. Die Abbildung 8 soll dies grafisch darstellen.

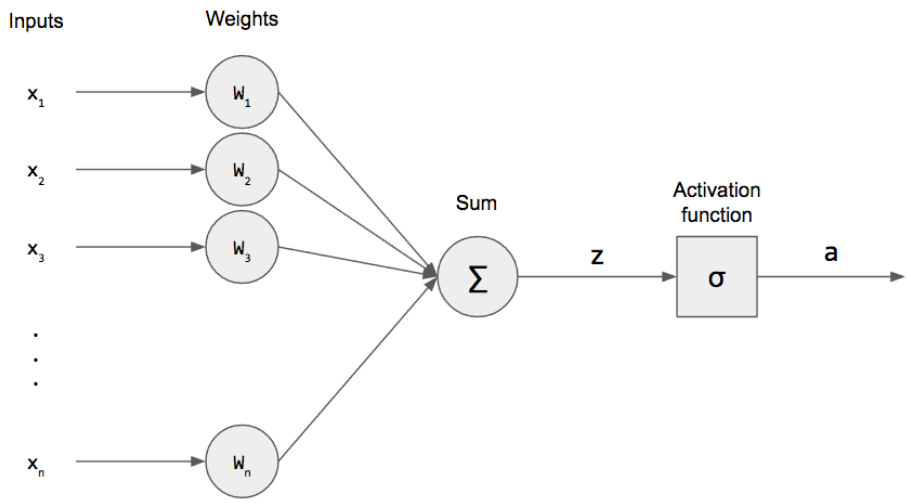

Abbildung 8: Einlagiges Perzeptron

https://1569708099.rsc.cdn77.org/wp-content/uploads/2017/09/Single-Perceptron.png?x23721

Für die Berechnung der Ausgabe wird zunächst die gewichtete Summe der angelegten Eingabewerte wie in Formel 5 gebildet.

$$sum = \sum_{i=1}^{n} x_i * w_i$$

Formel 5: Berechnung der gewichteten Summe der Eingabewerte

Nachdem die Summe bekannt ist, wird die Ausgabe mit Hilfe einer Aktivierungsfunktion bestimmt. Hier ist es die Binäre Schwellenwertfunktion.

$$\varphi(s) = \left\{\frac{1, wenn\ s > 0}{0, sonst}\right\}$$

Formel 6: Binäre Schwellenwertfunktion

5.1.1 Delta Regel

Die Delta Regel ist nur für einschichtige Perzeptren definiert. Die Deltaregel beruht auf der Methode des steilen Abstiegs und schätzt den Gradienten auf einfache Art. Der Algorithmus arbeitet Zeitrekursiv. Das heißt das mit jedem neuen Datensatz der Algorithmus einmal durchlaufen wird und somit wird die Lösung aktualisiert. Die Delta Regel hat den Vorteil das sie nicht für binäre Schwellenwertfunktionen geeignet ist und bei einer großen Entfernung zum Lernziel dies schneller erreicht. Sie berechnet eine Veränderung der Gewichte anhand des Einflusses der an der Verbindung beteiligten Neuronen auf den Fehler.

Zuerst wird eine bestimmte Eingabe, dessen korrekte Ausgabe bekannt ist, durch das Perzeptron propagiert. Anschließend wird für jedes Ausgabeneuron der Fehlerwert δ berechnet.

$$\delta = (p_i - o_i)$$

Formel 7: Fehlerwert

Dabei stellt p_i den erwarteten Wert dar und o_i dessen tatsächlichen Wert.
Die Lernrate welche die Intensität des Trainings angibt wird durch ε repräsentiert. Die Gewichtsanpassung erfolgt schlussendlich durch die untenstehende Formel.

$$w_{neu} = w_i + \Delta_{wi}$$

Formel 8: Gewichtsanpassung

5.1.2 Beispiel des Perzeptron

Die Abbild 3 zeigt ein einfaches Perzeptron das zwei Eingabewerte besitzt. Mit diesem Perzeptron wird die Logische ODER Verknüpfung realisiert. Diese Eingabewerte haben jeweils eine Gewichtung von $\frac{1}{2}$.

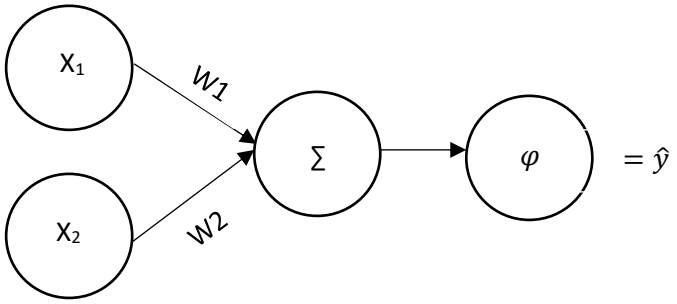

Abbildung 9: Beispiel des Perzeptron

$$\Rightarrow f(x_1, x_2) = y$$

Die Tabelle 1 zeigt für die Eingabe von x1 und x2 die Wahrheitswerte für die Logische ODER Verknüpfung an.

	X (1)	X (2)	X (3)	X (4)
X_1	0	0	1	1
X_2	0	1	0	1
y	0	1	1	1

Tabelle 1: Wahrheitswerte

$$w_1 = \frac{1}{2} \; w_2 = \frac{1}{2}$$

Formel 9: Gewichtung

Nachdem das Perzeptron mit den obenstehenden Gewichten initialisiert wurde. Anschließend werden die Werte der Wahrheitstabelle 1 mit den vorher definierten Gewichten multipliziert, um den Output des Perzeptron zu erhalten. Die Berechnungen des Outputs des Perzeptron werden in der Formel 10 wiedergegeben.

$$x^{(1)}: \frac{1}{2} * 0 + \frac{1}{2} * 0 = 0$$

$$x^{(2)}: \frac{1}{2} * 0 + \frac{1}{2} * 1 = \frac{1}{2} > 0 => 1$$

$$x^{(3)}: \frac{1}{2} * 1 + \frac{1}{2} * 0 = \frac{1}{2} > 0 => 1$$

$$x^{(4)}: \frac{1}{2} * 1 + \frac{1}{2} * 1 = 1 > 0 => 1$$

Formel 10: Berechnung der Outputs

5.2 Mehrlagiges Perzeptron

Im Gegensatz zu einlagigen Perzeptren besitzen mehrlagige Perzeptren weitere Schichten. Diese Schichten werden als Hidden Layer bezeichnet. Das Mehrlagige Perzeptron konnte die Beschränktheit der Einlagigen Perzeptren lösen. Diese Netzwerke gehören zu der Klasse der Feedforward-Netze in dem alle Neuronen einer Schicht mit allen Neuronen der darüber liegenden Schicht verknüpft sind. Ein Mehrlagiges Perzeptron kann mit dem Backpropagation Algorithmus trainiert werden. Zur Anwendung kommen solche Netze bei der Bildverarbeitung und Mustererkennung aber auch bei der Spachverarbeitung und Spracherkennung, sowie in der Regelungstechnik.

6. Bildverarbeitung

Unter Bildverarbeitung versteht man die Verarbeitung von Signalen, die Bilder repräsentieren wie Fotografien oder Einzelbilder aus Videos. Die Bildverarbeitung liefert als Ergebnis wieder ein Bild oder eine Menge von Merkmalen des Eingangsbildes. Die Bildverarbeitung wird heute nahezu in allen Wissenschaftsdisziplinen und Ingenieurdisziplinen eingesetzt. Durch die rasche Entwicklung in der Bildsensorik und der immer leistungsfähigeren Computersystemen, nimmt das Interesse an weiteren Möglichkeiten der moderne Bildverarbeitung stetig zu. Durch diese stetige Zunahme der Leistungsfähigkeit der Computer und der Bildsensoren werden diese heute schon in einer Vielzahl von Bereichen zur Darstellung von Bildern und Bildinhalten

eingesetzt. Durch die Mustererkennung können die Computer die zu verarbeitenden Bilder auf Regelmäßigkeiten, Wiederholungen und Ähnlichkeiten überprüfen. Der zentrale Punkt in der Mustererkennung ist dabei das Erkennen von Mustern und deren Merkmalen, um diese zu Segmentieren und zu Klassifizieren. Durch die Mustererkennungsverfahren werden Computer und Maschinen befähigt auch weniger exakte Signale zu einer Natürlichen Umgebung zu verarbeiten. Heutzutage unterscheidet man drei Ansätze zur Mustererkennung die Syntaktische, statische und strukturelle Mustererkennung.

Die älteste Form der Mustererkennung ist die Syntaktische Mustererkennung. Bei dieser Form werden Dinge durch Folgen von Symbolen beschrieben, sodass Objekte der Gleichen Kategorie dieselbe Beschreibung aufweisen. Das Problem bei dieser Art ist, die Suche nach einer geeigneten formalen Grammatik.

Das Ziel der statischen Mustererkennung ist es, zu einem Objekt die Wahrscheinlichkeit zu bestimmen, dass es zu der einen oder einer anderen Kategorie gehört. Bei der statischen Mustererkennung werden keine Merkmale nach vorher festgelegten Regeln ausgewertet, stattdessen werden bei dieser Methode Zahlenwerte gemessen und diese in einem Merkmalsvektor zusammengefasst. Durch Mathematische Funktionen werden anschließend die Merkmalsvektoren eindeutig zu einer Kategorie zugeordnet. Die größte Stärke dieses Verfahrens ist es, dass es auf nahezu alle Sachgebiete angewendet werden kann.

Die strukturelle Mustererkennung verbindet syntaktische und statische Verfahren zu einem neuen Verfahren. Bei der strukturellen Mustererkennung werden übergeordnete strukturelle Verfahren wie das Bayes’sche Netz führen Einzelergebnisse zusammen. Daraus wird das Gesamtergebnis berechnet und die Kategoriezugehörigkeit bestimmt. Die grundliegende Merkmalserkennung wird dabei dem statistischen Verfahren überlassen, während übergeordnete Inferenzverfahren Spezialwissen über das Sachgebiet einbringen.

Um die Mustererkennung weiter zu verbessern wird versucht die Verarbeitung der visuellen Signale beim Menschen und bei Tieren zu imitieren und diese nachzubilden. Für die Verarbeitung von optischen Signalen bei Menschen und Tieren sind die in den Augen vorhandenen Photorezeptoren, die als einzelne Bildpunkte zu verstehen sind, verantwortlich. Diese Photorezeptoren dienen im Auge als einzelne Bildpunkte, die wiederum im visuellen Cortex verarbeitet werden. In der Informatik werden statt der Photorezeptoren Rastergrafiken eingesetzt. Eine Rastergrafik ist eine Form eines Bildes die ein Computer lesen kann. Diese Grafiken bestehen aus einer Rasterförmigen Anordnung von Pixeln, denen jeweils

eine Farbtiefe zugeordnet ist. Die Einzelnen Informationen der Pixel können dann wie im visuellen Cortex verarbeitet werden. Sie können beispielsweise als Eingabe für ein Künstliches Neuronales Netz verwendet werden. Zum jetzigen Erkenntnisstand werden Neuronale Netze nicht nur verwendet, um vorgegebene Muster zu erkennen, sondern dass diese Netze aus den zugefügten Daten selbstständig Muster erlernen. Damit ein künstliches Neuronales Netz die Fähigkeit des selbstständigen Erkennens von Mustern besitzt muss das Netz die Faltung von Bildern beherrschen. Wie die Faltung im Einzelnen funktioniert wird im nächsten Kapitel dieser Arbeit beschrieben.

7. Convolutional Neuronal Network

Für die Probleme der Bildverarbeitung und dem maschinellen Sehen werden heute vor Allem Convolutional Neuronal Networks eingesetzt. Die Grundlagen für diese Art von Netzwerken wurde 1998 von LeCun geschaffen. Die Funktionsweise von Convolutional Neuronal Networks kann sowohl der klassischen Bildverarbeitung, als auch aus den Arbeiten von Hubel, Yang und Wiesel abgeleitet werden. Diese beschäftigen sich mit dem rezeptiven Feld im optischen Kortex von Katzen.

Für die Bildverarbeitung werden sehr häufig Neuronale Netze eingesetzt, da sie sich selbst anpassen können und daher sehr universell einzusetzen sind. Der Grundsätzliche Aufbau eines Convolutional Neuronal Network besteht aus einem Inputlayer, einem oder mehreren Convolutional Layer der gefolgt von einem Pooling Layer ist. Dieser Aufbau der einzelnen Layer kann beliebig oft in einem solchen Netzwerk wiederholt angewandt werden, um komplexe Aufgaben zu lösen. Ein Convolutional Neuronal Network wird überwacht trainiert.

7.1 Convolutional Layer

Der Convolutional Layer ist in einem Convolutional Neuronal Network der wichtigste Layer. Diese Art von Layer faltet das eingehende Bild. Eine Faltung ist eine Anordnung von Pixeln, wie zum Beispiel einer Kante, Belichtung oder Farbunterschiede also der Wechsel der Intensitäten von benachbarten Pixeln entsprechend einem Muster.

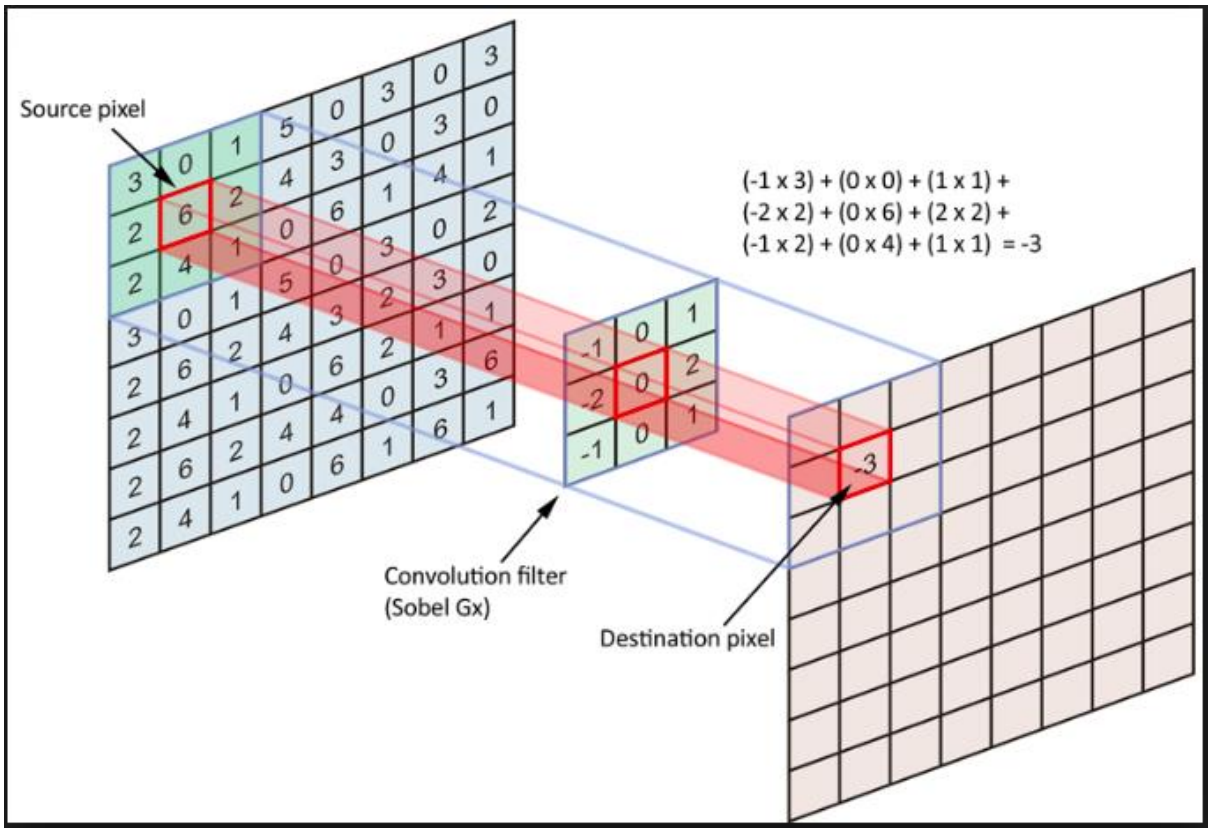

Abbildung 10: Prinzip der Faltung

*https://cdn-images-1.medium.com/max/1600/1*EuSjHyyDRPAQUdKCKLTgIQ.png*

Die Abbildung 10 zeigt, wie mit Hilfe einer Filtermaske in der Größe 3x3 Pixeln, die gewichtete Summe der Pixel einer Bildnachbarschaft berechnet wird. Um die Faltungen zu berechnen werden die Eingangs- und Ausgangsbilder sowie die Filtermaske als Matrix definiert. Die zu verwendenden Filter müssen immer in einer ungeraden Größe vorliegen, um den eindeutigen Mittelpunkt bestimmen zu können. Hat man ein RGB Bild vorliegen muss für jeden Farbkanal ein Filter angewandt werden. Um die Faltung durchzuführen wird der Filter auf das Eingangsbild gelegt. Somit wird die gewichtete Summe der benachbarten Pixel gebildet. Anschließend wird der Filter in der Regel einen Pixel verschoben und wieder angewandt. Dies erfolgt über das gesamte Eingangsbild. Dabei liefert der Filter an einer Position im Eingangsbild den entsprechenden Bild Wert im Ausgangsbild. Der Filter in dem Convolutional Layer hat keine Konstanten Werte. Die Werte des Filters sind die trainierbaren Gewichte in dem Convolutional Neuronal Network.

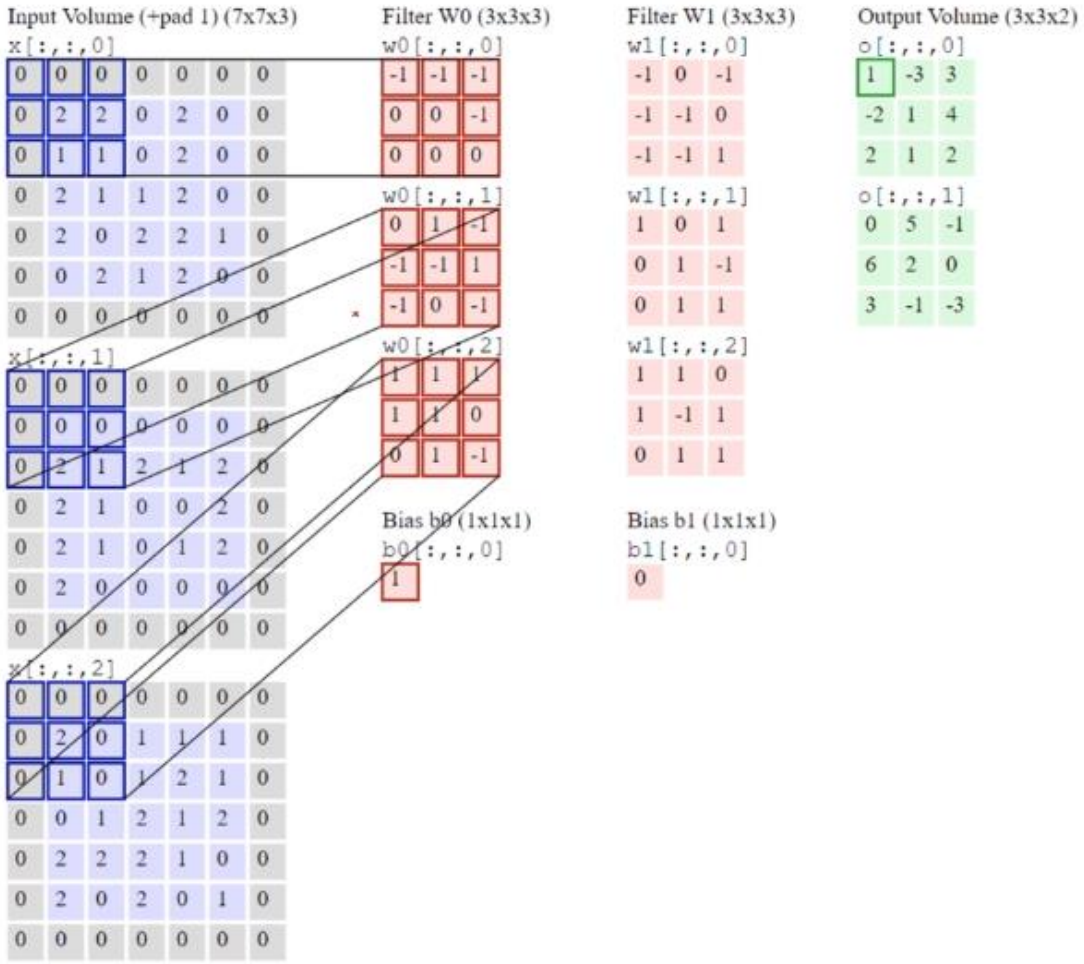

Abbildung 11: Berechnung des Outputs

*https://cdn-images-1.medium.com/max/1600/0*9J3MK1gd2zrFDzDN.gif*

$$\begin{pmatrix}0 & 0 & 0\\0 & 2 & 2\\0 & 1 & 1\end{pmatrix}\begin{pmatrix}-1 & -1 & -1\\0 & 0 & -1\\0 & 0 & 0\end{pmatrix} = -2$$

$$\begin{pmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 2 & 1\end{pmatrix}\begin{pmatrix}0 & 1 & -1\\-1 & -1 & 1\\-1 & 0 & -1\end{pmatrix} = -1$$

$$\begin{pmatrix}0 & 0 & 0\\0 & 2 & 0\\0 & 1 & 0\end{pmatrix}\begin{pmatrix}1 & 1 & 1\\1 & 1 & 0\\0 & 1 & -1\end{pmatrix} = 3$$

$$Ergebnis = -2 + (-1) + 3 = 0 + 1 = 1$$

Formel 11: Berechnung des Outputs

Die Abbildung 11 zeigt ein Bild mit der Größe 7x7, mit einer Tiefendimension von 3. Auf dieses Bild wird ein 3x3 Filter angewandt, der jeweils um Zwei Pixel verschoben wird. Durch das Verschieben des Filters ändert sich in der Ausgabe auch die Größe des Bildes auf 3x3 Pixel.

Durch das Anwenden der Filtern W0 und W1 ändert sich auch die Tiefendimension von Drei auf Zwei. Das Berechnungsbeispiel zeigt die Rechnung für den ersten Pixel in der Ausgabe des Convolutional Layers. Wenn der Filter ein Feature gefunden hat, dass er gesucht hat, so wird das Ergebnis in der Ausgabematrix höher sein.

Um zu verhindern, dass die Ausgabe eine andere Größe hat als die Eingabe, muss je nachdem wie die Filterverschiebung durchgeführt wird, das Eingabebild mit zusätzlichen Pixeln erweitert werden. Diese hinzugefügten Pixel werden mit dem Wert Null in das Bild geschrieben. Bei einer Verschiebung des Filters um 2 Pixel wird das Eingabebild jeweils um Zwei neue Spalten und Zeilen erweitert. Die Erweiterung des Eingangsbilds wird in Abbildung 12 ersichtlich.

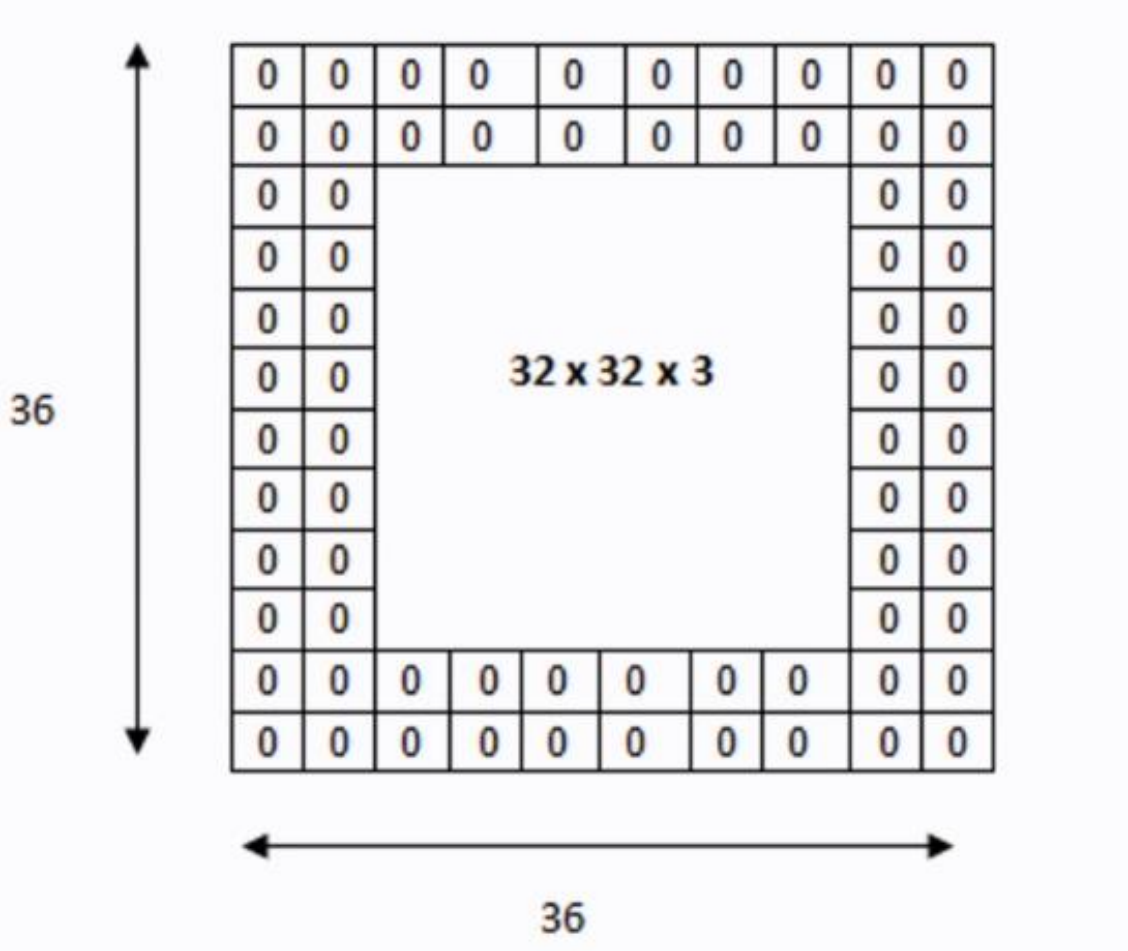

Abbildung 12: Erweiterung Der Eingabe für eine Ausgabe in der gleichen Dimension

https://adeshpande3.github.io/assets/Pad.png

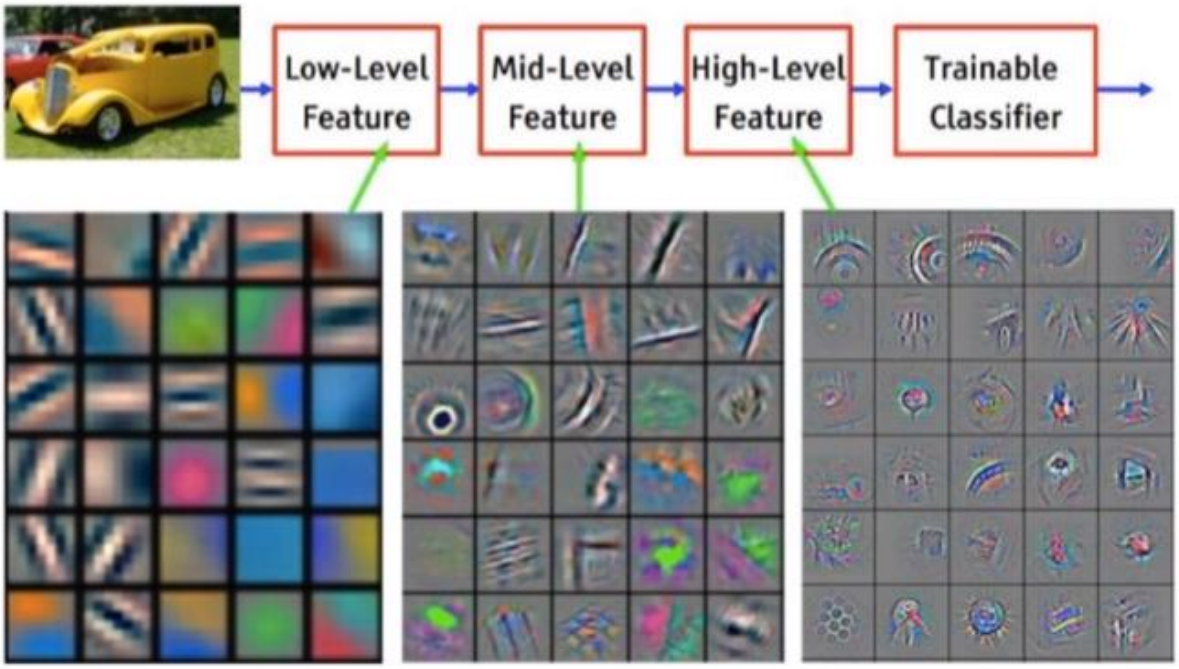

Abbildung 13: Feature Visualisierung auf Basis eines Convolutional Network (ImageNet)

*https://cdn-images-1.medium.com/max/2600/1*Ji5QhY9QXBlpNNLH4qAcNA.png*

Wie oben bereits erwähnt, werden in einem Convolutional Neuronal Network mehrere Convolutional Layer verwendet. Die ersten Convolutional Layer sind in der Lage Low-Level Features wie Kanten, Kontraste oder diverse Beleuchtungsunterschiede zu erkennen. Darauf aufbauend befinden sich weitere Convolutional Layer, die dann Mid-Level Features wie zum Beispiel Kreise oder Kanten erkennen können. Und darauf wiederum befindet sich eine weitere Schicht mit Convolutional Layern die dann High-Level Features wie Räder oder Teile eines Kühlergrills erkennen können.

7.2 Pooling Layer

Ein Pooling Layer aggregiert die Ergebnisse von vorher einhergehenden Convolutional Layer, indem der Pooling Layer das jeweils stärkste Signal weitergibt. Der Pooling Layer verwendet den höchsten Wert einer vorher definierten Kernel-Matrix, die anderen Werte werden verworfen. Die von einem 2x2 Kernel erstellten Vier Matrix Ergebnisse werden so auf nur eine Zahl reduziert. Durch den Pooling Layer wird versucht die Komplexität in einem Netzwerk zu reduzieren. Der Pooling Layer reduziert die Dimension eines Bildes gleichermaßen in der Weite und Breite. Die Abbildung 14 zeigt ein Pooling Layer in der Anwendung. Wie auch bei dem Convolutional Layer gibt es auch bei einem Pooling Layer einen Filter und eine Verschiebung des Filters. Auf das Bild in Abbildung 12 wird ein 2x2 Filter mit einer

Verschiebung von Zwei angewandt. Nach der Anwendung des Filters wird aus einem 4x4 Bild ein 2x2 Bild.

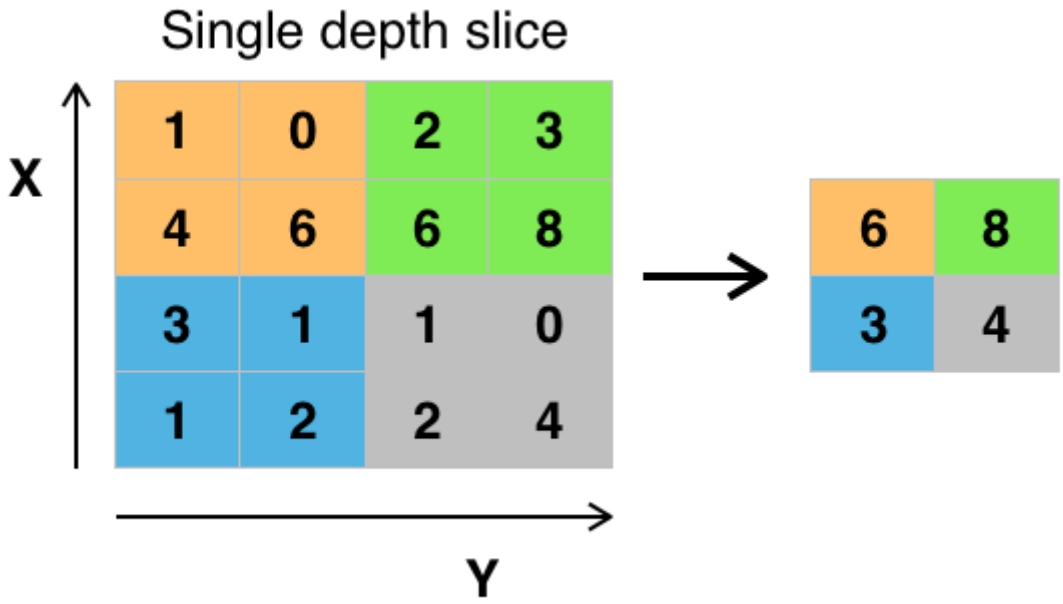

Abbildung 14: Max Pooling Layer

https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/Max_pooling.png/314px-Max_pooling.png

Das gerade beschriebe Verfahren nennt sich Max Pooling. Für diesen Typ Layer gibt es noch weitere Arten. Eine weitere Art ist das Average Pooling. Bei dieser Art wird nicht der höchste Wert in einem Filter weitergereicht, sondern der Mittelwert der Filtermatrix.

$$Filter\ 1: \frac{1+0+4+6}{4} = 2{,}75 \approx 3$$

$$Filter\ 2: \frac{2+3+6+8}{4} = 4{,}75 \approx 4$$

$$Filter\ 3: \frac{3+1+1+2}{4} = 1{,}75 \approx 2$$

$$Filter\ 4: \frac{1+0+2+4}{4} = 1{,}75 \approx 2$$

Formel 12: Berechnung mit Max Pooling

Weiterhin gibt es noch das minimal Pooling. Bei dieser Art würde die Filtermatrix den kleinsten Wert weitergeben. Diese Art des Poolings ist für ein Convolutional Neuronal Network nicht geeignet. Da der Filter des Convolutional Layers wenn er etwas gefunden hat, was der Filter sucht, den Wert hoch setzten wird. Hohe Werte werden in gefalteten Bildern immer darstellen, dass etwas gefunden wurde. Deswegen wird in der Regel in einem Convolutional Neuronal Network immer das Max Pooling oder das Average Pooling angewandt.

Die Tabelle 2 zeigt einen Praxistest der Zwei unterschiedlichen Pooling Layer zwischen Max Pooling und dem Average Pooling existiert ein Performanceunterschied von 2,19 %.

	Max Pooling	Average Pooling
Genauigkeit	87,71%	85,52%

Tabelle 2: Praxistest Unterschied Max Pooling und Average Pooling

Die Abbildung 15 zeigt ein Max Pooling. Ein Bild mit der Dimension 224x224 wird durch ein 2x2 Filter auf die Dimension 112x112 heruntergerechnet. Durch die Anwendung der Filter im Convolutional Layer werden erkannte Objekte auf dem Bild heller. Es wurden also Objekte erkannt. Durch die Anwendung des Max Poolings sind im Ergebnisbild immer noch die wichtigen Informationen enthalten. Nur das Bild ist jetzt nur noch halb so groß.

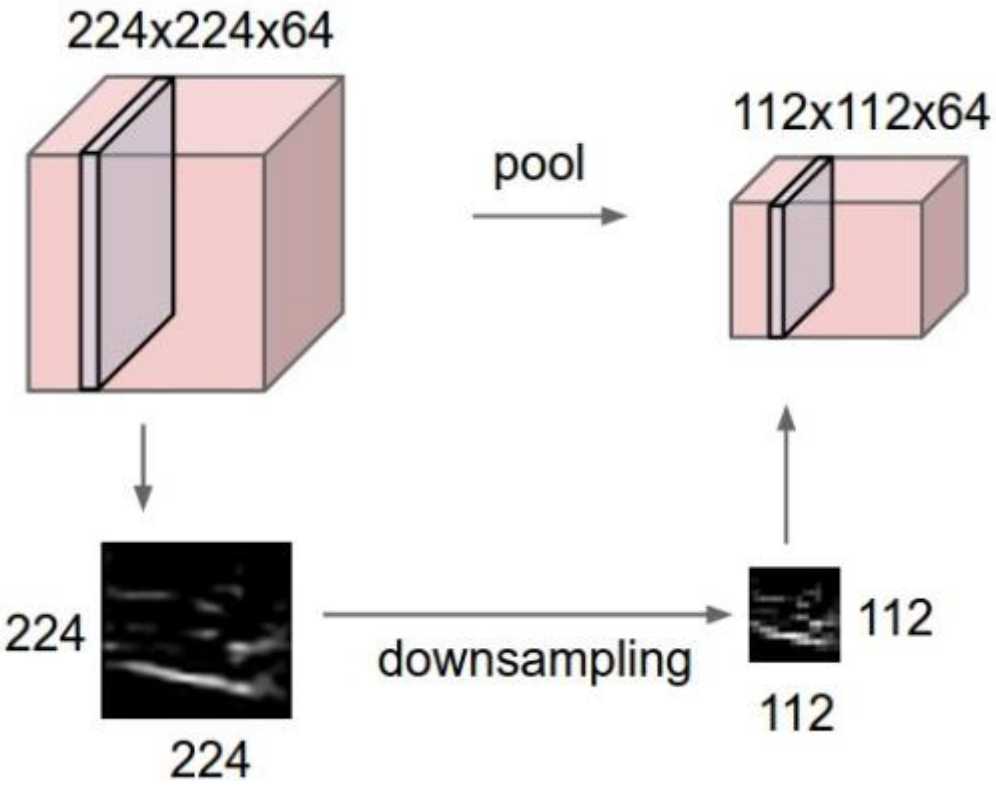

Abbildung 15: Max Pooling an einem Bild

http://cs231n.github.io/assets/cnn/pool.jpeg

Durch die damit entstandene Datenreduktion reduziert sich die Performance des Netzwerkes nicht. Die Reduktion der Bildgröße bietet signifikante Vorteile in der Berechnungsgeschwindigkeit. Weiterhin lassen sich durch die Reduzierung der Bildgrößen tiefere Netzwerke realisieren, die wiederum komplexere Aufgaben lösen können. Die weniger an dem Problem des Overfittings leiden.

7.3 Fully-connected Layer

Nach einer wiederholten Einheitenblöcken bestehend aus Convolutional- und Poolinglayern kann das Netzwerk mit einem oder mehreren Fully-connected Layern abschließen. Diese Art von Layer wird für Klassifikationsprobleme verwendet.

8. Das Training im Neuronalen Netz

Ein Training in einem Neuronalen Netz besteht aus mehreren Epochen. In einer Epoche wird der gesamte Trainingsdatensatz durch das Neuronale Netzwerk propagiert. Der Trainingsdatensatz in einer Epoche wird in mehrere Batches unterteilt. Die Größe der Batches hängt immer von der Leistungsfähigkeit der eingesetzten Computer ab. Die Updates der Gewichte der einzelnen Epochen werden dann anschließend mit einem Stochastic Gradient Descent Verfahren berechnet.

8.1 Das Stochastic Gradient Descent (SGD) Verfahren

Das Stochastic Gradient Descent Verfahren bildet den Gradienten des Trainingsdatensatzes nicht auf einmal. Der Gradient wird über die einzelnen Batches gebildet. Es wird immer eine Batch durch das komplette Netzwerk bis zur Ausgabeschicht propagiert um zu erfahren ob das Netzwerk einen Fehler gemacht hat. Hat das Netzwerk einen Fehler bei der Klassifizierung gemacht, so wird für die eine Batch die Error Backpropagation durchgeführt und die Gewichte werden auf Basis der Batch angepasst. Das Update der Gewichte wird in Relation zu der vorher definierten Lernrate durchgeführt. Nach der Anpassung der Gewichte wird die nächste Batch durch das Netzwerk propagiert um wieder zu sehen ob das Netzwerk einen Fehler gemacht hat um die Gewichte bei einem Fehler wieder anzupassen. So werden die einzelnen Batches in einer Epoche durch das Netzwerk propagiert, um die Fehlerwerte zu minimieren. Um den Fehlerwert zu minimieren gibt es eine Vielzahl von Optimizern. In dem Neuronalen Netzwerk, das in dieser Arbeit entwickelt wird, werden 4 verschiedene Optimizer auf ihre Nutzbarkeit für das vorliegende Klassifikationsproblem zum Einsatz kommen. Zum Einsatz im Praktischen Teil kommen die Optimizer Stochastic Gradient Descent (SGD) und Adaptive Moment Estimation (Adam).

8.1.1 Gewichtsanpassung ohne Lernrate

Die Abbild 14 zeigt einen Funktionsplot der Funktion $f(x) = x^2 + x + 1$. Die Funktion hat ein Globales Minimum bei $x - \frac{1}{2}$ .Dieses globale Minimum wird durch den mittleren Pfeil in der Abbildung 14 gekennzeichnet. Um den Gradienten Abstieg durchzuführen muss von dieser Funktion die erste Ableitung gebildet werden $f'(x) = 2x + 1$. In diesem Beispiel wird angenommen das $x_0 = 1$ ist. Dies ist auch der Startpunkt für den Gradienten Abstieg was mit dem rechten Pfeil in der Abbildung 16 gekennzeichnet ist. Nach dem Anwenden der Updateregel würde $x_0 = -2$ sein.

$$x_0 = x_0 - \nabla f'(x_0)$$

Formel 13: Updateregel ohne Lernrate

$$x_0 = 1 - 2 * 1 + 1$$

$$x_0 = 1 - 2 + 1 = -2$$

Formel 14: Berechnung der Updateregel

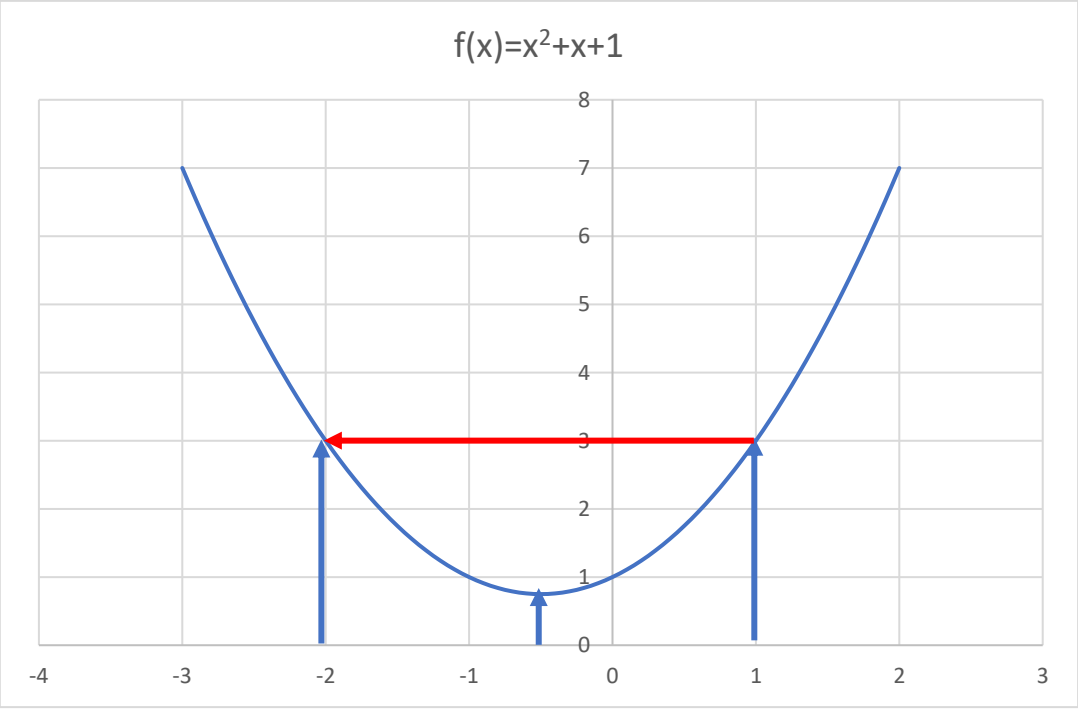

Abbildung 16: Funktionsgraph ohne Lernrate

8.1.2 Gewichtsanpassung mit Lernrate

Wie in der Abbildung 16 leicht zu sehen ist, ist nach der Anwendung der Updateregel der Fehlerwert gleich. Mit der Durchführung des Updates würde der Fehlerwert in diesem Beispiel

immer gleich bleiben. Durch ein Update sollte sich der Fehlerwert immer weiter an das Globale Minimum gefunden werden angeglichen werden.

Damit das gerade Beschriebene nicht eintritt, wird die Updatefunktion um eine weitere Variable erweitert. Die hinzugefügte Variable ist die Lernrate η. Die Lernrate ist Intervall von 0 – 1 im Bereich der Reellen Zahlen frei wählbar.

Die Lernrate wird in dem unten Aufgeführten Beispiel mit 0,2 festgelegt.

$$x_0 = x_0 - \eta\nabla * f'(x_0)$$

Formel 15: Updateregel mit Lernrate

$$x_0 = 1 - 0{,}2 * (2 * 1 + 1) = 0{,}4$$

$$x_0 = 0{,}4 - 0{,}2 * (2 * 0{,}4 + 1) = 0{,}04$$

$$x_0 = 0{,}04 - 0{,}2 * (2 * 0{,}04 + 1) = -0{,}176$$

$$x_0 = -0{,}176 - 0{,}2 * (2 * (-0{,}176) + 1) = -0{,}3056$$

$$x_0 = -0{,}3056 - 0{,}2 * (2 * (-0{,}3056) + 1) = -0{,}38336$$

$$x_0 = -0{,}38336 - 0{,}2 * (2 * (-0{,}38336) + 1) = -0{,}430016$$

$$x_0 = -0{,}430016 - 0{,}2 * (2 * (-0{,}430016) + 1) = -0{,}4580096$$

Formel 16: Berechnungen der Updateregel mit Lernrate

Die Sieben Berechnungen zeigen den immer weiter abnehmenden Fehlerwert, der in Abbildung 17 grafisch zu sehen ist. Jede einzelne Berechnung des Fehlerwertes stellt eine Epoche im Training dar.

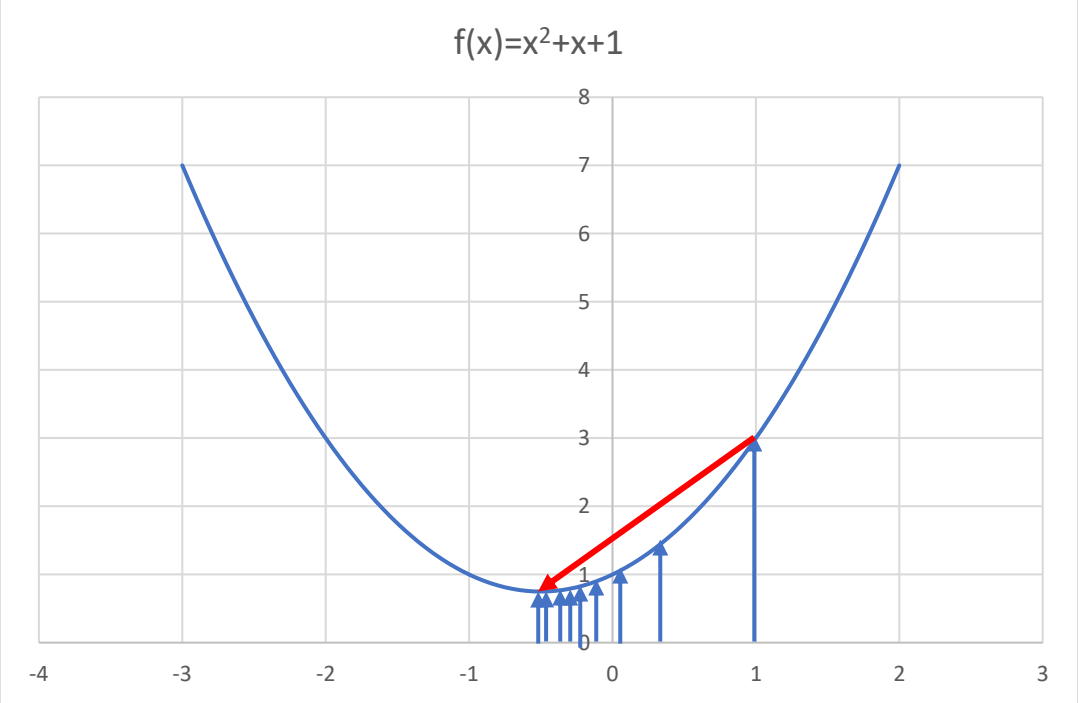

Abbildung 17: Funktionsgraph mit Lernrate

Während der ersten Iterationsschritte senkt sich der Fehlerwert noch mit größeren Schritten der dann aber mit dem zunehmenden Trainingsfortschritt immer geringer wird. Wie bereits erwähnt wird die Stärke der Anpassung des Fehlerwerts durch die Lernrate und den Gradienten bestimmt.

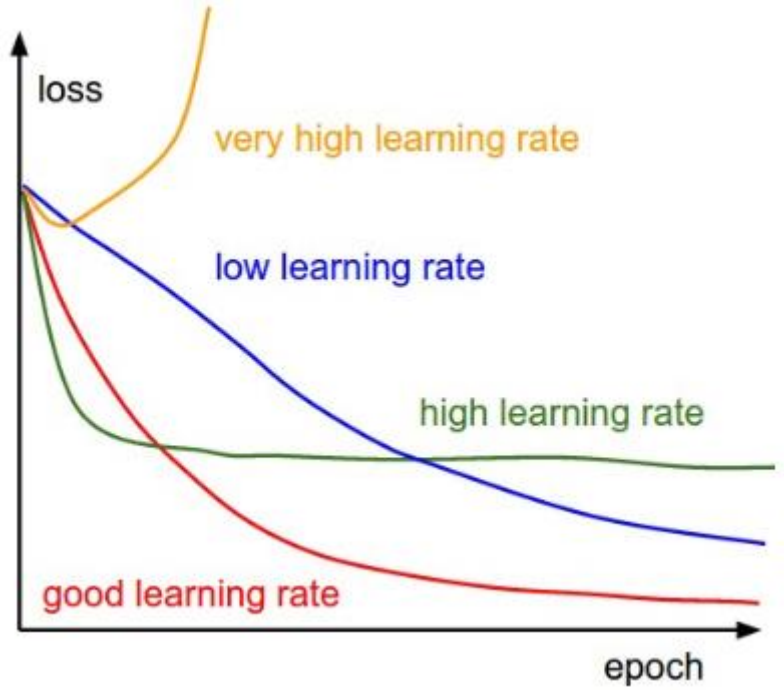

Abbildung 18: Bestimmung der Lernrate

Quelle: http://cs231n.github.io/neural-networks-3/

Durch eine viel zu hoch gewählte Lernrate wird mit steigender Anzahl an Epochen der Fehlerwert immer größer werden. Wird die Lernrate zu gering gewählt kann der Fehlerwert in einem Lokalen Minimum hängenbleiben oder sich erst nach einer Vielzahl von Trainingsschritten an das Globale Minimum annähern. Dies wird in den Abbildungen 19 und 20 ersichtlich. Um aus den Lokalen Minima zu entkommen hilft eine adaptive Lernrate sowie die einzelnen Batches pro Trainingsdurchlauf. Da nach jeder Batch die Gewichte angepasst werden. Ist die Lernrate aber besonders schlecht gewählt, so können die einzelnen Batches und die adaptive Lernrate keine großen Erfolge bei der Bestimmung des Fehlerwertes liefern.

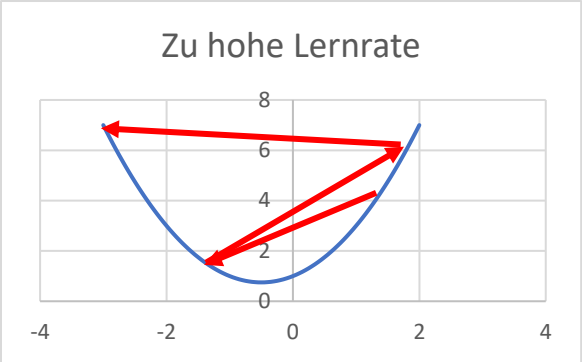

Abbildung 19: Lernrate zu Hoch

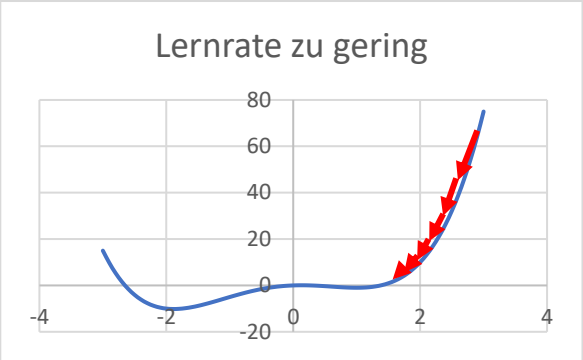

Abbildung 20: Lernrate zu gering

Die Lernrate muss für jede Problemstellung neu bestimmt werden, auch Änderungen am Netzwerk können zu einem anderen Ergebnis führen bei gleicher Lernrate. Die globalen Minima können nicht bewiesen werden. Man weiß nie, ob das gefundene Minimum ein gutes Minimum ist. Wie die Lernrate gewählt wird, hängt von dem zu lösenden Problem und dem eingesetzten Gradientenabstiegsverfahren ab. Anhand von mehreren Trainingsdurchläufen kann die Lernrate manuell eingestellt werden. Weiterhin kann die Lernrate mit einer Grid oder Random Suche bestimmt werden.

8.2 Adaptive Moment Estimation (Adam)

Der Adam Optimizer ist eine Weiterentwicklung von dem klassischen SGD Verfahren. Das Verfahren aktualisiert die Gewichte iterativ, basierend auf dem Trainingsdatensatz. Dieser Optimizer wurde 2015 von Diederik Kingma von OpenAI und Jimmy Ba von der University of Toronto 2015 vorgestellt. Während der klassische SGD Optimizer eine feste Lernrate für das Update der Gewichte hat, die sich auch des Trainings nicht ändert, so hat der Adam Optimizer eine individuelle adaptive Lernrate für die einzelnen Gewichte. Diese adaptive Lernrate wird durch Schätzungen der ersten und zweiten Ordnung der Gradienten bestimmt.

Er kombiniert die Vorteile von den Optimizern AdaGrad und RMSProp. Der AdaGrad funktioniert sehr gut mit spärlichen Verläufen. Der RMSProp funktioniert gut mit nicht stationären Zielen. Durch diese Eigenschaften lässt der Adam Optimizer sich für eine Vielzahl von Problemstellungen anwenden. Diese beiden Optimizer werden in dieser Arbeit nicht weiter erklärt und angewandt.

Der Adam Optimizer kann als eine Kombination aus den Optimizern RMSProp und SGD mit Momentum betrachtet werden. Der Optimizer verwendet die quadrierten Gradienten um die Lernrate wie bei RMSProp zu skalieren. Dadurch das der Adam Optimizer eine adaptive Methode ist berechnet der Adam für die verschiedenen Gewichte individuelle Lernraten. Er

schätzt die Gradienten der ersten und zweiten Ordnung, um die Lernrate für jedes Gewicht in einem Neuronalen Netzwerk anzupassen. Das erste Moment ist der Mittelwert und das zweite Moment ist die nichtzentrierte Varianz. Um die Momente abschätzen zu können verwendet der Adam die exponentiellen gleitende Durchschnitte die über den Gradienten der aktuellen Batch berechnet werden.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Formel 17: Mittelwert des Verlaufs

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Formel 18: Mittelwert des Quadratischen Verlaufs

In den Formeln 17 und 18 sind die Werte $m\ und\ v$ die gleitenden Durchschnitte und g ist der Gradient der aktuellen Batch. Der Adam hat auch weitere Hyperparameter die in den Formeln 17 und 18 durch das β gekennzeichnet sind. Die Standartwerte dieser neuen Hyperparameter sind für $\beta_1 = 0{,}9$ und für $\beta_2 = 0{,}999$. In der ersten Iteration werden die Vektoren für die gleitenden Durchschnitte mit Nullen initialisiert. In der späteren Praktischen Umsetzung werden die Werte für $\beta_1 und \beta_2$ nicht verändert da diese Standardwerte schon sehr gute Ergebnisse liefern.

Um das Update der Gewichte durchzuführen, muss eine Bias Korrektur durchgeführt werden. Diese Korrektur muss für alle Optimizer die mit gleitenden Durchschnitten Arbeiten durchgeführt werden. Die Bias Korrektur wird mit den Formeln 19 und 20 durchgeführt.

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

Formel 19: Bias Schätzer der ersten Ordnung

$$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

Formel 20: Bias Schätzer der zweiten Ordnung

Anhand der Formeln für den Mittelwert des Verlaufs und des Quadratischen Verlaufs lässt sich die Lernrate für jedes Gewicht individuell skalieren. Um nun eine Gewichtsanpassung vorzunehmen wird die Formel 21 verwendet.

$$w_t = w_{t-1} - \eta \frac{\widehat{m_t}}{\sqrt{\widehat{v_t} + \epsilon}}$$

Formel 21: Gewichtupdate Adam

8.3 Grid und Random Search

Die Grid Search ist ein Verfahren bei dem verschiedene Lernraten und verschiedene Optimizer gegeneinander antreten, um das beste Model zu finden. Hierfür muss dem Netzwerk verschiedene Lernraten und Optimizer zur Verfügung gestellt werden. Die Grid Search bildet dann aus den verschiedenen Lernraten, sowie Optimizern, ein Grid und trainiert damit das Netzwerk auf dem gesamten Trainingsdatensatz. Anschließend werden die Ergebnisse der verschiedenen Modelle auf Basis des Testdatensatzes untereinander verglichen.
Die Random Search funktioniert ähnlich wie die Grid Search. Der einzige Unterschied hierbei ist das die unterschiedlichen Lernraten nicht fix vorliegen. Die Lernraten werden bei der Random Search in einem Intervall angegeben. Daraus erstellt die Random Search verschiedene zufällige Modelle, die dann miteinander verglichen werden.

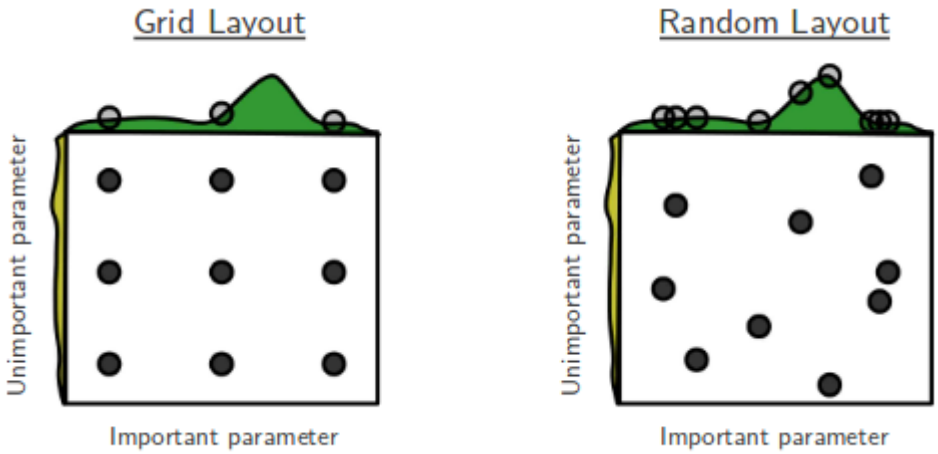

Abbildung 21: Grid und Random Search
https://i.stack.imgur.com/cIDuR.png

8.4 Kreuzvalidierung

Diese eben beschriebene Grid und Random Search kann mit einer Kreuzvalidierung kombiniert werden. Bei der Kreuzvalidierung wird der Trainingsdatensatz in k- Bestandteile zerlegt. Diese Bestandteile (Folds) sind gleich groß. Wie die Abbildung 22 zeigt wurde in diesem Beispiel ein Datensatz in 6 gleich große Folds zerlegt. Dabei ist jeder Fold einmal im Validationsset und 5-mal im Trainingsset. Somit ist jeder Fold einmal für die Messung der Performance da. Anschließend wird der Mittelwert der einzelnen Folds gebildet. Dadurch kann die Genauigkeit des Netzwerkes besser bestimmt werden. Da jeder Fold einmal Validationsset enthalt ist und somit für die Performancemessung relevant ist.

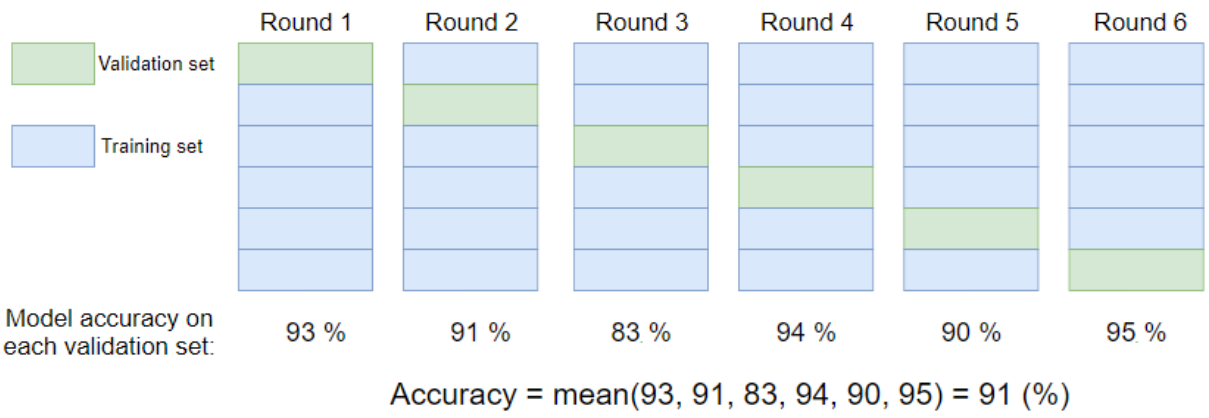

Abbildung 22: Kreuzvalidierung
http://www.machinelearningtutorial.net/2017/04/01/training-set-vs-test-set-vs-validation-set-whats-the-deal/

8.4.1 ReduceLerningRateOnPlateau

Die Lernrate kann außerdem noch mit der Methode ReduceLerningRateOnPlateau angepasst werden. Die Methode ReduceLerningRateOnPlateau senkt die Lernrate, wenn sich der Fehlerwert des Validationssets innerhalb von einer Festgelegten Trainingsepochenanzahl nicht verringert hat. Wenn der Fehlerwert minimiert wird, dann wird die Genauigkeit maximiert. Die Senkung der Lernrate bei dieser Methode wird in Prozent angegeben.

9. Backpropagation

In diesem Kapitel dieser Arbeit wird anhand eines Beispiels die Backpropagation geklärt. Dieses Netzwerk besitzt einen Inputlayer mit zwei Inputfeatures einen Hiddenlayer mit einem Neuron und einem Outputlayer mit einem reellen Outputfeature. Das Netzwerk besitzt 5 Gewichte, die zu Beginn fest definiert sind. Der Konten w hat die Gewichte $-0{,}25\ und\ 1{,}0$, der Konten b hat das Gewicht $0{,}1$, der Konten u hat das Gewicht $1{,}25$ und der Konten c hat das Gewicht $0{,}125$. Dabei sind die Konten b und c die Biasneuronen in diesem Netzwerk.
Im ersten Schritt der Backpropagation wird der Input des Konten X mit den Gewichten des Knoten W multipliziert. Das Ergebnis dieser Multiplikation wird in dem Knoten j gespeichert.

$$(-0{,}25 * 2) + (1{,}0 * 2) = 1{,}5$$

Formel 22: Berechnung für Knoten j

Im nächsten Schritt wird auf dieses Ergebnis der Bias addiert. Somit hat der Knoten i den Wert $1{,}6$. Dieser Wert wird anschließend mit der Aktivierungsfunktion Relu h ausgewertet. Die Relu

sendet diesen Wert an den Hiddenlayer weiter da er größer Null ist. Im Hiddenlayer wird auf diesen Wert das Gewicht des Knoten u multipliziert.

$$1{,}25 * 1{,}6 = 2$$

Somit hat der Knoten g den Wert 2. Auf diesen Wert wird wieder das Bias addiert somit hat der Konten f den Wert $2{,}125$. Dieser Wert ist der Output des Netzwerkes. Der Konten Y enthält das Ergebnis, das hätte rauskommen sollen. Da dieses Netzwerk mit dem überwachten Lernen lernt. Um den Fehlerwert zu bestimmen wird in diesem Beispiel die Mittlere Quadratische Abweichung berechnet. Dies erfolgt mit einer Subtraktion von dem Knoten Y und dem Knoten e, dieses Ergebnis wird anschließend im Knoten d quadriert. Die Anpassung der Gewichte erfolgt mit dem Stochastic Gradient Descent Verfahren.

$$2 - 2{,}125 = -0{,}125$$
$$-0{,}125^2 = 0{,}015625$$

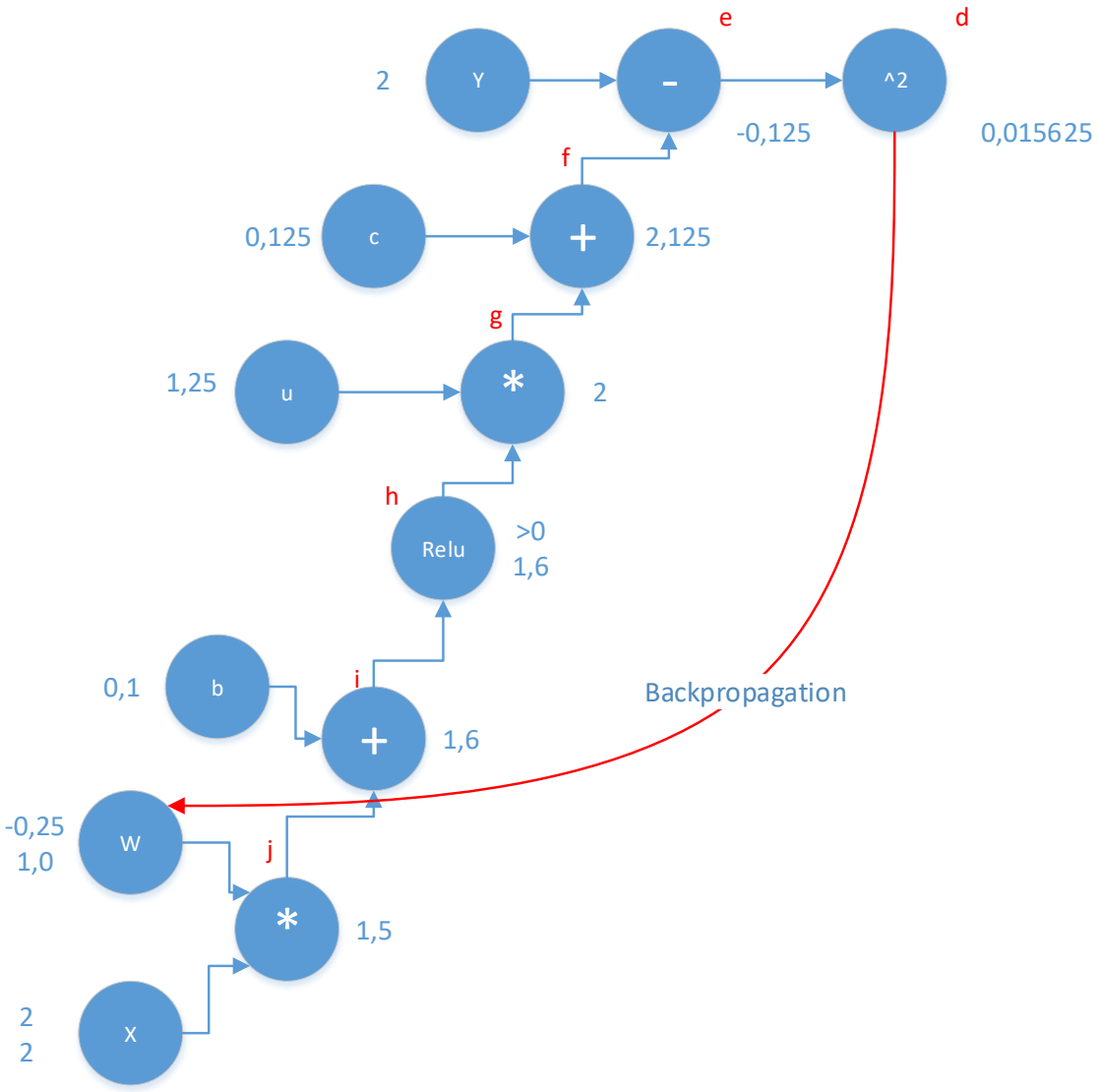

Abbildung 23: Der Berechnungsgraph

Der Backpropagation Algorithmus rechnet rekursiv das heißt der Algorithmus fängt bei dem Fehlerwert des Knoten d an und rechnet rückwärts gerichtet durch das Netzwerk. Die Idee hinter dem Backpropagation Algorithmus ist es die Teilgradienten der einzelnen Konten von hinten beginnend zu berechnen, um somit die Gewichte updaten zu können. Damit der Backpropagation Algorithmus funktioniert müssen die einzelnen Rechenoperationen jedes Knoten bekannt sein. Diese Rechenoperationen befinden sich in der Tabelle 3. In dieser Tabelle befindet sich für jeden Knoten von d bis j die Berechnung und die dazugehörige Ableitung nach der Variablen.

$d = e^2 \ \frac{dd}{de} = 2e$	$e = y - f \ \frac{de}{dy} = 1, \frac{de}{df} = -1$	$f = c + g \ \frac{df}{dc} = 1, \frac{df}{dg} = 1$
$g = u * h \ \frac{dg}{du} = h, \frac{dg}{dh} = u$	$h = relu(i) \frac{dh}{di} = \begin{cases} 0, i \leq 0 \\ 1, i > 0 \end{cases}$	$i = b * j \ \frac{di}{db} = 1, \frac{di}{dj} = 1$
$j = w * x \ \frac{dj}{dw1} = x1, \frac{dj}{dw2} = x2, \qquad \frac{dj}{dx1} = w1, \frac{dj}{dx2} = w2$		

Tabelle 3: Aufstellungen der Formeln und der zugehörigen Ableitungen

Diese einzelnen Ableitungen der Tabelle 3 sind nötig, weil die direkten Ableitungen von $\frac{dd}{dw1}$ und $\frac{dd}{dw2}$ nicht möglich sind. Es muss der gesamte Pfad von dem Konten d bis zu den Gewichten des Knoten w abgeleitet werden, um die Gewichte des Kontens W mit dem Stochastic Gradient Descent Verfahren zu updaten. Die Nachfolgenden Berechnungen der Konten e bis j.

$$Berechnung\ für\ Knoten\ e = \frac{dd}{de} = 2 * -0{,}125 = -0{,}25$$

$$Berechnung\ für\ Knoten\ f = \frac{dd}{df} = \frac{dd}{de} * \frac{de}{df} = -0{,}25 * (-1) = 0{,}25$$

$$Berechnung\ für\ Knoten\ c = \frac{dd}{dc} = \frac{dd}{df} * \frac{df}{dc} = 0{,}25 * 1 = 0{,}25$$

$$Berechnung\ für\ Knoten\ i = \frac{dd}{dg} = \frac{dd}{df} * \frac{df}{dg} = 0{,}25 * 1 = 0{,}25$$

$$Berechnung\ für\ Knoten\ h = \frac{dd}{dh} = \frac{dd}{dg} * \frac{dg}{dh} = 0{,}25 * 1{,}25 = 0{,}3125$$

$$Berechnung\ für\ Knoten\ u = \frac{dd}{du} = \frac{dd}{dg} * \frac{dg}{du} = 0{,}25 * 1{,}6 = 0{,}4$$

$$Berechnung\ für\ Knoten\ i = \frac{dd}{di} = \frac{dd}{dh} * \frac{dh}{di} = 0{,}3125 * 1 = 0{,}3125$$

$$Berechnung\ für\ Knoten\ b = \frac{dd}{db} = \frac{dd}{di} * \frac{di}{db} = 0{,}3125 * 1 = 0{,}3125$$

$$Berechnung\ für\ Knoten\ j = \frac{dd}{dj} = \frac{dd}{di} * \frac{di}{dj} = 0{,}3125 * 1 = 0{,}3125$$

Formel 23: Berechnungen der einzelnen Konten e bis j

$$Berechnung\ für\ den\ Gradient\ 1 = \frac{dd}{dw1} = \frac{dd}{dj} * \frac{dj}{dw1} = 0{,}3125 * 2 = 0{,}625$$

$$Berechnung\ für\ den\ Gradient\ 2 = \frac{dd}{dw2} = \frac{dd}{dj} * \frac{dj}{dw2} = 0{,}3125 * 2 = 0{,}625$$

Formel 24: Berechnungen für die Gradienten 1 und 2

Mit den Berechnungen für die neuen Gradienten lassen sich nun die Gewichte nach dem Stochastic Gradient Descent berechnen. Die nun unten aufgeführte Berechnung zeigt die neuen Gewichte w1 und w2.

$$Berechnung\ für\ Gewicht\ w1 = 2 - \boldsymbol{\eta} * \frac{dd}{dw1} = 2 - 0{,}1 * 0{,}625 = 1{,}9375$$

$$Berechnung\ für\ Gewicht\ w2 = 2 - \boldsymbol{\eta} * \frac{dd}{dw2} = 2 - 0{,}1 * 0{,}625 = 1{,}9375$$

Formel 25: Berechnung der neuen Gewichte w1 und w2

Mit dem Update der Gewichte wurde nun ein kompletter Trainingsdurchlauf in diesem Netzwerk mit dem Stochastic Gradient Descent Verfahren durchgeführt. Im nächsten Schritt wird mit den neuen Gewichten wieder durch das gesamte Netzwerk propagiert, um zu sehen wie der Fehler in dem neuen Trainingsdurchlauf ist.

$$(1{,}9375 * 2) + (1{,}9375 * 2) = 7{,}75$$

$$7{,}75 + 0{,}3125 = 8{,}0625$$

$$relu > 0 = 8{,}0625$$

$$0{,}4 * 8{,}0625 = 3{,}225$$

$$0{,}25 + 3{,}225 = 3{,}475$$

Formel 26: Berechnung bis zur Outputschicht

$$2 - 3{,}475 = -1{,}475$$

$$-1{,}475^2 = -2{,}175625$$

Nach dem zweiten Trainingsdurchlauf ist der Fehler noch größer geworden. Jetzt würde wieder der Backpropagation Algorithmus angewandt werden, um die Gewichte zu updaten. Dieses Beispiel so dazu dienen wie der Backpropagation Algorithmus funktioniert.

10. Overfitting

Beim Trainieren von neuronalen Netzen mit einer großen Zahl an Parametern und der damit eingehenden Komplexität des Netzwerkes kann es leicht zu einer Über- oder Unteranpassung kommen. Eine Überanpassung wird durch eine hohe Varianz bei einem niedrigen Bias

ausgelöst. Das Problem der Unteranpassung wird durch einen zu hohen Bias ausgelöst. Eine Überanpassung ist eines der Hauptprobleme beim Training von Neuronalen Netzen, dieser tritt deutlich häufiger auf als die Unteranpassung. Je stärker das Rauschen der Trainingsdaten, umso größer ist die Gefahr des Übertrainierens. Neuronale Netze neigen stärker zu einer Überanpassung, je mehr Verbindungen und Neuronen im Netzwerk enthalten sind. Mit zunehmender Größe eines neuronalen Netzwerkes steigt auch die Fähigkeit komplexere Sachen zu erlernen, damit steigt aber auch die Gefahr der Überanpassung.

Um auf das Problem der Überanpassung reagieren zu können, wird ein weiterer Datensatz, der Testdatensatz verwendet. Dieser Datensatz wird nach jeder Epoche des Trainings durch das Netzwerk propagiert. Der damit ermittelte Fehlerwert wird nicht wie bei dem Trainingsdatensatz zur Anpassung der Gewichte verwendet. Sollte eine Überanpassung vorliegen äußert sich dies in einem Anstieg des Fehlers im Testdatensatzes bei einer weiteren Abnahme des Fehlers beim Trainingsdatensatzes. Um die optimalen Trainingsepochen zu bestimmen sollte das Training bei dem kleinsten Fehlerwert des Testdatensatzes abgebrochen werden. Für eine optimale Eignungsprüfung wird ein dritter Datensatz der Validierungsdatensatz, angewendet. Dieser Datensatz wird weder zur Anpassung der Gewichte noch zur Bestimmung des Abbruchs des Trainings verwendet. Dieser Datensatz bietet dadurch die Möglichkeit einer ungebundenen Überprüfung der Performance des Neuronalen Netzwerkes. Bei der Aufteilung der verschiedenen Datensätze muss sichergestellt werden, dass alle Teildatensätze die gleichen Eigenschaften aufweisen damit die Ergebnisse nicht verfälscht werden.

In der Abbildung 24 ist zu sehen wie der Fehler im Trainingsdatensatz mit der Anzahl der Epochen immer weiter abnimmt und sich der Null immer weiter annähert.

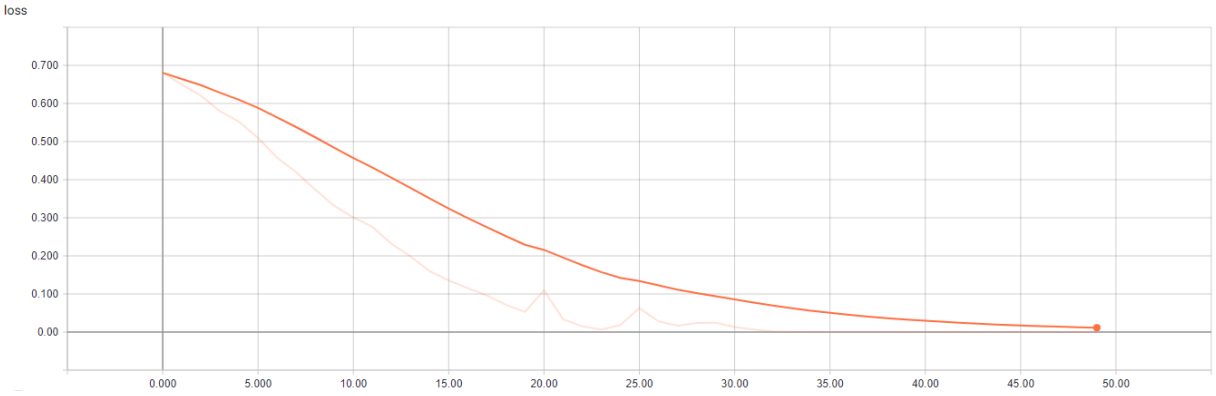

Abbildung 24: Trainingsfehler

Die Abbildung 25 zeigt den Fehler auf dem Validationsdatensatz, der bis Epoche 16 kontinuierlich abnimmt. Ab der Epoche 16 nimmt der Fehlerwert bis zum Ende des Trainings wieder zu. In Epoche 50 ist der Fehlerwert sogar höher als zum Beginn des Trainings. Die untenstehende Grafik zeigt eine klare Überanpassung. Das Training hätte bei Epoche 16 enden müssen.

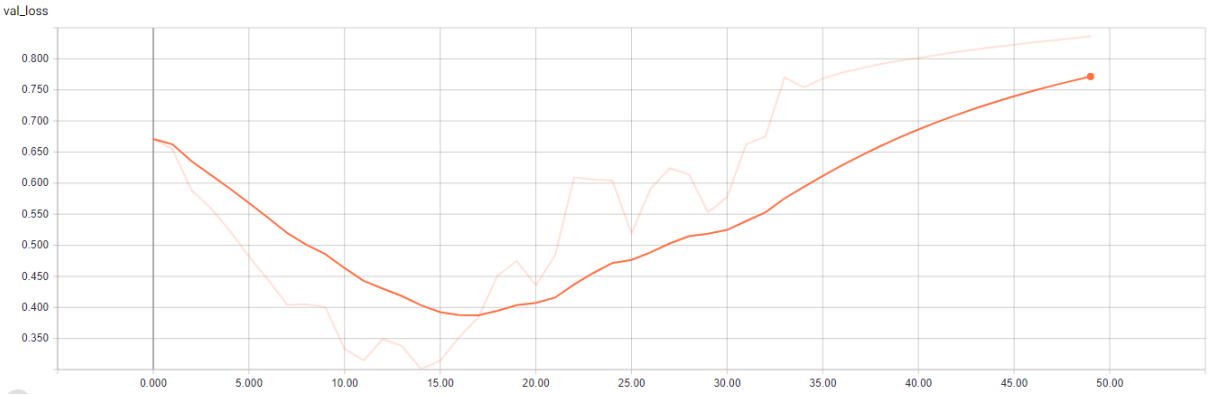

Abbildung 25: Validationsfehler mit Overfitting

10.1 EarlyStopping

Das EarlyStopping ist eine Methode, welche den Validationsdatensatz überwacht. Wenn sich innerhalb einer vorher definierten Anzahl an Epochen keine Verbesserung des Validationsfehlers ergeben hat, wird das Training abgebrochen und die Gewichte der Epoche wiederhergestellt, bei der noch eine Verbesserung eingetreten ist. Somit kann während des Trainings überprüft werden, ob das Netzwerk in das Overfitting geraten wird.

11. Optimierungsverfahren

11.1 Dropout

Große neuronale Netze mit einer großen Anzahl an Trainierbaren Parametern sind sehr leistungsfähige Lernsysteme. Das Overfitting ist jedoch in solchen Netzen ein sehr ernstes Problem. Eine Möglichkeit dem entgegenzuwirken, ist der Dropout. Die Idee hinter dem Dropout ist es, während des Trainings zufällige Neuronen abzuschalten. Dadurch wird verhindert, dass sich die Neuronen zu stark an den Trainingsdatensatz anpassen. Außerdem wird durch diese Methode eine verbesserte Trainingsgeschwindigkeit erreicht. Durch den Dropout wird die Neuroneninteraktion reduziert, was dazu führt, dass das Neuronale

Netzwerk besser auf neue Daten reagieren kann. Nach dem Training werden die entfernten Neuronen mit ihren Ursprünglichen Gewichten wieder aktiviert.

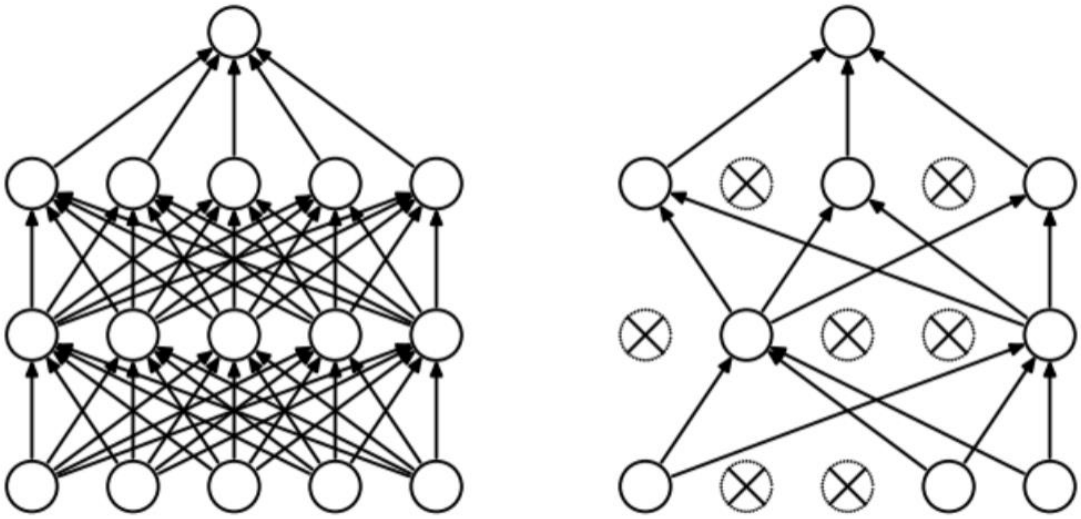

Abbildung 26: Neuronales Netzwerk ohne und mit Dropout

*https://cdn-images-1.medium.com/max/800/1*iWQzxhVlvadk6VAJjsgXgg.png*

11.2 Batch-Normalisierung

Die Batch Normalisierung ist eine trainierbare Normalisierung der Daten der Hiddenlayer. Damit werden die Daten der vorrangegangenen Schichten Batchweise normalisiert. Dadurch dass die Daten ohne die Batchnormalisierung zwischen den einzelnen Schichten nicht Normalisiert vorliegen, muss sich das Netzwerk immer an eine neue Verteilung der Daten anpassen, was zu einer Verlangsamung des Trainings führt. Dieses Problem wird als Kovariatsverschiebung bezeichnet. Die Batch-Normalisierung ist eine Methode, mit der die Eingaben jeder Schicht normalisiert werden können, um das Problem der Hiddenlayer der zu bekämpfen. Für die Werte der vergangen Layer können keine globalen Mittelwerte oder Varianzen berechnet werden, diese Werte müssen angenähert werden. Dies erfolgt mit der folgenden Berechnungsvorschrift.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Formel 27: Batch Mittelwert

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Formel 28: Batch Varianz

$$\overline{x_\iota} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Formel 29: Batch Normalisierung

$$y_i = \Upsilon \overline{x_\iota} + \beta$$

Formel 30: Skalen und Verschiebungsparameter

Damit Formel 29 eine einfache Normalisierung von x durchgeführt wird, kann es zu einer Veränderung der Darstellung der betroffenen Schicht kommen. Weiterhin wird in der Batch Normalisierung γ eingeführt. Dieses γ stellt sicher das die Transformation eine Identitätstransformation darstellt und somit jedes x vollständig durch γ ersetzt wird. Somit wird mit der Batch Normalisierung der Skalen- und der Verschiebungsparameter γ β eingeführt. Diese Parameter sind ebenfalls erlernbar. Durch dieses Parameter steigen die zu erlernenden Parameter an, dies wird aber mit einer Erhöhung der Lerngeschwindigkeit und insgesamt weniger Arbeitsschritte kompensiert.

$$E_x = \frac{1}{m}\sum_{i=1}^{j} \mu_B^{(i)}$$

Formel 31: Inferenz Mittelwert

$$Var_x = \left(\frac{m}{m-1}\right)\frac{1}{m}\sum_{i=1}^{j} \sigma_B^{2(i)}$$

Formel 32: Inferenz Varianz

$$y = \frac{\Upsilon}{\sqrt{Var_x + \epsilon}}x + \left(\beta + \frac{\Upsilon E_x}{\sqrt{Var_x + \epsilon}}\right)$$

Formel 33: Inferenz Skalen und Verschiebungsparameter

12. Umsetzung

Das im Rahmen dieser Arbeit zu entwickelnde Convolutional Neuronal Network soll mit Hilfe eines aktuellen Frameworks durchgeführt werden. Der zu entwickelnde Prototyp verwendet als Grundlegende Daten den kaggle dogs and cats Datensatz. Dieser Datensatz beinhaltet eine Sammlung 25000 Farbbilder aufgeteilt auf 2 Klassen. Pefinder.com stellte Microsoft Research

über 3 Millionen Bilder zur Verfügung. Diese Bilder wurden von Menschen in Tausenden Tierheimen in den USA manuell klassifiziert. Über die kaggle.com wurde ein Teil dieses Datensatzes für die nicht kommerzielle Forschung veröffentlicht. Diese Bilder liegen in keiner einheitlichen Größe vor und müssen vor der Verwendung in dem Convolutional Neuronal Network erstmal auf eine identische Größe gebracht werden. Die Klassen des Datensatzes besitzen die Titel Cat und Dog. Die gesamten Bilder liegen im JEPG Format vor und weisen als Beschriftung eine Numerische Aufzählung beginnend mit Null, auf. Der gesamte Datensatz hat eine Dateigröße von 1,93 Gigabyte.

Der hier in dieser Arbeit entwickelte Prototyp wird in der Programmiersprache Python entwickelt. Um die Programmierung des Prototypen zu erleichtern, finden die Frameworks Tensorflow und Keras Anwendung in dieser Arbeit. Tensorflow ist eine Open-Source Programmbibliothek für künstliche Intelligenzen, die vom Google-Brain-Team für die interne Verwendung entwickelt wurde, später wurde Tensorflow unter der Apache-2.0 Open Source Lizenz veröffentlicht. Mit Tensorflow lassen sich mathematische Berechnungen mit Hilfe von Datenflussgraphen realisieren.

Das Framework Keras ist auch eine Open Source Deep-Learning-Bibliothek, die im März 2015 unter der MIT Lizenz veröffentlicht wurde. Wie auch Tensorflow ist Keras in Python geschrieben. Keras bietet eine einheitliche Schnittstelle für verschiedene Deep-Learning Frameworks wie Tensorflow, Microsoft Cognitive Toolkit und Theano. Seit der Tensorflow Version 1.4 gehört Keras zur Core API von Tensorflow, dieses besteht aber eigenständig weiter da es als Schnittstelle für viele Bibliotheken gedacht war.

Um die Performance des zu entwickelnden Convolutional Neuronal Network zu verbessern, wurde dieses auf einer Grafikkarte trainiert. Hierdurch wird der Speicher der Grafikkarte zum limitierenden Faktor bezüglich der Größe des Netzes, Größe der Eingangsbilder und der Größe der Batches. Die eingesetzte Grafikkarte ist eine Nvidia GTX 965m Refresh. Der Grafikchip dieser Karte (GM204) basiert auf der Maxwell Architektur von Nvidia und besitzt 4 Gigabyte GDDR 5 Arbeitsspeicher.

12.1 Vorverarbeitung des Datensatzes

Um eine höhere Performance beim Training zu erreichen, werden die einzelnen Bilder der zwei Klassen in einem Array Byteweise abgespeichert. Dadurch muss das Neuronale Netz beim Training nicht jedes einzelne Bild öffnen und wieder schließen, was die Zeit des Trainings erheblich beschleunigt. Bevor die Bilder in dem Array gespeichert werden können, wird der Datensatz auf Fremddateien überprüft. Diese Fremddateien können nicht eingelesen werden und müssen gelöscht werden, bevor die Bilder in dem Array gespeichert werden. Die Speicherung erfolgt in einem Numpy Array wobei die Bilder in einem Array Namens x.npy liegen und die dazugehörigen Labels in einem weiteren Array Namens y.npy. Weiterhin werden in der Vorbereitung alle Bilder auf dieselbe Größe konvertiert, da das Neuronale Netz nur Bilder mit der gleichen Auflösung erwartet. In der ersten Umsetzungsphase werden 3 verschieden Arrays mit unterschiedlichen Bildgrößen erstellt, um zu überprüfen ob die Auflösung der Bilder einen nennenswerten Unterschied in der Performance des Netzwerkes ergibt. Die Auflösung der Bilder im ersten Array haben eine Größe von 32x32 Pixeln die Bilder im zweiten Array haben eine Auflösung von 64x64 Pixeln und die Bilder im letzten Array haben die Auflösung von 96x96 Pixeln.

Auf noch größere Bilddateien wurde in dieser Arbeit verzichtet da die Leistung der eingesetzten Grafikkarte für noch größere Bilddateien nicht ausreichte. Es wurde am Anfang der Umsetzung an einem Prototyp gearbeitet, der Bilder in der Dimension von 128x128 Pixeln verarbeiten kann. An diesem Prototyp wurde aber nicht weitergearbeitet, da die Bearbeitungszeit dieser Arbeit begrenzt ist und die Trainingszeit pro Epoche in keinem Verhältnis zur angegebenen Zeit stand.

Abbildung 27: Originalbild aus dem Datensatz in der Auflösung 500x258

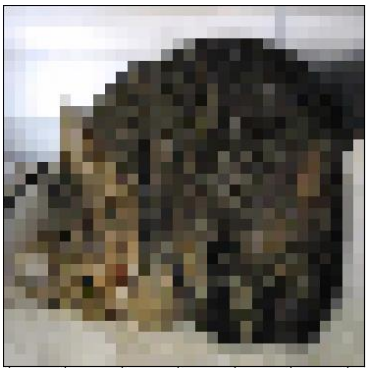

Abbildung 28: Auflösung 32x32

Abbildung 29: Auflösung 64x64

Abbildung 30: Auflösung 96x96

Weiterhin wurde in der Vorverarbeitung der Datensatz in einen Testdatensatz und in einen Trainingsdatensatz aufgeteilt. Nach der Aufteilung besitzt der Trainingsdatensatz 70 % des vollständigen Datensatz und der Testdatensatz hat nach der Aufteilung eine Größe von 30 % gegenüber dem vollständigen Datensatz.

Da ein Convolutional Neuronal Network nur Bilder in der gleichen Größe verarbeiten kann, werden in der Vorverarbeitung die Bilder in die vorher festgelegte Größe konvertiert. Außerdem werden aus Labels des Trainings- und Testdatensatzes One-Hot Felder erstellt. Diese Felder dient dem Netzwerk später für die Klassifikation. Die One-Hot Kodierung erstellt aus dem Label, dass eine Integer Zahl ist ein Feld der Größe 2. Um das One-Hot Feld später für die Klassifikation zu verwenden, muss der Datentyp von Integer in Float32 geändert werden, da der Datentyp Integer keine Gleitkommazahlen verwenden kann. Das One-Hot Feld speichert zu jedem Bild die entsprechende Zugehörigkeit in Prozent zu den einzelnen Klassen. Um später die Klasse eines Bildes zu bestimmen wird auf das One-Hot Feld später noch die Funktion arg_max angewandt. Das Ergebnis der Funktion liefert die Position des Maximalen

Wertes. Die Abbildung 30 zeigt ein Bild, das mit dem Convolutional Neuronal Network klassifiziert wurde. Wobei das One-Hot Feld mit ausgegeben wurde. Das Tier auf dem Bild gehört zu einer Wahrscheinlichkeit von 55,08565 % zu der Klasse Cat und zu 44,914347 % zur Klasse Dog.

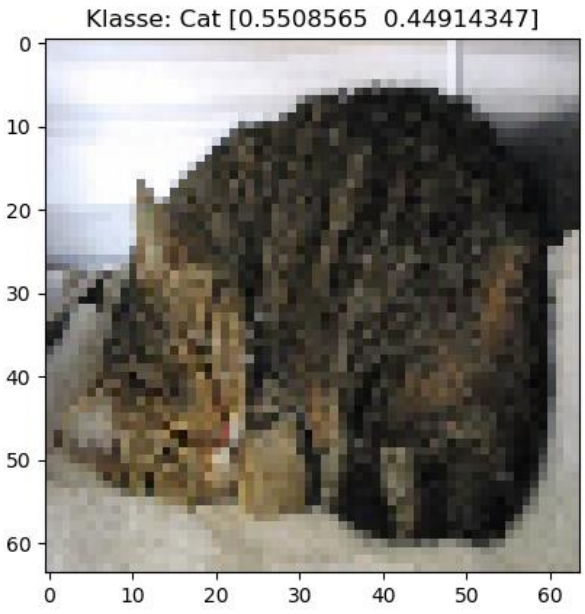

Abbildung 31: Klassifiziertes Bild mit One-Hot Feld

Umso größer der Datensatz zum Anlernen eines Convolutional Neuronal Networks ist umso höher kann die Genauigkeit eines Convolutional Neuronal Network sein. Um den Datensatz künstlich zu erweitern bietet Keras die Option eines ImageDataGenerators an. Mit dieser Option ist man in der Lage den Trainingsdatensatz durch das Verändern der Bilder künstlich zu erweitern. Dies bietet den Vorteil, dass man den vorliegenden Datensatz einfach um mehrere Bilder erweitern kann, ohne diese Bilder selbst zu erstellen oder manuell in den Datensatz einpflegen muss. Der in der Arbeit verwendete ImageDataGenerator wird verwendet um Bilder in einer zufälligen weise um in die Bilder zu zoomen, um die Bilder vertikal und horizontal zu drehen und um die Bilder in der Höhe und weite zu verschieben. In dem ersten Prototyp wird anhand der Datenerweiterung gezeigt, dass durch die Verwendung eines ImageDataGenerators die Genauigkeit des Netzwerkes gesteigert werden kann.

12.2 Der erste Prototyp

In diesem Unterkapitel der Arbeit wird der erste Prototyp vorgestellt. Hier werden die verschiedenen Bildgrößen gegeneinander antreten und anhand ihrer Performance und Genauigkeit gemessen. Weiterhin werden hier die verschieden Aktivierungsfunktionen, die in Kapitel 4 beschrieben wurden, verwendet, um zu sehen welche der Funktionen die beste Genauigkeit in dem Netzwerk liefert. Auch wird der erste Prototyp dafür verwendet, um die Optimizer gegeneinander antreten zu lassen und ob die Bilder Größe einen Einfluss auf die Performance und die Genauigkeit des Neuronalen Netzes hat.

Der erste Prototyp besteht aus einer Eingabeschicht, 2 Convolutional Blöcken und einer Ausgabeschicht. Ein Convolutional Block besteht aus einem Convolutional Layer, einer Aktivierungsfunktion, einem weiteren Convolutional Layer der wieder gefolgt von einer Aktivierungsfunktion ist und einem Pooling Layer. Die Convolutional Layer im ersten Block beinhalten 32 Filter und die Convolutional Layer im zweiten Block enthält 64 Filter. Die Lernrate lag bei allen Durchläufen fest bei 0,002.

Im ersten Schritt wurden die unterschiedlichen Datensätze in den Größen 32x32, 64x64 und 96x96 Pixeln erzeugt und mit den verschiedenen Aktivierungsfunktionen und den verschiedenen Optimizern auf ihre Performance und Genauigkeit hin untersucht. Die Abbildung 32 zeigt den gesamten Aufbau des Netzwerkes von der Eingabeschicht bis zur Ausgabeschicht inklusive des Optimizers für die Fehlerrückführung.

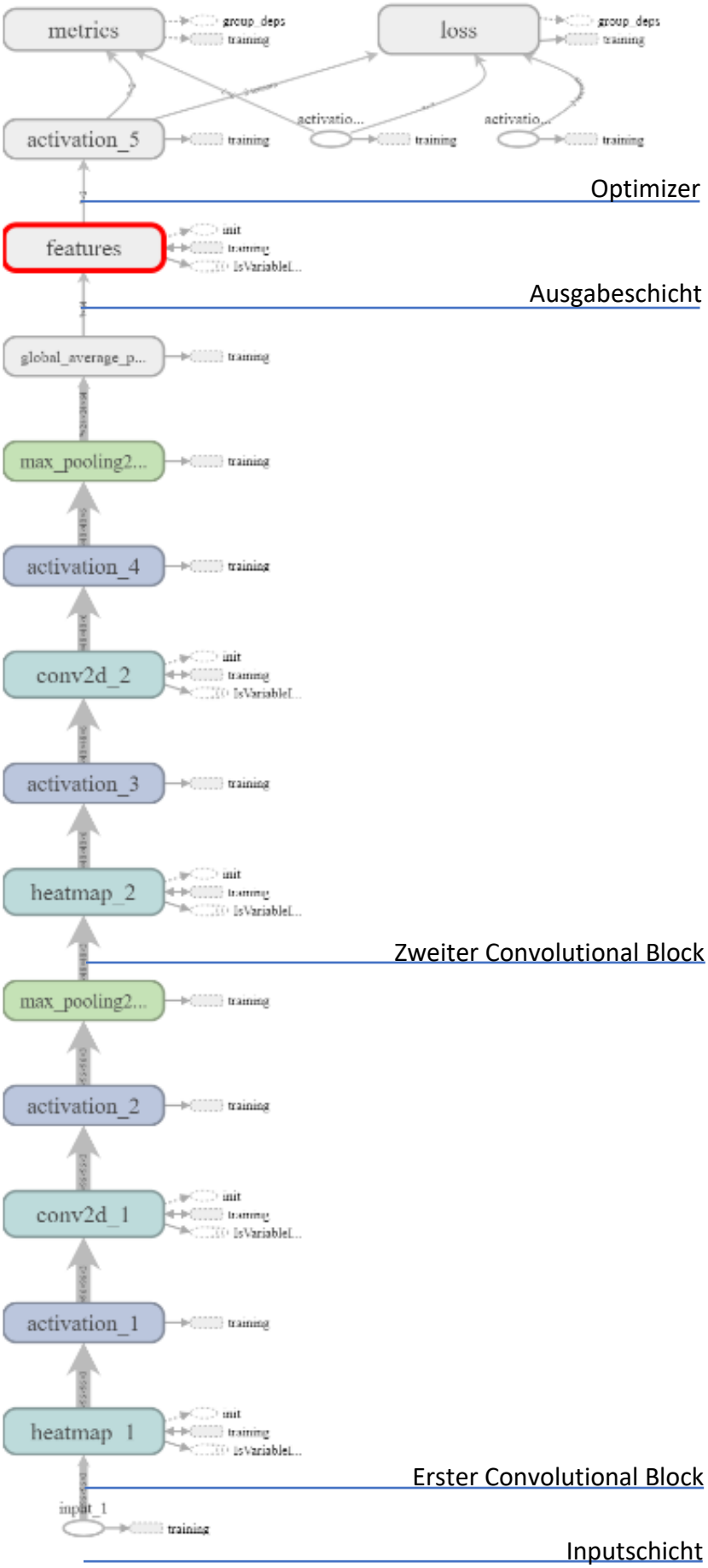

Abbildung 32: Modell des ersten Prototyps

12.2.1 Testdurchläufe mit dem Adam Optimizer

Im ersten Teil dieses Testdurchlaufs wurde das Netzwerk mit dem 32x32 Pixel großen Datensatz zur Bestimmung der Genauigkeit verwendet. Die unten aufgeführte Tabelle 4 zeigt die Genauigkeit der verschiedenen Aktivierungsfunktionen auf dem Test Set, das eine Größe von 7500 Bilder hat. Die Trainingszeit lag bei allen Aktivierungsfunktionen in dem 32x32 Pixel großen Datensatz bei circa 6 Sekunden pro Epoche somit ergibt sich eine Trainingszeit von 2:30 Minuten pro Testdurchlauf.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	81,58 %	6 Sekunden
Sigmoid	60,72 %	6 Sekunden
Tanh	79,34 %	6 Sekunden

Tabelle 4: Performance des Netzwerkes (32x32) mit dem Optimizer Adam und den verschiedenen Aktivierungsfunktionen

Die Abbildung 33 zeigt die Genauigkeit der einzelnen Epochen auf dem Validation Set der einzelnen Aktivierungsfunktionen unter dem Adam Optimizer. Wie auch schon in der Tabelle 4 ersichtlich ist der Durchlauf mit der ReLu (Orange) am genausten mit 81,58%. Der Test mit der Tanh (Rot) zeigt eine ähnlich gute Genauigkeit, wobei die ReLu um 2,24% besser abschneidet. Die Sigmoid hingegen kam lediglich auf eine Genauigkeit von 60,72 %.

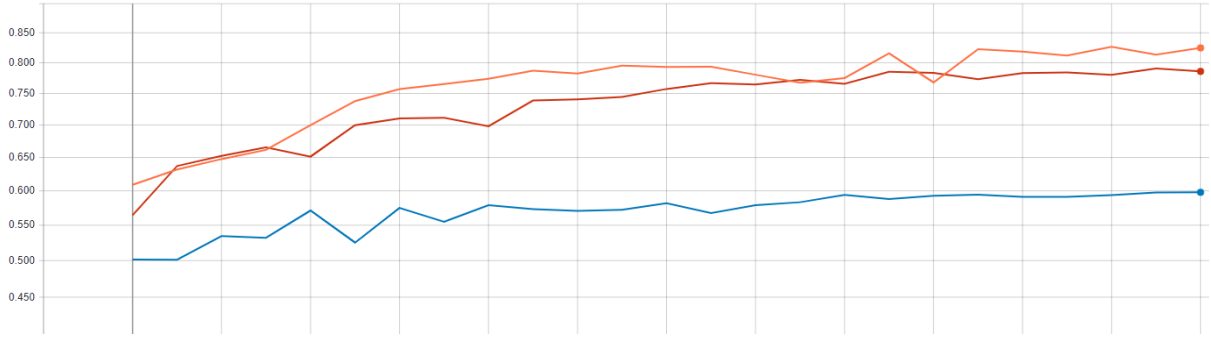

Abbildung 33: Performance der Aktivierungsfunktionen in der Anwendung im Validation Set auf dem 32x32 Datensatz

Im zweiten Teil wurden die verschiedenen Aktivierungsfunktionen mit dem 64x64 Pixel großen Datensatz getestet. Auch hier ist die Sigmoid wieder das Schlusslicht mit 56,77% was nochmal eine Verschlechterung von 3,95% ergab. Die Tests mit der ReLu und die Tanh ergab auch eine Verschlechterung gegenüber dem Datensatz mit 32x32 Pixeln von 3,95%

beziehungsweise 2,44%. Die Trainingszeit bei dem 64x64 großen Datensatz lag pro Epoche bei 16 Sekunden, somit ergab sich eine Trainingszeit von 6:40 Minuten.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	76,60 %	16 Sekunden
Sigmoid	56,77 %	16 Sekunden
Tanh	76,90 %	16 Sekunden

Tabelle 5: Performance des Netzwerkes (64x64) mit dem Optimizer Adam und den verschiedenen Aktivierungsfunktionen

Die Abbildung 34 zeigt auch wieder den Trainingsfortschritt der einzelnen Epochen unter Verwendung der verschieden Aktivierungsfunktionen. Der Grüne Graph zeigt den Fortschritt der ReLu. Der Pinke Graph zeigt den Fortschritt der Sigmoid Aktivierungsfunktion und der Türkise Graph zeigt den Fortschritt der Tanh Aktivierungsfunktion. Hierbei lässt sich wieder schön sehen, dass das Training mit der Sigmoid Aktivierungsfunktion keinen Nennenswerten Fortschritt über das gesamte Training bietet. Die ReLu und die TanH Funktion zeigen auch hier eine gute Verbesserung wobei die ReLu in der letzten Epoche einen Genauigkeits Rückgang von circa 4% aufweist.

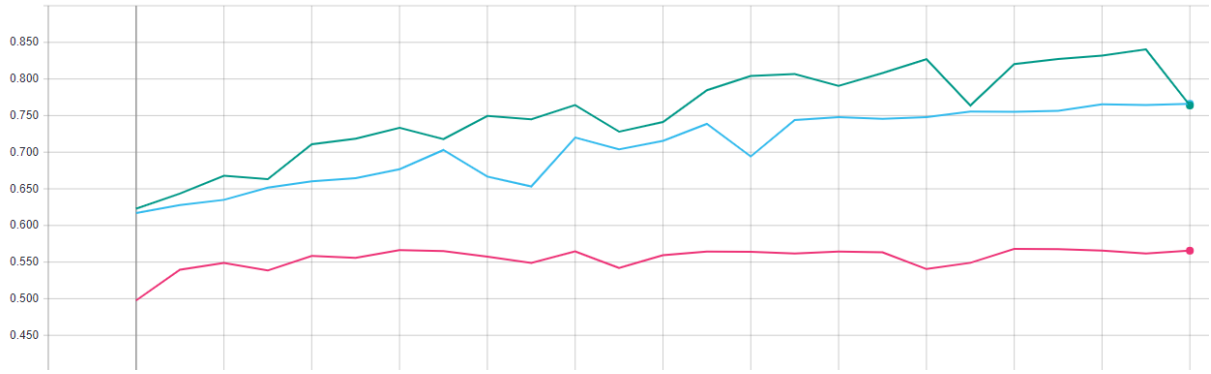

Abbildung 34: Performance der Aktivierungsfunktionen in der Anwendung im Validation Set auf dem 64x64 Datensatz

Im dritten Testdurchlauf wurde der Datensatz mit den Bildgrößen von 96x96 Pixeln verwendet. Hierbei lag die Trainingszeit pro Epoche bei 36 Sekunden was eine gesamte Trainingszeit pro Aktivierungsfunktion von 15 Minuten hatte. Hierbei konnte festgestellt werden, dass die Genauigkeit der ReLu Aktivierungsfunktion wieder zugenommen hat, auf 82,85%, und dies der höchste Wert aller getesteten Datensätze ist. Die Tanh Aktivierungsfunktion hat im Vergleich zum 64x64 Datensatz einen Verlust der Genauigkeit von 5,82%. Das Schusslicht ist in diesem Test ist auch wieder die Sigmoid mit 57,59%.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	82,85 %	36 Sekunden
Sigmoid	57,59 %	36 Sekunden
Tanh	71,08 %	36 Sekunden

Tabelle 6: Performance des Netzwerkes (96x96) mit dem Optimizer Adam und den verschiedenen Aktivierungsfunktionen

Wie auch schon weiter oben, zeigt die Abbildung 35 den Trainingsfortschritt der einzelnen Prototypen mit den verschiedenen Aktivierungsfunktionen. Der Orange Graph zeigt hier den Fortschritt der ReLu Aktivierungsfunktion. Die Tanh Aktivierungsfunktion wird durch den roten Graphen gekennzeichnet und die Sigmoid Aktivierungsfunktion wird durch den blauen Graphen dargestellt. Die ReLu und die Tanh Aktivierungsfunktion weisen eine stetige Verbesserung der Genauigkeit über den gesamten Zeitraum des Trainings auf. Die Sigmoid Aktivierungsfunktion hat auch bei diesem Datensatz keinen signifikanten Fortschritt in der Genauigkeit erreichen können.

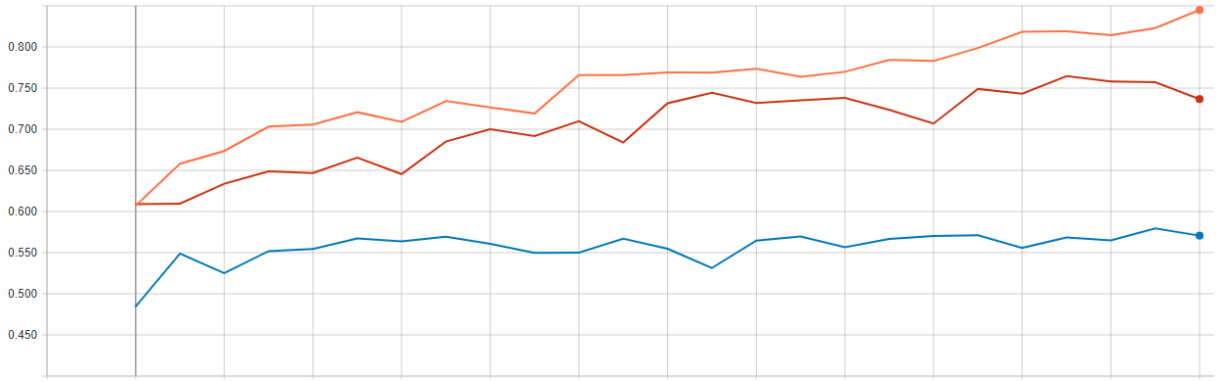

Abbildung 35: Performance der Aktivierungsfunktionen in der Anwendung im Validation Set auf dem 96x96 Datensatz

12.2.2 Testdurchläufe mit dem SGD Optimizer

Im nachfolgenden werden die Ergebnisse der Tests mit dem SGD Optimizer präsentiert. Wie auch schon bei dem Adam Optimizer werden die verschiedenen Aktivierungsfunktion auf die zuvor erstellten Datensätze angewandt.

Als erstes wurde der Prototyp mit dem 32x32 Pixel große Datensatz auf die verschiedenen Aktivierungsfunktionen angewandt. Auch bei diesen Durchläufen betrug die Trainingszeit pro Epoche 6 Sekunden, was eine gesamte Trainingszeit von 2:30 Minuten pro Aktivierungsfunktion ergab. Die Ergebnisse der Genauigkeit auf dem Testdatensatz der verschiedenen Aktivierungsfunktionen werden in Tabelle 7 präsentiert.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	59,73 %	6 Sekunden
Sigmoid	49,98 %	6 Sekunden
Tanh	60,87 %	6 Sekunden

Tabelle 7: Performance des Netzwerkes (32x32) mit dem Optimizer SGD und den verschiedenen Aktivierungsfunktionen

Die Abbildung 36 zeigt auch hier den Trainingsfortschritt der einzelnen Aktivierungsfunktion über die 25 Epochen. Die Tanh (Türkis) zeigt bei dem SGD Optimizer die beste Genauigkeit in dem 32x32 Datensatz. Auch die ReLu (Grün) kann einen Lernfortschritt verzeichnen, der aber unter dem Niveau der Tanh liegt. Die Sigmoid (Pink) kann mit den vorher fest definierten Parametern keinen keine Lernfortschritt verzeichnen. Bei einem Klassifikationsproblem mit 2 Klassen beträgt Raten 50%. In diesem Durchlauf hat die Sigmoid sogar diesen Wert um 0,02% unterschritten.

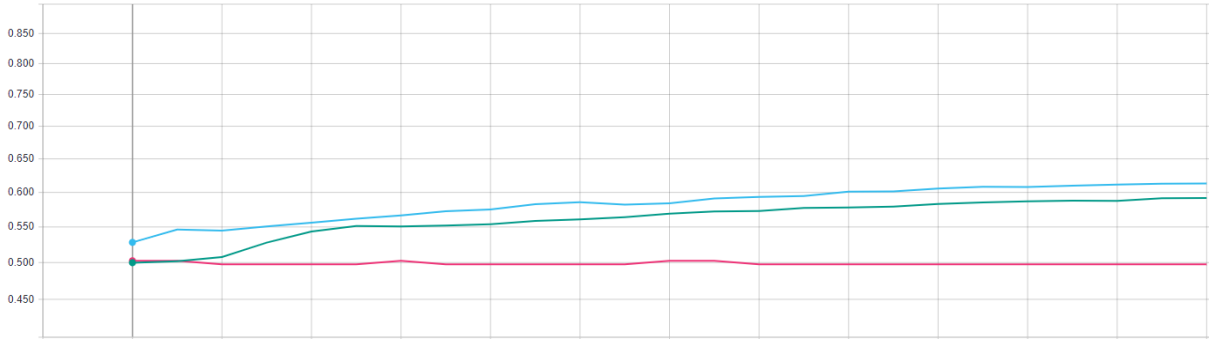

Abbildung 36: Performance der Aktivierungsfunktionen in der Anwendung im Validation Set auf dem 32x32 Datensatz

Im zweiten Durchlauf wurden die Aktivierungsfunktionen auf dem 64x64 Pixel großen Datensatz angewandt. Hierbei ergab sich eine Trainingszeit von 36 Sekunden pro Epoche was eine gesamte Trainingszeit pro Aktivierungsfunktion von 15 Minuten ergab. Auch hier zeigte sich ein ähnliches Bild der Genauigkeit wie bei dem Datensatz mit 32x32 Pixeln. Es gab so gut wie keine Veränderungen in Genauigkeit.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	60,02 %	36 Sekunden
Sigmoid	50,37 %	36 Sekunden
Tanh	60,11 %	36 Sekunden

Tabelle 8: Performance des Netzwerkes (64x64) mit dem Optimizer SGD und den verschiedenen Aktivierungsfunktionen

Die Abbildung 37 zeigt auch hier wieder den Trainingsfortschritt der 25 Epochen auf dem Validation Set. Die ReLu, hier in Orange, zeigte in den ersten 3 Epochen einen guten Trainingsfortschritt, stagnierte aber dann bis zum Ende des Trainings bei circa 60%. Die Tanh Aktivierungsfunktion zeigte einen stetigen Fortschritt bis Epoche 12 und stagnierte ab dann ebenfalls bei circa 60%. Die Sigmoid Aktivierungsfunktion zeigte auch bei dem 64x64 Pixel großen Datensatz keinen Lernfortschritt.

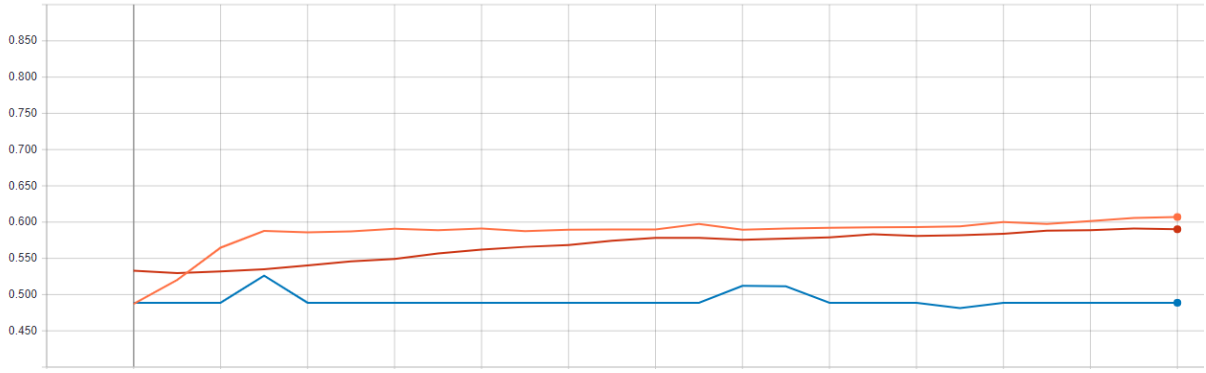

Abbildung 37: Performance der Aktivierungsfunktionen in der Anwendung im Validation Set auf dem 64x64 Datensatz

Im dritten Durchlauf wurden die Aktivierungsfunktionen auf dem 96x96 großen Datensatz angewandt. Auch hier lag die Trainingszeit pro Epoche bei 36 Sekunden was eine Gesamttrainingszeit von 15 Minuten ergab. Die Tabelle 9 zeigt auch hier wieder die Ergebnisse der Genauigkeit der unterschiedlichen Aktivierungsfunktionen.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	57,65 %	36 Sekunden
Sigmoid	50,26 %	36 Sekunden
Tanh	59,51 %	36 Sekunden

Tabelle 9 Performance des Netzwerkes (96x96) mit dem Optimizer SGD und den verschiedenen Aktivierungsfunktionen

Die Abbildung 38 zeigt den Lernfortschritt der einzelnen Aktivierungsfunktionen auf dem Validation Set an. Hier konnten die vorherigen Ergebnisse des SGD Optimizer in ihrer Genauigkeit bestätigt werden. Die Sigmoid in Pink hat hier wieder keine Lernfortschritt erzielen können. Die ReLu in Grün und die Tanh in der Türkis Aktivierungsfunktion konnten sich bis Epoche 16 stetig verbessern, stagnierten aber dann bis zum Ende des Trainings bei circa 56% beziehungsweise 60%.

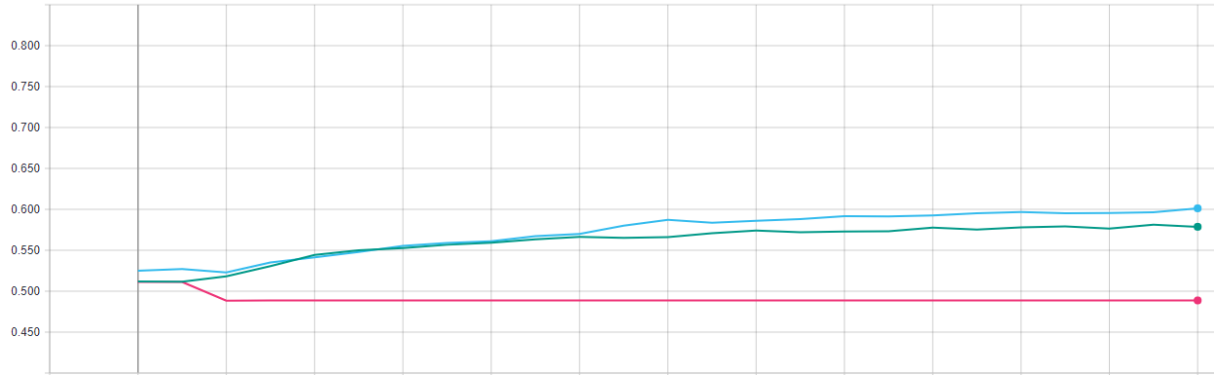

Abbildung 38: Performance der Aktivierungsfunktionen in der Anwendung im Validation Set auf dem 96x96 Datensatz

Bei allen getesteten Variationen der Optimizer und der Aktivierungsfunktionen ergab das der Optimizer SGD und die Aktivierungsfunktion Sigmoid in der Genauigkeit nicht mit dem Adam und der ReLu, Tanh mithalten konnte. In allen Netzwerktestes gab es ein Gewichtanzahl von 65698 die alle trainiert werden konnten.

12.2.2.1 Auswertung der Ergebnisse

Im weiteren Verlauf dieser Arbeit wird auf den Optimizer SGD und die Aktivierungsfunktion Sigmoid verzichtet, da sie in dem ersten Prototyp keine guten Ergebnisse erzielten konnten. Es konnte gezeigt werden, dass die verschiedenen Bildgrößen einen signifikanten Einfluss auf die Performance des Netzwerkes haben, da sich die Trainingszeit pro Epoche und Datensatz immer weiter vergrößert hat. Bei der Genauigkeit hingegen konnte kein beziehungsweise ein nur sehr kleiner Einfluss festgestellt werden. Die Genauigkeit nahm bei dem Adam Optimizer von dem 32x32 Pixel Datensatz zu dem 64x64 Pixel Datensatz um circa 4 % ab, was durch die größeren Bilder zu erwarten war. Dies konnte aber bei dem nächst größeren Datensatz nicht bestätigt werden, weil die Genauigkeit wieder anstieg. Die Ergebnisse des SGD Optimizers waren nahezu über alle Datensatzgrößen identisch schlecht.

12.2.3 Erweiterung des Prototyps

Um die Genauigkeit des Netzwerkes weiter zu verbessern, wurde im nächsten Schritt ein weiteres Convolutional Block mit 96 Filtern hinzugefügt. Durch den neu hinzugefügten Convolutional Block stiegen die Gewichte auf 204194 an. Hierbei wurden wieder die verschiedenen Bildgrößen berücksichtigt. Weitere Änderungen an dem Netzwerk wurden nicht vorgenommen, die Lernrate und Batch Größe blieben unverändert, um die Ergebnisse zu den ersten Tests vergleichbar zu halten. Die Abbildung 39 auf der nächsten Seite zeigt den Prototyp mit dem neu hinzugefügten Convolution Block.

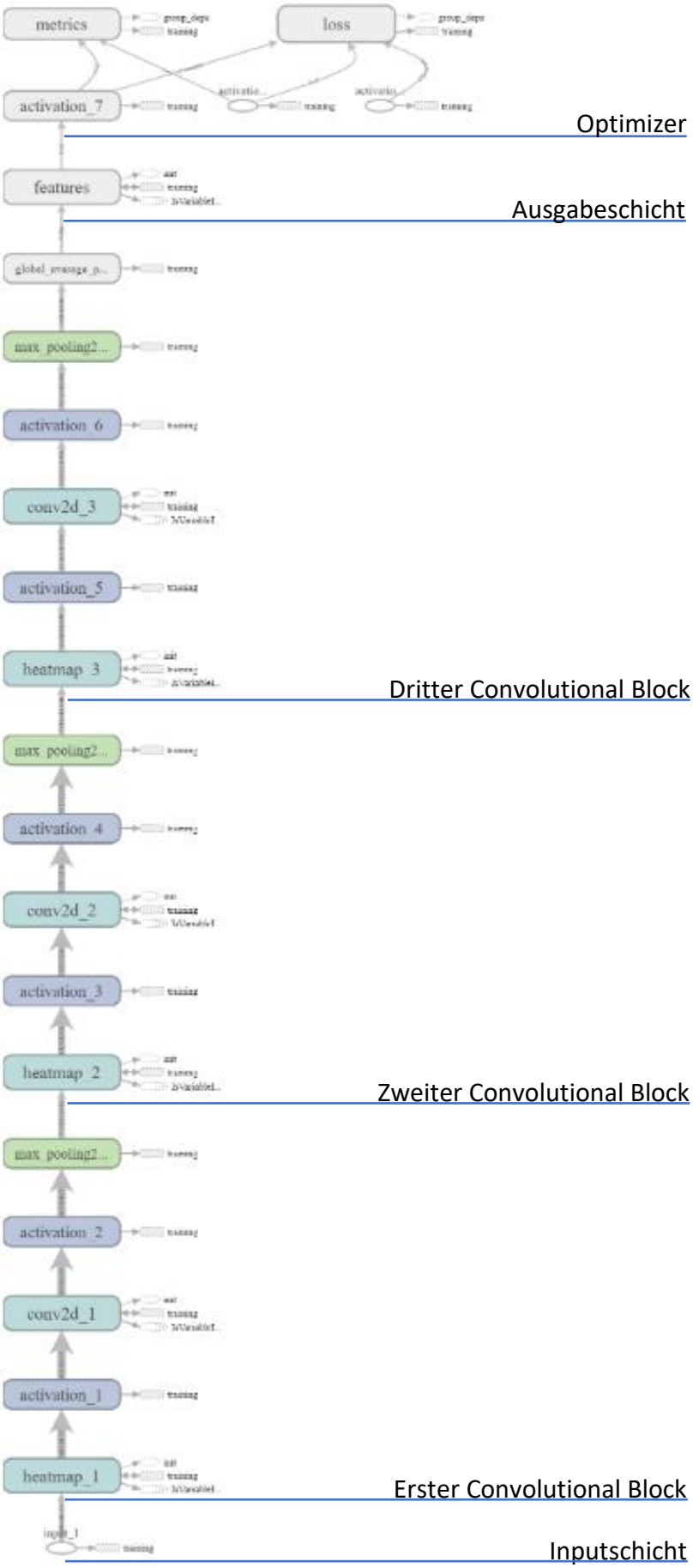

Abbildung 39: Modell des erweiterten Prototypen um einen weiteren Convolution Block

Das Netzwerk konnte die Genauigkeit gegenüber dem Netzwerk mit 2 Convolutional Blöcken leicht steigern. Die ReLu, hier in Grau, konnte ihre Genauigkeit um 0,41% steigern, die Tanh, in Orange, konnte die Genauigkeit um 0,94% erhöhen. Die Trainingszeiten hatten sich bei dieser Veränderung an den Prototypen für den 32x32 Pixel Datensatz nicht verändert.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	81,99 %	6 Sekunden
Tanh	80,28 %	6 Sekunden

Tabelle 10: Performance des Netzwerkes mit 3 Convolutional Blöcken 32x32

Die beiden verwendeten Aktivierungsfunktionen zeigen auch weiter einen stetigen Anstieg der Genauigkeit bis Epoche 9, danach wurden keine oder nur geringe Fortschritte der Genauigkeit mehr erzielt.

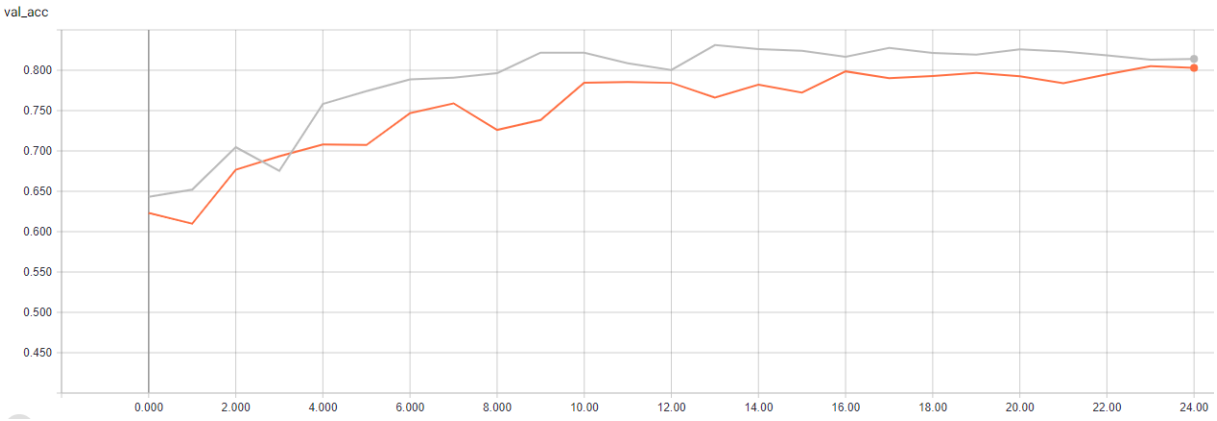

Abbildung 40: Genauigkeit auf dem Validation Set mit 3 Convolutional Blöcken 32x32

In der Abbildung 41 ist zu sehen, dass das Netzwerk nun mit beiden Aktivierungsfunktionen an Overfitting leidet. Das Netzwerk mit der ReLu senkt den Fehlerwert bis Epoche 10, danach steigt dieser wieder bis Epoche 22 an, wobei der Fehlerwert dann bis zum Ende des Trainings fällt aber immer noch größer ist als zu Beginn des Trainings. Die Tanh senkt den Fehlerwert bis Epoche 13, dann steigt auch hier der Fehlerwert bis zum Ende des Trainings. Die Epochenanzahl kann in diesem Netzwerk auf 13 Epochen beschränkt werden da es ab dieser Epoche keinen Zugewinn in der Genauigkeit gibt außerdem lässt sich damit das Overfitting vermeiden.

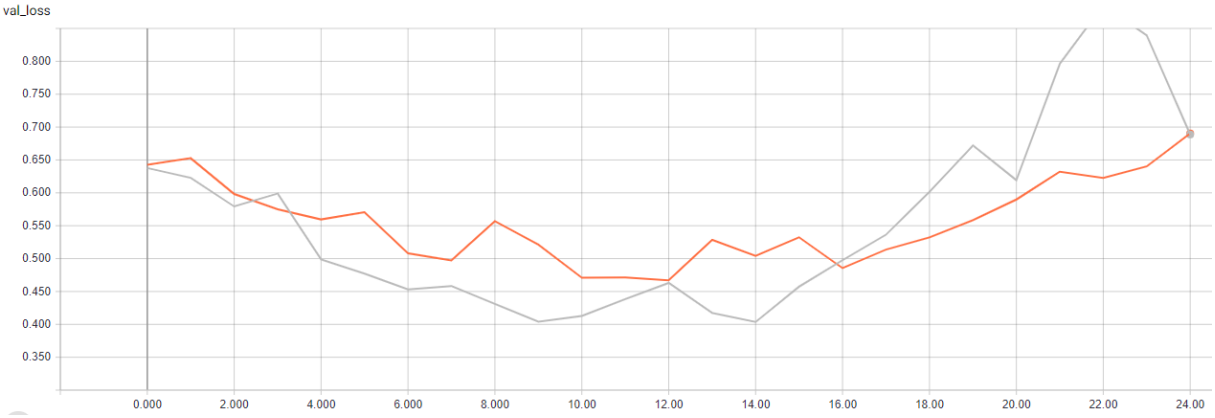

Abbildung 41: Fehlerwert auf dem Validation Set 32x32

Durch das Hinzufügen eines weiteren Convolutional Blocks konnte das Netzwerk mit dem 64x64 Datensatz die Genauigkeit der ReLu um 10,02% steigern. Auch die Tanh konnte ein Genauigkeitsgewinn von 5,47% erreichen. Die Trainingszeit pro Epoche ist um 3 Sekunden auf 19 Sekunden gestiegen, was eine Gesamttrainingszeit von 7:55 ergibt.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	86,62 %	19 Sekunden
Tanh	82,37 %	19 Sekunden

Tabelle 11: Performance des Netzwerkes mit 3 Convolutional Blöcken 64x64

Die Abbildung 42 zeigt wieder die Genauigkeit pro Epoche auf dem Validation Set des Netzwerkes mit dem 64x64 großen Datensatz. Hier ist zu sehen, dass die beiden Aktivierungsfunktionen einen Trainingsfortschritt bis Epoche 18 erzielen und ab dann nahezu seitwärts bis zum Ende des Trainings laufen.

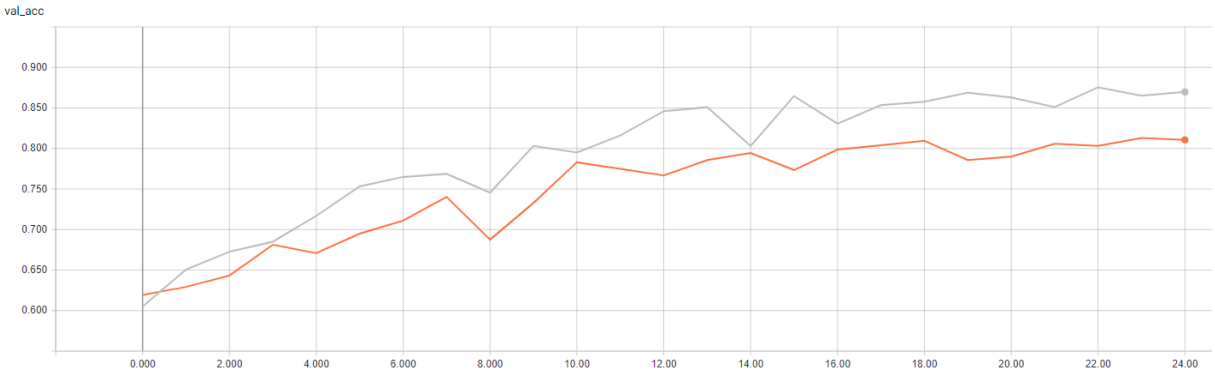

Abbildung 42: Genauigkeit auf dem Validation Set mit 3 Convolutional Blöcken 64x64

In der Abbildung 43 ist der Fehlerwert des Prototyps abzulesen. Die Aktivierungsfunktion ReLu (Grau) senkt den Fehlerwert bis Epoche 18 ab. Danach läuft der Fehlerwert der ReLu seitwärts bis Epoche 23, danach steigt dieser an. Daraus lässt sich erkennen, dass das Netzwerk beginnt sich überanzupassen, also in das Overfitting geriet. Der Durchlauf mit der Tanh Aktivierungsfunktion senkt den Fehlerwert ebenfalls bis Epoche 18 ab, danach läuft dieser seitwärts. Da es nach Epoche 18 keine Steigerung Genauigkeit mehr gibt kann die Epochenanzahl für diesen Prototyp auf noch 18 Epochen gesenkt werden.

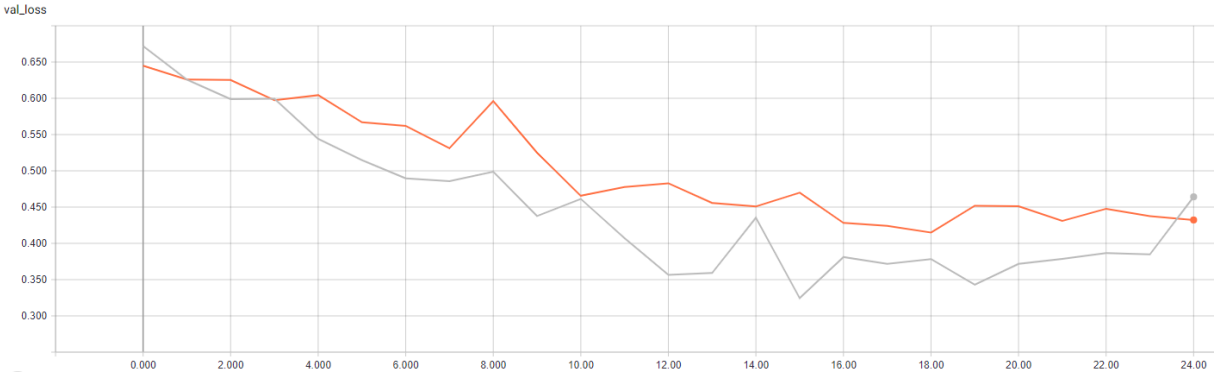

Abbildung 43: Fehlerwert auf dem Validation Set 64x64

Auch das Netzwerk mit dem 96x96 Bilder großen Datensatz konnte sich in der Genauigkeit durch das Hinzufügen eines weiteren Convolutional Blocks verbessern. Die ReLu konnte sich um 3 % auf 85,85 % verbessern, die Tanh konnte sich ebenfalls um 10,28 % auf nun 81,36 % verbessern. Durch das Hinzufügen eines weiteren Convolutional Blockes vergrößerte sich die Trainingszeit pro Epoche von 36 Sekunden auf nun 41 Sekunden was eine gesamte Trainingszeit von 17:05 Minuten ergibt.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	85,85 %	41 Sekunden
Tanh	81,36 %	41 Sekunden

Tabelle 12: Performance des Netzwerkes mit 3 Convolutional Blöcken 96x96

Die Abbildung 44 zeigt auch hier den Verlauf des Trainings auf dem Validation Set. Die ReLu hier in Grau zeigt eine Verbesserung bis Epoche 18, danach wird gibt es keine weiter Verbesserung der Genauigkeit. Die Tanh, hier in Orange, verbessert die Genauigkeit über die

gesamte Dauer des Trainings, die Lernfortschritte werden zum Ende aber stark reduziert. Hier hätte das Training bei Epoche 18 abgebrochen werden können.

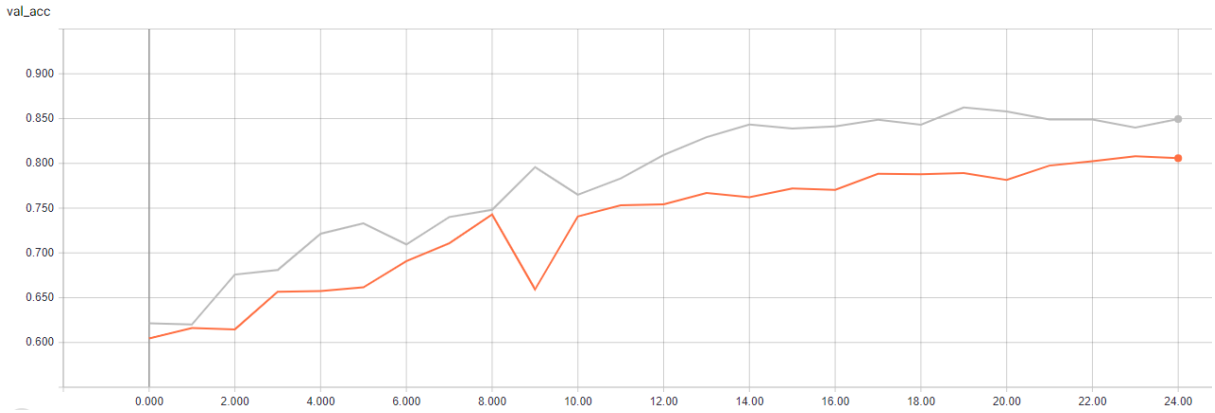

Abbildung 44: Genauigkeit auf dem Validation Set mit 3 Convolutional Blöcken 96x96

Der Fehlerwert der Tanh senkt sich über die gesamte Dauer des Trainings auch die ReLu senkt den Fehlerwert bis Epoche 19. Nach der 19. Epoche steigt der Fehlerwert des Netzwerkes unter der ReLu wieder an was zu einer Überanpassung an den Datensatz führt.

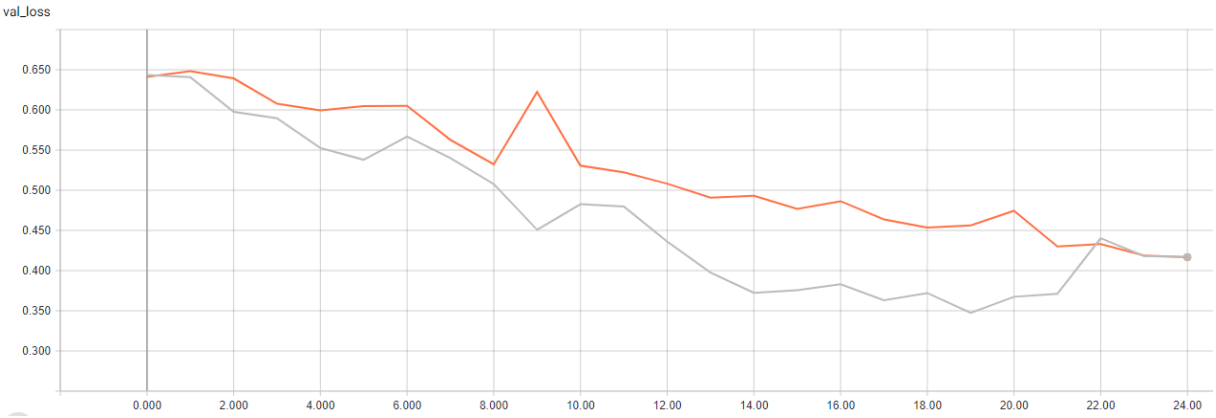

Abbildung 45: Fehlerwert auf dem Validation Set 96x96

12.2.3.1 Ergebnisse der ersten Erweiterung

Die Genauigkeit des Netzwerkes konnte sich im Allgemeinen durch das Hinzufügen eines weiteren Convolutional Blocks verbessern. Durch das Hinzufügen eines weiteren Convolutional Blocks stiegen auch die Trainingszeiten der unterschiedlichen Netzwerke bis auf das 32x32 Pixel Netzwerk an. Das Netzwerk mit dem 64x64 Datensatz konnte die Genauigkeit verbessern, litt aber durch die beiden Aktivierungsfunktionen unter der Überanpassung an

den Datensatz. Der 96x96 große Datensatz hingegen litt nur unter der ReLu an einer leichten Überanpassung am Ende des Trainings.

12.2.4 Zweite Erweiterung des Prototypen

Im nächsten Schritt werden die einzelnen Prototypen um einen weiteren Convolutional Block erweitert, um die Genauigkeit weiter zu steigern. Der nun neu hinzugefügte Block hat 128 Filter. Somit steigen die trainierbaren Gewichte auf 462562 an. Es werden wieder die Netzwerke mit den 64x64 Pixeln und den 96x96 Pixeln gegeneinander antreten, inklusive der Aktivierungsfunktionen ReLu und Tanh. Die Abbildung 46 zeigt auch wieder das Netzwerk mit dem neu hinzugefügten Convolution Block.

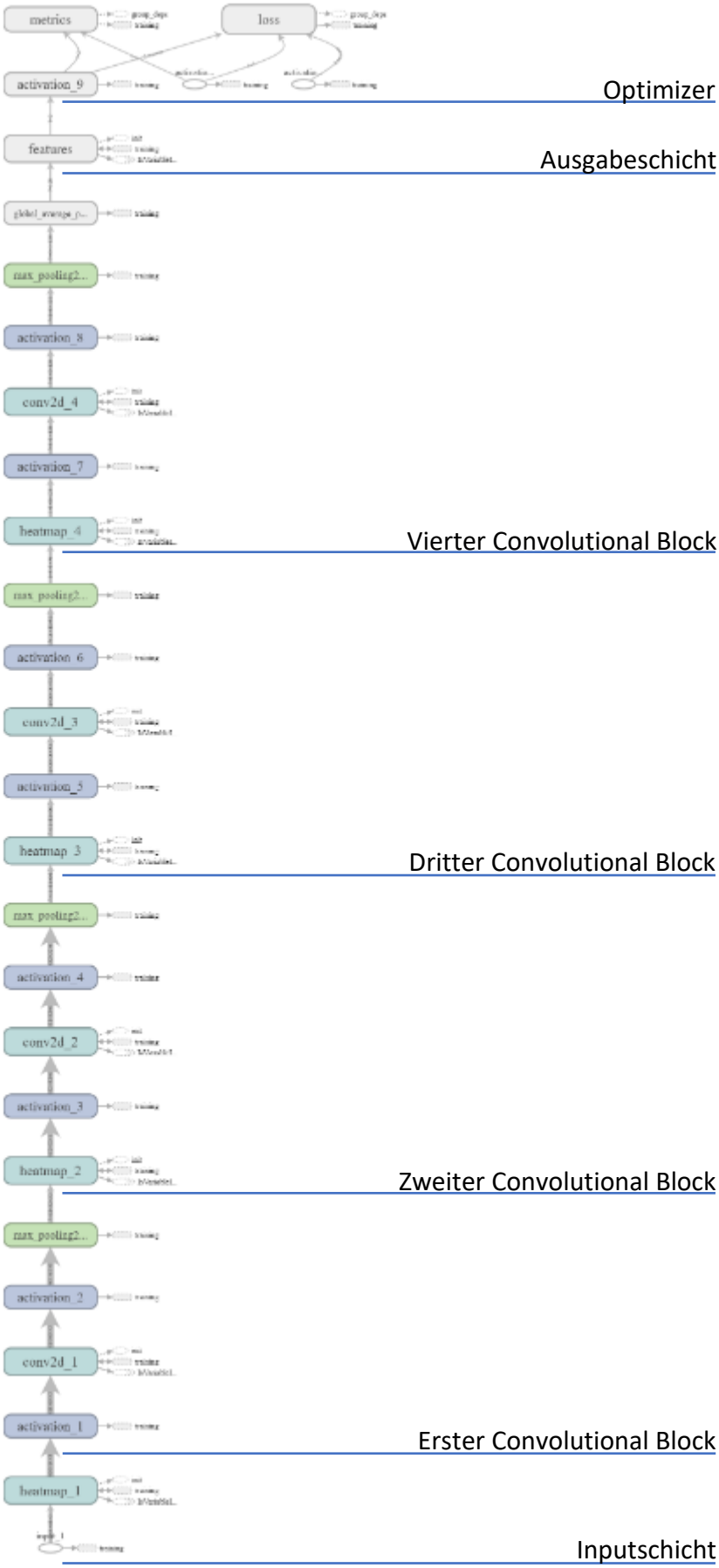

Abbildung 46: Modell des erweiterten Prototypen um einen weiteren Convolution Block (2)

Im ersten Schritt wurde wieder das Netzwerk mit dem 64x64 Pixel großen Datensatz auf die Genauigkeit untersucht. Durch den weiteren Convolutional Block senkte sich die Genauigkeit bei allen beiden Aktivierungsfunktionen. Die Trainingszeit erhöhte sich um eine Sekunde und lag bei 20 Sekunden pro Epoche, somit ergab sich eine gesamte Trainingszeit von 8:20 Minuten. Das Training mit der ReLu verschlechterte sich um 1,04 % auf 85,58 %. Auch das Training mit der Tanh verschlechterte sich um 1,79 % auf 82,37%.

Aktivierungsfunktion	Genauigkeit	Performance pro Epoche
ReLu	85,58 %	20 Sekunden
Tanh	84,16 %	20 Sekunden

Tabelle 13: Performance des Netzwerkes mit 4 Convolutional Blöcken 64x64

Die Abbildung 47 zeigt auch hier wieder den Verlauf des Trainings auf dem Validation Set an. Die ReLu in Blau verbessert die Genauigkeit bis Epoche 14, ab dieser Epoche gibt es keinen Zugewinn der Genauigkeit bis zum Ende des Trainings. Die Tanh in Rot verbessert die Genauigkeit bis zum Ende des Trainings.

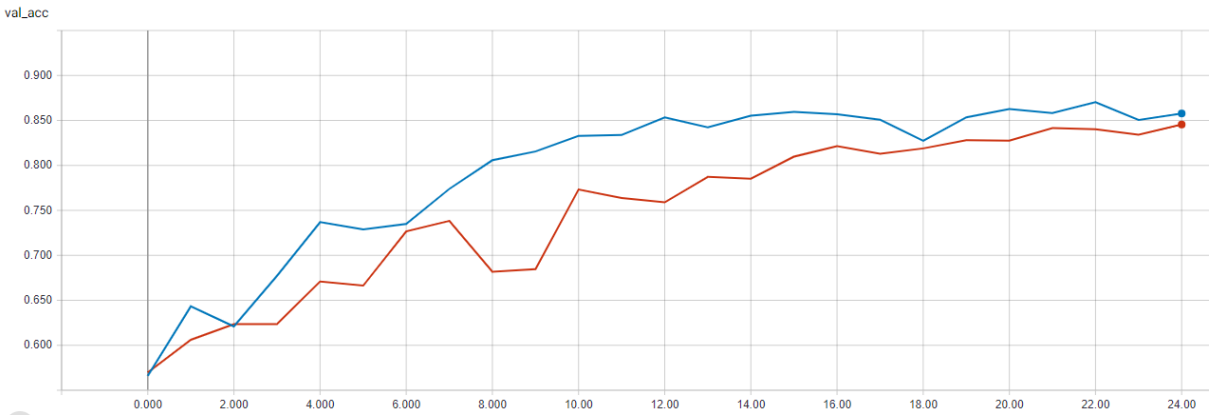

Abbildung 47: Genauigkeit auf dem Validation Set mit 4 Convolutional Blöcken 64x64

Das Netzwerk mit der ReLu zeigt ab Epoche 14 einen starken Anstieg des Fehlerwertes an, was zu einer signifikanten Überanpassung an den Datensatz führt. Die Tanh senkt den Fehlerwert bis Epoche 21 dann steigt auch hier der Fehlerwert moderat was auch zu einer Überanpassung führt jedoch nicht so stark wie bei der ReLu.

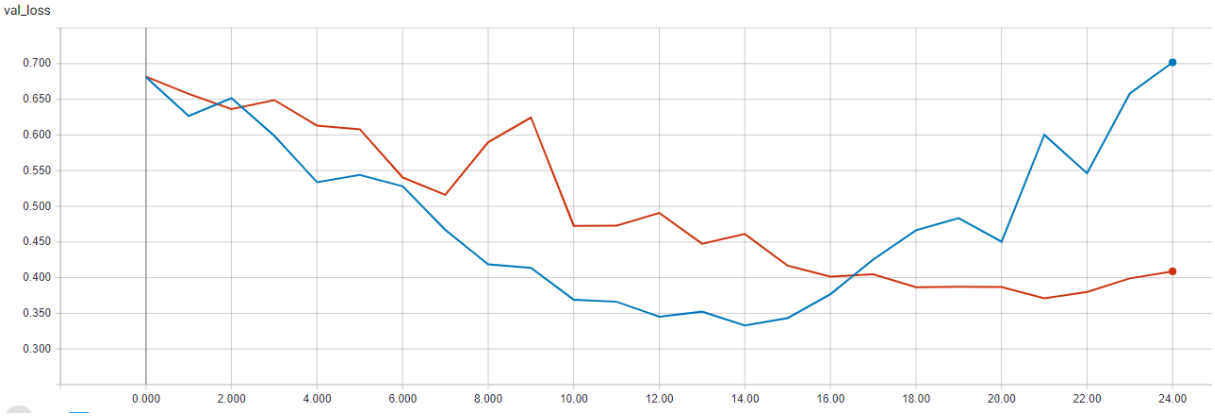

Abbildung 48: Fehlerwert auf dem Validation Set 64x64

Das Netzwerk mit dem 96x96 Pixel großen Datensatz kann durch das Hinzufügen eines weiteren Convolutional Blocks seine Genauigkeit auf dem Testdatensatz unter der ReLu um weitere 2,14 % auf 87,99 % steigern. Die Tanh Aktivierungsfunktion hingegen konnte von dem neu hinzugefügten Convolutional Blocks nicht profitieren hier senkte sich die Genauigkeit um 2,34 % auf 79,02 %.

Aktivierungsfunktion	Genauigkeit	Performance
ReLu	87,99 %	51 Sekunden
Tanh	79,02 %	51 Sekunden

Tabelle 14: Performance des Netzwerkes mit 4 Convolutional Blöcken 96x96

Die Abbildung 49 zeigt wieder die Genauigkeit auf dem Validation Set. Die ReLu zeigt eine Verbesserung der Genauigkeit bis Epoche 16 an, danach gab es bis zum Ende des Trainings keine Verbesserung. Die Tanh zeigt wieder eine stetige Verbesserung bis zum Ende des Trainings. Sie erreicht aber auch hier nicht die Genauigkeit des Netzwerkes mit der ReLu Aktivierungsfunktion.

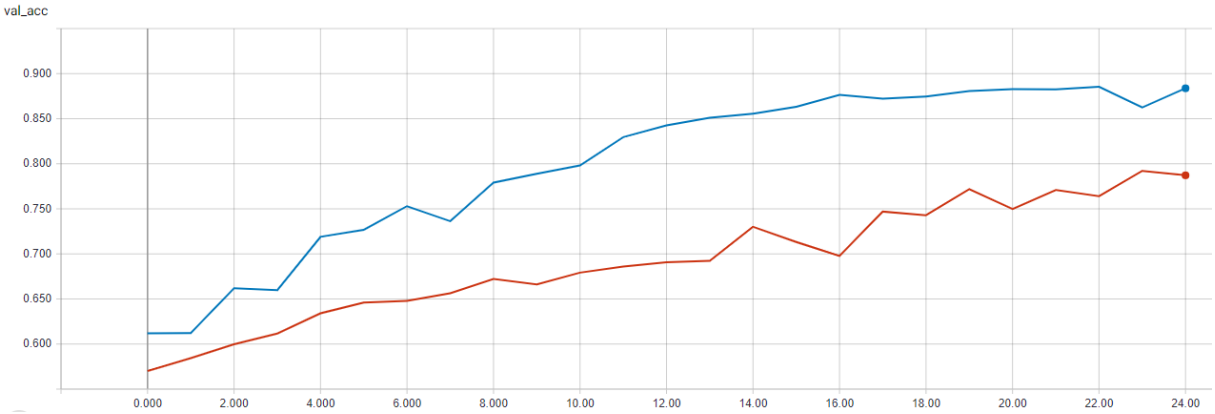

Abbildung 49: Genauigkeit auf dem Validation Set mit 4 Convolutional Blöcken 96x96

Der Fehlerwert nimmt unter der ReLu stärker ab, als unter der Tanh Aktivierungsfunktion aber auch in diesem Testdurchlauf zeigt die ReLu auf dem Validation Set eine Überanpassung ab Epoche 16. Die Tanh zeigt auf dem Validation Set keine Gefahr der Überanpassung.

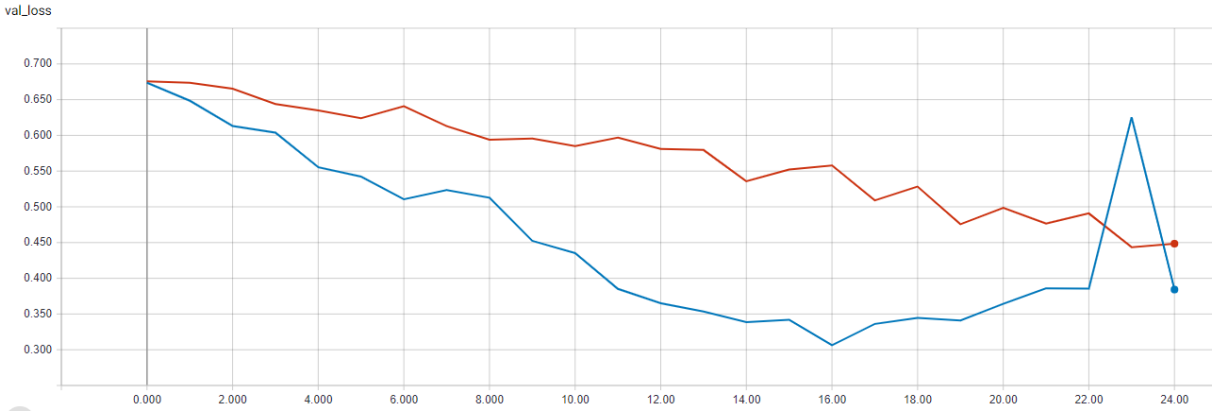

Abbildung 50: Fehlerwert auf dem Validation Set 96x96

12.2.4.1 Ergebnisse der zweiten Erweiterung des Netzwerkes

Mit dem Hinzufügen eines weiteren Convolution Blockes konnte die Genauigkeit auf dem 96x96 Pixel großen Datensatz mit der ReLu weiter gesteigert werden. Das Netzwerk mit der Tanh Funktion konnte mit einer Vergrößerung der Bilder und dem weiteren Convolutional Block die Genauigkeit nicht weiter steigern, hier sank die Genauigkeit sogar von 81,36% auf 79,02% auf dem Validation Set. Es zeigte sich auch, dass Netzwerke mit der ReLu Aktivierungsfunktion ehr dazu neigen sich Überanzupassen.

Abschließend kann gesagt werden, dass Datensätze mit kleinen Bildern eine gute Performance in der Trainingszeit aufweisen, diese aber bei tieferen Netzwerken ehr Gefahr laufen an der Überanpassung zu leiden. Für die weitere Optimierung wird das Netzwerk mit dem 96x96 Pixel

großen Datensatz und der ReLu verwendet, da dieses Netzwerk die höchste Genauigkeit unter allen getesteten Netzwerken mit 87,99 % aufweist.

12.2.5 Optimierung des Prototyps

Im nächsten Schritt wird mit dem weiterverwendeten Netzwerk aus dem vorherigen Kapitel eine Random Search durchgeführt, um die Genauigkeit durch eine „verbesserte“ Lernrate weiter steigern zu können. Die Random Search mit Kreuzvalidierung teilt den Datensatz in 3 Folds und testet jeden Fold mit der dazugehörigen zufälligen Lernraten. Das Intervall für die Random Search ist von 0,0001 bis 0,001. Aus diesem Intervall sollen 10 zufällige Lernraten angewandt werden. Somit werden 30 Durchläufe durchgeführt um 10 verschiedene Lernraten zu erhalten. Die gesamte Random Search dauerte 5 Stunden und 30 Minuten. Durch die Random Search konnte eine Lernrate gefunden werden, die das Netzwerk auf dem Testdatensatz in der Genauigkeit weiter verbessern konnte. Durch die neue Lernrate konnte sich die Genauigkeit auf dem Testdatensatz von 87,99 % auf 89,78 % gesteigert werden. Die Abbildung 51 zeigt die verschiedenen Lernraten und die dazugehörigen Genauigkeiten auf dem Validation Set der Random Search an.

```
Best: 0.859214 using {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.000764433628332873}
0.844429 (0.013219) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00048839036456075 49}
0.780214 (0.004638) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00015538553590138 31}
0.792643 (0.006998) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00019753744088543866}
0.836714 (0.015194) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00035603437058913146}
0.852857 (0.011995) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00072288859553788 76}
0.821000 (0.028264) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00039585901636771895}
0.836714 (0.009299) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00043526601395241286}
0.825500 (0.014841) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00037134990422248566}
0.846643 (0.008476) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00092958333052355 95}
0.859214 (0.014506) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.000764433628332873}
```

Abbildung 51: Ergebnisse der Random Search

Nach der Random Search wurde der Datensatz mit Hilfe der Datenerweiterung um 15000 Bilder erweitert. Durch die Datenerweiterung werden die neu hinzugefügten Bilder in ihrer Rotation, Zoom und der Verschiebung in die Höhe und Breite verändert. Diese Bilder haben nach ihrer Veränderung immer noch die gleiche Klassenzugehörigkeit und sind für das Netzwerk komplett neue Bilder was dem Training zu Gute kommen kann. Durch das Hinzufügen trainiert das Netzwerk nun mit 26000 Bildern, das Validation Set hat nun eine Größe von 6500 Bildern. Die Abschließende Genauigkeit wird mit dem Test Set, dass nun aus 7500 Bildern besteht bestimmt.

Die Abbildung 52 zeigt den Genauigkeitsverlauf auf dem Validation Set mit der Datenerweiterung. Der Orange Graph zeigt die Genauigkeit des besten Ergebnisses der Random Search auf dem Validation Set an. Alle weiteren Optimierungen müssen sich anhand des Ergebnisses der Random Search in ihrer Genauigkeit messen. Der Pinke Graph zeigt die Genauigkeit auf dem Validation Set mit der Batch Größe von 256 Bildern an. Durch die Datenerweiterung stieg auch Trainingszeit von 51 Sekunden pro Epoche auf 85 Sekunden an. Die Batchgröße musste im weiteren Verlauf der Optimierung gesenkt werden, da die Leistung des eingesetzten Computers nicht mehr ausreichte, um das Netzwerk zu trainieren.
Der grüne Graph zeigt die Genauigkeit mit einer verringerten Batchgröße von 128 Bildern. Diese Durchläufe wurden erstellt, um die weiteren Ergebnisse untereinander vergleichen zu können. Der pinke Graph zeigte auf dem Validation Set eine Genauigkeit von 93,12 % und auf dem Test Set eine Genauigkeit von 90,21 % Der grüne Graph zeigte eine Genauigkeit von 94,05 % und auf dem Test Set 91,44 %. Somit hat die Batchgröße einen Einfluss auf die Genauigkeit des Netzwerkes. Durch die reduzierte Batchgröße steigerte sich auch die Trainingszeit von 83 Sekunden auf 85 Sekunden.

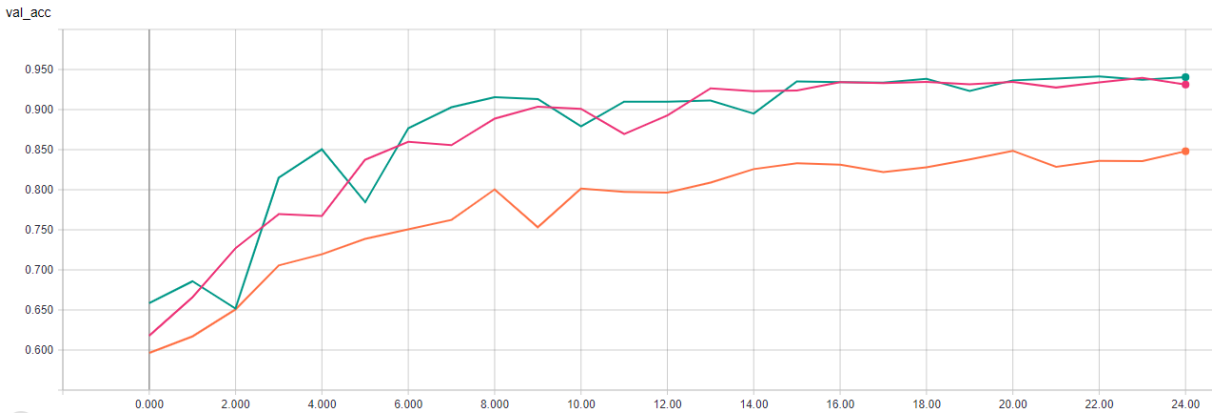

Abbildung 52: Datenerweiterung Genauigkeit auf dem Validation Set

Nachdem die Genauigkeit mit der Datenerweiterung gesteigert werden konnte, wurde nun versucht die Genauigkeit mit der Batch Normalisierung ebenfalls weiter zu steigern. Diese Normalisierung wird nach jeder Faltungsschicht einmal ausgeführt, was bedeutet, dass die Batch Normalisierung in dem Netzwerk 8-mal eingesetzt wird. Durch die Normalisierung stieg die Trainingszeit pro Epoche auf 127 Sekunden an was auch darauf zurückzuführen ist, dass die Batchgröße halbiert wurde. Die Abbildung 53 zeigt die Genauigkeiten der Random Search, der Datenerweiterung und der Batch Normalisierung hier in Türkis an. Durch die Batch

Normalisierung stieg die Genauigkeit sowohl auf dem Validation und dem Test Set an. Die Genauigkeit stieg von 91,44 % auf nun 93,26 % auf dem Test Set nach 25 Epochen an. In der Abbildung 53 lässt sich gut erkennen das die Genauigkeit durch die Batch Normalisierung stärker schwankt als ohne diese.

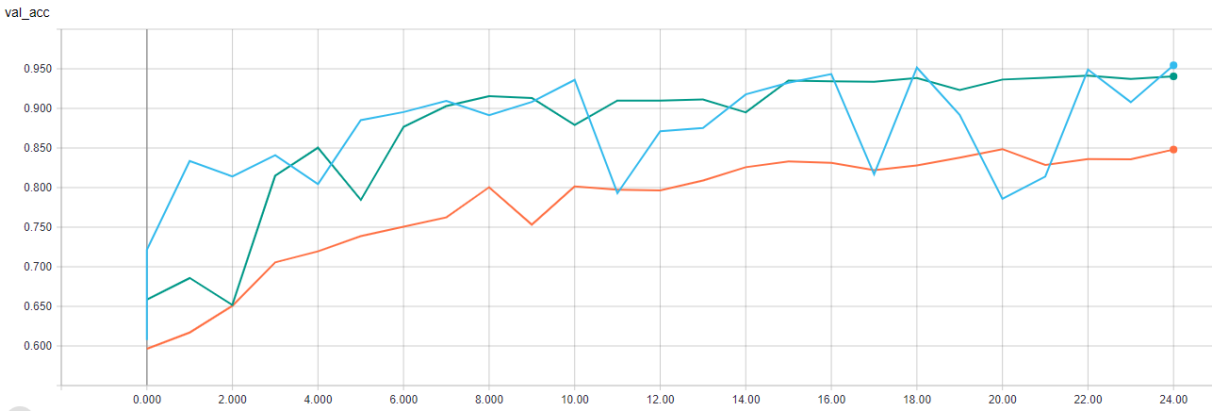

Abbildung 53: Genauigkeit auf dem Validation Set mit der Batch Normalisierung

Im nächsten Schritt wurde mit verschiedenen Einstellungen des Dropouts versucht die Genauigkeit auf dem Testdatensatz weiter zu steigern. Im ersten Durchlauf wurden 10 % der Neuronen der einzelnen Blöcke ausgeschalten, um eine Überanpassung des Netzwerkes zu verhindern. Durch das Ausschalten der Neuronen hatte das Netzwerk nun eine Genauigkeit von 86,40 % auf dem Test Set. Die Abbildung 54 zeigt die Verläufe der Random Search, der Datenerweiterung und des Dropouts mit 10 % (grauer Graph). Der Durchlauf mit dem Dropout zeigt eine Steigerung der Genauigkeit gegenüber der Random Search, liegt aber deutlich hinter dem Ergebnis der Datenerweiterung.

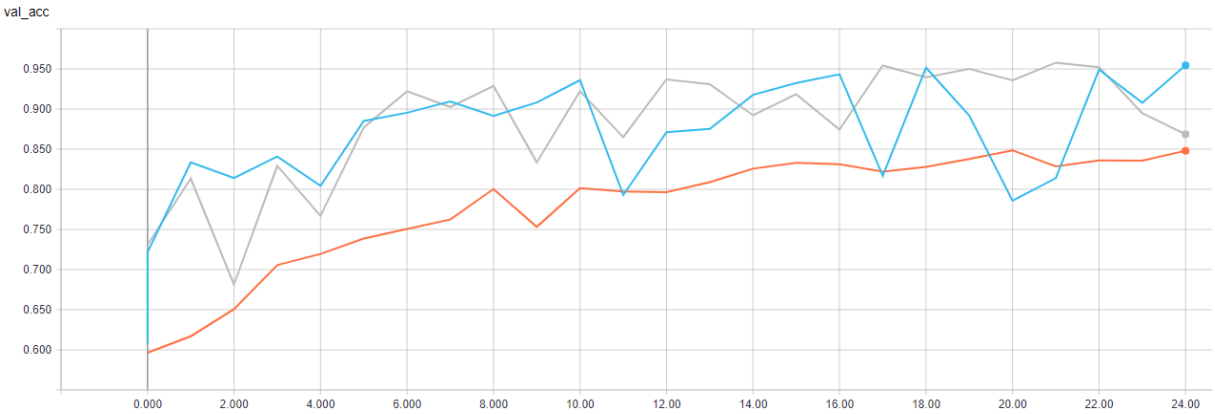

Abbildung 54: Genauigkeit auf dem Validation Set mit dem Dropout 10%

Da das Ergebnis des Dropouts schlechter war als der vorherige wurden die auszuschaltenden Neuronen in den einzelnen Blöcken nun immer um 10 % erhöht, dass nun im letzten Faltungsblock 40 % aller Neuronen während des Trainings abgeschaltet wurden. Die Abbildung 55 zeigt auch hier wieder den Verlauf der Random Search, der Datenerweiterung und des Dropouts mit der aufsteigenden Abschaltung der Neuronen bis 40 % (Orangener Graph). Auch dieser Durchlauf konnte die Genauigkeit des Netzwerkes nicht weiter steigern. Durch den aufsteigenden Dropout konnte eine Genauigkeit von 92,99 % auf dem Test Set erreicht werden. Diese Genauigkeit ist zwar höher als bei dem Dropout mit 10 %, sie liegt aber immer noch unter der Genauigkeit der Datenerweiterung mit 93,26 %.

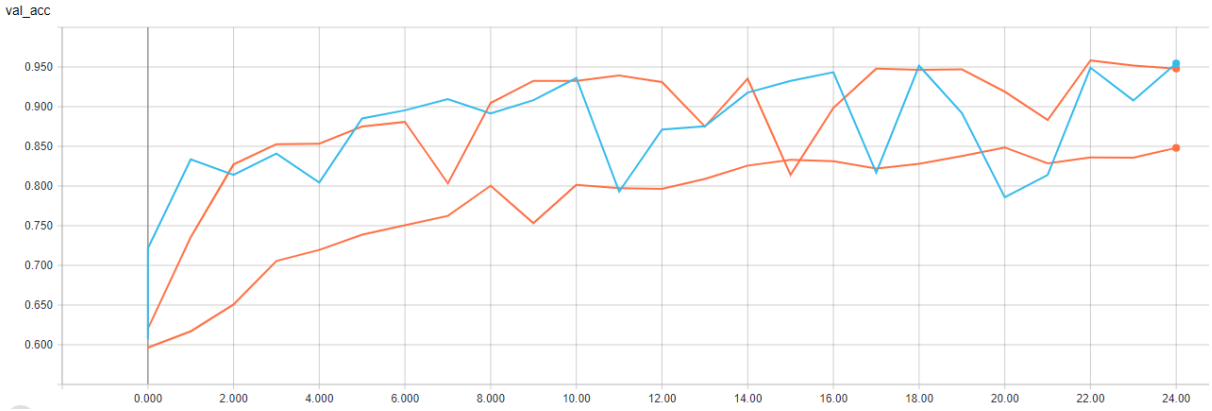

Abbildung 55: Genauigkeit auf dem Validation Set mit dem Dropout aufsteigend bis 40%

Im letzten Schritt wurde der Dropout absteigend angewandt, sodass im ersten Faltungsblock 40 % und im letzten Faltungsblock 10 % der Neuronen während des Trainings abgeschaltet wurden. Durch dieses Anwenden des Dropouts konnte die Genauigkeit des Netzwerkes auf dem Test Set von 93,26 % auf 93,56 % gesteigert werden. Die Abbildung 56 zeigt auch hier wieder die Ergebnisse der Random Search, der Datenerweiterung und des absteigenden Dropouts (grüner Graph) von 40 % auf 10 %.

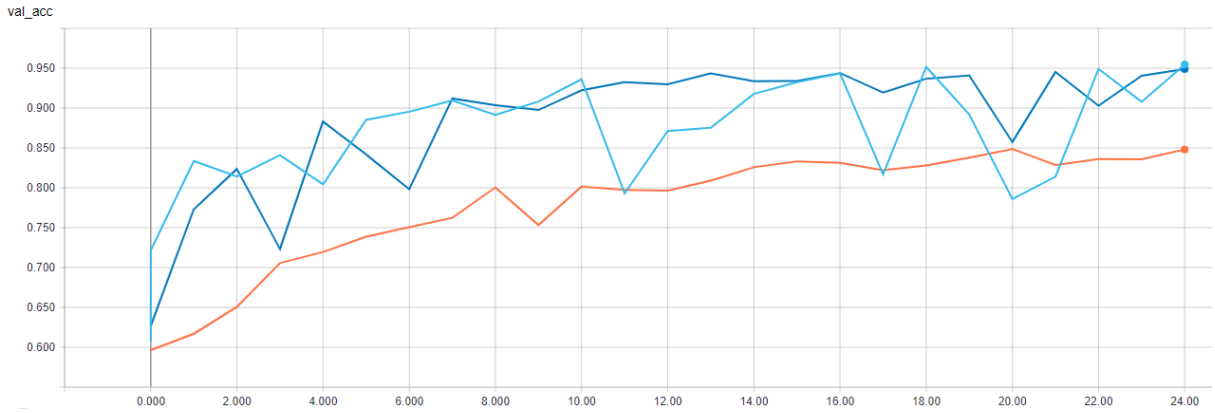

Abbildung 56: Genauigkeit auf dem Validation Set mit dem Dropout absteigend von 40% bis 10%

Nach den Optimierungen wurde nochmals eine Random Search durchgeführt, um eventuell eine noch bessere Lernrate zu finden, die unten aufgeführte Abbildung zeigt die erneuten Ergebnisse dieser Random Search.

```
Best: 0.931000 using {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0008894769940900988}
0.901462 (0.032179) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0003728508310766569}
0.915654 (0.002314) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0005853591495051357}
0.915462 (0.017399) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0008564616364025258}
0.633346 (0.078243) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00010250422997399776}
0.896923 (0.020776) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00034480002864990735}
0.931000 (0.007254) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0008894769940900988}
0.922462 (0.020516) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0009677222115063953}
0.859846 (0.066406) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.00035619991534033084}
0.917923 (0.031828) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0007918772016600696}
0.929192 (0.006477) with: {'optimizer': <class 'keras.optimizers.Adam'>, 'lr': 0.0008894037037478153}
```

Abbildung 57: Erneutes Anwenden der Random Search

Es wurden wieder 10 Lernraten-Kandidaten aus dem Intervall von 1e-3 bis 1e-4 erprobt. Der Trainingsdatensatz wurde auch schon wie bei der vorherigen Suche in 3 Folds geteilt. Das erneute Anwenden der Random Search mit allen vorherigen Optimierungen nahm eine Zeit von 19 Stunden und 30 Minuten in Anspruch. Die beste Lernrate aus der erneuten Random Search lieferte auf dem Test Set eine Genauigkeit von 92,93 %. Somit konnte durch das erneute Anwenden der Random Search die Genauigkeit nicht verbessert werden. In der Abbildung 58 ist das Ergebnis des besten Kandidaten der Random Search(hellblauer Graph) im Verhältnis zu dem besten Ergebnis sowie der ersten Random Search zu sehen.

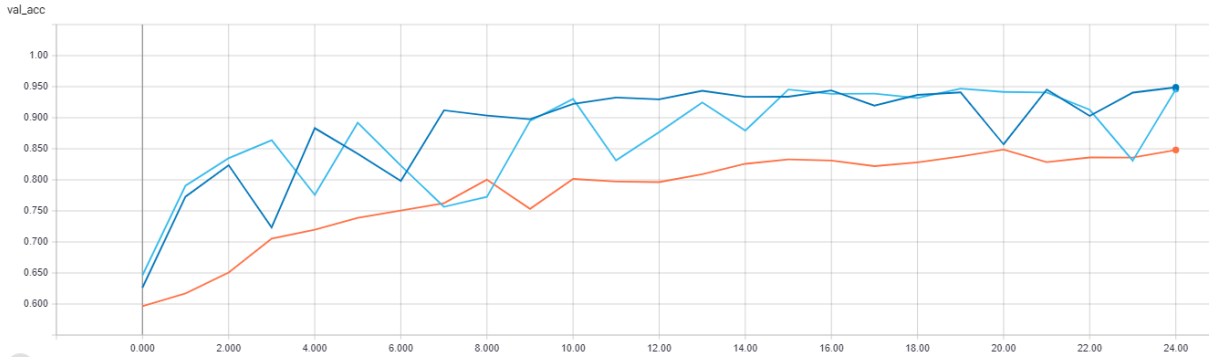

Abbildung 58: Genauigkeit auf dem Validation Set mit erneuter Anwendung der Random Search

Um nun die Genauigkeit noch weiter steigern zu können wurde eine Callback Methode mit dem Namen ReduceLerningRateOnPlateau angewandt, die die Lernrate nach einem vorher festgelegten Parameter reduzieren kann. In dem Netzwerk wird durch die Methode der Fehlerwert auf dem Validation Set überwacht, wenn sich dieser nach 2 Epochen nicht groß ändert so wird die Lernrate um eine vorher festgelegte Prozentzahl verringert. Im ersten Durchlauf wurde die Lernrate um 10 % reduziert, wenn der Fehlerwert auf dem Validation Set sich innerhalb von 2 Epochen nicht großartig ändert. Nach dem Anwenden der Methode sank die Genauigkeit auf dem Test Set von 93,56 % auf 90,28 % ab. Die Abbildung 59 zeigt den Verlauf des Trainings auf dem Validation Set, der rote Graph zeigt hier den Trainingsverlauf der Methode ReduceLerningRateOnPlateau an. Weiterhin ist in der Abbildung 58 gut zu erkennen das sich die Genauigkeit auf dem Validation Set von Epoche 24 zu 25 von 94,84 % zu 89,95 % reduziert hat.

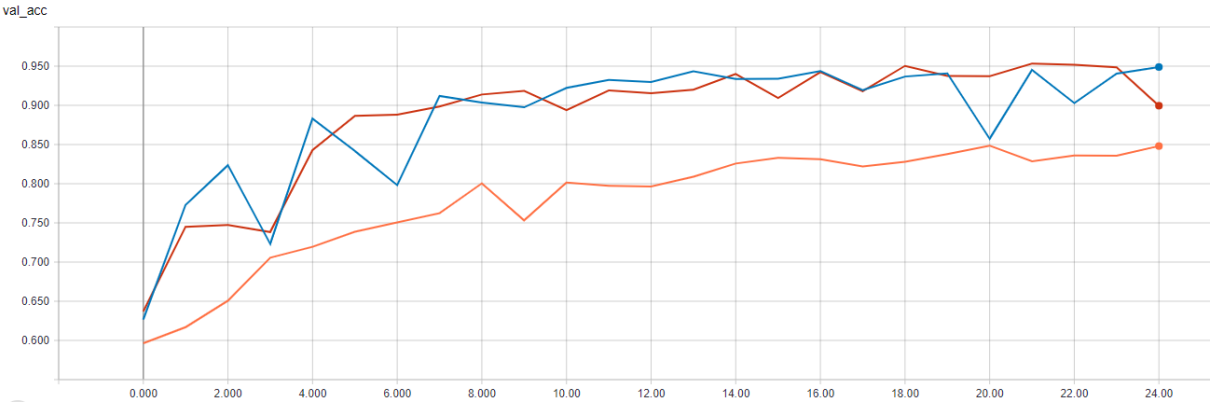

Abbildung 59: Genauigkeit auf dem Validation Set mit der Methode ReduceLerningRateOnPlateau 10%

Um die Genauigkeit weiter steigern zu können wurde im Anschluss ein Durchlauf durchgeführt, ob eine Reduzierung der Epochen auf 24 die Genauigkeit auf dem Test Set erhöhen kann. Die Abbildung 60 zeigt den Verlauf des Trainings auf dem Validation Set. Der neue grüne Graph zeigt hier in der Abbildung das Training mit 24 Epochen. Durch das herabsetzen der Epochen konnte die Genauigkeit auf dem Validation Set von 89,95 % auf 95,83 % gesteigert werden. Auch die Genauigkeit auf dem Test Set konnte von 90,28 % auf 93,61 % gesteigert werden. Durch das Anwenden dieser Methode und durch das Absenken der Epochen konnte die Genauigkeit im Vergleich zum absteigenden Dropout um 0,05 % gesteigert werden.

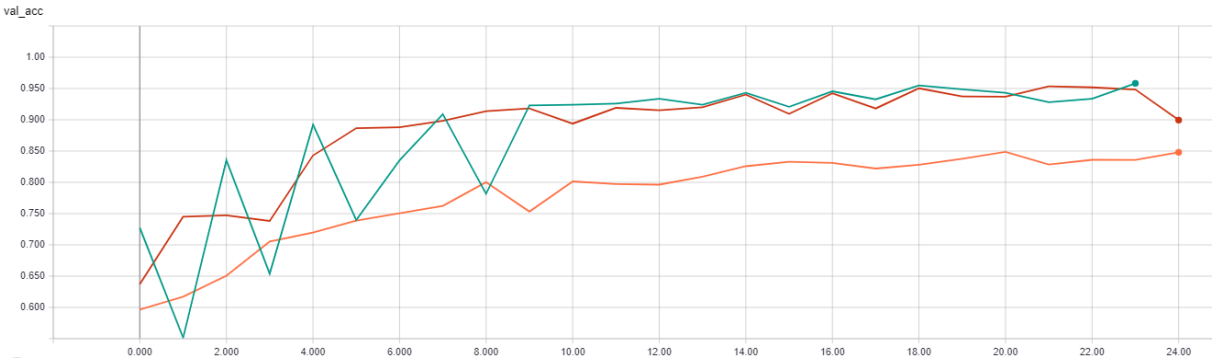

Abbildung 60: Genauigkeit auf dem Validation Set mit der Methode ReduceLerningRateOnPlateau 10% mit 24 Epochen

Um die Genauigkeit weiter zu steigern, wurde verschiedene Einstellungen der Methode ReduceLerningRateOnPlateau angewandt. Der nächste Durchlauf wurde mit einer Reduzierung der Lernrate von 10 % auf 1 % durchgeführt. Dies führte bei 25 Epochen auch wieder zu einem schlechteren Ergebnis, die Genauigkeit sank von 93,61 % auf 87,68 % nach 25 Epochen auf dem Test Set.

Auch hier lag die höchste Genauigkeit auf dem Validation Set bei Epoche 24 bei 95,71 % gegenüber 88,05 % bei Epoche 25. Die Abbildung 61 zeigt auch wieder den Einbruch bei Epoche 24 des Türkisen Graphs.

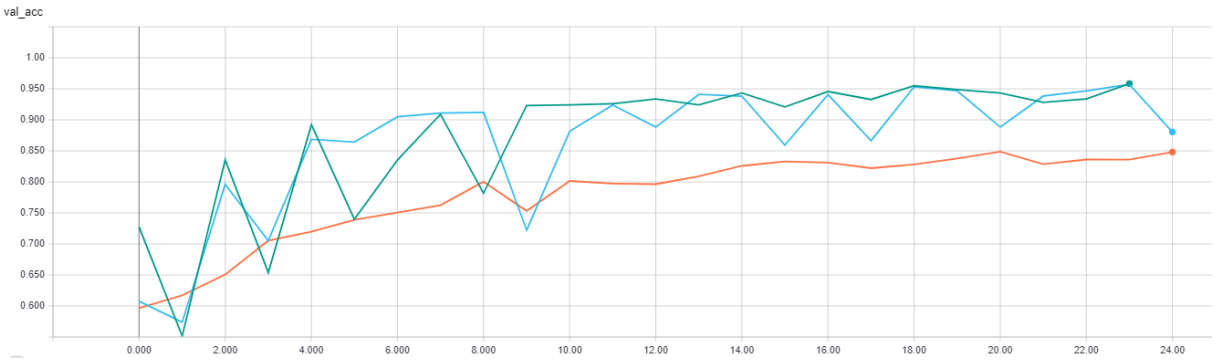

Abbildung 61: Genauigkeit auf dem Validation Set mit der Methode ReduceLerningRateOnPlateau 1%

Hier wurde wieder überprüft, ob eine Reduzierung der Epochenanzahl auf 24 eine Steigerung der Genauigkeit hervorruft. Die Genauigkeit konnte auf dem Validation Set von 88,05 % auf 93,38 % erhöht werden, auch auf dem Test Set konnte eine Positive Zunahme der Genauigkeit von 87,68 % auf 91,93 % festgestellt werden. In Abbildung 62 wird der aktuelle Durchlauf durch den grauen Graphen dargestellt. Es wurde auch hier gezeigt, dass die Reduzierung der Epochenzahl einen positiven Einfluss auf die Genauigkeit des Netzwerkes hat. Durch den 1%igen Abfall der Lernrate bei Erreichung eines Plateaus konnte sich die Genauigkeit durch eine Reduzierung der Epochenzahl verbessern, diese Verbesserung reichte allerdings nicht aus, um das Netzwerk im Allgemeinen in der Genauigkeit zu verbessern.

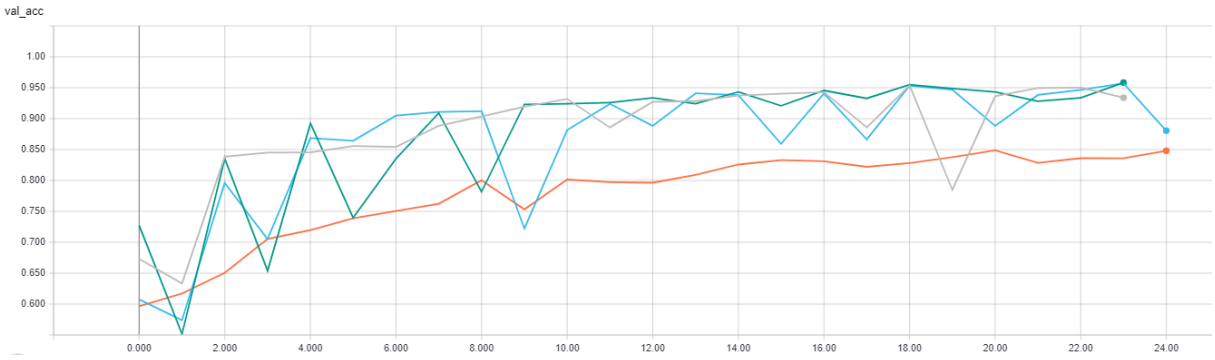

Abbildung 62: Genauigkeit auf dem Validation Set mit der Methode ReduceLerningRateOnPlateau 1% mit 24 Epochen

Es wurde noch ein weiterer Versuch unternommen die Genauigkeit des Netzwerkes zu erhöhen. Dieses Mal wurde die ReduceLerningRateOnPlateau Methode so angepasst, dass durch Erreichen eines Plateaus die Lernrate um 5% abnimmt. Nach Epoche 25 hatte das Netzwerk eine Genauigkeit von 93,28 % auf dem Test Set, was auch hier wieder eine Verschlechterung von 93,28 % auf 93,61 % darstellt. Auch in diesem Durchlauf senkte sich der

Fehlerwert ab Epoche 23. Die Abbildung 63 zeigt den Verlauf des Trainings mit der ReduceLerningRateOnPlateau Methode mit dem 5%igen Abfall in Pink, den ersten Durchlauf der Random Search und der ReduceLerningRateOnPlateau Methode mit dem 10%igen Abfall mit 23 Epochen.

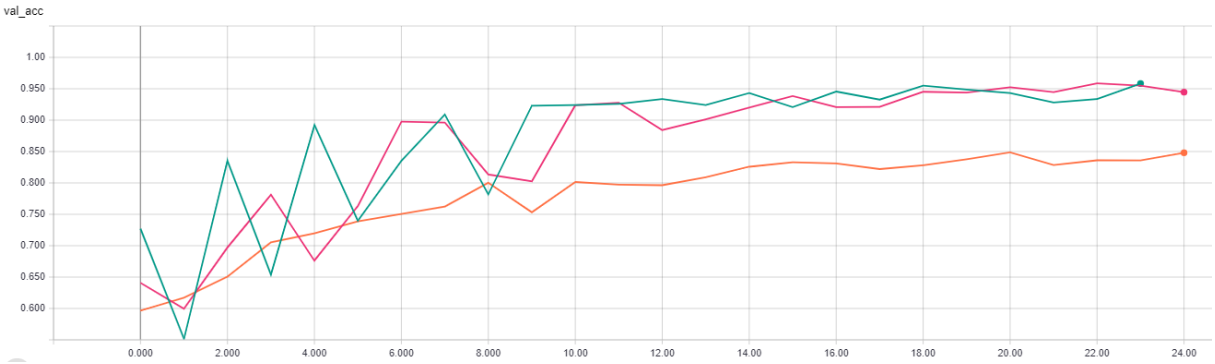

Abbildung 63: Genauigkeit auf dem Validation Set mit der Methode ReduceLerningRateOnPlateau 5%

Hier konnte ebenfalls durch die Reduzierung der Epochen auf 23 die Genauigkeit gesteigert werden. Durch die Reduzierung konnte die Genauigkeit auf dem Validation Set 94,46 % auf 95,12 % und auf dem Test Set von 93,28 % auf 93,47 % gesteigert werden. Die untere Abbildung zeigt auch hier den Verlauf der Genauigkeiten auf dem Validation Set. Der Orange Graph zeigt hier den aktuellen Durchlauf an. Durch die Anpassung der Epochenzahl konnte auch hier wieder die Genauigkeit gesteigert werden, diese Steigerung reicht aber auch hier nicht aus, um die Genauigkeit des Netzwerkes im Allgemeinen zu verbessern.

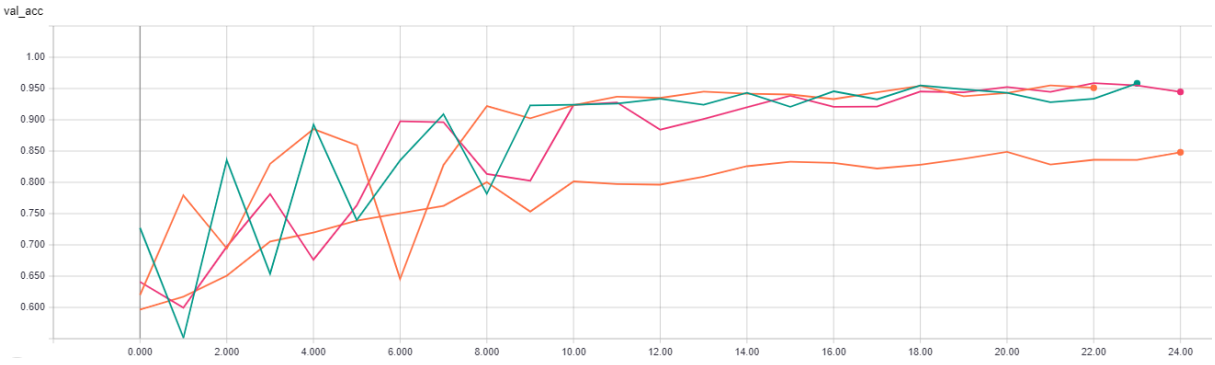

Abbildung 64: Genauigkeit auf dem Validation Set mit der Methode ReduceLerningRateOnPlateau 5% mit 23 Epochen

13. Der Finale Prototyp

In den vorherigen Kapiteln dieser Arbeit wurde der Aufbau und die Optimierung des zu entwickelnden Convolutional Neuronal Network beschrieben und durchgeführt. In diesem Kapitel wird auf den Finalen Prototypen eingegangen. Der Finale Prototyp besteht aus 4 Convolutional Blöcken. Die wie in der Abbildung 65 aufgebaut sind.

```
x = Conv2D(filters=32, kernel_size=3, padding="same", name="heatmap_1", kernel_initializer=init_w)(input_img)
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = Conv2D(filters=32, kernel_size=3, padding="same", kernel_initializer=init_w)(x)
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = MaxPool2D()(x)
x = Dropout(rate=0.4)(x)
```

Abbildung 65: Convolutional Block

Sie unterscheiden sich nur in der Anzahl an verwendeten Filter und der Dropoutrate. Im ersten Block sind pro Convolution Schicht 32 Filter, in dem zweiten Block sind 64 Filter pro Convolution Schicht, in dem dritten Block befinden sich 96 Filter pro Convolution Schicht und im vierten Block sind 128 Filter pro Convolution Schicht in der Verwendung. Die Dropoutrate verringert sich pro Block um 10 %. Der Verwendete Optimizer in dem Finalen Prototyp ist der Adam. Die verwendete Lernrate für das Netzwerk ist 0.000764433628332873, diese wird durch die Callback Methode ReduceLerningRateOnPlateau um 10 % reduziert sobald der Validation Fehlerwert sich innerhalb von 2 Epochen nur geringfügig oder gar nicht verändert. Durch diese Einstellungen weist der Prototyp eine Genauigkeit von 93,47 % bis 97,09 % auf. Diese Schwankungen in der Genauigkeit entstehen durch das erneute Testen der Genauigkeit auf dem Test Set bei unangetasteten Gewichten. Die Abbildungen 66 und 67 zeigen den Finalen Prototypen mit allen Schichten.

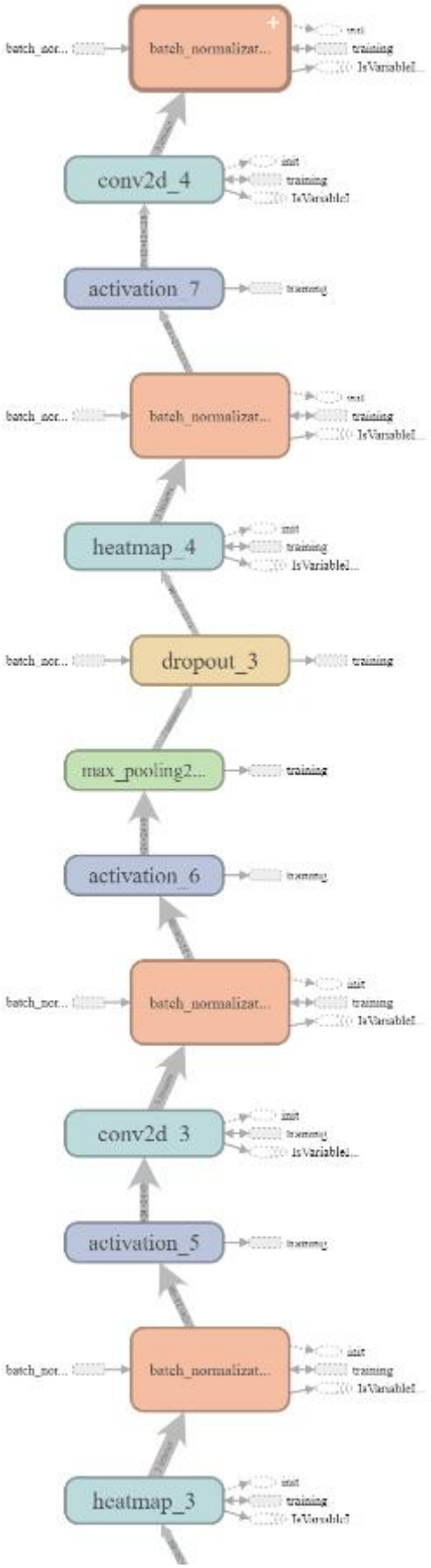

Abbildung 66: Der Finale Prototyp 1-2

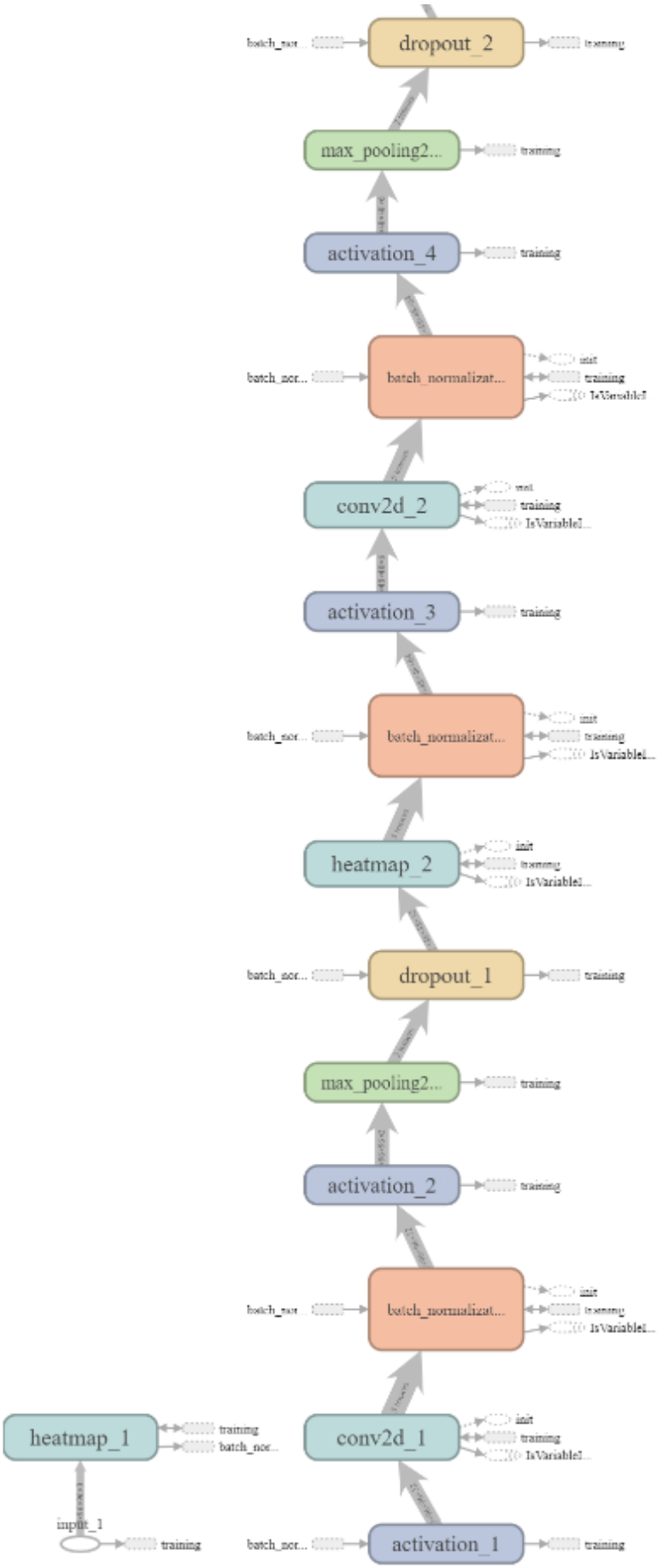

Abbildung 67: Der Finale Prototyp 2-2

Die unteren Abbildungen zeigen die jeweiligen richtigen Label und die vom Netzwerk falsch erkannten Label der Bilder der unterschiedlichen Genauigkeiten an. Wobei das Label 0 die Klasse Katze repräsentiert und das Label 1 die Klasse Hund repräsentiert.

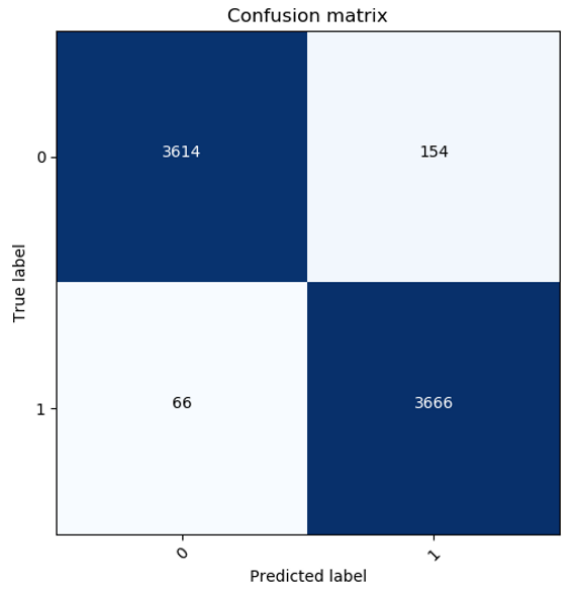

Abbildung 68: Confusion Matrix1

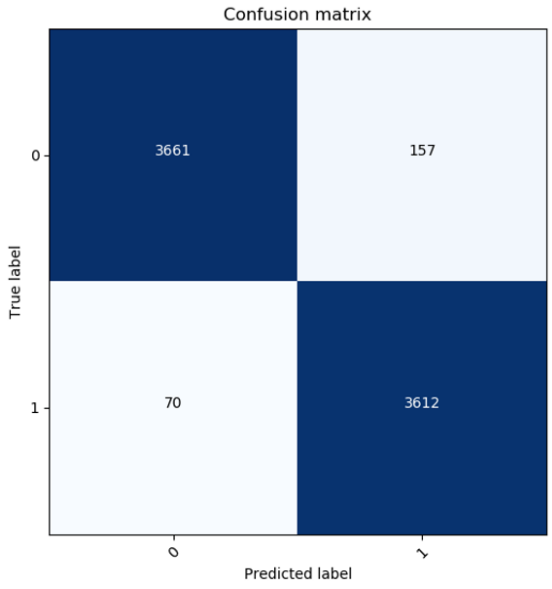

Abbildung 69: Confusion Matrix2

Das fertige Netzwerk zeigt bei der Erkennung von Katzen eine mehr als doppelt so hohe Fehlerwahrscheinlichkeit auf, wie bei der Erkennung der Hunde auf. Diese Unterschiede in der Fehlerwahrscheinlichkeit sind auf den verwendeten Datensatz zurückzuführen. Eventuell würde eine Vergrößerung des Datensatzes die Fehlerwahrscheinlichkeit des Netzwerkes weiter senken. Auch sind die Bilder in dem Datensatz teilweise zu dunkel oder zu hell, sodass keine klare Trennung von den zu erkennenden Objekten zu dem Hintergrund sattfinden kann. Auch verschwinden durch zu dunkle oder zu helle Bilder die Konturen der Tiere. Dadurch wird es unmöglich für das Netzwerk Dinge auf den Bildern, die zu dunkel oder zu hell sind zu erkennen. Die unteren Abbildungen zeigen Bilder, die das Netzwerk falsch klassifiziert hat. In der ersten Abbildung lässt sich ein Hund erahnen, ihm fehlen aber sämtliche Konturen. Im zweiten Bild sieht man eine Katze wobei der rechte, obere Übergang von der Katze zum Hintergrund nicht gegeben ist. Das dritte Bild zeigt auch wieder eine Katze, die zu hell auf dem Bild ist und somit die Konturen auch wieder verschwinden.

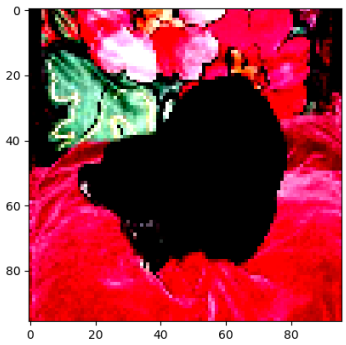

Abbildung 70: Falschklassifikation Konturen zu dunkel

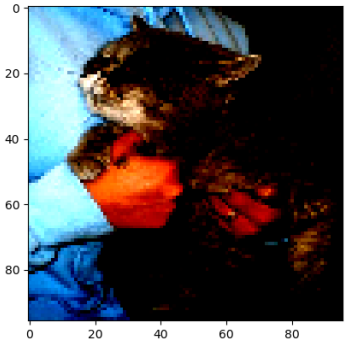

Abbildung 71: Falschklassifikation keine Trennung vom Hintergrund

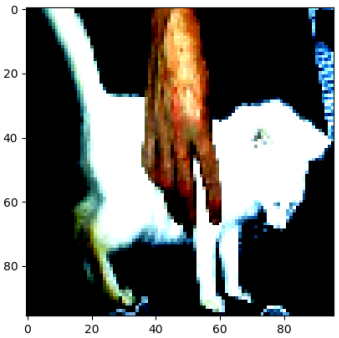

Abbildung 72: Falschklassifikation Konturen zu hell

Dies ist aber ein generelles Problem mit der JPG Komprimierung. Würden die Bilder im RAW Format vorliegen könnten diese vor der Umwandlung in JPG aufgehellt oder abgedunkelt werden. Das RAW Format ist ein Rohdatenformat, dass alle Bildinformationen eines Bildes speichern kann. Eine Aufhellung oder Abdunkelung ist mit den Bildern im Datensatz nicht möglich, da durch die JPG Komprimierung und die geringe Auflösung der Bilder die entsprechenden Pixelinformationen fehlen und schwarz immer schwarz bleibt und weiß immer weiß bleibt. Durch ein Abdunkeln oder Aufhellen würden diese Bereiche eher zu Grau werden, was dem Netzwerk nicht hilft.

Es wurde erst nach dem Training und der Ausgabe der falsch klassifizierten Bilder ersichtlich, dass in dem Datensatz auch Bilder enthalten sind, die nichts mit der Problemstellung zu tun haben. Diese Bilder verfälschen dann natürlich das Ergebnis des Netzwerkes. Um dieses Problem zu lösen müsste der gesamte Datensatz manuell auf diese Art von Bildern überprüft werden.

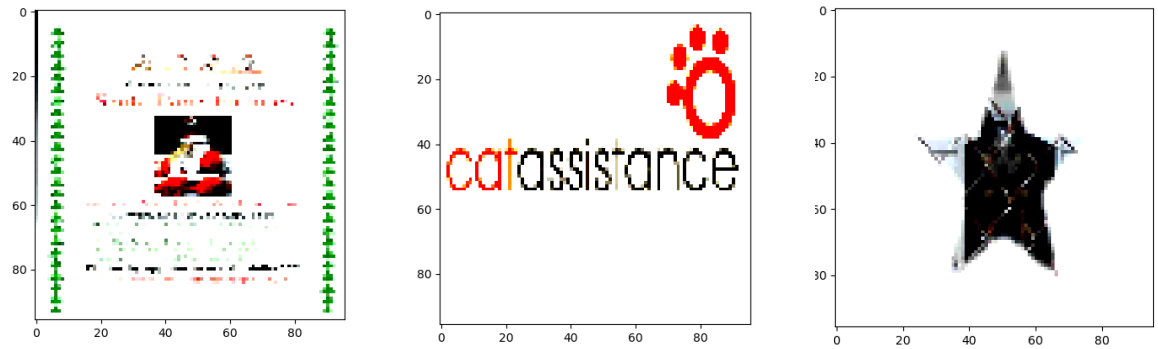

Abbildung 73: Falsche Bilder im Datensatz

Die Gewichte, die für diese Genauigkeit zuständig sind, wurden in der catsvsdogs_96.h5 abgespeichert. Somit kann das Convolutional Neuronal Network zum Testen eigener Bilder auf die vorher gespeicherten Werte der Gewichte zurückgreifen und muss nicht neu angelernt

werden, was zu einer signifikanten Beschleunigung der Performance führt. Die eigenen Bilder, die in das Netzwerk gegeben werden, werden durch das Netzwerk im ersten Schritt in die zu verarbeitende Größe von 96x96x3 Pixeln konvertiert wobei der dritte Wert die Informationen der Farbtiefe beinhaltet. Dieses so konvertierte Bild wird anschließend durch das gesamte Netzwerk propagiert, um zu einem Ergebnis zu kommen. Die Programmierte Ausgabe gibt das Bild in der für das Netzwerk richtigen Größe inklusive der Klassenzugehörigkeit und dem dazugehörigen One-Hot Vektor aus. Die untere Abbildung zeigt dieses. Das Bild gehört mit einer Wahrscheinlichkeit von 72,04 % zu Klasse Cat und zu 27,95 % zur Klasse Dog.

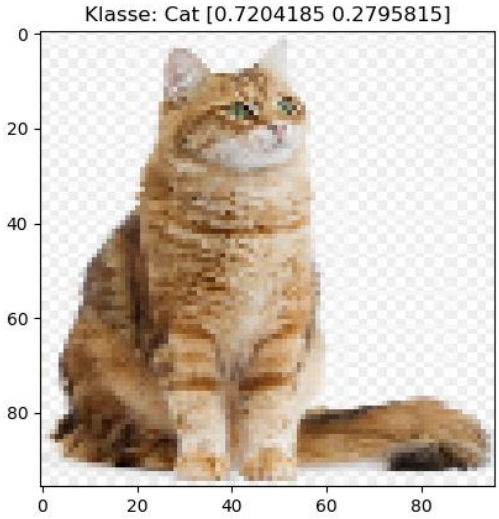

Abbildung 74: Erkanntes Bild mit Klassenzugehörigkeit und One-Hot Vektor

Zu der obigen Abbildung folgen nun die dazugehörigen Filter der einzelnen Convolution Blöcke. Hierbei wurde dem ersten Convolutional Layer eines Blockes ein Name gegeben und jeweils die ersten 16 Filter ausgegeben. Die Abbildung 75 zeigt diese Filter. In den helleren Bereichen hat der Filter etwas erkannt nachdem dieser gesucht hat. Nachdem die Filter angewandt worden sind erhöht sich auch die Tiefeninformation des Bildes von 96x96x3 Pixel zu 96x96x32 Pixel.

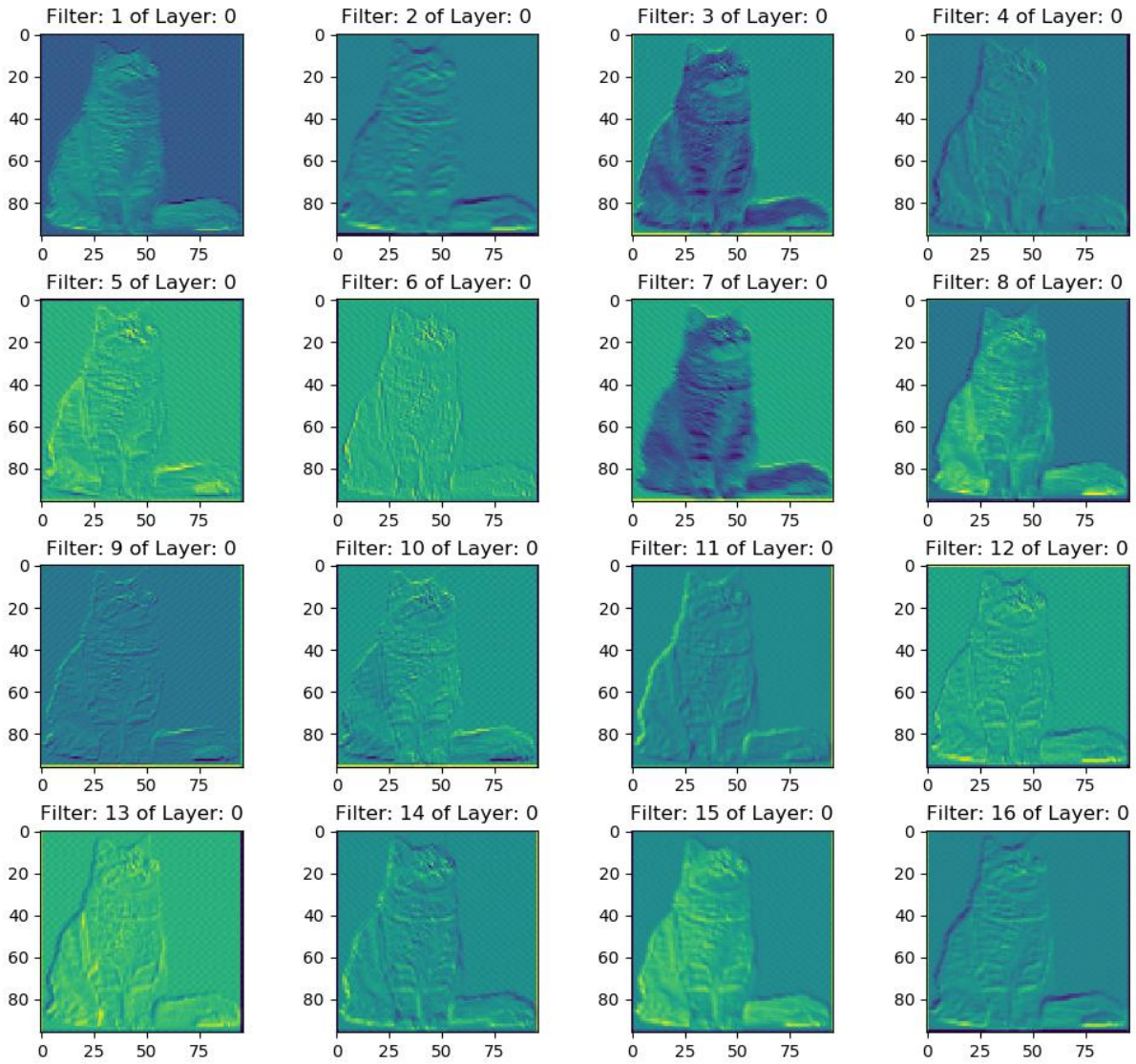

Abbildung 75: Filter des ersten Convolutional Layers im ersten Block

Der erste Convolutional Layer im zweiten Block erhält das Bild in der Dimension 48x48x32 Pixel. Durch Anwenden des Max Pooling Layers halbiert sich die Größe des Bildes aber nicht dessen Tiefeninformation. Die Abbildung 76 zeigt die ersten 16 Filter aus dem ersten Convolutional Layer im zweiten Block. Nach dem Anwenden der Filter hat das ausgehende Bild eine Dimension von 48x48x32 Pixel. Durch das Anwenden des Max Pooling Layers halbiert sich die Ausgabe des Blocks auf 24x24x32 Pixel.

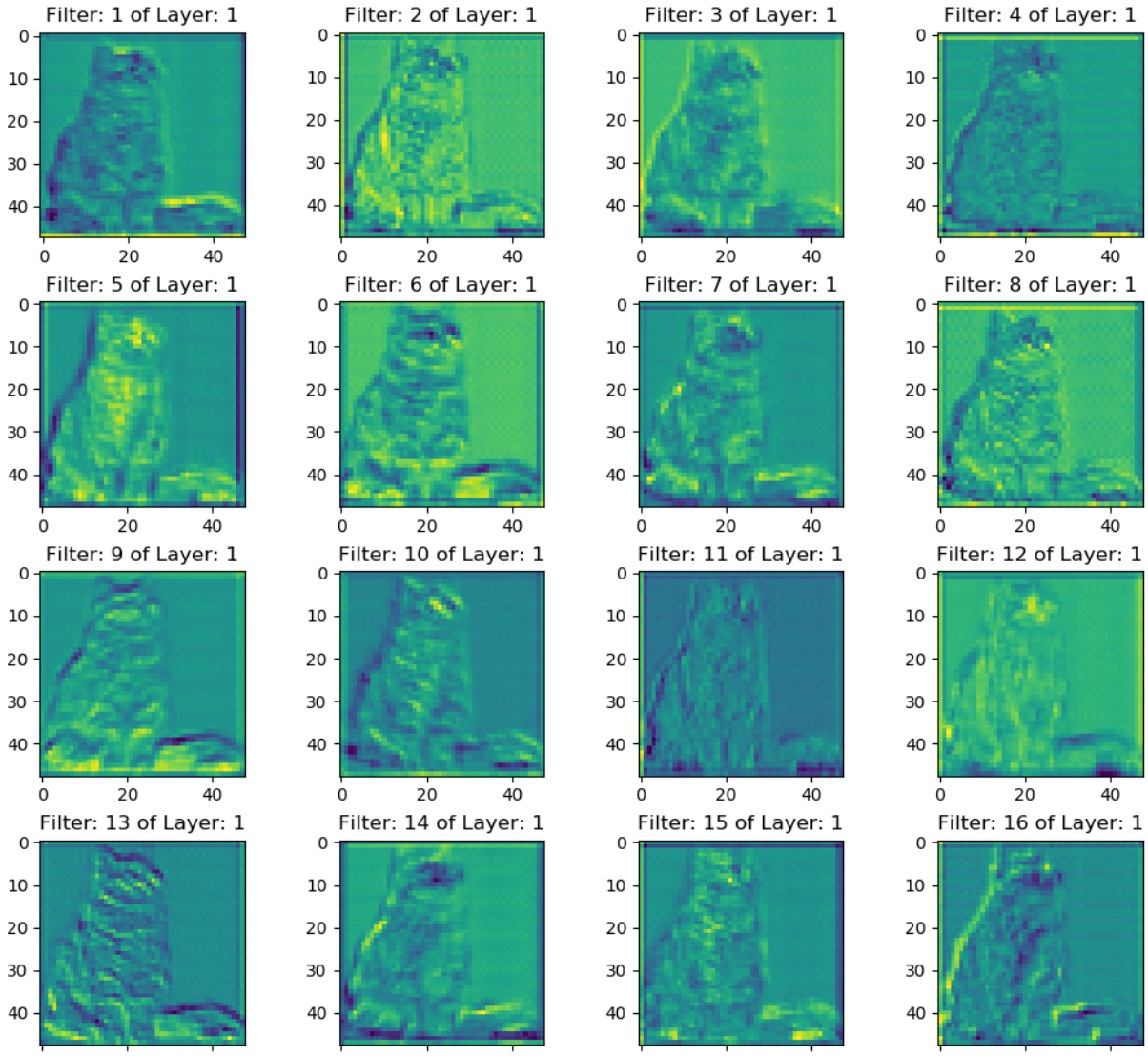

Abbildung 76: Filter des ersten Convolutional Layers im zweiten Block

Der erste Convolutional Layer im dritten Block erhält das Bild in der Dimension 24x24x64 Pixel. Auf dieses Bild werden dann die 96 Filter des Convolutional Layers des dritten Blocks angewandt, von denen 16 Filter in Abbildung 77 zu sehen sind. Nach dem Anwenden der Filter ändert sich die Dimension des Bildes von 24x24x64 zu 24x24x96 Pixel. Darauf wird im Max Pooling Layer die Bildgröße auf 12x12x96 reduziert.

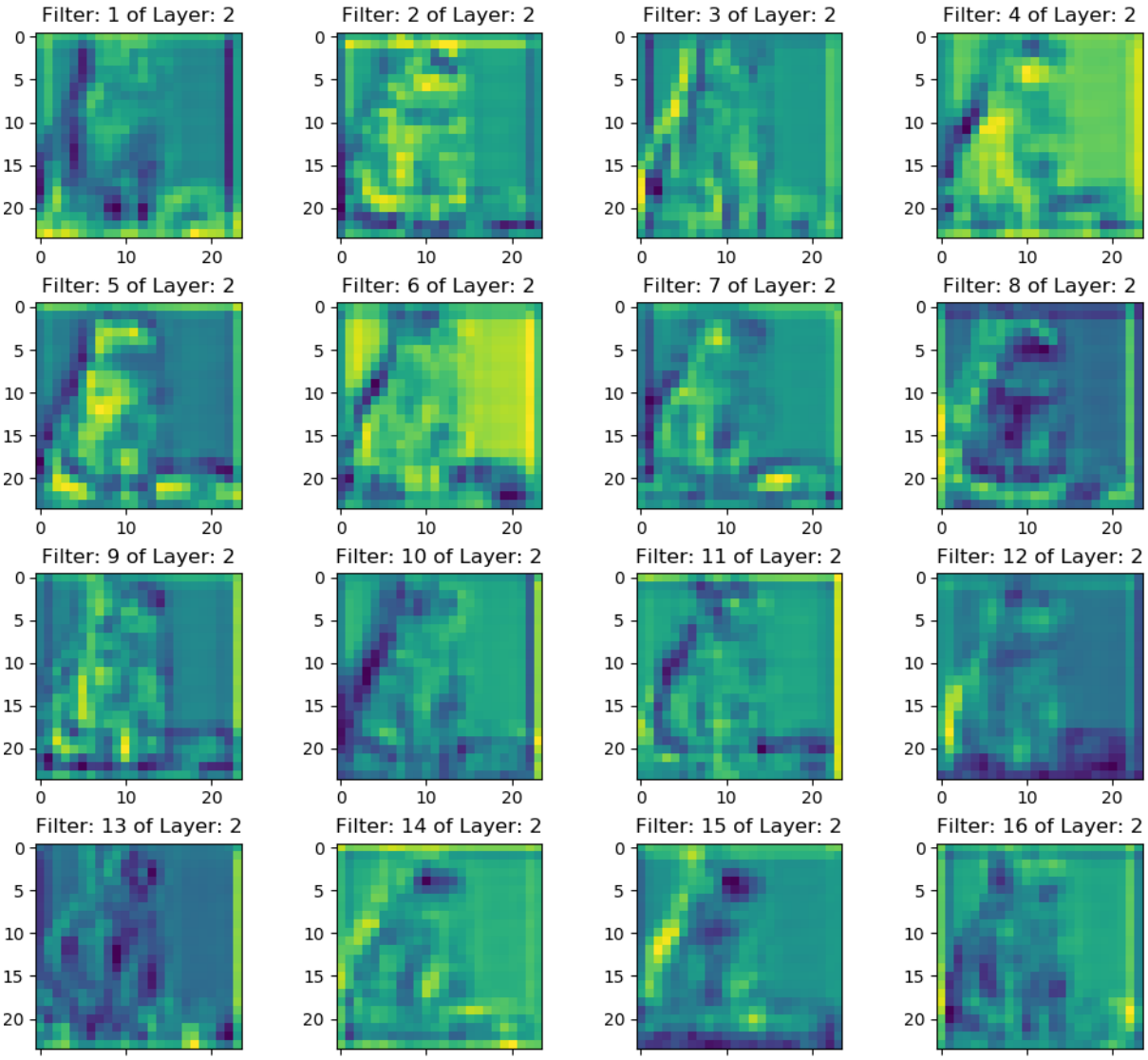

Abbildung 77: Filter des ersten Convolutional Layers im dritten Block

Nachdem das Bild von dem vorherigen Max Pooling Layer auf die Größe von 12x12x96 konvertiert wurde. Wird im letzten Convolutional Block das Bild mit 128 Filtern bearbeitet. Auch hier zeigt die untere Abbildung einen Teil dieser Filter. Am Ende dieses Blockes wird die Dimension des Bildes nochmals halbiert und an die nächste Schicht weitergegeben.

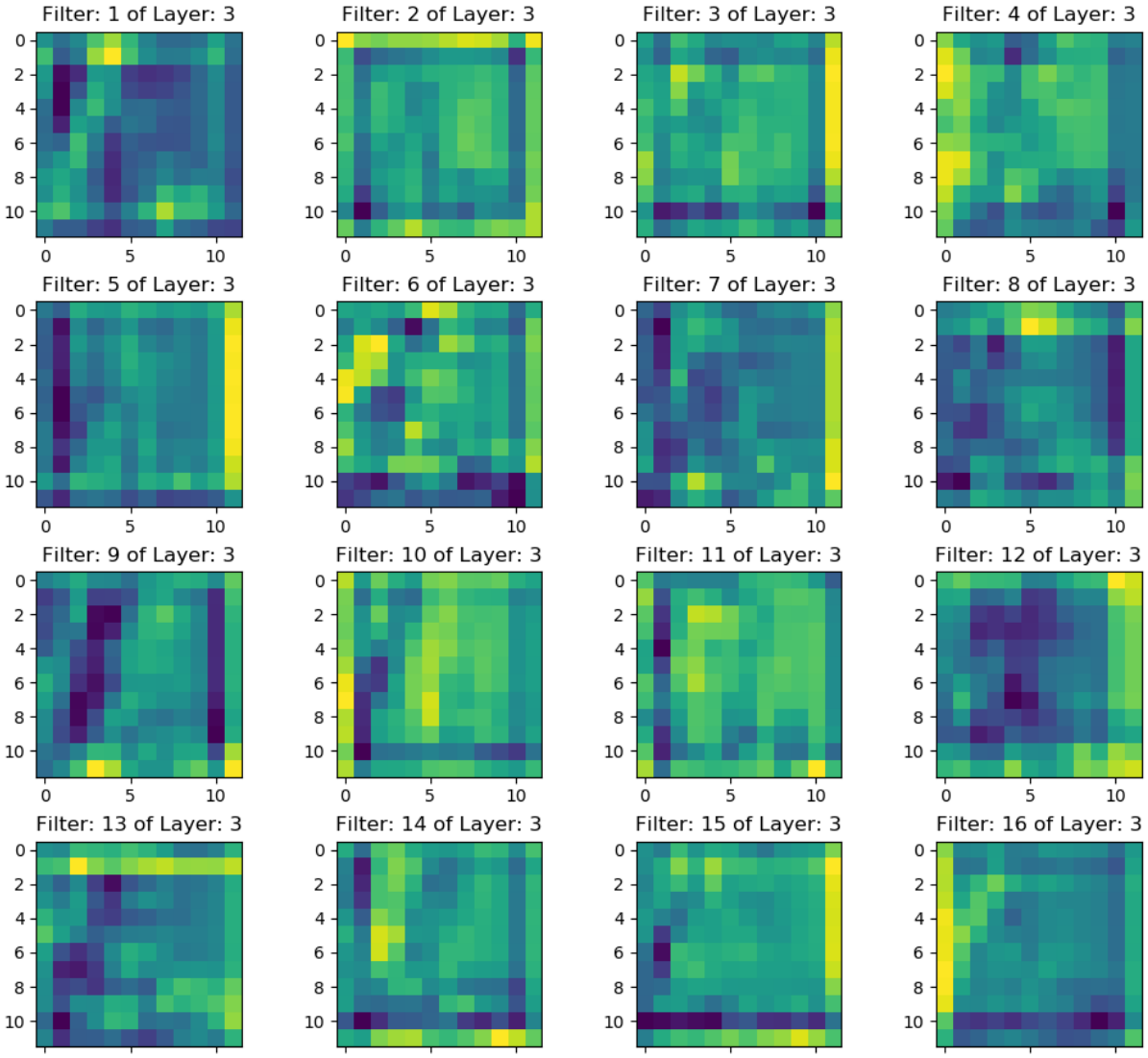

Abbildung 78: Filter des ersten Convolutional Layers im vierten Block

Aus den Bildern der Filter aus den ersten 3 Convolutional Blöcken lässt sich aus Eingangsbild noch gut erkennen. Obwohl die Größe des Bildes mit jedem Block um die Hälfte reduziert wird. Im letzten Block benötigt man eine Menge Fantasie, um hier noch die Katze des Eingangsbildes zu erkennen.

14. Resümee

Ziel dieser Arbeit war zu zeigen, dass mit Hilfe von Convolutional Neuronal Networks große Bildermengen klassifiziert werden können. Dieses Ziel konnte mit begrenzten Ressourcen erfolgreich umgesetzt werden. Diese Arbeit zeigt auch die ungeahnten Möglichkeiten der Neuronalen Netzwerke. Die Arbeit zeigte auch das für das Training von Neuronalen Netzen noch immer starke Leistungsfähige Computer mit schnellen Grafikkarten eingesetzt werden müssen. Aufgrund der gegenseitigen Abhängigkeit der Rechenleistung und dem immensen Zeitbedarf für das Training konnte in der angegebenen Zeitspanne ein sehr gutes Ergebnis erreicht werden. Der Programmieraufwand der Arbeit betrug circa 20 % der gesamten Zeit, 80 % der Zeit wurde für die Optimierung und das Training aufgebrauch. Oftmals kam es zu Trainingsabbrüchen da der Grafikspeicher der eingesetzten Grafikkarte für die enormen Datenmengen zu klein war. Durch diese Arbeit wurde sich intensiv mit den Mathematischen Grundlagen der linearen Algebra und der Methoden der Statistik auseinandergesetzt. Die dadurch geschaffene Anwendung lässt sich für weitere Anwendung in der Zukunft weiter ausbauen, woran meinerseits ein sehr großes Interesse besteht. Auch ist es eine Möglichkeit das Training in die Cloud zu verlagern um die Hardware kosten und die Trainingszeit drastisch senken zu können. In der Bearbeitungszeit war es mir nicht mehr möglich die Cloud für das Training zu verwenden.

I. Abbildungsverzeichnis

II. Formelverzeichnis

III. Tabellenverzeichnis

IV. Literaturverzeichnis

(31. 01 2019). Von neuronalesnetz.de: http://www.neuronalesnetz.de/aktivitaet.html abgerufen

(05. 01 2019). Von archive: https://web.archive.org/web/20160828193438/https://webdocs.cs.ualberta.ca/~sutton/RL-FAQ.html abgerufen

(14. 02 2019). Von medium.com: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5 abgerufen

(20. 02 2019). Von github.io: http://cs231n.github.io/convolutional-networks/ abgerufen

(11. 03 2019). Von www.machinelearningtutorial.net: http://www.machinelearningtutorial.net/2017/04/01/training-set-vs-test-set-vs-validation-set-whats-the-deal/ abgerufen

arxiv.org. (11. 03 2019). Von https://arxiv.org/abs/1212.5701 abgerufen

chatbotslife.com. (19. 03 2019). Von https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9 abgerufen

de.wikipedia.org. (14. 03 2019). Von https://de.wikipedia.org/wiki/Keras abgerufen

heise. (15. 10 2018). Von https://www.heise.de/ct/ausgabe/2016-6-Die-Mathematik-neuronaler-Netze-einfache-Mechanismen-komplexe-Konstruktion-3120565.html abgerufen

HUBEL, D. H. (1959). RECEPTIVE FIELDS OF SINGLE NEURONES IN THE CAT'S STRIATE CORTEX. Wilmer Institute, The Johns Hopkins Hospital and University, Baltimore Maryland, U.S.A: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1959.sp006308.

jaai.de. (14. 03 2019). Von https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/ abgerufen

kaggle.com. (14. 03 2019). Von https://www.kaggle.com/c/dogs-vs-cats abgerufen

keras.io. (21. 03 2019). Von https://keras.io/preprocessing/image/ abgerufen

learnopencv.com. (21. 03 2019). Von https://www.learnopencv.com/batch-normalization-in-deep-networks/ abgerufen

machinelearningmastery.com. (10. 03 2019). Von https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ abgerufen

machinelearningmastery.com. (18. 03 2019). Von https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/ abgerufen

medium.freecodecamp.org. (18. 02 2019). Von https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050 abgerufen

moodle2.cs. (11. 03 2019). Von https://moodle2.cs.huji.ac.il/nu15/pluginfile.php/316969/mod_resource/content/1/adam_pres.pdf abgerufen

notebookcheck.com. (14. 03 2019). Von https://www.notebookcheck.com/Test-NVIDIA-GeForce-GTX-965M-2016-Refresh-N16E-GR.157942.0.html abgerufen

skymind.ai. (19. 03 2019). Von https://skymind.ai/wiki/convolutional-network abgerufen

stackoverflow.com. (21. 03 2019). Von https://stackoverflow.com/questions/38553927/batch-normalization-in-convolutional-neural-network abgerufen

towardsdatascience.com. (11. 03 2019). Von https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c abgerufen

towardsdatascience.com. (21. 03 2019). Von https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad abgerufen

uibk.ac.aa. (13. 03 2019). Von https://www.uibk.ac.at/psychologie/mitarbeiter/leidlmair/neuronale_netze.pdf abgerufen

Wikipedia. (15. 10 2018). Von https://de.wikipedia.org/wiki/Neuronales_Netz abgerufen

Wikipedia. (15. 10 2018). Von https://de.wikipedia.org/wiki/Neuronales_Netz#/media/File:Neurons_big1.jpg abgerufen

Wikipedia. (15. 10 2018). Von https://de.wikipedia.org/wiki/Sigmoidfunktion abgerufen

Wikipedia. (05. 02 2019). Von https://de.wikipedia.org/wiki/Best%C3%A4rkendes_Lernen abgerufen

wikipedia.org. (18. 03 2019). Von https://de.wikipedia.org/wiki/1-aus-n-Code abgerufen

youtube.com. (18. 03 2019). Von https://www.youtube.com/watch?v=BecEHOVmx9o abgerufen

zeit.de. (13. 03 2019). Von https://www.zeit.de/digital/internet/2016-10/deep-learning-ki-besser-als-menschen abgerufen

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Ich versichere außerdem, dass ich keine andere als die angegebene Literatur verwendet habe. Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Zeichnungen, Skizzen, bildlichen Darstellungen und dergleichen.
Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Vitzenburg,
den 01.04.2019
Ort, Datum

David Ebert