

Projekt

Automatische Auswertung von Gebärdensprache mittels Maschinellen Lernens.

Mario Gruda

Matrikelnummer: 10053795

E-Mail: gruda.mario@fh-swf.de

Sebastian Schmidt

Matrikelnummer: 10053783

E-Mail: schmidt.sebastian2@fh-swf.de

Prüfer: Prof. Dr. Heiner Giefers

17. Juli 2020

Eigenständigkeitserklärung

Wir erklären, dass wir die Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel benutzt haben. Alle uns, die dem Wortlaut oder dem Sinn nach anderen Werken, gegebenenfalls auch elektronischen Medien, entnommen sind, sind von mir durch Angabe der Quelle als Entlehnung kenntlich gemacht. Entlehnungen aus dem Internet sind durch Angabe der Quelle und des Zugriffsdatums belegt. Weiterhin haben wir die vorliegende Arbeit an keiner anderen Stelle zur Abgabe vorgelegt.

Datum, Ort

Unterschrift

Datum, Ort

Unterschrift

Inhaltsverzeichnis

1. Einleitung	1
2. Fingeralphabet	1
3. Eingrenzung	3
4. Theoretische Grundlagen	4
4.1. Allgemeine Grundlagen des Maschinellen Lernens	4
4.1.1. Arten von Verfahren des Maschinellen Lernens	4
4.1.2. Typische Probleme beim Maschinellen Lernen	6
4.1.3. Datenvorverarbeitung für Bilddaten	8
4.1.4. Qualitätsmaße	9
4.1.5. Testen und Validieren	12
4.2. Neuronale Netze	13
4.2.1. Biologische Grundlagen	14
4.2.2. Künstliche Neuronen	14
4.2.3. Einschichtige Neuronale Netze	16
4.2.4. Gradientenabstiegsverfahren	17
4.2.5. Mehrschichtige Neuronale Netze	19
4.2.6. Backpropagation-Verfahren	20
4.2.7. Aktivierungsfunktionen	21
4.2.8. Regularisierungsverfahren	22
4.3. Konvolutionelle Neuronale Netze	23
4.3.1. Convolutional Layer	24
4.3.2. Pooling Layer	27
5. Datensätze	28
5.1. Sign Language MNIST Dataset	28
5.2. ASL Alphabet Dataset	28
5.3. Videodatensätze	29
6. Entwicklungsprozess	29
6.1. Datenvorverarbeitung	29
6.1.1. Sign Language MNIST Dataset	30

6.1.2. ASL Alphabet Dataset	31
6.1.3. DatasetParser Pakete	32
6.2. Entwickelte Modelle	34
6.2.1. Grundaufbau des Modelltrainings	34
6.2.2. Neuronale Netze	37
6.2.3. Konvolutionelle Neuronale Netze	39
6.2.4. Transfer Learning mittels VGG19	41
6.3. Backend der Webanwendung	43
6.4. Frontend der Webanwendung	46
6.5. Ergebnisse der Entwicklung	51
7. Fazit	51
8. Ausblick	52
8.1. Data Augmentation	52
8.2. Ausbau des Traingsdatensatzes	53
8.3. Performance des Frontends	53
8.4. Textausgabe im Frontend	53
8.5. Intervall der Handzeichenerkennung	53
9. Quellen	55

Abbildungsverzeichnis

1	Fingeralphabet	2
2	Overfitting in einem mehrschichtigen Neuronalen Netz	7
3	Beispiel einer Konfusionsmatrixe	10
4	Klassenweises Beispiel für TP, TN, FN, FP in einer Konfusionsmatrixe . .	11
5	Beispiel für Kreuzvalidierung	13
6	Aufbau eines biologischen Neurons	14
7	Aufbau eines künstlichen Neurons	15
8	Einschichtiges Neuronales Netz	16
9	Fehler im Neuronalen Netz	18
10	Gradientenabstiegsverfahren in zwei Dimensionen	19
11	Mehrschichtiges Neuronales Netz	20
12	Aufbau des visuellen Cortex	24
13	Einzelner Filter eines Convolutional Layers	25
14	Beispiel für horizontale und vertikale Filter	25
15	Beispiel für verschiedene visualisierte Filter	26
16	Beispiel eines CNNs mit mehreren Feature Maps	26
17	Arbeitsweise des Max Pooling	27
18	Eingesetztes Neuronales Netz mit fünf Hidden-Layern	38
19	Eingesetztes CNN für den Sign Language MNIST Dataset	40
20	Eingesetztes CNN für den ASL-Alphabet Dataset	40
21	Aufbau des VGG19	41
22	Angepasstes VGG19	42
23	„D“ Handzeichen, als Icon und im Original	47
24	Bild der Webcam	49
25	Punktedarstellung eines Handzeichen in Mediapipe	50

Listingverzeichnis

1	Kern der Sign Language MNIST Dataset Vorverarbeitung.	30
2	Kern der Sign Language MNIST Dataset Vorverarbeitung.	32
3	Beispielhafter Aufruf der Funktion zum Einlesen von Daten	33
4	Beispielhafte Vorbereitung für das Training	34
5	Kompilierung eines Modells	35
6	Logging Funktion für Trainingsvorgänge	35

7	Start des Trainings mit Keras	36
8	Auswertung eines Modells mit Testdaten	36
9	Anlegen der HTTP-Endpunkte.	43
10	Abfrage der möglichen Modelle.	45
11	Verarbeitung der Bilder im HTTP-Endpunkt.	45
12	Laden der frontendseitigen MediaPipe Hands Modelle.	47
13	Erkennen einer Hand mit MediaPipe Hands.	48

Formelverzeichnis

1	Kreuzentropie	10
2	Vorhersagegenauigkeit	11
3	Recall	11
4	Precision	12
5	F1-Score	12
6	Mathematische Definition eines Künstlichen Neurons	15
7	Schwellenwertfunktion eines Perzeptrons	15
8	Matrizendarstellung eines einschichtigen Neuronalen Netzes	17
9	Mathematische Definition eines einschichtigen Neuronalen Netzes	17
10	Ziel des Gradientenabstiegsverfahren	17
11	ReLU	22
12	ELU	22
13	Softmax	22

1. Einleitung

Im Rahmen der Veranstaltung Spezielle Algorithmen des Masterstudiengangs für angewandte Informatik der Fachhochschule Südwestfalen soll zum Abschluss ein Projekt im Bereich Machine Learning durchgeführt werden. Das nachfolgende Dokument beschäftigt sich mit der Erkennung von Buchstaben, die mithilfe des amerikanischen Fingeralphabets gezeigt wurden. Ebenso wie in sprachlichen Übersetzungen, wird im Bereich der Erkennung von Gebärden und des Fingeralphabets viel geforscht, um Sprachbarrieren zu überwinden. Anders als in sprachlichen Übersetzungen steht keine Textuelle Analyse in Bezug auf Grammatik und Semantik im Vordergrund, sondern die Analyse von Bildmaterial. Das Dokument ist in einen theoretischen und einen praktischen Teil gegliedert. Der theoretische Teil beschäftigt sich vorerst mit den Grundlagen des maschinellen Lernens, woraufhin dann die Algorithmen und die genutzten Modelle näher vorgestellt werden. Der praktische Teil beschreibt die Integrierung der Modelle mit einem webbasierten Frontend, um sie in einer realen Umgebung testen zu können. Eventuelle Optimierungen für den praktischen Teil werden nur kurz angesprochen, aber nicht in aller Tiefe implementiert und getestet. Das Hauptaugenmerk wird aufgrund des Umfangs auf die Theorie gelegt. Abschließend werden aufgetretene Probleme und ein Ausblick näher beleuchtet.

2. Fingeralphabet

Das internationale Einhand-Fingeralphabet erlaubt es, die lateinischen Buchstaben ohne Unterscheidung der Groß- und Kleinschreibung mithilfe von Handzeichen zum Ausdruck zu bringen. Jedes Zeichen kann mit nur einer Hand nachgebildet werden, wobei die Handzeichen den Kleinbuchstaben nachempfunden sind. So ist beispielsweise das „a“ eine geschlossene Faust mit angelegtem Daumen. Das Fingeralphabet wird üblicherweise mit der dominanten Hand gebildet und wird vor allem genutzt, um die Gebärdensprache zu ergänzen, sollte es noch keine Gebärde geben. Alternativ kann mit dem Fingeralphabet ein Wort besonders betont werden. Anders als die Gebärdensprache, wird das Fingeralphabet nicht genutzt, um eine ganze Unterhaltung zu führen. Je nach Land unterscheiden sich die Handzeichen in Form und Menge. So musste das deutsche Fingeralphabet um Umlaute ergänzt werden. Außerdem unterscheiden sich beispielsweise die Buchstaben M

und N zwischen dem deutschen und amerikanischen Alphabet. Da das Fingeralphabet genutzt wird, um die Gebärdensprache zu ergänzen, wird in diesem Projekt mit dem Begriff Gebärdensprache das Fingeralphabet bezeichnet.

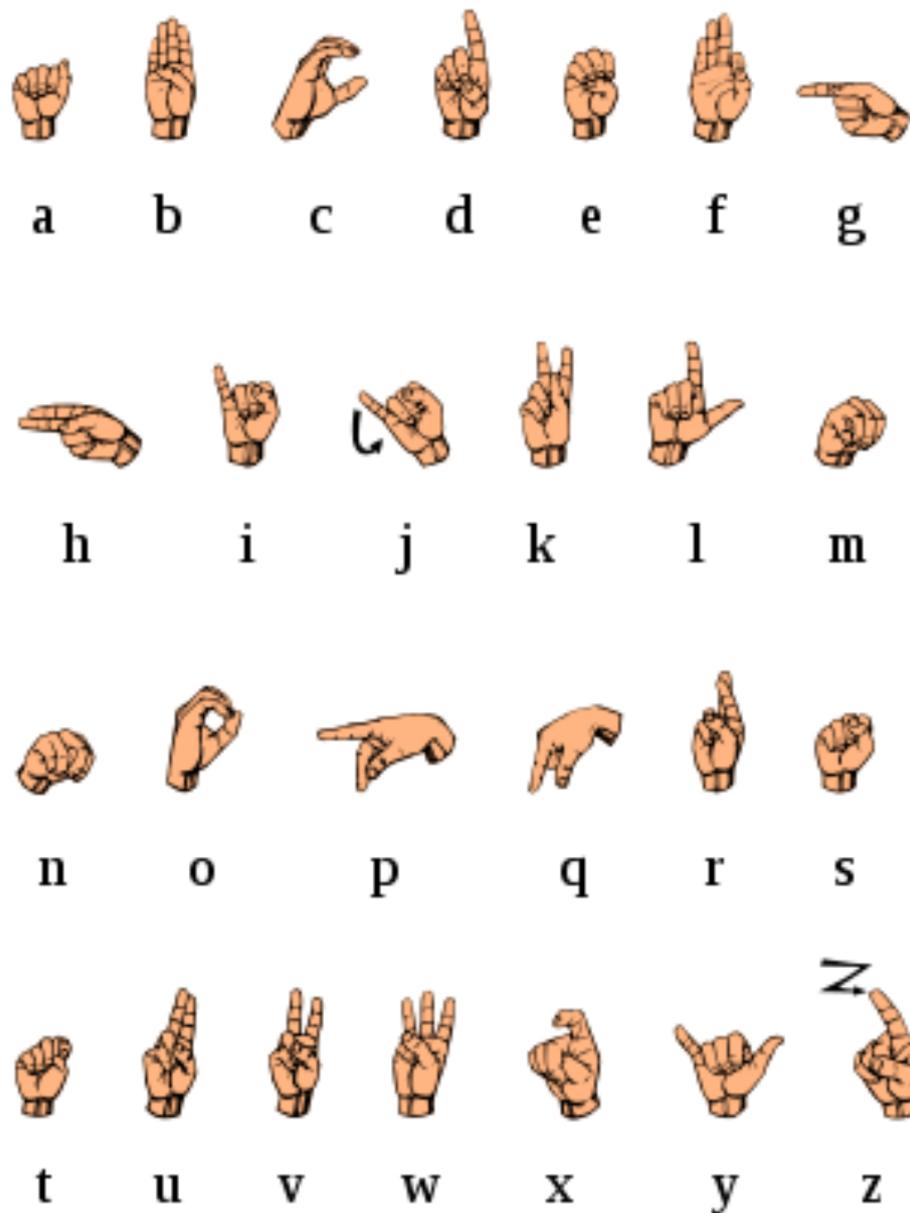


Abbildung 1: Darstellung des Fingeralphabets [1].

3. Eingrenzung

Das Ergebnis dieses Projekts soll eine schriftliche Ausarbeitung und Quellcode für das Modell-Training sowie für ein Python Backend und Angular Frontend sein. Die Zeichenerkennung wurde auf das unter Abbildung 1 dargestellte amerikanische Fingeralphabet begrenzt. Umlaute oder Zahlen werden also nicht erkannt. Die Buchstaben J und Z sind zwar in einem der verwendeten Datensätze enthalten, erfordern aber eine gewisse Gestik, sodass die Erkennung in Standbildern erschwert ist. Da das hier vorgestellte Problem auf die Erkennung von Bildern zurückzuführen ist, wird in diesem Projekt auf Konvolutionelle Neuronale Netze gesetzt, welche im ImageNet Large Scale Visual Recognition Challenge (ILSVRC) die besten Ergebnisse erzielen. Vor Projektvergabe wurde auf einem kleinen Datensatz mit 28x28 Merkmalen eine Support Vektor Maschine mit RBF Kernel trainiert. Die erreichte Vorhersagegenauigkeit von ca. 79% auf dem Testdatensatz wurde mit einem CNN aber schnell überboten. Zusätzlich wurden zunächst einfachere Neuronale Netze antrainiert, welches unter 6.2.2. beschrieben werden. Auch hier wurden Tests und Optimierungen aufgrund von ernüchternden Ergebnissen zügig abgebrochen und dienen in der Ausarbeitung nur als Vergleichswert. Im Rahmen der Vorlesung wurden unter anderem die Bibliotheken ScikitLearn, NumPy, Tensorflow und Keras bearbeitet, sodass sich in der Projektarbeit größtenteils darauf beschränkt wird. Die Nutzung der trainierten Modelle zielt darauf ab, Bilder einer Webcam in einem Intervall an ein Python Backend zu senden. Im Backend werden die bereits trainierten Modelle hinterlegt, welche dann eine Voraussage für das gesendete Bild treffen. Das Ergebnis entspricht einer Klassifikation, wobei jeder mögliche Buchstabe einer Klasse zugeordnet ist. Das Frontend soll als Antwort die jeweiligen Wahrscheinlichkeiten und den wahrscheinlichsten Buchstaben als separates Property bekommen, der entsprechend im Frontend visualisiert wird. Das Python Backend gibt über einen Endpunkt die hinterlegten Modelle aus, sodass diese im Frontend jederzeit gewechselt werden können. Als Schnittstelle zwischen Frontend und Backend dienen einfache HTTP-Anfragen, sodass das Frontend ohne Probleme ausgetauscht oder durch ein weiteres ergänzt werden kann.

4. Theoretische Grundlagen

In diesem Abschnitt sollen theoretische Grundlagen des Maschinellen Lernens vorgestellt werden. Es wird zunächst auf allgemeine Grundlagen eingegangen, um im Folgenden Neuronale Netze und Konvolutionelle Neuronale Netze zu betrachten.

4.1. Allgemeine Grundlagen des Maschinellen Lernens

Maschinelles Lernen gibt einem Computer die Möglichkeit, Aufgaben zu erfüllen, ohne explizit von einem Entwickler für diese programmiert worden zu sein. Tom Mitchell beschreibt Maschinelles Lernen 1997 als ein Computerprogramm, das ohne Veränderungen des Programmcodes aus Erfahrungen E im Bezug auf eine Aufgabe T und ein Maß für die Leistung P lernt, indem seine Leistung P mit der Erfahrung E anwächst [2, S. 4]. Erfahrungen sammelt ein System, indem Daten mit speziellen Algorithmen betrachtet werden. Auf Basis dieser Daten erstellt der Algorithmus dann eine Strukturbeschreibung, die auch als Modell bezeichnet wird [3, S. 2].

In diesem Projekt war die Aufgabe das Klassifizieren von Gebärdensprache. Die Leistung des Modells wird durch die korrekte Klassifizierung bestimmt, welche idealerweise während des Trainingsvorgangs beim Anlernen von Datensätzen zu Gebärdensprache ansteigen sollte.

4.1.1. Arten von Verfahren des Maschinellen Lernens

Verfahren des Maschinellen Lernens werden in der Literatur häufig mittels verschiedener Kategorien in unterschiedliche Arten aufgeteilt. Auch wenn viele Arten von Verfahren in dieser Arbeit nicht betrachtet werden, ist es wichtig, diese zu benennen, um ein passendes Verfahren auszuwählen. Es werden typischerweise drei Kategorien unterschieden, die Verfahren des Maschinellen Lernens einteilen [2, S.8-14][4, S.2].

Zunächst wird ein Verfahren danach eingeteilt, welche Informationen zu den Trainingsdaten anhand von Labels nötig sind. Muss jeder Datenpunkt mit einem Label versehen sein, so handelt es sich um Überwachtes Lernen. Demgegenüber steht

das Unüberwachte Lernen, bei dem ein Algorithmus versucht, Aussagen über mögliche Label anhand der Struktur der Daten zu treffen. Das Halbüberwachte Lernen kombiniert beide Eigenschaften und der Algorithmus versucht zunächst Label anzulernen, um im Folgenden autonom fortzufahren. Abschließend sei noch das Reinforcement Lernen zu nennen. Dieses bestraft und belohnt Aktionen auf eine vorher definierte Weise. Der Algorithmus versucht dann, möglichst viele Belohnungen in möglichst wenig Zeit zu erhalten und gleichzeitig Bestrafungen zu verhindern [2, S.8-14][4, S.2].

Eine weitere Kategorie ist die Art, wie ein Lernverfahren mit Daten versorgt werden muss. Beim Batch-Learning muss das Verfahren auf Basis eines kompletten Datensatzes trainiert und kann danach nicht mehr mit weiteren Daten verbessert werden. Es muss von Grund auf neu trainieren. Demgegenüber steht das Online-Learning, welches nach und nach trainiert wird, und noch eine spätere Verfeinerung ermöglicht [2, S.14-17][4, S.2].

Mit einer weiteren Kategorie wird festgelegt, wie ein Verfahren Daten verallgemeinert, um neue Aussagen treffen zu können. Beim Modellbasierten Lernen wird anhand der Daten ein mathematisches Modell erstellt, welches versucht, die Struktur der Daten korrekt zu beschreiben, um neue Vorhersagen treffen zu können. Instanzbasiertes Lernen vergleicht bei vorherzusagenden Daten die Merkmale mit bereits angelernten Datensätzen. Werden hier Ähnlichkeiten in den bereits erlerten Instanzen entdeckt, werden diese für eine neue Aussage genutzt [2, S.14-17][4, S.2].

Ebenfalls lassen sich Verfahren nach der Art des zu lösenden Problems einteilen. Bei der Klassifikation werden Daten anhand ihrer Labels einer gewissen Klasse zugeordnet. Das Ziel ist es nun, die Klasse von neuen Datensätzen korrekt vorherzusagen. Demgegenüber stehen Regressionsprobleme, bei denen ein Label einen konkreten Wert angibt, welcher bei einem neuen Datensatz vom Modell korrekt vorhergesagt werden muss [2, S.8-9][4, S.2].

Bei dem in dieser Arbeit zu lösenden Problem handelt es sich um ein typisches Klassifikationsproblem des Überwachten Lernens. Ein Modell wird auf Basis von vorklassifizierten Daten, also typischerweise Datensätze mit Bildern von Händen, die ein Zeichen in Gebärdensprache repräsentieren, angelernt. Dieses soll dann im Fol-

genden weitere Bilder einem Zeichen in Gebärdensprache korrekt zuweisen können. Instanzbasiertes Lernen könnte bei Bilddaten zu Problemen führen, da Bilder in der Regel relativ viele Daten beinhalten und dadurch zu sehr großen Modellen führen können [2, S.18]. Aus diesem Grund sollte Modellbasiertes Lernen vorgezogen werden. Es ist allerdings sowohl Batch-Learning, als auch Online-Learning denkbar.

4.1.2. Typische Probleme beim Maschinellen Lernen

Beim Maschinellen Lernen können viele Probleme auftreten, die den Lernerfolg eines Modells behindern können. Um diesen entgegenwirken zu können, sollten typische Probleme vor dem Entwickeln eines Modells betrachtet werden.

Schon bei einfachen Problemen ist eine ausreichende Datenmengen nötig, um Modelle antrainieren zu können. Abhängig von der Komplexität eines Problems und des genutzten Algorithmus können Tausende bis Millionen von Datensätzen für einen Lernvorgang nötig sein [2, S.23][4, S.4].

Wurden Daten fehlerhaft gesammelt und repräsentieren nicht das Spektrum der möglichen Eingabe im Einsatz, so kann ein Modell keine korrekten Vorhersagen für den geplanten Einsatz treffen, da hier das Wertespektrum nicht das Gelernte repräsentiert. Auch eine ungünstige Verteilung von Klassen in den Daten kann zu Problemen führen; und zwar besonders dann, wenn dieser Fall mit unpassenden Qualitätsmaßen ausgewertet wird [2, S.24-25][4, S.4].

Minderwertige Daten erschweren ebenfalls den Lernvorgang. Dies kann sich durch verrauschte, fehlerhafte und fehlende Daten sowie Ausreißer äußern. Eine manuelle oder automatische Vorverarbeitung könnte dies verbessern. Sogar ein Auslassen bestimmter Merkmale kann sich als sinnvoll herausstellen. [2, S.26][4, S.3]. Hilft dies nicht, ist ein Sammeln von neuen Datensätzen sinnvoll.

Irrelevante oder schlecht gewählte Merkmale für das Antrainieren eines Datensatzes führen ebenfalls zu einem schlechteren oder gar keinem Lernerfolg. Für das Vorhersagen von Gebärdensprache sollte zum Beispiel das Datum des vorherzusagenden Bildes keinen Einfluss haben. Wird diese Information im Trainingsvorgang trotzdem einbezogen, so ist mit einer schlechteren Performanz des Lernvorgangs

zu rechnen [2, S.26][4, S.4].

Overfitting tritt dann auf, wenn ein Modell für die Trainingsdaten zu komplex gewählt ist. Es führt dazu, dass ein Modell nicht korrekt verallgemeinert, sondern zufällige Variationen in den Trainingsdaten lernt. Es äußert sich in einer guten Performance im Training, während diese in Tests nachlässt. Verhindern lässt sich dies im Modell durch eine Verringerung der Zahl der Parameter oder bestimmter Techniken, die dem Modell Restriktionen aufliegen, der sogenannten Regularisierung. Außerdem lässt es sich durch Sammeln neuer Daten, sowie der Verringerung von Rauschen in bereits Vorhandenen reduzieren [2, S.27][4, S.13].

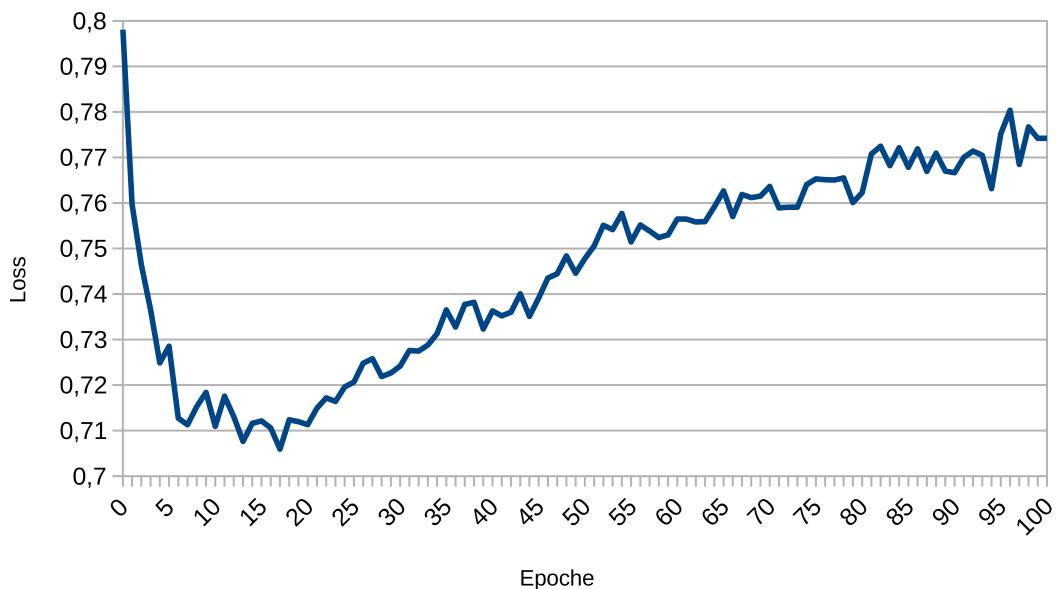


Abbildung 2: Nach der 20. Epoche tritt in diesem mehrschichtigen Neuronalen Netz ein Overfitting auf [4, S.14].

Zuletzt sei das Underfitting genannt, welches dem Overfitting gegenübersteht. Ein Modell ist dabei zu einfach, um die Strukturen in den Trainingsdaten anzulernen. Modelle mit mehr Parametern, bessere Merkmale oder eine Verringerung der Regularisierung reduzieren Underfitting in der Regel [2, S.29][4, S.14].

4.1.3. Datenvorverarbeitung für Bilddaten

Wie im vorherigen Kapitel dargestellt, würden qualitativ unzureichende Daten zu einer schlechteren Performanz beim Anlernen eines Modells führen. Dies kann auch bei Bilddaten der Fall sein und sollte durch Vorverarbeitungsschritte verhindert werden. In dieser Sektion sollen einige typische Schritte der Bildvorverarbeitung dargestellt werden.

Verrauschte Bilder können bei der Bilderkennung große Probleme darstellen. Ist ein Bild nicht rauschfrei, wurde gezeigt, dass Verfahren zum Entfernen von Rauschen eine positiven Effekt haben können [5]. Dieses Verfahren wurde im Praxisteil zwar aufgrund des qualitativ hochwertigen Datensatzes nicht genutzt, stellt aber für die Zukunft, besonders im Einsatz, wo die Qualität der Bilddaten nicht immer gewährleistet ist, eine gute Methode für eine möglichen Verbesserung dar.

Ein weiterer Datenvorverarbeitungsschritt ist das Extrahieren von relevanten Bildausschnitten. Die meisten Datensätze für die Erkennung von Gebärdensprache haben hier bereits die relevanten Ausschnitte vorverarbeitet, sodass nur noch die Hände zu sehen sind. Dies ist jedoch ein Problem für die Praxis, da ein Bild einer Webcam in der Regel mehr als nur die Hand beinhaltet. Es würde hier das Problem entstehen, dass die Daten in der Praxis nur wenig mit dem Angelernten übereinstimmen. Hier ist es also sinnvoll, den relevanten Teil eines Bildes zu extrahieren. Dazu wurde im Praxisteil ein Machine-Learning Framework namens MediaPipe genutzt [6].

Zuletzt müssen Bilder in der Regel vor der Nutzung skaliert werden. Modelle des Maschinellen Lernens erwarten in der Regel eine feste Eingabegröße. Dies gilt auch für Bilder, sodass beispielsweise das VGG19 Konvolutionelle Neuronale Netzwerk für Objekterkennung eine Eingabegröße von 224 x 224 Pixeln erwartet [7, S.2]. Auch mit Blick auf die Laufzeit ergibt eine Skalierung von Bildern Sinn. Moderne Webcams erreichen häufig schon eine Auflösung von 1920 x 1080 Pixeln. Dies würde bei unskalierten und nicht extrahierten Bildinformationen zu einer Eingabegröße von 2 Millionen Parametern führen, was abhängig vom Modell zu sehr hohen Laufzeiten führen kann. Typische Algorithmen zum Herunterskalieren von Bildern sind die Nächste-Nachbarn, Bilineare und Bikubische Interpolation [7, S.2].

4.1.4. Qualitätsmaße

Um die Leistung eines Modells korrekt bewerten zu können, müssen abhängig vom Ziel Qualitätsmaße eingeführt werden, die es möglich machen, Modelle untereinander zu vergleichen. Dazu wird mit der Kreuzentropie ein typisches Qualitätsmaß für Klassifikationsprobleme eingeführt. Außerdem werden die Begriffe Konfusionsmatrix, Accuracy, Precision, Recall und F1-Score diskutiert.

In praktisch allen Fällen können Algorithmen des Maschinellen Lernens nicht mit Klassenbezeichnungen in Form von Zeichenketten umgehen. Eine Menge von Klassenbezeichnungen wie $\{a, b, c\}$ könnte nur schwierig direkt von Algorithmen als Label genutzt werden. Hier werden in der Regel zwei Möglichkeiten genutzt, um Bezeichnungen in nützlichere Label umzuwandeln. Diese wären:

- Klassen-IDs: Die vorher angegebene Menge von Klassenbezeichnungen könnte in eine Menge von repräsentierenden IDs transformiert werden. So wird die Menge $\{a, b, c\}$ in die IDs $\{0, 1, 2\}$ umgewandelt. Der Algorithmus muss bei der Klassifikation nun eine korrekte IDs vorhersagen, welche eine Klasse repräsentiert. Aufgrund der Arbeitsweise vieler Algorithmen nehmen diese allerdings an, dass IDs, die näher beieinander liegen, zueinander ähnlicher sind. Dies ist nicht für alle Problemstellungen ideal [4, S.12][2, S.63].
- One-Hot-Kodierung: Bei diesem Verfahren wird für jede Klasse eine Zahl in einem Vektor reserviert. Eine vorliegende Klasse wird dann in der Regel mit einer 1 und der Rest der Elemente mit einer 0 markiert. Für die Beispieldmenge $\{a, b, c\}$ ergäbe sich daher die Menge von Vektoren $\{(1 \ 0 \ 0), (0 \ 1 \ 0), (0 \ 0 \ 1)\}$. Korrekt klassifiziert wurde nun, wenn die vorliegenden Klassen mit den höchsten Vorhersagen markiert werden. Da in diesem Fall keine Beziehung zwischen den Klassen erkannt wird, ist diese Darstellung auch ideal für die betrachtete Problemstellung [4, S.12][2, S.63].

Die bereits vorher erwähnte Kreuzentropie ist ein ideales Qualitätsmaß für Klassifikationsprobleme in der One-Hot-Kodierung. Niedrige Vorhersagen bei den vorliegenden Klassen werden dabei abgestraft, während andere Vorhersagen ignoriert werden. Für einen Datensatz mit m Tupeln, seinen One-Hot-kodierten Labels \mathbf{y} mit K -Klassen und den One-Hot-kodierten Vorhersagen \mathbf{y}' mit K -Klassen gilt [4,

S.13][2, S.63]:

$$J = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \mathbf{y}_{ik} \cdot \log(\mathbf{y}'_{ik}) \quad (1)$$

Nun wurde ein Qualitätsmaß eingeführt, mit dem die Korrektheit einer Vorhersage bestimmt werden kann. Im Folgenden werden nun Wege beschrieben, wie sich die Performanz eines Modells auf einem gesamten Datensatz auswerten lässt. Eine typische und ausführliche Auswertungsmöglichkeit bei Klassifikationsproblemen ist eine Konfusionsmatrix, für die ein Beispiel in Abbildung 3 zu sehen ist. Jede Zeile in einer Konfusionsmatrix steht für eine tatsächliche Klasse, während eine Spalte eine vorhergesagte Kategorie repräsentiert (dies ist definitionsabhängig und kann auch getauscht werden). Die Wertepunkte in der Matrize beschreiben damit, als welche Klasse ein Datenpunkt einer tatsächlichen Klasse vorhergesagt wird. Sie werden meistens als feste Anzahl oder Anteile in Prozent aller Vorhersagen zu dieser Klasse angegeben. Die Hauptdiagonale der Matrize beschreibt alle korrekt klassifizierten Vorhersagen [2, S.86-87].

	A	B	C
A	90	4	6
B	12	85	2
C	4	19	80

Abbildung 3: Beispiel einer Konfusionsmatrix für die Klassen A, B, C.

Eine Konfusionsmatrix ist sehr ausführlich und beinhaltet besonders bei vielen Klassen viele Informationen, die nicht unbedingt für die Lösung des betrachteten Problems relevant sind. In diesem Fall können andere Qualitätsmaße sinnvoll sein. Um diese zu errechnen, ist ein Verständnis über die Begriffe True Positive (TP), True Negative (TN), False Positive (FP) und False Negative (FN) nötig [2, S.87]. Diese sind beispielsweise, abhängig von Klasse A, in der Abbildung 4 zu erkennen. Es gilt:

- True Positive (grün): Beschreibt den korrekt klassifizierten Anteil der betrachteten Klasse.
- True Negative (hellgrün): Beschreibt den korrekt klassifizierten Anteil der nicht betrachteten Klassen.

- False Positive (rot): Beschreibt den Anteil, mit dem die nicht betrachteten Klassen fälschlicherweise als die Betrachtete eingeordnet werden.
- False Negative (orange): Beschreibt den Anteil, mit dem eine betrachtete Klasse fälschlicherweise als die nicht betrachteten Klassen eingeordnet wird.

	A	B	C
A	90	4	6
B	12	85	2
C	4	19	80

Abbildung 4: Beispiel für TP, TN, FN, FP der Klasse A in einer Konfusionsmatrix.

Nun können auf Basis der eingeführten Begriffe einige Qualitätsmaße eingeführt und ihr Einsatz abhängig vom Datensatz diskutiert werden. Die Vorhersagegenauigkeit A stellt alle korrekt klassifizierten Datenpunkte (TP, TN), der Menge aller klassifizierten Datenpunkte (M) gegenüber (2). Die Vorhersagegenauigkeit ist ein beliebtes Qualitätsmaß, welches jedoch einige Probleme aufweist. Ist ein Datensatz unbalanciert, ist eine Klasse also deutlich mehr vertreten als eine andere, so wird diese Klasse die Vorhersagegenauigkeit stärker beeinflussen als andere. Dies kann zu einem Klassifikator führen, der lediglich die am stärksten vertretene Klasse antizipiert [2, S.85-86].

$$A = \frac{TP + TN}{M} \quad (2)$$

Der Recall (3) ist ein Maß, welches die korrekt klassifizierten Datenpunkte allen Datenpunkten mit dieser Klasse gegenüberstellt. Er ist damit sehr gut für unterrepräsentierte Klassen geeignet, da ein Fokus auf die korrekte Klassifikation gesetzt wird [2, S.87]. Ein möglicher Anwendungsfall wäre damit die korrekte Klassifikation von Krankheitsfällen, da diese in der Regel unterrepräsentiert sind.

$$R = \frac{TP}{TP + FN} \quad (3)$$

Weiterhin gibt es mit der Precision (4) ein Qualitätsmaß, welches dem Recall mit einer Wechselbeziehung gegenübersteht. Statt die False Negatives zu betrachten, betrachtet es mit den False Positives die Datenpunkte, die fälschlicherweise als die

betrachtete Klasse klassifiziert wurden. Die Precision steigt also, wenn weniger Datenpunkte falsch als die betrachtete Klasse dargestellt werden [2, S.87]. Ein typischer Anwendungsfall hierfür ist ein Spamfilter, da möglichst wenig Nachrichten fälschlicherweise als Spam markiert werden sollten.

$$P = \frac{TP}{TP + FP} \quad (4)$$

Zuletzt sei mit dem F1-Score (5) ein beliebtes Qualitätsmaß genannt, welches Recall und Precision in ihrem harmonischen Mittelwert vereint. Es wird genutzt, wenn sowohl Recall als auch Precision relevante Tatsachen darstellen.

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \quad (5)$$

Da nun alle typischen Qualitätsmaße eingeführt wurden, können im nächsten Kapitel typische Richtlinien zum Vergleichen der Performanz von Modellen beschrieben werden.

4.1.5. Testen und Validieren

Eine gute Performanz während des Trainings eines Modells bedeutet noch keine gute tatsächliche Leistung. Overfitting oder andere Effekte können eine Qualität vortäuschen, welche im echten Einsatz aufgrund einer Spezialisierung des Modells auf die Trainingsdaten nicht erreicht wird. Es haben sich hier zwei typische Verfahren etabliert, um damit umzugehen:

Zum einen kann ein Trainingsdatensatz in drei Teile, einen tatsächlichen Trainingsdatensatz, einen Validierungsdatensatz und einen Testdatensatz unterteilt werden. Eine typische Unterteilung hierfür ist, 80% für das Training und jeweils 10% für Validierung und Test zu reservieren. Das Modell wird dann wie bisher mit dem Trainingsdatensatz angelernt und im Folgenden mit dem Validierungsdatensatz geprüft. Dann wird das Modell optimiert und der Vorgang wiederholt, bis die gewünschte Qualität erreicht wurde. Ist dies abgeschlossen, ist eine letzte Überprüfung nötig. Da während des Optimierungsvorgangs eine Spezialisierung auf Trainings- und Validierungsdaten stattgefunden haben könnte, muss mit dem bisher unbe-

kannten Testdatensatz eine erreichte Verallgemeinerung des Wissens im Modell überprüft werden. Wurde keine ausreichende Performanz festgestellt, muss der gesamte Trainingsvorgang wiederholt werden, bis ein Modell die gewünschte Qualität erreicht hat [2, S.30-31][4, S.43-44].

Eine weitere beliebte Möglichkeit zur Validierung von Modellen ist die Kreuzvalidierung. Ein Datensatz wird dabei in k-Teile (beispielsweise $k = 10$) unterteilt. Einer dieser Teile wird als Testdatensatz ausgewählt, während alle anderen Teile zum Training kombiniert werden. Dann wird das Training ausgeführt und die Leistung des Modells vermerkt. Nun wird dies für einen anderen der k-Teile als Testdatensatz und dem nicht trainierten Modell wiederholt, bis jeder Teil einmal getestet wurde. Aus den k-Leistungsdaten wird dann eine Gesamtleistung errechnet, welche eine gute Repräsentation für die Leistung des Modells ist. Zum genaueren Testen nach der Kreuzvalidierung ist auch eine vorherige Abspaltung weiterer Testdaten denkbar [3, S.22][4, S.44].

Datensatz					
1. Durchlauf	Validierung				Training
2. Durchlauf	Training	Validierung			Training
3. Durchlauf	Training		Validierung		Training
4. Durchlauf		Training		Validierung	Training
5. Durchlauf			Training		Validierung

Abbildung 5: Beispiel für eine 5-fache Kreuzvalidierung [4, S.44].

4.2. Neuronale Netze

In dieser Arbeit wurde sich, wie heutzutage üblich, bei der Bilderkennung auf Neuronale Netze und besonders die Konvolutionellen Neuronalen Netze fokussiert. Diese sind bei der Bilderkennung in vielen Bereichen State of the Art und werden von vielen bekannten vortrainierten Modellen wie dem VGG19 genutzt. Um diese zu verstehen, sollen allerdings erst die Grundlagen der Neuronalen Netze betrachtet werden.

4.2.1. Biologische Grundlagen

Neuronen sind für viele Menschen vor allem aus ihren eigenen Gehirn bekannt. In diesem arbeiten viele der kleinen Nervenzellen zusammen, um Informationen zu verarbeiten. Jedes Neuron nimmt elektrische Signale über seine stark verzweigten Dendriten auf und sammelt sie in seinem Zellkörper, dem Soma, an. Ist eine gewissen Stärke des elektrischen Signals vorhanden, so gibt ein Neuron dieses über sein Axon weiter. Axone sind lange, verzweigte Fäden, die mittels Synapsen mit den Dendriten anderer Neuronen verbunden sind. Synapsen arbeiten mit chemischen Transmittern und können eine erregende oder hemmende Wirkung auf das zu übertragene Signal haben. Biologische Neuronale Netze lernen nun, indem die Stärke der Verbindung über die Synapsen gesteuert wird. Außerdem können Verbindungen getrennt und neu eingegangen werden, was ebenfalls einen Lerneffekt erzeugt [4, S.24][3, S.43-45][8, S.29-30].

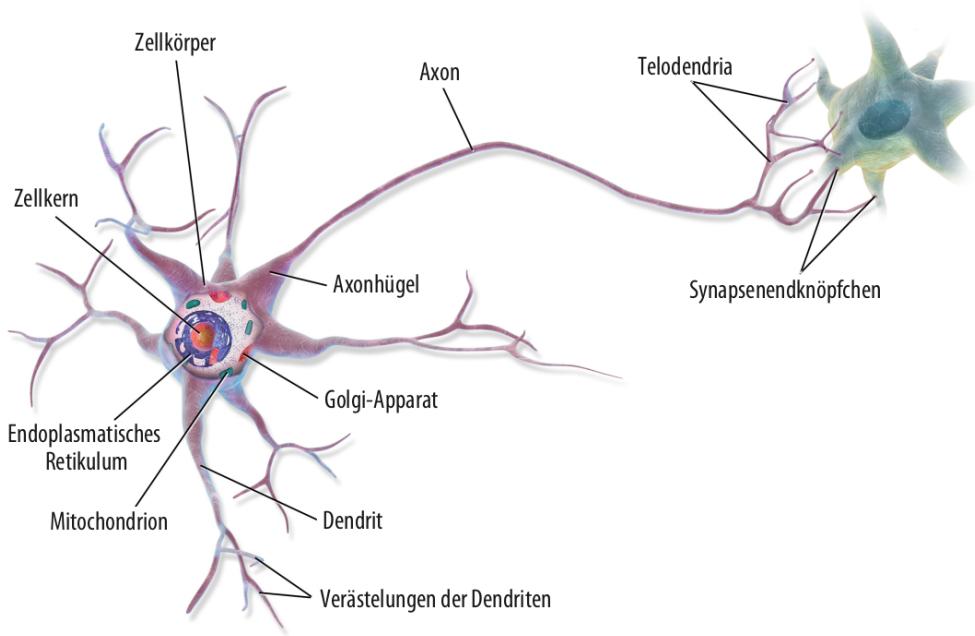


Abbildung 6: Aufbau eines biologischen Neurons [2, S.255].

4.2.2. Künstliche Neuronen

In künstlichen Neuronalen Netzen wird nun eine auf den biologischen Grundlagen basierende künstliche Variante von Neuronen genutzt. Diese nehmen über ihre

Eingänge, ähnlich zu den verbundenen Axonen, Signale auf, welche als ein Vektor \mathbf{x} mit n -Elementen dargestellt werden. Jedes dieser Eingangssignale wird nun über einen eigenen Wert in einem Gewichtungsvektor \mathbf{w} gehemmt oder verstärkt, was ähnlich zu der Arbeitsweise von Synapsen ist. Nun werden ebenfalls, ähnlich zum biologischen Vorbild, alle eingegangenen Signale aufaddiert, jedoch an eine Aktivierungsfunktion f_{akt} weitergegeben, welche ein Signal nochmal auf eine bestimmte Weise verarbeitet und sich komplett anders zum biologischen Vorbild verhalten kann. Zum Schluss kann das errechnete Signal des Künstlichen Neurons an weitere Neuronen, ähnlich zum ausgehenden Axon der biologischen Variante, weitergeleitet werden [4, S.25][8, S.30-31].

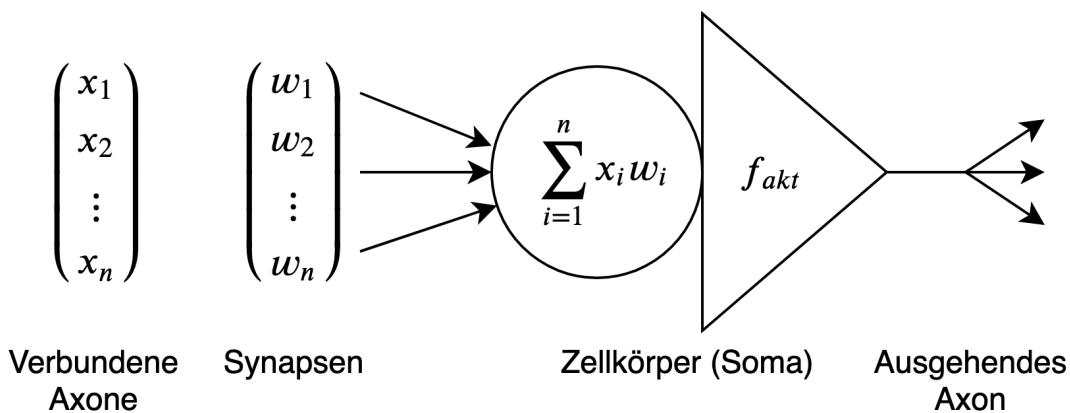


Abbildung 7: Aufbau eines künstlichen Neurons, angelehnt an das biologische Vorbild [4, S.25].

Für n Eingänge lässt sich ein Künstliches Neuron nun folgendermaßen definieren [4, S.25][8, S.30]:

$$y = f_{akt} \left(\sum_{i=1}^n x_i w_i \right) = f_{akt}(\mathbf{x}^T \cdot \mathbf{w}) \quad (6)$$

Wird eine Schwellenwertfunktion (7) als Aktivierungsfunktion genutzt, so wird ein Künstliches Neuron in der Regel als Perzepron bezeichnet [4, S.26][3, S.49].

$$f_{akt} = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (7)$$

4.2.3. Einschichtige Neuronale Netze

Liegen mehrere Künstliche Neuronen oder Perzeptrons in einer einzigen Schicht vor, entsteht ein einschichtiges Neuronales Netz. Dabei wird jeder Eingang in das Neuronale Netz mit jedem Neuron in der Schicht des Netzes verbunden und über Gewichtungen verändert. Die Eingänge werden dabei häufig in der Notation als Schicht von Eingabeneuronen dargestellt, welche lediglich die Eingabewerte an die Künstlichen Neuronen weitertragen. Weiterhin wird in die Eingabeschicht dieser Netze häufig ein spezielles Neuron eingeführt, welches lediglich eine Eins als Ausgabe liefert und über Gewichtungen verändert werden kann. Dies wird als Bias-Neuron bezeichnet. Die Ausgänge der Künstlichen Neuronen des Netzes können nun als Ausgabevektor \mathbf{y} ausgelesen werden [4, S.26][2, S.258]. Ein einschichtiges Neuronales Netz mit einer vereinfachten Darstellung der Neuronen ist in der Abbildung 8 dargestellt.

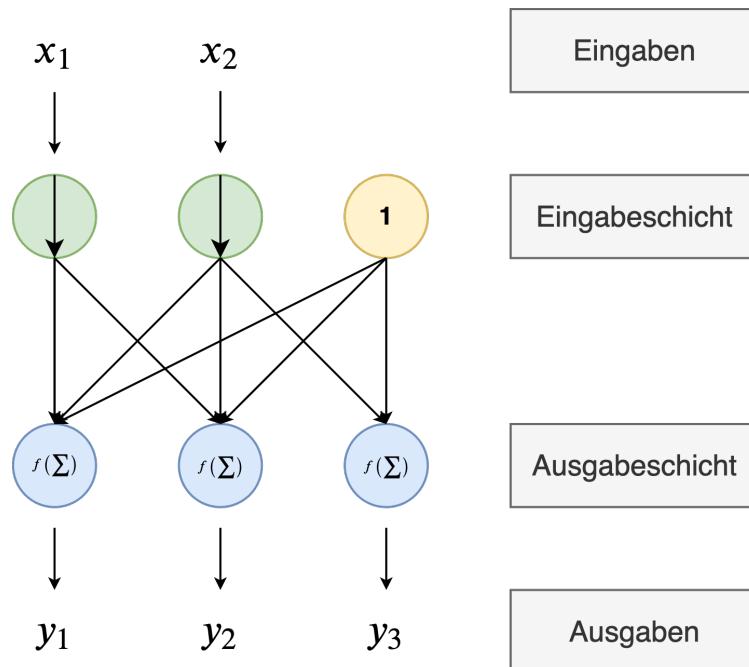


Abbildung 8: Aufbau eines einschichtigen Neuronalen Netzes [4, S.27][2, S.258].

Die Gewichtungen zwischen der Eingabeschicht und der Ausgabeschicht werden zur einfacheren Berechnung typischerweise als Matrix \mathbf{W} dargestellt (8) [4, S.26][2,

S.258].

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{b1} & w_{b2} & w_{b3} \end{pmatrix} \quad (8)$$

Mittels dieser Darstellung kann bei einer elementweisen Auswertung der Aktivierungsfunktion und einer Matrizenmultiplikation sehr einfach eine ähnliche Formel wie (6) angewandt werden, um die Ausgabe des Netzes zu errechnen:

$$\mathbf{y} = f_{akt}(\mathbf{x}^T \cdot \mathbf{W}) \quad (9)$$

Doch wie lernen einschichtige Neuronale Netze, sich an neue Informationen anzupassen? Früher und für nicht überwachtes Lernen wurde hier häufig die an die menschliche Biologie angelehnte Hebb'sche Lernregel angewandt. Diese ist jedoch für überwachtes Lernen wie im Fall der Problemstellung eher ungeeignet. Es wird also das in der Praxis häufig eingesetzte Gradientenabstiegsverfahren betrachtet [2, S.259].

4.2.4. Gradientenabstiegsverfahren

Die Idee hinter dem Gradientenabstiegsverfahren ist eine schrittweise Verringerung des Gesamtfehler eines einschichtigen Neuronalen Netzes F unter einer Fehlerfunktion $E(\mathbf{y}', \mathbf{y})$ für die Vorhersagen \mathbf{y}' und die Labels \mathbf{y} , indem eine optimale Kombination von Gewichten \mathbf{W}_{min} gesucht wird (10) [4, S.28].

$$\min_{\mathbf{W}} F = E(f_{akt}(\mathbf{x}^T \cdot \mathbf{W}), y) \quad (10)$$

In einer dreidimensionalen Gebirgslandschaft, welche durch zwei Gewichte und den Fehler beschrieben ist, kann man sich die Arbeitsweise dieses Algorithmus leicht so vorstellen: Ein Bergsteiger mit eingeschränkter Sicht in dieser Landschaft kann sich lediglich über den Punkt des steilsten Abstiegs langsam und schrittweise in ein Tal wagen. Er versucht also immer weiter dem steilsten Abstieg zu folgen, bis er ein Tal der Fehlerfunktion findet. Dies lässt sich in 9 erkennen [4, S.28][9, S.39].

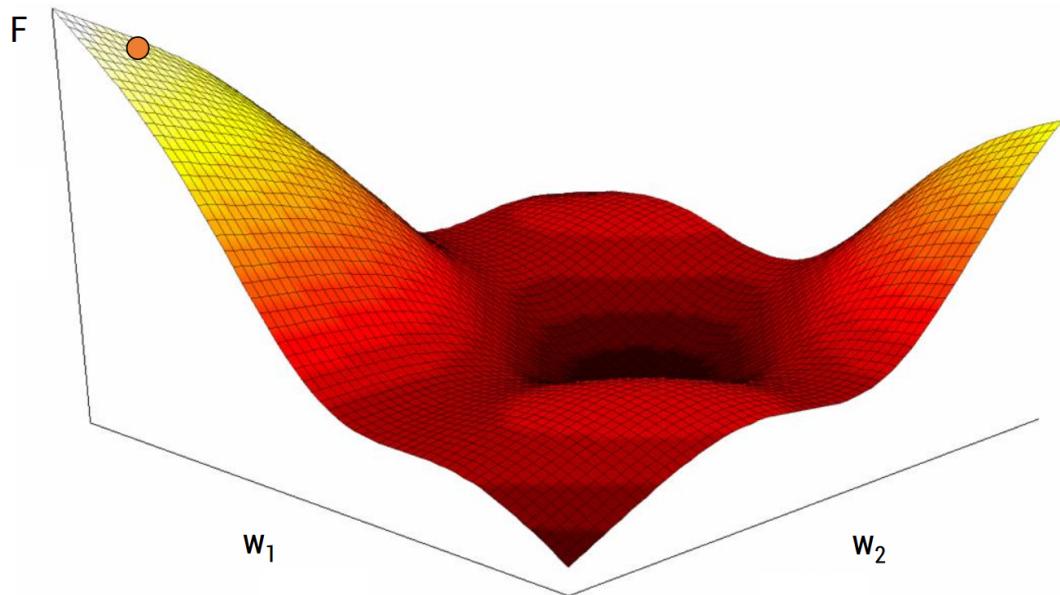


Abbildung 9: Gebirgslandschaft des Fehlers in einem Neuronalen Netz [10].

Im Algorithmus funktioniert dies durch partielle Ableitungen der Fehlerfunktion nach jedem Gewicht und einem Anwenden des daraus resultierenden Gradientenvektors in negativer Richtung angepasst durch die gewählte Lernrate, welche die Geschwindigkeit des Abstiegs zum Minimum definiert. Dies wird wiederholt, bis ein gewünschtes Minimum oder eine gewählte Zahl an Wiederholungen erreicht wurde. Die Wiederholungen werden bei Neuronalen Netzen über alle Trainingsdaten auch als Epochen bezeichnet [4, S.29-30][9, S.41][3, S.31]. Häufig wird jede Epoche aber auch noch in kleinere Teile, die sogenannten Batches, unterteilt.

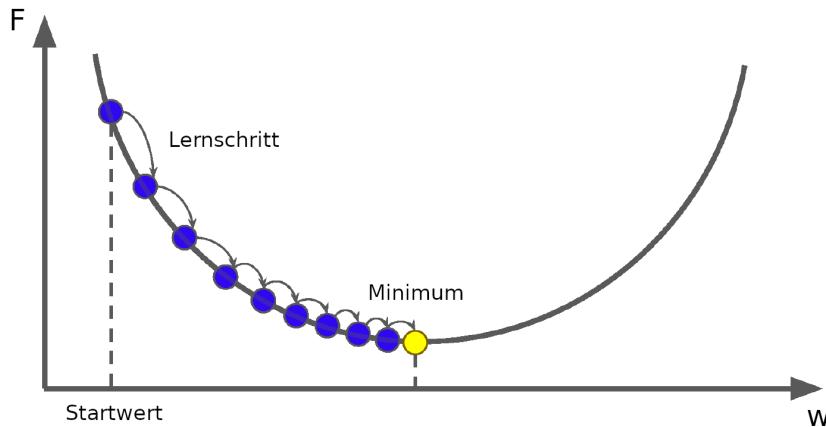


Abbildung 10: Das Gradientenabstiegsverfahren in zwei Dimensionen [2, S.113].

Bereits durch die Beschreibung des Algorithmus lassen sich einige Probleme abschätzen. Zum einen ist nicht gewährleistet, dass ein globales Minimum gefunden wird. Stoppt der Algorithmus in einem lokalen Minimum, so kann nicht weiter optimiert werden. Eine höhere Lernrate oder ein Momentum-Term, welcher dem Algorithmus beim Durchlaufen der mehrdimensionalen Gebirgslandschaft eine gewisse Trägheit gibt, können dafür sorgen, dass lokale Minima übersprungen werden. Eine hohe Lernrate kann aber auch zu Problemen führen. So ist in diesem Fall eine ein- oder mehrschrittige Oszillation zwischen den Talrändern ein typisches Problem, welches auftreten kann. Auch kann sie zum Überspringen des globalen Minimum führen [4, S.30][9, S.46-47].

Ein weiteres Problem tritt bezüglich der bisher bekannten Aktivierungsfunktion, der Schwellenwertfunktion, auf. Für diese kann kein Gradient ungleich null berechnet werden, wodurch kein Minimum gefunden werden kann. Dies kann durch das Einführen anderer Aktivierungsfunktionen gelöst werden und wird in einem späteren Kapitel betrachtet [4, S.31-32][2, S.262].

4.2.5. Mehrschichtige Neuronale Netze

Leider lassen sich mit einschichtigen Neuronalen Netzen nur einfache Probleme lösen. Dies lässt sich durch das Einfügen weiterer Schichten lösen, indem die Ausgaben der vorherigen Schicht durch weitere erlernbare Gewichtungen an die nächste Schicht weitergegeben werden. Alle Schichten zwischen der Eingabe- und Ausga-

beschicht werden dabei als verborgene Schichten oder Hidden-Layer bezeichnet [4, S.31-32][2, S.261]. Ein Beispiel hierfür ist in der folgenden Abbildung zu erkennen:

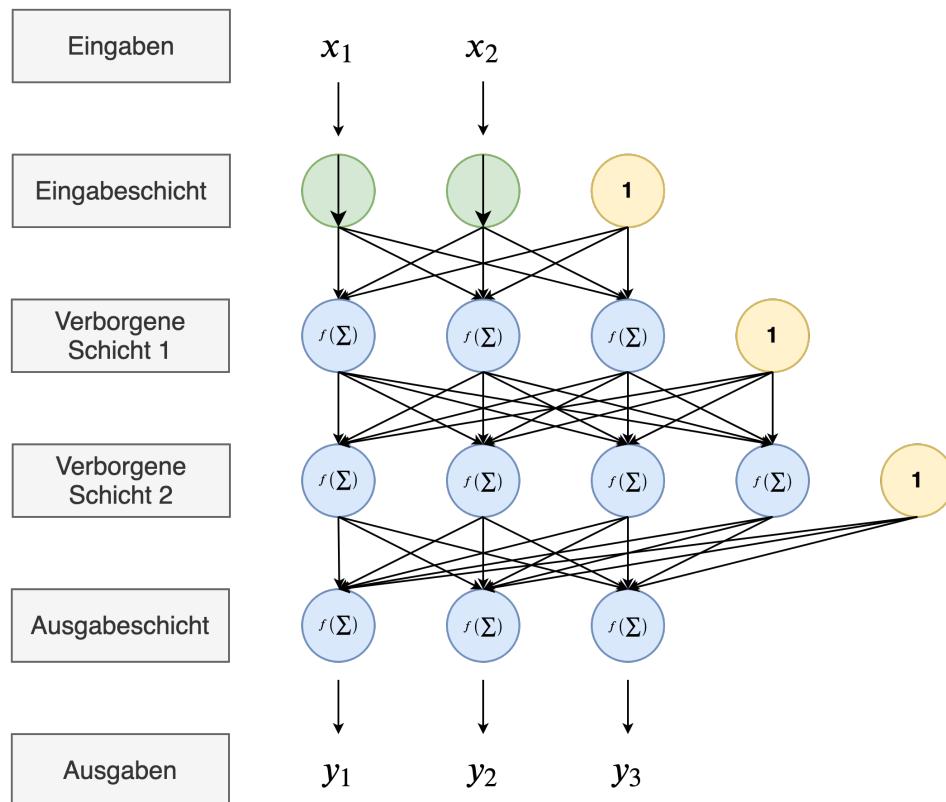


Abbildung 11: Ein mehrschichtiges Neuronales Netz mit zwei verborgenen Schichten [4, S.32][2, S.261].

Leider funktioniert das Gradientenabstiegsverfahren in der bisherigen Funktionsweise nur mit einem einzigen Layer, da der Fehler von verborgenen Schichten nicht direkt errechnet werden kann. Es fehlt damit eine Lernmethode für das überwachte Lernen. Die Lösung hierfür ist das Backpropagation-Verfahren [2, S.261].

4.2.6. Backpropagation-Verfahren

Über das Backpropagation-Verfahren wird der Fehler eines mehrschichtigen Neuronalen Netzes über seine Schichten zurückgerechnet, um eine Veränderungen der Gewichtungen in den verborgenen Schichten zu ermöglichen [9, S.51-52].

Das Backpropagation-Verfahren wird folgendermaßen ausgeführt: Nachdem die

Netzausgabe bestimmt wurde, wird für diese der Fehler zu den gewünschten Ausgaben errechnet. In der Ausgabeschicht wird mit dem Gradientenabstiegsverfahren für jedes Gewicht der negative Gradient zum letzten Hidden-Layer errechnet. Dieser wird nun durch die Gewichte der verbundenen Hidden-Neuronen zurückgerechnet und damit auf diese verbundenen Hidden-Neuronen verteilt. Wird dies für alle Neuronen der betrachteten Schicht wiederholt, kann mittels des Gradientenabstiegsverfahrens der Fehler der verbundenen verborgenen Schicht errechnet werden. Dies wird wiederholt, bis die Eingabeschicht erreicht wurde [4, S.33][9, S.52-53]. Wurden die Gradienten errechnet, können die Gewichtungen mit einem vorher definierten Lernparameter angepasst werden [2, S.262]. Der Algorithmus wird nun batch- und epochenweise solange wiederholt, bis der Fehler ein Limit unterschreitet oder eine maximale Anzahl an Wiederholungen erreicht wurde [9, S.52].

Typischerweise können bei dem Backpropagation-Verfahren ähnliche Probleme wie beim Gradientenabstiegsverfahren auftreten. Außerdem können Gradienten aufgrund des Zurückrechnens sehr klein ausfallen und zu einer immer kleiner werdenden Anpassung der Gewichte in den ersten Schichten führen. Dies würde zu einem immer kleiner werdenden Lerneffekt führen. In wenigen Fällen kann auch ein gegenteiliges Problem, nämlich explodierende Gradienten, auftreten [4, S.33-34][2, S.275-276]. In der Regel lässt sich dies über sinnvoll ausgewählte Aktivierungsfunktionen und spezielle Regularisierungsverfahren lösen, die in den nächsten Kapiteln betrachtet werden.

4.2.7. Aktivierungsfunktionen

Moderne Neuronale Netze verwenden in der Regel Aktivierungsfunktionen, welche auf der ReLU (Rectified Linear Units) basieren. Die ReLU hat im Gegensatz zu anderen, typischerweise früher eingesetzten Aktivierungsfunktionen wie der Sigmoidfunktion oder dem Tangens hyperbolicus, den Vorteil, dass sie keinen Sättigungseffekt durch eine sehr kleine Ableitung für positive Werte erfährt. Dies wirkt dem Problem der schwindenden Gradienten entgegen [4, S.34-35]. Für die Eingabe x ist die ReLU definiert als [2, S.245, S.279]:

$$f_{akt} = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (11)$$

Wie zu sehen ist, ist die ReLU für positive Werte lediglich äquivalent zu der originalen Eingabe. Damit ist sie sehr einfach zu berechnen. Leider leidet sie unter einem großen Problem: Ein Neuron mit einer negativen Eingabesumme vor der Aktivierungsfunktion liefert eine Konstante 0 und trägt nicht mehr zum Lernen bei. Tritt dies bei vielen Neuronen auf, kann dies ebenfalls den Lerneffekt beschränken [2, S.279].

In vielen Fällen wird dies über die Verwendung leicht abgewandelter ReLU-Varianten wie der Leaky-ReLU oder der ELU (Exponential Linear Unit), die auch in diesem Projekt verwendet wurde, gelöst. Die ELU hat im Positiven ein lineares Verhalten, während sie sich im negativen Bereich wie eine Exponentialfunktion im Negativen verhält, die sich einem Parameter a annähert. Es gilt [4, S.36]:

$$f_{akt} = \begin{cases} x & x \geq 0 \\ a(e^x - 1) & x < 0 \end{cases} \quad (12)$$

Mit diesen Aktivierungsfunktionen können bereits einige typische Probleme beim Maschinellen Lernen mit mehrschichtigen Neuronalen Netzen und Backpropagation verhindert werden. Eine weitere verwendete Aktivierungsfunktion, welche vor allem bei der Ausgabeschicht von Klassifikationsproblemen mit One-Hot kodierten Labeln nützlich ist, ist die Softmax-Funktion. Im Gegensatz zu anderen Aktivierungsfunktionen erhält sie die Ausgabe aller Neuronen einer Schicht. Diese Ausgaben werden dann mit der Summe aller Ausgaben normalisiert, was in einem Vektor von Zahlen zwischen 0 und 1 resultiert. Diese werden in der Regel als Wahrscheinlichkeiten für eine vorliegende Klasse interpretiert. Für den Vektor mit Neuronensummen \mathbf{s} mit n -Elementen ist die Softmax definiert als [4, S.37][2, S.141-142, S.263]:

$$f_{akt} = \frac{e^{s_i}}{\sum_{i=1}^n e^{s_i}} \quad (13)$$

4.2.8. Regularisierungsverfahren

Zuletzt werden nun zwei typische Regularisierungsverfahren für Neuronale Netze betrachtet, mit denen deren Performanz auf bestimmte Weisen erhöht werden

kann.

Die Batch-Normalisierung ist ein ideales Verfahren, um schwindenden Gradienten sowie Overfitting entgegenzuwirken und die Stabilität von Neuronalen Netzen zu erhöhen. Diese wird schicht- und batchweise ausgeführt, indem die Summe aller Neuronen einer Schicht vor der Aktivierungsfunktion standardisiert wird. Das Verfahren arbeitet, indem der Mittelwert auf 0 zentriert und durch die Standardabweichung der Batch geteilt wird. Dann wird das Ergebnis über zwei lernbare Parameter skaliert und verschoben. Dadurch kann das Netz die ideale Skalierung und den Mittelwert der Eingaben für eine Schicht erlernen. Schwindende Gradienten werden dadurch so stark bekämpft, dass sogar der Einsatz der Sigmoidfunktion oder des Tangens hyperbolicus wieder denkbar wäre [4, S.37-38][2, S.282-283].

Eine weitere Möglichkeit zur Regularisierung ist ein Dropout in verborgenen Schichten. Das Prinzip vom Dropout ist, dass Neuronen in einer Schicht während eines Schritts des Trainingsvorgangs mit einer gewissen Wahrscheinlichkeit ausgelassen werden. Wird der nächste Schritt erreicht, so werden die Dropouts neu ausgewürfelt. Durch das Auslassen von gewissen Neuronen werden andere dazu gezwungen, ihr Wissen zu verallgemeinern, da sie sich nicht auf die ausgeschiedenen Neuronen verlassen können. Dies wirkt Overfitting entgegen [4, S.38-39][8, S.205-206].

4.3. Konvolutionelle Neuronale Netze

Heutzutage werden für Aufgaben der Bilderkennung in der Regel keine einfachen Neuronalen Netze, sondern Konvolutionelle Neuronale Netze eingesetzt. Diese wurden zum ersten mal 1980 genutzt worden und sind dem Aufbau des visuellen Cortex des Menschen nachempfunden. In diesem arbeiten Neuronen in verschiedenen Schichten von Filtern, die aufeinander aufbauen. Die oberen Filter erkennen viele einfache Bildelemente, wie beispielsweise unterschiedliche Ausrichtungen von Linien. Diese werden weiter an untere Schichten geleitet, die das Gefilterte immer weiter zu komplexen Mustern und Bildern kombinieren können [2, S.360].

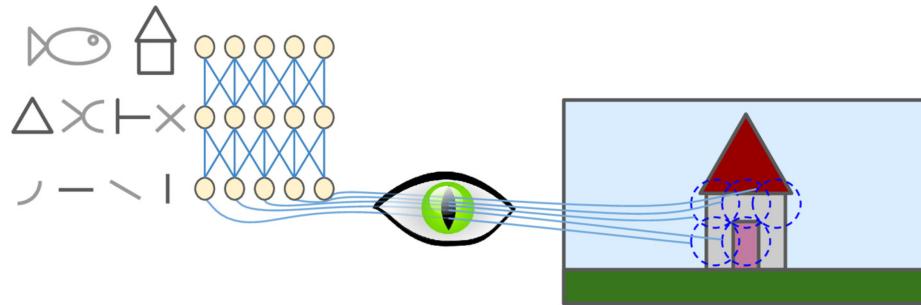


Abbildung 12: Aufbau des visuellen Cortex eines Menschen [2, S.360].

Diese Erkenntnisse wurden dann genutzt, um die ersten Konvolutionellen Neuronalen Netze zu entwickeln und für die Erkennung von handgeschriebenen Ziffern zu nutzen. Ein typischer Aufbau für Konvolutionelle Neuronale Netze besteht aus drei verschiedenen Schichten. Die letzten Schichten sind dabei in der Regel die bereits betrachteten, vollständig verbundenen Neuronalen Netze. Am Anfang befinden sich allerdings abwechselnd Convolutional und Pooling Layer, welche im Folgenden betrachtet werden [2, S.361].

4.3.1. Convolutional Layer

Convolutional Layer bestehen vom Grundprinzip aus einer zweidimensionalen Gewichtsmatrix von Neuronen einer gewissen Höhe und Breite, auch Filter genannt, welche mit einer bestimmten Schrittweite (Stride) über ein Eingabebild- oder ein bereits vorher durch ein von Convolutional Layer betrachtetes Bild wandern. Der Filter multipliziert dabei in jedem durch die Schrittweite betrachteten Ausschnitt des Bildes die Bildpunkte mit eigenen, im Filter vorhandenen und lernbaren Gewichtungen. Dann werden die gewichteten Bildpunkte aufsummiert. Es entsteht ein neuer Bildpunkt, welcher, wenn ein zum Filter passendes Feature betrachtet wurde, einen hohen Wert aufweisen wird. Damit Randpixel betrachtet werden können, muss abhängig von der Filtergröße ein Padding am Rand des Bildes hinzugefügt werden. Hier sind unterschiedliche Arten, wie ein Zero-Padding, also ein Auffüllen des Randes mit Nullen, denkbar. In der Abbildung 13 ist ein Convolutional Layer mit einer Filtergröße von 3x3 Pixeln, einer Schrittweite von 2 und ein Zero-Padding erkennbar. Bei dieser Schrittweite ist klar zu sehen, dass das Bild durch diese verkleinert wird. Dies ist nützlich, da hierdurch Bildauflösungen verringert werden können [2, S.361-363].

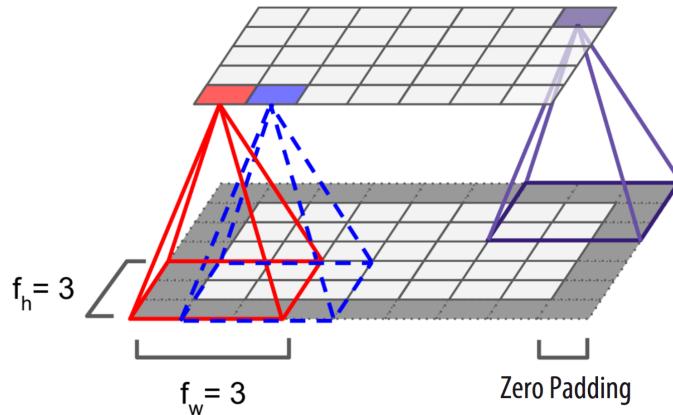


Abbildung 13: Ein einzelner Filter eines Convolutional Layers [2, S.362].

Die Arbeitsweise eines Filters ist sehr einfach zu visualisieren. Ein Filter, der in einer einzelnen Linie mit vertikaler Ausrichtung hohe Gewichtungen aufweist, würde in einem Bild vertikale Linie hervorheben und horizontale Linie verschwimmen lassen. Demgegenüber würde ein horizontaler Linienfilter horizontale Linien hervorheben und vertikale verschwimmen lassen [2, S.363-364]. Ein Beispiel hierfür ist in folgender Abbildung zu erkennen:

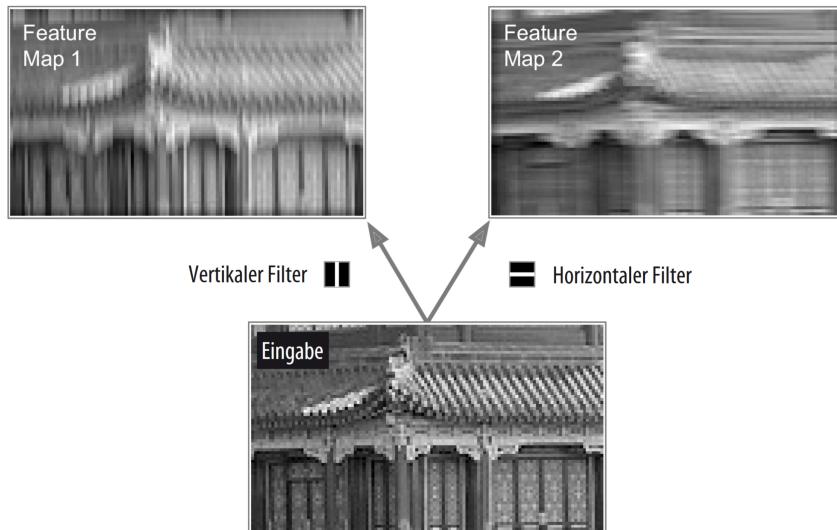


Abbildung 14: Ein Beispiel für horizontale und vertikale Filter [2, S.364].

In der Abbildung 15 ist außerdem ein typisches Beispiel für visualisierte Filter des ersten Convolutional Layers in einem CNN zu erkennen. Es wird klar ersichtlich, dass die dargestellten Filter verschiedene Strukturen erkennen können [11, S.132].



Abbildung 15: Ein Beispiel für verschiedene visualisierte Filter im ersten Convolutional Layers eines CNNs [11, S.132].

Wie in der Abbildung zu sehen, gibt es pro Convolutional Layer gleich mehrere Filter, die sogar Farben verarbeiten können. Zwar lassen sich mit mehreren bisher beschriebenen Filtern pro Convolutional Layer auch schon verschiedene Features von Grauwertbildern (ein Grauwert pro Bildpunkt) sehr gut extrahieren, jedoch gehen auf diese Weise viele Bildinformationen der Farbe verloren, deshalb sind tatsächliche Filter in modernen CNNs eher dreidimensionale Matrizen. Deren Höhe und Breite ist weiterhin dieselbe definierte Größe wie bei bisher bekannten Filtern, jedoch weisen sie in der Tiefe weitere Reihen von Filtern auf. Diese können Informationen aus den Farbkanälen extrahieren. Weiterhin können pro Convolutional Layer, wie in der Abbildung zu sehen war, mehrere Filter eingesetzt werden. Daraus resultiert ein Stapel sogenannter Feature Maps, auf denen erneut ein dreidimensionaler Filter angewendet werden kann [2, S.364-365]. Dies ist in folgender Abbildung gut zu erkennen:

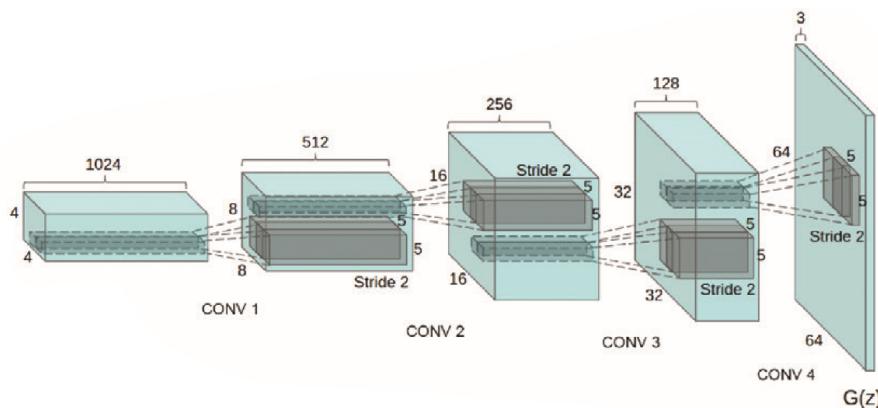


Abbildung 16: Ein Beispiel eines CNNs mit steigenden Feature Maps pro Layer [11, S.138].

Die Anzahl der Feature Maps steigt in der Regel mit der Tiefe an, jedoch sollte sich ihre Größe verringern. Dies geschieht entweder über die Schrittweite oder weitere Methoden wie Pooling Layer.

4.3.2. Pooling Layer

Pooling Layer werden genutzt, um die Größe von Convolutional Layern zu verringern, indem sie zwischen diesen platziert werden und jede Feature Map oder jeden Kanal auf eine gewisse Weise verarbeiten. Die Art der Verarbeitung ist dabei abhängig vom Typ des Pooling Layers, jedoch sind Average- oder Max Pooling die typischen Varianten. Beim Max Pooling wird aus einem Filter einer gewissen Höhe und Breite der größte Wert in diesem Bereich der Feature Map entnommen. Dadurch kann dessen Größe stark verringert werden. Auch können Pooling Layer wieder eine gewisse Schrittweite haben, mit denen die Verringerung der Größe einer Feature Map erhöht werden kann [12, S.18-19][2, S.369-370]. In der folgenden Abbildung wird die Arbeitsweise eines 2×2 Max Pooling mit einem Stride von 2 dargestellt:

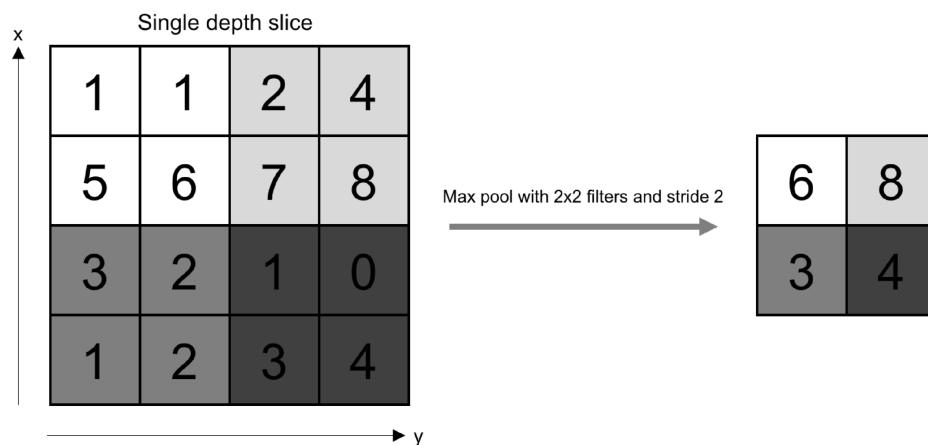


Abbildung 17: Die Arbeitsweise des Max Pooling mit einem 2×2 Kernel und Stride von 2 [12, S.19].

Mit der Definition der Pooling Layer sind nun die wichtigsten Grundlagen für das Maschinelle Lernen mit Bildern gegeben. Es kann also nun mit dem praktischen Aspekten begonnen werden.

5. Datensätze

Die verwendeten Datensätze stammen von der Online-Community Kaggle. Kaggle veranstaltet Data-Science Wettbewerbe und erlaubt es, Datensätze zu finden und bereitzustellen. Die Erkennung von Fingeralphabeten ist ein übliches Problem der Bilderkennung, sodass diverse Datensätze zur Auswahl stehen. Im Projekt wurden zwei Datensätze genutzt, die nachfolgend näher beschrieben werden.

5.1. Sign Language MNIST Dataset

Der „Sign Language MNIST“ Datensatz wurde im Kapitel 3. in Bezug auf Support Vektor Maschinen bereits kurz angesprochen. Der Datensatz war bereits aufbereitet und die Merkmale der Bilder wurden in Form von Graustufen (0-255) in eine CSV Datei geschrieben. Die CSV-Datei enthielt mit einer Klasse und 28x28 Pixeln pro Datensatz 785 Spalten. Die Buchstaben Z und J wurden nicht als Klassen hinterlegt, da sie eine gewisse Gestik benötigen, die in einem Standbild nicht einzufangen ist. Durch Extrahieren der Merkmale konnten die Bilder zurückgewonnen und manuell analysiert werden. Aufgefallen ist, dass die Bilder stets mit einem ruhigen und einfarbigen Hintergrund aufgenommen wurden. Insgesamt lagen in einem Verhältnis von 80 zu 20 27455 Trainingsdaten und 7172 Testdaten vor [16]. Die geringe Anzahl an Merkmalen erlaubte ein schnelles Training, allerdings mussten für eine reale Anwendung die Bilder herunterskaliert werden. Trotz hoher Accuracy auf dem Testdatensatz konnten Bilder in einer praktischen Anwendung mit einer Webcam nicht zuverlässig erkannt werden. Verursacht wurde dieses Problem vermutlich durch die geringe Informationsdichte der Bilder. Daraufhin wurde der Datensatz gewechselt.

5.2. ASL Alphabet Dataset

Der zweite Datensatz mit den Namen „ASL Alphabet“ hat 87000 Bilder mit Farbinformationen und 200x200 Pixeln [17]. Die Input-Größe, definiert als Breite x Höhe x Farbinformation, ist für ein Konvolutionelles Neuronales Netz dank der Filter und des möglichen Trainierens auf einem Grafikprozessor kein Problem. Das ConvNet ist in der Lage, alle wichtigen Informationen selbst zu extrahieren und die Unwichtigen herauszufiltern. In dem Datensatz sind zusätzlich die Klassen Space, Delete

und Nothing enthalten, wobei nur Letztere im Frontend berücksichtigt wurde. Die Handzeichen sind durchweg vor dem gleichen Hintergrund aufgenommen, variieren aber in Lichtverhältnissen, Bildschärfe und Position der Hand. Außerdem ist der Hintergrund nicht einfarbig, sondern die Bilder scheinen vor einem Dachfenster aufgenommen worden zu sein. Anders als der „Sign Language MNIST Datensatz“ sind die Bildinformationen nicht bereits in eine CSV Datei extrahiert worden, sondern liegen tatsächlich als Bilder vor. Die einzelnen Handzeichen sind in Ordnern mit dem Namen des jeweiligen Buchstabens organisiert. Ein Testdatensatz ist bereits vorbereitet, enthält aber lediglich ein Bild jeder Klasse, sodass dieser noch erweitert werden muss. Das genaue Vorgehen wird unter [6.1.2.](#) beschrieben.

5.3. Videodatensätze

Videodatensätze erlauben es, Gestik und Mimik zu erkennen, was für komplexe Gebärdensprache unerlässlich ist. Dazu müsste der Stream der Webcam in Echtzeit an das Backend übertragen werden. Da Videos lediglich eine schnelle Folge von Bildern sind, könnte eine naive Idee vorsehen, die Frames einzeln durch das Netz zu schicken. Bei der Vorhersage wird die Anzahl der letzten Frames K betrachtet und ein Durchschnitt der Predictions gebildet. Das Label mit der höchsten Wahrscheinlichkeit wird als Gesamtergebnis verwendet. Da das Projekt sich auf Standbilder des Fingeralphabets bezieht, wurde diese Möglichkeit allerdings final nicht weiter untersucht.

6. Entwicklungsprozess

Da nun die genutzten Datensätze beschrieben wurden, kann mit der Dokumentation des Entwicklungsprozesses fortgefahren werden. Hier wird sich, wie bei Maschinellen Lernen üblich, zunächst auf die Datenvorverarbeitung konzentriert.

6.1. Datenvorverarbeitung

Da die beiden hauptsächlich verwendeten Datensätze in einer für ihre Auflösung ausreichenden Qualität vorhanden sind, müssen nur noch wenige Schritte in der

Datenvorverarbeitung beachtet werden. Dies bezieht sich vor allem auf das Einlesen der Datensätze und das Skalieren der Bilder. Hier liefern Keras und Tensorflow in den neuesten tf-nightly Versionen (getestet mit 2.3.0a20200613) mit `image_dataset_from_directory` eine sehr mächtige Funktion, welche lediglich auf Basis von Ordnerstrukturen Bilddatensätze mit passenden Labels erstellt und auch das Skalieren von Bildern übernehmen kann [13]. Dabei wird beim Aufruf ein Tensorflow Dataset zurückgegeben, das bereits Pipelining einer gewissen Batchgröße von der Festplatte unterstützt. Dies ist bei sehr großen Datensätzen nützlich. Nun müssen die Datensätze lediglich in passende Ordnerstrukturen umgewandelt werden, bei denen alle Bilder einer Klasse einen Ordner mit zugehöriger Bezeichnung nutzen. Außerdem ist eine Teilung in Test, sowie Trainings- und Validierungsdaten nötig, denn `image_dataset_from_directory` unterstützt lediglich eine Zweiteilung.

6.1.1. Sign Language MNIST Dataset

Dieser Datensatz besteht im Kern aus zwei CSV Dateien für Training und Test. Die Zeilen repräsentieren dabei die 28x28 Pixel eines Grauwertbildes sowie ein Label in der ersten Spalte. Die Label sind Zahlen von 0-25 und repräsentieren alle Buchstaben des Alphabets außer Z (25) und J (9), welche ausgelassen wurden. Trotz des Aufbaus in CSV Dateien wurde sich dazu entschieden, die Werte zu extrahieren und als einzelne PNGs in den erforderlichen Ordnerstrukturen abzuspeichern. Der Kern der Skriptdatei zur Vorverarbeitung des Sign Language MNIST Dataset ist dabei folgende Funktion:

```

36 def to_image_at_dir(target_dir, row_data, image_name):
37     """
38     Wandelt die Zeilen einer CSV Datei im Datensatz in ein Bild um.
39     :param target_dir: Zielordner
40     :param row_data: Daten der Zeile.
41     :param image_name: Name des Bildes.
42     :return: None.
43     """
44     # Label entnehmen.
45     label = row_data[0]
46     # Daten entnehmen in Integer umwandeln.
47     list_data = list(map(int, row_data[1::]))
48
49     # Image Data in Numpy 28x28 Array umwandeln.

```

```

50     image_data = numpy.reshape(numpy.array(
51         list_data, dtype=numpy.uint8), (28, 28))
52     label_folder = target_dir + str(label) + "/"
53
54     # Ordner erstellen, wenn nicht vorhanden.
55     os.makedirs(label_folder, exist_ok=True)
56     # Bild erstellen.
57     image = Image.fromarray(image_data)
58     # Bild als PNG speichern.
59     image.convert("RGB").save(label_folder + image_name + ".png")

```

Listing 1: Kern der Sign Language MNIST Dataset Vorverarbeitung.

Diese Funktion ist sehr einfach und nimmt lediglich einen Zielordner, die Daten einer Zeile des Datensatzes sowie ein Name des Bildes (zum Beispiel eine Bildnummer) an. Dann werden in Zeile 45 und 47 Label und Daten getrennt. Eine Umwandlung der Daten in Integer wird ebenfalls über eine Map-Funktion vorgenommen. Es folgt in Zeile 50 bis 52 eine Umwandlung in ein 28x28 Numpy-Array mit Datenpunkten als 8-Bit Unsigned Integer (uint8). Zuletzt werden die Daten des Numpy-Arrays lediglich mit der bekannten Bildverarbeitungsbibliothek für Python, „Pillow“, in ein Bild umgewandelt und anschließend als PNG in RGB abgespeichert. Es wurde sich hier für PNGs statt Grauwertbilder entschieden, da so ein problemloses Verwenden von `image_dataset_from_directory` möglich wird.

6.1.2. ASL Alphabet Dataset

Der ASL Alphabet Dataset ist vom Aufbau deutlich einfacher zu verarbeiten, da hier die Bilder bereits in passende Subordner des Trainingsdatensatzes sortiert sind. Die Testdaten benötigen allerdings Vorverarbeitung, da sie pro Zeichen der ASL nur ein Bild beinhalten. Dies ist nicht hinreichend für Testdaten. In der Vorverarbeitung wird also eine bestimmte Anzahl zufälliger Bilder pro ASL Zeichen aus den Trainingsdaten ausgewählt und in das passend erstellte Verzeichnis für die Testdaten verschoben.

Dies funktioniert im Kern über folgende Funktion, welche jeden Ordner für ein ASL Zeichen durchläuft und mittels der Bibliothek `glob` alle Dateien ausliest. Dann werden mit `random.sample` eine vorher definierte zufällige Anzahl von Samples (in der Praxis 300) ausgewählt und in das Zielverzeichnis des Testordners für die-

ses ASL Zeichen verschoben.

```

44 # Bewege die in rand_num_to_move definierte Anzahl
45 # zufälliger Elemente pro Klasse in den Testordner.
46 for target_dir_name in glob.glob(target_directory + "/" +
47     train_directory_name + "/*"):
48
49     target_dir_classes = glob.glob(target_dir_name + "/*")
50     # Wähle für diese Klasse rand_num_to_move zufällige Elemente aus.
51     random_elements = random.sample(target_dir_classes,
52         rand_num_to_move)
53     # Kopiere die Elemente um.
54     for move_element in random_elements:
55         new_dir = move_element.replace(train_directory_name,
56             test_directory_name)
57         os.makedirs(os.path.dirname(new_dir), exist_ok=True)
58         shutil.move(move_element, new_dir)

```

Listing 2: Kern der Sign Language MNIST Dataset Vorverarbeitung.

6.1.3. DatasetParser Pakete

Das Ziel der DatasetParser Python Pakete ist es, für alle vorverarbeiteten Datensätze eine vereinigte Schnittstelle zum Bereitstellen von Datensätzen zu liefern. Diese bestehen aus einer DatasetParser Datei mit den Funktionen `get_train_and_val` und `get_test`, welche Trainings-, Test- und Validierungsdaten bereitstellen, sowie den Konstanten `class_names` und `image_size`, die Klassennamen (im Bezug auf die Ordner der vorverarbeiteten Datensätze) und eine standardmäßige Bildgröße definieren. Eine weitere in den Paketen nötige Skriptdatei ist `DatasetPrepareOrig`. Diese führt mittels einer separaten Ausführung eine in den vorherigen Kapiteln beschriebene Vorverarbeitung aus. Der Datensatz selbst ist nun in `DatasetOrig` als nicht verarbeiteter Datensatz und in `DatasetPrepared` als durch das Skript vorverarbeiteter Datensatz definiert.

Die Arbeitsweise der DatasetParser Dateien ist nun sehr einfach zu beschreiben. Diese führen beim Aufruf von `get_train_and_val` zweimal die Tensorflow-Funktion `image_dataset_from_directory` mit vom Datensatz abhängigen Parametern auf und geben die Ergebnisse dieser zurück. Wird unter der Angabe des Parameters `validation_split` diese Funktion auf ein Verzeichnis zweimal ausgeführt,

so gibt Tensorflow beim ersten Aufruf das Dataset der Trainings- und beim zweiten das Dataset der Validierungsdaten zurück. Die Testdaten der Funktion `get_test` werden über einen einzelnen Aufruf ohne `validation_split` zurückgegeben.

```
1 return image_dataset_from_directory(  
2     directory="datasets/kaggle_1/DatasetPrepared/train",  
3     label_mode='categorical',  
4     color_mode='grayscale',  
5     image_size=image_size_param,  
6     batch_size=batch_size,  
7     shuffle=shuffle,  
8     seed=seed,  
9     validation_split=validation_split,  
10    subset="training",  
11    interpolation='bilinear',  
12    follow_links=False  
13 )
```

Listing 3: Beispielhafter Aufruf der Funktion zum Einlesen von Daten

Im Listing 3 wird ein Aufruf der Funktion `image_dataset_from_directory` beschrieben. Die wichtigsten Parameter haben dabei folgende Aufgaben:

- `directory`: Das Verzeichnis, aus dem der Datensatz gelesen wird.
- `label_mode`: Bei `categorical` werden die Subordner als Kategorien gesehen und als One-Hot kodierten Vektor beschrieben.
- `color_mode`: Umgewandelte Farbkanäle der Bilder.
- `image_size`: Skalierte Größe der Bilder.
- `batch_size`: Größe der Batches des Datasets.
- `shuffle`: Wenn wahr, werden die Bilder nach dem Einlesen zufällig angeordnet.
- `validation_split`: Anteil der Validierungsdaten.
- `subset`: Das zurückgegebene Subset für dieses Verzeichnis. Es ist entweder `training` oder `validation`.
- `interpolation`: Genutztes Verfahren für die Skalierung.

Da die Datensätze für ihre Auflösung alle ein sehr rauschfreies Bild aufweisen und in der Regel schon recht gut auf die Hände fokussiert sind, ist damit nun die Datenvorverarbeitung abgeschlossen und die Bilder der Datensätze können sehr einfach in passende Tensorflow-Datasets gelesen werden. Nun wird die Entwicklung und Auswertung von Modellen beschrieben.

6.2. Entwickelte Modelle

In diesem Kapitel wird der Grundaufbau der Python-Dateien zum Maschinellen Lernen beschrieben und die Ergebnisse einiger Modelle vorgestellt. Dabei werden für Neuronale Netze, Konvolutionelle Neuronale Netze und Transferlearning einige der besten Versuche dokumentiert.

6.2.1. Grundaufbau des Modelltrainings

Alle Dateien für das Training von Modellen haben grundsätzlich einen ähnlichen Aufbau. Zunächst müssen die DatasetParser der gewünschten Datensätze inkludiert werden, mit denen ein Großteil der Datenvorverarbeitung schon erledigt werden kann. Danach folgen einige Definitionen für Batchgröße und Bildgröße, mit welchen nun die Funktionen `get_train_and_val` und `get_test` aufgerufen werden können. Dank sinnvoller Default-Parameter ist dieser Aufruf in der Regel recht kurz. Es wurde ein Validation Split im Verhältnis von 80 zu 20 unter einer in Kapitel [4.1.5.](#) beschriebenen Validierung in drei Stufen verwendet. Am Ende der Vorbereitung wird noch die Anzahl der Klassen für die Ausgabe und die Größe der Eingabe für den ersten Layer des Modells definiert. Ein Beispiel für die Vorbereitung des Trainings ist im folgenden Listing zu erkennen:

```
1 from sign_language_image.datasets.kaggle_1.DatasetParser
2     import get_train_and_val, get_test, class_names
3 ...
4 # Batchgröße
5 batch_size = 100
6 # Bildgröße
7 image_size = (28, 28)
8 train_ds, val_ds = get_train_and_val(batch_size,
9     image_size_param=image_size)
10 test_ds = get_test(image_size_param=image_size)
11
```

```

12 # Anzahl der Klassen aus den Namen der Klassen extrahieren
13 num_of_classes = len(class_names)
14 # Input Shape als Bildgröße + Farbkanäle
15 input_shape = (image_size[0], image_size[1], 1)

```

Listing 4: Beispielhafte Vorbereitung für das Training

Danach kann dann schon ein Modell definiert werden. Diese können sich je nach Aufbau etwas unterscheiden, daher wird auf sie und ihre Ergebnisse in den nächsten Kapiteln eingegangen.

Nach der Definition eines Modells kann es kompiliert und mit einem Optimizer und Qualitätsmaß versehen werden. Hier wurde aufgrund des Aufbaus der Daten und des Lernziels die Kreuzentropie genutzt. Außerdem wird ein Modell während der Optimierung über die Vorhersagegenauigkeit auf Trainings- und Validierungsdaten ausgewertet. Dies ist aufgrund der gleichmäßigen Verteilung und Priorität der Klassen in den Datensätzen möglich. Als Optimierer wird eine Erweiterung des stochastischen Gradientenverfahrens, der Adam-Optimierer, genutzt.

```

1 # Modell kompilieren mit Kreuzentropie und Adam Optimizer.
2 model.compile(loss='categorical_crossentropy',
3                 optimizer=tf.keras.optimizers.Adam(lr=0.00005),
4                 metrics=['accuracy'])

```

Listing 5: Kompilierung eines Modells

Dann wurde eine Funktion für das Loggen entwickelt. Diese kopiert das momentan aktive Skript in einen eigenen Ordner des übergebenen Logverzeichnisses. Dieses dient als Referenz, wenn die originale Trainingsdatei verändert wurde. Dann wird, abhängig von einem Parameter, die Ausgabe der Konsole in eine Logdatei in das vorher erstellten Verzeichnis umgeleitet, um später den Trainingsvorgang nachvollziehen zu können. Zuletzt werden zwei Tensorflow-Callbacks für das Verzeichnis erstellt, welche TensorBoard-Logs erstellen und die Modelle mit dem geringsten Fehler abspeichern.

```

1 def log_run(log_dir, log_to_file=True):
2     script_path = sys.argv[0]
3     script_name = os.path.basename(script_path)
4     log_dir_script = log_dir + script_name + "_" + datetime.now().
5         strftime("%d-%m-%Y_%H%M%S") + "/"
6     # Erstelle das Log Verzeichnis

```

```

6     os.makedirs(log_dir_script)
7     # Kopiere das Script, welches ausgeführt wurde in das Log
8     Verzeichnis
9     shutil.copy(script_path, log_dir_script + script_name)
10    # Leite Ausgabe der Konsole um
11    if log_to_file:
12        print("Starting Log To File!")
13        sys.stdout = open(log_dir_script + script_name +
14                           ".log.txt", "w")
15
16    # Erstelle und gebe Callbacks für das Log-Verzeichnis zurück.
17    return tf.keras.callbacks.TensorBoard(
18        log_dir=log_dir_script, profile_batch=5), \
19        tf.keras.callbacks.ModelCheckpoint(log_dir_script + 'model.
mdl_wts.hdf5', save_best_only=True,
20        monitor='val_loss', mode='min'), log_dir_script

```

Listing 6: Logging Funktion für Trainingsvorgänge

Im Folgenden wird lediglich nur noch ein Callback eingeführt, der das Training nach einer gewissen Zahl von Durchläufen, in denen sich der Fehler des Modells nicht verbessert, abbricht. Dann kann das Modell mit Trainings- und Validierungsdaten zur Überprüfung für eine maximale Anzahl von Epochen folgendermaßen antrainiert werden:

```

1 model.fit(train_ds,
2           epochs=50,
3           verbose=1,
4           validation_data=val_ds,
5           callbacks=[tensorboard_callback, checkpoint, es])

```

Listing 7: Start des Trainings mit Keras

Nach dem Trainingsvorgang wird nun die beste Epoche aus dem Logverzeichnis geladen und mit den Testdaten überprüft. Dann wird zunächst die evaluate Methode des Keras-Modells genutzt, um eine Vorhersagegenauigkeit und den Fehler der Testdaten zu bestimmen. Zum Schluss wird mittels Scikit-Learn in der eigens definierten evaluate Funktion eine Konfusionsmatrix und Auswertungstabelle mit Precision, Recall und F1-Score zu den Testdaten erstellt. Dies ist im folgenden Listing erkennbar:

```

1 # Lade das beste Modell
2 model = load_model(run_path + "model.mdl_wts.hdf5")

```

```
3 # Score auf den Testdaten
4 score = model.evaluate(test_ds, verbose=1)
5 print('Test loss:', score[0])
6 print('Test accuracy:', score[1])
7
8 # Entnehme aus dem Test Datensatz, zur Anwendung mit sklearn
9 test_data_labels = []
10 for data, label in test_ds.take(-1):
11     test_data_labels.extend(label)
12
13 # Prediction für den Test Datensatz
14 pred = model.predict(test_ds, verbose=1)
15 # Evaluiere die Ergebnisse vom Testdatensatz mit sklearn
16 evaluate(numpy.argmax(pred, axis=1),
17         numpy.argmax(test_data_labels, axis=1))
```

Listing 8: Auswertung eines Modells mit Testdaten

Abschließend sei erwähnt, dass die Auswertungsergebnisse der Testdaten nur in Betracht gezogen wurden, wenn bereits einige andere Modelle mit Validierungsdaten getestet worden sind. Ein anderes Auswertungsverhalten würde die in Kapitel 4.1.5. genannten Grundsätze verletzen.

6.2.2. Neuronale Netze

Zunächst wurde der Einsatz einfacher Neuronaler Netze erprobt. Da diese grade bei großen Bildern in der Regel von CNNs abgelöst wurden, wurden hier nur wenige Tests ausgeführt. Selbst bei großen Modellen und ohne vorliegenden Overfitting Effekten konnte maximal eine Vorhersagegenauigkeit von 80% erreicht werden. Grundsätzlich wurde versucht, in der Eingabeschicht etwa ein Neuron pro Bildpunkt zu nutzen. Diese sollten dann in jedem Layer mit einer abnehmenden Schichtgröße immer weiter heruntergerechnet werden, bis es sinnvoll wurde, die Ausgabeschicht anzusetzen. Weiterhin folgten auf alle versteckten Schichten ein Dropout und eine Batch Normalisierung. Auf dieser Basis wurden viele Netzstrukturen erprobt. Eines dieser Modelle, welches für das Sign Language MNIST Dataset verwendet wurde, ist in folgender Abbildung zu erkennen:

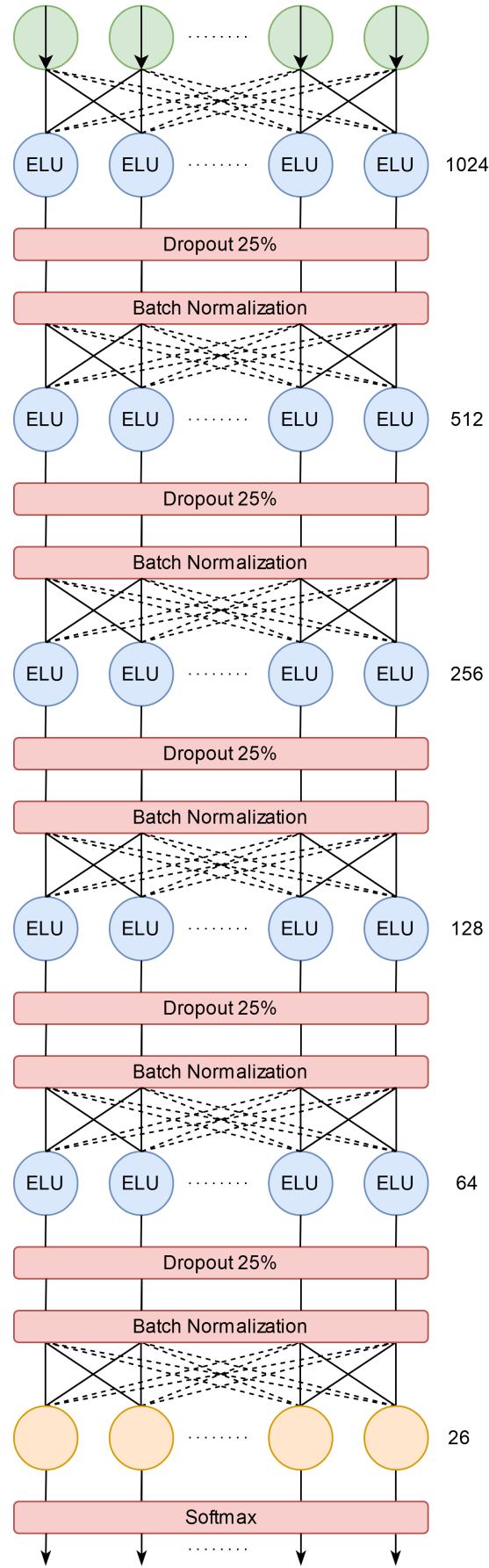


Abbildung 18: Ein eingesetztes Neuronales Netz mit fünf Hidden-Layern.

Dieses Modell erreicht auf dem genutzten Datensatz lediglich eine Vorhersagegenauigkeit von 78% auf den Validierungsdaten. Auf den Testdaten erreicht es eine Vorhersagegenauigkeit von 81%. Dies ist zwar etwas besser, jedoch noch nicht in dem gewünschten Bereich.

Mit dem ASL-Alphabet Dataset gibt es noch größere Probleme beim Einsatz normaler Neuronaler Netze. Die Bilder weisen alle eine Größe von 200x200 Pixeln auf, was in einer Eingabegröße von 40000 Pixeln mit drei Kanälen resultieren würde. Die eingegebenen Bilder werden beim Einlesen daher auf 50x50 Grauwerte reduziert. Selbst damit musste ein siebter Layer mit 2048 ELU-Neuronen an den Anfang des Netzes eingeführt werden, um die 2500 Eingabewerte sinnvoll aufnehmen zu können. Trotz dieser Änderung und dem Test weiterer Netzstrukturen zeigen sich die Probleme mit diesem Datensatz klar. Es konnte dabei maximal eine Vorhersagegenauigkeit von 67% erreicht werden. Grade im ASL-Alphabet Dataset wird daher klar, dass für die Lösung dieser Aufgabe Konvolutionelle Neuronale Netze sinnvoller sind.

6.2.3. Konvolutionelle Neuronale Netze

Beim Training der Konvolutionellen Neuronale Netze wurden besonders für den ASL-Alphabet Dataset viele Modellstrukturen erprobt. Es wurde sich für eine wie in dem VGG19 vorhandene typische Verdopplung der Feature Maps entschieden [14], wobei gleichzeitig weniger Convolutional Layer verwendet wurden, um Overfitting zu vermeiden. Gleichzeitig konnten mit dem Einsatz von Stride zum einfachen Herunterrechnen der Bildinformationen gute Erfahrungen erzielt werden, auch wenn dies im Falle des Sign Language MNIST Dataset durch die geringe Bildgröße in der Regel nur in den ersten ein bis zwei Convolutional Layern verwendet werden konnte. Für die vollständig verbundenen Schichten wurde sich für einen ähnlichen Aufbau wie im vorherigen Kapitel entschieden, inklusive des Dropouts und der Batch-Normalisierung. Mit dem Sign Language MNIST Dataset konnten mit CNNs zwar bessere Ergebnisse erzielt werden als mit einfachen Neuronalen Netzen, jedoch zeigt sich hier klar die geringe Auflösung der Bilder mit 28x28 Pixeln als Problemfaktor. In allen erprobten Modellstrukturen wurden schlechtere Ergebnisse erzielt als für den ASL-Alphabet Dataset. Ein eingesetztes Modell mit guten Ergebnissen für dieses Dataset ist in Folgender Abbildung dargestellt:

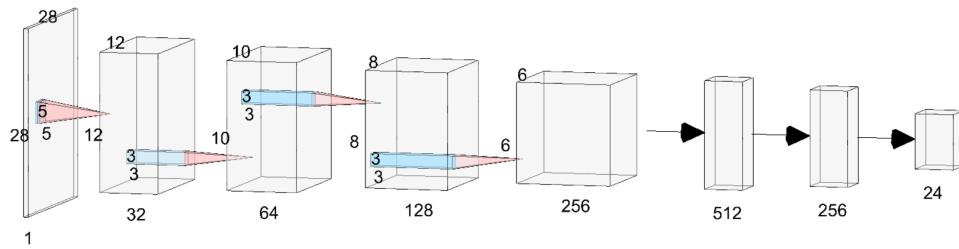


Abbildung 19: Ein Eingesetztes CNN für den Sign Language MNIST Dataset.

Zwar konnte mit diesem Modell eine hundertprozentige Genauigkeit für die Validierungsdaten erreicht werden, es war jedoch nie eine komplette Generalisierung für die Testdaten zu erkennen. Diese erreichen maximale Genauigkeiten von 75%. Die Gründe für das schlechtere Abschneiden konnten bisher nicht herausgefunden werden. Es wären Fehler in der Vorverarbeitung, dem Einlesen, eine starke Abweichung der Testdaten von den Trainings- und Validierungsdaten oder Overfitting denkbar. Leider konnte dies nicht perfekt gelöst werden und aufgrund besserer Ergebnisse für den ASL-Alphabet Dataset wurde sich nun auf diesen konzentriert.

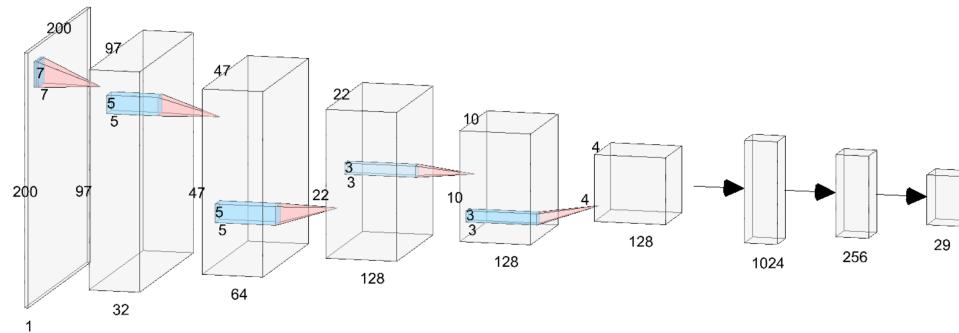


Abbildung 20: Ein Eingesetztes CNN für den ASL-Alphabet Dataset.

Eines dieser Modelle ist in der oberen Abbildung zu erkennen, es konnte sehr gute Ergebnisse erzielen und ist erneut leicht an das VGG19 mit weniger Convolutional Layern angelehnt. Aufgrund der höheren Bildgrößen im ASL-Alphabet Dataset weißt es deutlich höhere Parameterzahlen als das vorherige Modell auf. Insgesamt konnte es eine Vorhersagegenauigkeit von 99% mit Test- und Validierungsdaten erreichen. Es ist damit bisher und insgesamt eines der besten Modelle für die Vorhersage von Fingeralphabeten in dieser Arbeit.

6.2.4. Transfer Learning mittels VGG19

Bei vielen Problemen des maschinellen Lernens ist es sinnvoll, ganze oder Teile von vortrainierten Modellen zu verwenden. Dies ist besonders bei vielen Bereichen der Bilderkennung sinnvoll und üblich. Die Wiederverwendung von Modellen wird auch als Transfer Learning bezeichnet [2, S.287]. Ein solches Modell soll nun auch in diesem Projekt angewendet werden. Es wurde sich für das VGG19 entschieden, da dieses laut der basierenden Arbeit eine sehr gute Generalisierung für andere Datensätze besitzt und von Keras unterstützt wird [14]. Es ist folgendermaßen aufgebaut:

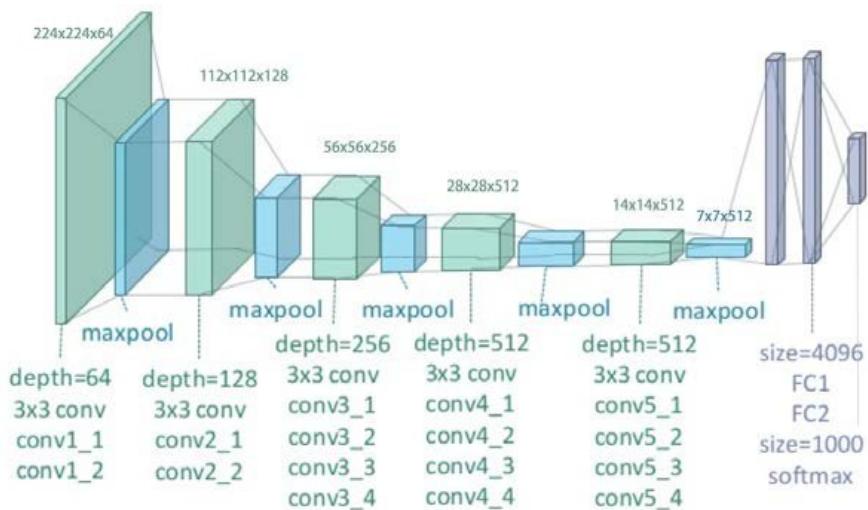


Abbildung 21: Der Aufbau des VGG19 [15].

Es ist zu beachten, dass sich Filter derselben Größe mehrfach wiederholen, sodass mit den vollständig verbundenen Schichten ein Konvolutionelles Neuronales Netz mit 19 Layern entsteht. Die vollständig verbundenen Schichten können in Keras ausgeschaltet werden, was in diesem Fall des Transfer Learning aufgrund der unterschiedlichen Anzahl von nützlichen Ausgabeneuronen (das Netz sieht 1000 vor sinnvoll ist. In der Entwicklung konnte das VGG19 mit angepassten, vollständig verbundenen Schichten nahezu perfekt abschneiden. Dabei wurden in der Regel die originalen Schichten des VGG19 aus dem Lernvorgang ausgeschlossen, sodass deren angelernten Features verwendet wurden, um die Zeichen der Hände in den folgenden, vollständig verbundenen Schichten zu klassifizieren. Durch das Transfer Learning konnten Genauigkeiten von nahezu 100% in der Validierung und in den Tests erreicht werden. Es ist damit das am besten abschneidende Verfahren. Eines der genutzten Netze ist in folgender Abbildung zu erkennen:

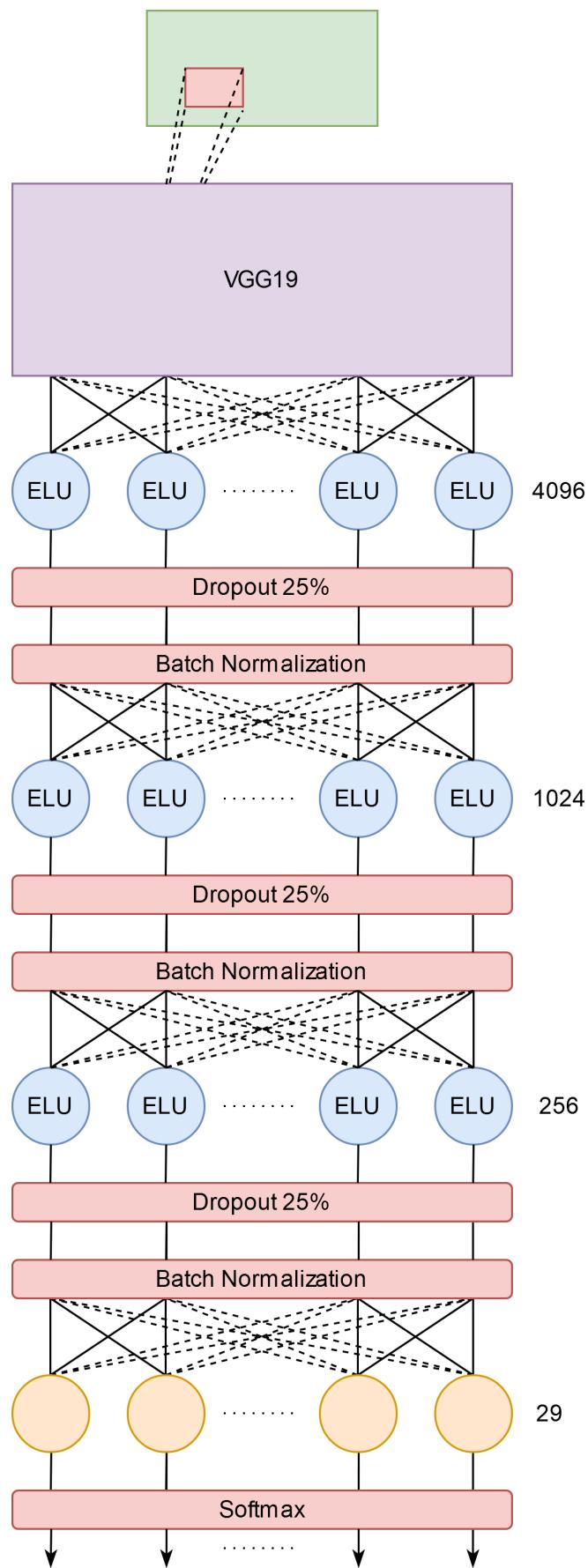


Abbildung 22: Angepasstes VGG19 mit sehr guten Ergebnissen.

6.3. Backend der Webanwendung

Das Backend des Projekts wurde mit Python implementiert, um die ebenfalls mit Python trainierten Modelle einfach einbinden zu können. Python wurde um die Bibliothek Flask ergänzt, welche erlaubt, mit wenig Aufwand HTTP Endpunkte zur Verfügung zu stellen. Das Backend dient dazu, die trainierten Modelle für die Erkennung des Fingeralphabets über das Netzwerkprotokoll HTTP und die angesprochenen Endpunkte durch beliebige Clients nutzbar zu machen. Das in diesem Projekt implementierte Frontend nutzt die HTTP-Schnittstelle des Backends und kann also durch beliebige andere Clients aus dritter Hand ersetzt werden. Um sicherzustellen, dass der Zugriff von externen Domains möglich ist, wird über Cross-Origin Resource Sharing die Same-Origin Policy aufgeweicht. Die Same-Origin Policy wird von Browsern implementiert und schränkt den Zugriff auf eine Domain ein, sodass dieser nur von der gleichen Domain mit dem gleichen Port erfolgen kann.

Einsprungspunkt des Python Backends ist die Datei `__init__.py`, welche Flask initialisiert und die durch Keras gespeicherten Modelle in eine globale Variable lädt. Genutzt wird dazu die Funktion `load_model` aus dem Import `tensorflow.keras.models` und das bewährte Dateiformat HDF5 (Hierarchical Data Format), welches auf die Speicherung großer Datenmengen in wissenschaftlichen Anwendungen zugeschnitten ist. Die geladenen Modelle werden in der Datei `services.py` als unterschiedliche HTTP-Endpunkte mit auf das Modell angepassten Parametern registriert. So muss beispielsweise je nach Input-Layer der Modelle das Bild eine gewisse Form aufweisen, die „Target-Shape“ wird mit dem Modell dementsprechend hart verzahnt.

```
6 # GültigeUrls für Modelle
7 model_urls = {
8     "150x150_5_Layer_CNN.hdf5": "/cnn_5_150_150",
9     "224x224_VGG19_CNN.hdf5": "/vgg19_224_224",
10    "224x224_VGG19_CNN_v2.hdf5": "/vgg19_224_224_v2"
11 }
12
13 # Infos der Modelle mit URLs zurückgeben
14 api.add_resource(ModelInfo, "/model_info", endpoint="model_info",
15     resource_class_kwargs={"model_urls": model_urls})
```

```

17 # Eigenes Modell mit Eingangsgröße 150x150 und 5 CNN Layern.
18 api.add_resource(SignLanguageCNN,
19     model_urls["150x150_5_Layer_CNN.hdf5"],
20     endpoint="cnn_5_150_150",
21     resource_class_kwargs={
22         "document_outputs": False,
23         "target_shape": (150, 150),
24         "model_config":
25             app.config["MODELS"]["150x150_5_Layer_CNN.hdf5"]})
26
27 # VGG19 mit Eingangsgröße 224x224 umtrainiert.
28 api.add_resource(SignLanguageCNN,
29     model_urls["224x224_VGG19_CNN.hdf5"],
30     endpoint="vgg19_224_224",
31     resource_class_kwargs={
32         "document_outputs": False,
33         "to_grayscale": False,
34         "target_shape": (224, 224),
35         "model_config":
36             app.config["MODELS"]["224x224_VGG19_CNN.hdf5"]})
37
38 # VGG19 mit Eingangsgröße 224x224 umtrainiert
39 # und mit VGG19_Preprozess der Bilder.
40 api.add_resource(SignLanguageCNN,
41     model_urls["224x224_VGG19_CNN_v2.hdf5"],
42     "/default",
43     endpoint="vgg19_224_224_v2",
44     resource_class_kwargs={
45         "document_outputs": False,
46         "to_grayscale": False,
47         "target_shape": (224, 224),
48         "model_config":
49             app.config["MODELS"]["224x224_VGG19_CNN_v2.hdf5"]})

```

Listing 9: Anlegen der HTTP-Endpunkte.

Der oberste Endpunkt /model_info ist in der Datei ModelInfo.py implementiert und erlaubt es, per HTTP-GET die hinterlegten Modelle und die dazugehörige URL abzufragen. So kann ein Client zwischen den Modellen hin- und herwechseln. Gerade für das Testen in der Praxis war das wichtig und die unterschiedlichen Leistungen konnten leichter herausgearbeitet werden. Das Backend wurde unter <https://jupiter.fh-swf.de/sign-language> gehosted, sodass für die

Abfrage der Modelle https://jupiter.fh-swf.de/sign-language/model_info per HTTP-GET angesprochen werden muss.

```

4  class ModelInfo(Resource):
5      """
6          Resource, die URL Infos zu den Modellen zurückgibt.
7      """
8  def __init__(self, model_urls):
9      """
10     Resource, die URL Infos zu den Modellen zurückgibt.
11     :param model_urls: URL Infos, die zurückgegeben werden.
12 """
13 self.model_urls = model_urls
14
15 def get(self):
16     """
17     Gibt die Model Infos mittels get-Request zurück.
18 """
19 return self.model_urls, 200

```

Listing 10: Abfrage der möglichen Modelle.

Der Hauptendpunkt ist in der Klasse SignLanguageCNN in der Datei SignLanguageCNN.py implementiert. Dieser erlaubt es, per HTTP-POST ein Bild an das Backend zu senden, welches je nach URL (siehe zwei Bilder darüber) durch ein unterschiedliches Netz verarbeitet wird. Als Parameter dienen image und image_base_64. Der Parameter image_base_64 wurde eigens für das Frontend hinzugefügt, da die Bildextraktion aus einem HTML5 Canvas lediglich Base64-kodiert gelingt. Je nach Modell ist eine Auswertung der Farbe nicht vorgesehen, sodass das Bild nach der Anpassung der Größe auf den Input Layer noch in Graustufen umgewandelt werden kann.

```

44 # Image per FileStorage im Request
45 if args.image:
46     image = args.image
47 # Image per Base64 in Post
48 else:
49     image = BytesIO(base64.b64decode(args.image_base_64))
50
51 # Öffne das Bild und resize es.
52 image_pil = Image.open(image).resize(self.target_shape)
53 color_dim = 3
54

```

```
55 # Wenn Grauwert ausgewertet
56 if self.to_grayscale:
57     image_pil = image_pil.convert("L")
58     color_dim = 1
59 # Wenn RGB ausgewertet
60 else:
61     image_pil = image_pil.convert("RGB")
62
63 # Image in ein Tensorflow Array, Reshape es in 4
64 # Dimensionen (BatchSize (1), Breite, Höhe, Farben)
65 image_tf = img_to_array(image_pil).reshape(
66     (1, self.target_shape[0], self.target_shape[1], color_dim)
67 )
68 # Prediction für das Bild, mit der genutzten Config.
69 prediction = self.model_config["MODEL"].predict(image_tf).tolist()
```

Listing 11: Verarbeitung der Bilder im HTTP-Endpunkt.

Das Ergebnis der Prediction ist eine Klassifizierung mit den jeweiligen Wahrscheinlichkeiten der zutreffenden Klasse. Der Wertebereich bewegt sich zwischen 0 und 1, wobei 1 als voll zutreffend gilt. Das Ergebnis wird in ein Dictionary bestehend aus Klasse und Prediction-Value überführt und mit einem Statuscode 200 an den aufrufenden Client zurückgeschickt.

6.4. Frontend der Webanwendung

Das webbasierte Frontend dient als Praxisanwendung, um die Modelle außerhalb der Testdaten zu nutzen. Das Frontend wurde mithilfe von Angular, Angular Material und RxJS als Progressive Web App programmiert und unter der URL <https://jupiter.fh-swf.de/sign-language-classification/> zur Verfügung gestellt. Ge-hosted wird die Applikation sowie auch das Backend hinter einem nginx Webserver. Aufgeteilt wurde die Anwendung in die eigentliche Zeichenerkennung und einer Übersicht aller möglichen Handzeichen, erreichbar über das Menü in der Topbar. Die möglichen Handzeichen wurden mit Bildern modelliert, die die Hand von Vorne zeigen. Diese Bilder entstammen einem Icon-Set für das unter 2. angesprochene amerikanische Fingeralphabet. Die tatsächliche Handstellung kann durch Rotation etwas abweichen, sodass Originalbilder einer menschlichen Hand in der Übersicht eingefügt wurden. Die Originalbilder stammen aus dem Trainingsdatensatz und sind mit dem Bewegen der Maus über das Handzeichen erreichbar. Die

Auflösung der Originale ist mit 200x200 Pixeln nicht besonders hoch, aber das Erkennen der richtigen Handstellung ist problemlos möglich. Die folgende Darstellung zeigt das Handzeichen „D“ als Icon und im Original.

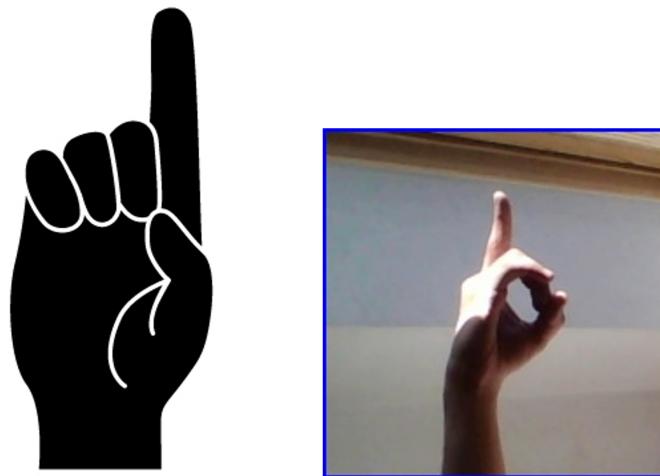


Abbildung 23: Das „D“ Handzeichen als Icon und im Original.

Im Frontend kann auf der Seite für die Zeichenerkennung mit dem Leerzeichen oder einem Klick auf „Start Prediction“ die Erkennung über eine Webcam gestartet werden. Der Zugriff auf die Webcam muss dabei aktiv vom Benutzer erlaubt werden. Frontendseitig wird mithilfe von Googles MediaPipe Hands die Hand im Bild in Echtzeit getrackt. MediaPipe Hands besteht aus mehreren Modellen, die über eine ML-Pipeline zusammenarbeiten. Das erste Modell operiert auf dem gesamten Bild und findet die Handfläche, danach nutzt das zweite Modell den Ausschnitt der gefundenen Handfläche und markiert 21 markante Punkte der Hand. MediaPipe Hands konnte mithilfe von TensorFlow.js frontendseitig eingebunden und angesprochen werden. Das Laden des Handtracking-Modells wird in der Datei `main.ts` durchgeführt und automatisch an das globale Window-Objekt angehängt.

```
16  (async () => {
17    const model = await handpose.load();
18    (window as any).tensorflowModel = model;
19    platformBrowserDynamic().bootstrapModule(AppModule)
20      .catch(err => console.error(err));
21 })();
```

Listing 12: Laden der frontendseitigen MediaPipe Hands Modelle.

Sobald das Modell frontendseitig registriert wurde, wird die Angular Applikation gestartet. Da sich das Frontend die Bibliothek lediglich zu Nutze macht und in das Trainieren in ein Handtracking-Modell keine Eigenleistung investiert wurde, wird technisch nicht tiefer auf MediaPipe Hands eingegangen. Das Bild der Webcam wird dem Benutzer normalerweise in einem HTML5 Video-Element angezeigt. Aber da auf so ein Element weder gezeichnet noch Einzelbilder extrahiert werden können, wurde für die Ausgabe ein Canvas-Element genutzt. Die Hauptaufgabe der frontendseitigen Handerkennung passiert in der Datei `webcam.component.ts` unter `src/app/sign-prediction/webcam/` in der Methode `estimateHands`. Hier wird nach Auslesen der Höhe und Breite des Video- und Canvas-Elements das Bild der Webcam in das Canvas übertragen. Dazu wird der Rendering-Context des Canvas-Elements genutzt und das Video-Element zuvor ausgeblendet. Über die zuvor angesprochene globale Variable kann auf das Handtracking-Modell zugegriffen und das aktuelle Webcam-Bild übergeben werden. Wird eine Hand gefunden, werden die Koordinaten extrahiert und in einen mit RxJS erstellten Stream übergeben.

```
124 private async estimateHands() {  
125  
126     const videoWidth = this.video.nativeElement.videoWidth;  
127     const videoHeight = this.video.nativeElement.videoHeight;  
128     const canvasWidth = this.overlay.nativeElement.width;  
129     const canvasHeight = this.overlay.nativeElement.height;  
130  
131     this.renderingContext.drawImage(  
132         this.video.nativeElement, 0, 0, videoWidth,  
133             videoHeight, 0, 0, canvasWidth,  
134             canvasHeight  
135     );  
136  
137     if (this.isPredictionActive) {  
138         const model: HandPose = (window as any).tensorflowModel;  
139         const hands: Array<any> =  
140             await model.estimateHands(this.video.nativeElement);  
141         const boundingBox = hands.length > 0 ?  
142             { topLeft: hands[0].boundingBox.topLeft,  
143                 bottomRight: hands[0].boundingBox.bottomRight } :  
144             { topLeft: ['0', '0'], bottomRight: ['0', '0'] };  
145  
146         this.croppedRenderingContext.clearRect(0, 0,  
147             this.croppedCanvas?.nativeElement.width,
```

```
149     this.croppedCanvas?.nativeElement.height);
150
151     this.ngZone.run(() =>
152       this.boundingBoxSubject.next(boundingBox)
153     );
154
155     requestAnimationFrame(this.estimateHands.bind(this));
156   }
157 }
```

Listing 13: Erkennen einer Hand mit MediaPipe Hands.

Mithilfe der `requestAnimationFrame` Methode wird pro Frame `estimateHands` erneut aufgerufen, bis der Benutzer abbricht. Das Handtracking wird mithilfe von WebGL auf der Grafikkarte ausgeführt, sodass es ohne Grafikkarte zu FPS-Einbrüchen kommen kann. Der RxJS Stream kümmert sich nachgelagert um das Einzeichnen einer roten Box um die Hand des Benutzers.

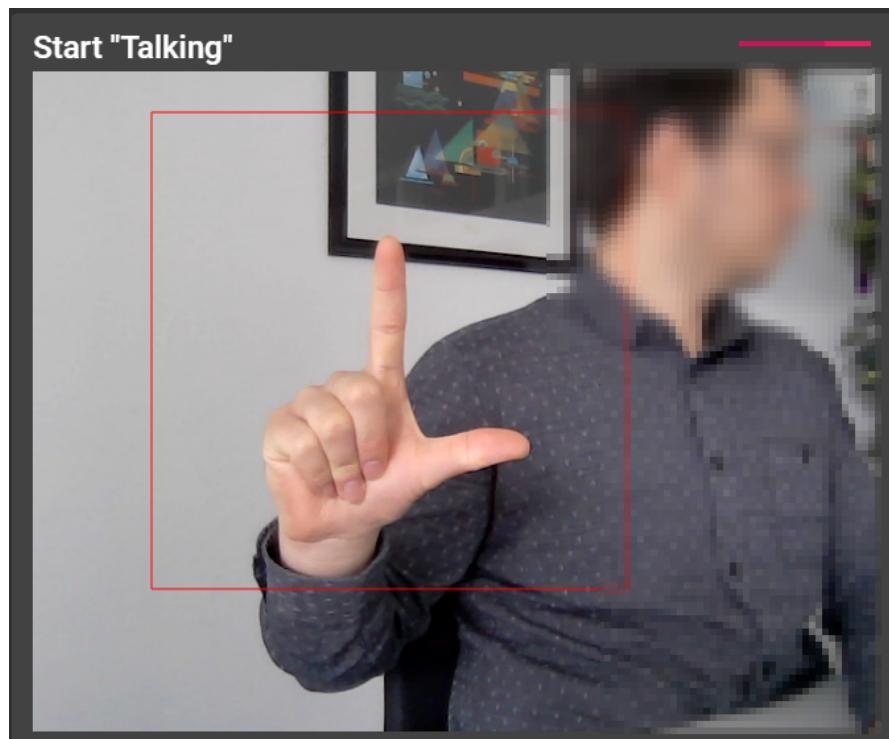


Abbildung 24: Bild der Webcam, welche an das Backend übertragen wird.

In einem 1,5 Sekunden-Intervall wird das aktuelle Bild der Webcam an das Python Backend übertragen und das Ergebnis im Frontend visualisiert. Da das trainierte

Modell für das Fingeralphabet auch eine Klasse für „kein Handzeichen des Fingeralphabets“ hat, wird das Bild auch ohne erkannte Hand im Frontend übertragen. Sollte mit Ablauen des Intervalls im Frontend eine Hand erkannt worden sein, wird diese mithilfe der Box quadratisch ausgeschnitten. Dies erleichtert dem Anwender die Positionierung der Hand in eine möglichst optimale Hintergrundumgebung ohne ablenkende Objekte oder Farben. Diese Optimierung war nötig, da die Trainingsdaten stark auf die Hand fokussiert sind und Hintergründe mit markanten Formen die Erkennung trotz automatischer Merkmalserkennung eines Konvolutionellen Neuronalen Netzes beeinflusst haben. Das Intervall ist momentan nicht konfigurierbar und wurde als guter Mittelwert für ungeübte Nutzer des Fingeralphabets eingesetzt. Während des Betriebs kann das aktuelle Modell mit der Auswahlbox über der Webcam-Ausgabe gewechselt werden, ohne das Tracking zu beenden.

Neben den Koordinaten der Hand erkennt das frontendseitige Modell wie bereits erwähnt zusätzlich 21 Punkte auf der Hand, wodurch die Fingerstellung genau abgelesen werden kann.

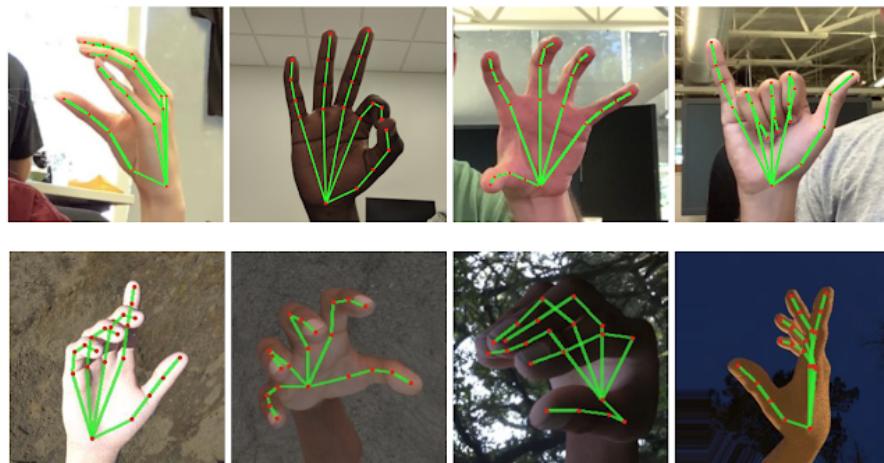


Abbildung 25: Punktedarstellung eines Handzeichen in Mediapipe.

Dies hätte die Vorhersage des Fingeralphabets stark vereinfacht und die Bildverarbeitung ersetzen können. Da das Projekt aber mit Konvolutionellen Neuronalen Netzen auf Bildklassifizierung abzielt, wurde in diesem Projekt die Möglichkeit nicht berücksichtigt.

6.5. Ergebnisse der Entwicklung

Aufgrund der geringen Informationsdichte und der daraus vermutlich resultierenden schlechten Praxisergebnisse wurde der Sign Language MNIST Datensatz schnell verworfen. Für den ASL Alphabet Datensatz wurde per zufälligem Ziehen ein Testdatensatz mit 300 Elementen angelegt. Auf Basis der Ordner-Struktur konnte mit der Funktion `image_dataset_from_directory` zügig ein Dataset zum Trainieren eines Modells erstellt werden. Sämtliche Trainingsdurchläufe wurden wie unter [6.2.1.](#) beschrieben geloggt und konnten so jederzeit nachvollzogen und erneut erzeugt werden. Die einfachen Neuronalen Netze konnten lediglich auf dem kleinen Sign Language MNIST Datensatz sinnvoll trainiert werden. Die Vorhersagegenauigkeit lag je nach Datensatz aber lediglich zwischen 67% und 81% für die jeweiligen Testdaten. Die trainierten Konvolutionellen Neuronalen Netze konnten auf dem ASL Alphabet Datensatz über 99% Vorhersagengenauigkeit auf Test- und Validierungsdaten erreichen und wurden nur mithilfe von Transfer Learning eines VGG19 Modells überboten. Die guten Ergebnisse waren wenig überraschend, da CNNs de facto als Standard für Bildverarbeitung gelten. Auch eine Konfusionsmatrix zeigt, dass es unter den sehr geringen Fehlern keine besonderen Ausreißer gibt, obwohl sich einige Buchstaben recht ähnlich sind. Somit wurde das VGG19 Modell als Standard im Frontend hinterlegt. Das Python Backend ist in der Lage, die unterschiedlichen Modelle in den Speicher zu laden und Bilder mit oder ohne Base64 Kodierung zu verarbeiten. Die Modelle sind über unterschiedliche Endpunkte per HTTP direkt ansprechbar und eine Prediction dauert unter 100 Millisekunden. Die im Backend hinterlegten Modelle können abgefragt werden. Das Frontend visualisiert die möglichen Handzeichen und ist in der Lage, über die Webcam und MediaPipe Hands die Hand zu verfolgen. In einem Intervall von 1,5 Sekunden wird das aktuelle Bild an das Backend übertragen und das Ergebnis dem Benutzer angezeigt. Damit konnte die geplante Entwicklung im Rahmen dieses Projekts erfolgreich abgeschlossen werden.

7. Fazit

Das Projekt hat neben den theoretischen Grundlagen von Machine Learning einfache Neuronale Netze, Konvolutionelle Neuronale Netze und Transfer Learning näher beleuchtet. Die besten Modelle erreichten nahezu 100% Vorhersagegenau-

igkeit auf Test- und Validierungsdaten. Darüber hinaus wurden die Modelle in eine Anwendung integriert, um die Vorhersagen auf echten Daten zu prüfen. Trotz der ausgezeichneten Ergebnisse auf den Datensätzen zeigen sich Schwächen mit Daten in der Praxis. Eine richtige Vorhersage der Handzeichen kann mit etwas Übung des Benutzers zwar erreicht werden, aber die Positionierung vor der Webcam nimmt noch zu großen Einfluss. Markante Formen und Farben, die in den Bildausschnitt der Hand hineinragen, erschweren zusätzlich die Erkennung, sodass der Benutzer nach Möglichkeit einen ruhigen Hintergrund wählen sollte. Auch wenn die aus diesem Projekt hervorgegangene Applikation noch nicht für eine reale Nutzung vollends geeignet ist, sind die Endergebnisse zufriedenstellend, vor allem mit dem Hintergrund, dass die Applikation bestehend aus Frontend und Backend vordergründig als On-Top Ergänzung zu den trainierten Modellen gedacht war. Uns war es aber wichtig, in diesem Projekt eine Applikation zu implementieren, da die Schwierigkeit beim Machine Learning gerade darin besteht, die Modelle praxistauglich aus der Testumgebung in die echte Welt zu übertragen.

8. Ausblick

Während der Planung und Entwicklung des Projekts sind einige Punkte aufgefallen, die zur Verbesserung zusätzlich umgesetzt werden könnten. Um im Rahmen des Projektumfangs zu bleiben, werden diese anschließend nur in schriftlicher Form vorgestellt.

8.1. Data Augmentation

Der genutzte ASL Alphabet Datensatz hat mit 87000 Bildern bereits eine angemessene Größe. Die Bilder scheinen allerdings alle von einer Person erzeugt worden zu sein. Die Bilder für einen Buchstaben unterscheiden sich zwar stärker als im Sign Language MNIST Datensatz, aber dennoch deutlich weniger als mit unterschiedlichen Personen. So könnten die Bilder manuell rotiert und umpositioniert werden, um einen breiteren Datensatz zu erzeugen und ein besseres Ergebnis für Praxisdaten zu erhalten.

8.2. Ausbau des Traingsdatensatzes

Alternativ zur Data Augmentation könnte der Datensatz durch Praxisdaten erweitert werden. Das Frontend sendet sämtliche Bilder Base64-kodiert an das Python Backend. Diese Bilder könnten gespeichert und manuell klassifiziert werden. Auf lange Sicht könnte der bestehende Datensatz sogar ersetzt werden, um das Ergebnis an die Bilder der Webcam anzupassen und zu verbessern.

8.3. Performance des Frontends

Das Frontend arbeitet selbst mit einem großen Modell mithilfe von TensorFlow.js, um die Hand in dem Bild der Webcam zu finden. Dies erlaubt es dem Benutzer, Feedback zu geben und die Hand bereits frontendseitig auszuschneiden. Das Modell wird mit WebGL direkt auf der Grafikkarte ausgeführt und wird mit dem Start der Applikation geladen. Insgesamt werden ca. 15MB übertragen, welche für eine durchschnittliche Internetverbindung keine Probleme darstellen. Dennoch könnte die Startzeit der Applikation mit geringerer Dateigröße verringert werden. Außerdem ist eine dedizierte Grafikkarte vorauszusetzen, um ein flüssiges Arbeiten zu garantieren. Mit dem Verzicht auf das Handtracking könnte das Modell wegfallen und die Aufbereitung aus dem Frontend in das Backend übertragen werden. So wäre ein flüssiges Arbeiten mit dem Frontend auf nahezu jeder Hardware möglich.

8.4. Textausgabe im Frontend

Momentan wird im Frontend der erkannte Buchstabe hervorgehoben. Das trainierte Modell bietet zusätzlich noch die Klassen Space und Delete, die dazu genutzt werden könnten, eine sinnvolle Textausgabe auf Basis der Vorhersagen zu realisieren.

8.5. Intervall der Handzeichenerkennung

Zurzeit ist das Intervall, in dem das Bild der Webcam an das Backend übertragen wird, auf ungeübte Nutzer ausgelegt. Dank der geringen Antwortzeit des Backends von unter 100ms könnte das Intervall deutlich verringert werden. Dadurch wäre

ein deutlich flüssigeres Arbeiten mit dem Frontend möglich. Alternativ könnte auf das Intervall gänzlich verzichtet werden, indem das frontendseitige Handtracking als Grundlage genutzt wird. Dazu könnte ein gewisses Delta für die Handkoordinaten im Bild festgelegt werden. Sobald dieses Delta überschritten wurde, wird ein neues Zeichen oder ein wiederholtes gleiches Zeichen angenommen und das Backend angesprochen.

9. Quellen

- [1] *Abbildung des Fingeralphabets*. url: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c8/Asl_alphabet_gallaudet.svg/380px-Asl_alphabet_gallaudet.svg.png (besucht am 16.07.2020).
- [2] Aurelien Geron. *Praxiseinstieg: Machine Learning mit Scikit-Learn & TensorFlow*. Heidelberg: O'Reilly, 2018. isbn: 978-3-96009-061-8.
- [3] Josh Patterson und Adam Gibson. *Deep Learning: A Practitioner's Approach*. Sebastopol: O'Reilly, 2017. isbn: 978-1-491-91425-0.
- [4] Sebastian Schmidt. "Automatische Analyse von Stimmmerkmalen zur Vorhersage von Persönlichkeitsprofilen mittels Künstlicher Intelligenz". Iserlohn: Fachhochschule Südwestfalen, 25. Okt. 2019.
- [5] Jonghwa Yim und Kyung-Ah Sohn. "Enhancing the Performance of Convolutional Neural Networks on Quality Degraded Datasets". Suwon, Korea: Department of Computer Engineering: Ajou University, 18. Okt. 2017. url: <https://arxiv.org/ftp/arxiv/papers/1710/1710.06805.pdf> (besucht am 05.07.2020).
- [6] Google LLC. *Cross-platform ML solutions made simple*. url: <https://google.github.io/mediapipe/> (besucht am 05.07.2020).
- [7] Erwin Quiring, David Klein, Daniel Arp, Martin Johns und Konrad Rieck. "Adversarial Preprocessing: Understanding and Preventing Image Scaling Attacks in Machine Learning". Braunschweig, Germany: Technische Universität Braunschweig, 2020. url: https://www.usenix.org/system/files/sec20fall_quiring_prepub.pdf (besucht am 05.07.2020).
- [8] Roland Schwaiger und Joachim Steinwendner. *Neuronale Netze programmieren mit Python*. Bonn: Rheinwerk, 2019. isbn: 978-3-8362-6142-5.
- [9] Günter Daniel Rey und Karl F. Wender. *Neuronale Netze: Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. Bern: Huber, 2011. isbn: 978-3-456-84881-5.
- [10] Günter Daniel Prof. Dr. Rey. *Künstliche neuronale Netze*. url: https://www.tu-chemnitz.de/phil/imf/psyler/lehre/WS18-19/S_KnN/3%20Lernregeln.pdf (besucht am 07.10.2019).

- [11] Terrence J. Sejnowski. *The Deep Learning Revolution*. MIT: The MIT Press, 2018. isbn: 978-0-262-03803-4.
- [12] Rajalingappa Shanmugamani. *Deep Learning for Computer Vision*. Birmingham: Packt, 2018. isbn: 978-1-78829-562-8.
- [13] keras. *Image data preprocessing*. url: <https://keras.io/api/preprocessing/image/>.
- [14] Andrew Zisserman und Karen Simonyan. "Very Deep Convolutional Networks for Large-Scale Image Recognition". Oxford: University of Oxford, 2014. url: <https://arxiv.org/pdf/1409.1556.pdf> (besucht am 14.07.2020).
- [15] Yufeng Zheng, Clifford Yang und Aleksey Merkulov. "Breast cancer screening using convolutional neural network and follow-up digital mammography". Connecticut: University of Connecticut, Mai 2018.
- [16] *Sign Language MNIST*. url: <https://www.kaggle.com/datamunge/sign-language-mnist>.
- [17] *ASL Alphabet*. url: <https://www.kaggle.com/grassknotted/asl-alphabet>.