

Bachelorarbeit

Automatische Analyse von Stimmerkmalen zur Vorhersage von Persönlichkeitsprofilen mittels Künstlicher Intelligenz

Sebastian Schmidt

Matrikelnummer: 10053783

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

Erstprüfer: Prof. Dr. Michael Rübsam

Zweitprüfer: Prof. Dr. Christian Gawron

25. Oktober 2019

Eigenständigkeitserklärung

Ich erkläre, dass ich die Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel benutzt habe. Alle Stellen, 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 habe ich die vorliegende Arbeit an keiner anderen Stelle zur Erlangung eines Abschlusses vorgelegt.

Datum, Ort

Unterschrift

Inhaltsverzeichnis

1. Einleitung	2
2. Eingrenzung	2
3. Theoretische Grundlagen	2
3.1. Allgemeine Grundlagen des Maschinellen Lernens	2
3.1.1. Arten von Verfahren des Maschinellen Lernens	2
3.1.2. Typische Probleme beim Maschinellen Lernen	4
3.1.3. Datenvorverarbeitung für Bilddaten	6
3.1.4. Qualitätsmaße	7
3.1.5. Testen und Validieren	10
3.2. Neuronale Netze	11
3.2.1. Biologische Grundlagen	12
3.2.2. Künstliche Neuronen	12
3.2.3. Einschichtige Neuronale Netze	14
3.2.4. Gradientenabstiegsverfahren	15
3.2.5. Mehrschichtige Neuronale Netze	17
3.2.6. Backpropagation-Verfahren	18
3.2.7. Aktivierungsfunktionen	19
3.2.8. Regularisierungsverfahren	20
3.3. Konvolutionelle Neuronale Netze	21
3.3.1. Convolutional Layer	22
3.3.2. Pooling Layer	25
4. Datensätze	26
4.1. Sign Language MNIST Dataset	26
4.2. ASL Alphabet Dataset	26
4.3. Videodatensätze	26
5. Entwicklungsprozess	26
5.1. Datenvorverarbeitung	26
5.1.1. Sign Language MNIST Dataset	27
5.1.2. ASL Alphabet Dataset	28
5.1.3. DatasetParser Pakete	29

5.2. Entwickelte Modelle	30
5.2.1. Grundaufbau des Modelltrainings	30
5.2.2. Neuronale Netze	34
5.2.3. Konvolutionelle Neuronale Netze	34
5.2.4. Transferlearning mittels VGG19	34
5.3. Backend der Webanwendung	34
5.4. Frontend der Webanwendung	34
5.5. Ergebnisse der Entwicklung	34
6. Fazit	34
7. Ausblick	34
8. Quellen	35

Abbildungsverzeichnis

1	Overfitting in einem mehrschichtigen Neuronalen Netz	5
2	Beispiel einer Konfusionsmatrix	8
3	Klassenweises Beispiel für TP, TN, FN, FP in einer Konfusionsmatrix	9
4	Beispiel für Kreuzvalidierung	11
5	Aufbau eines biologischen Neurons	12
6	Aufbau eines künstlichen Neurons	13
7	Einschichtiges Neuronales Netz	14
8	Fehler im Neuronalen Netz	16
9	Gradientenabstiegsverfahren in zwei Dimensionen	17
10	Mehrschichtiges Neuronales Netz	18
11	Aufbau des visuellen Cortex	22
12	Einzelner Filter eines Convolutional Layers	23
13	Beispiel für horizontale und vertikale Filter	23
14	Beispiel für verschiedene visualisierte Filter	24
15	Beispiel eines CNNs mit mehreren Feature Maps	24
16	Arbeitsweise des Max Pooling	25

Listingverzeichnis

1	Kern der Sign Language MNIST Dataset Vorverarbeitung.	27
2	Kern der Sign Language MNIST Dataset Vorverarbeitung.	28
3	Beispielhafter Aufruf der Funktion zum Einlesen von Daten	29
4	Beispielhafte Vorbereitung für das Training	31
5	Kompilierung eines Modells	31
6	Logging Funktion für Trainingsvorgänge	32
7	Start des Trainings mit Keras	33
8	Auswertung eines Modells mit Testdaten	33

Formelverzeichnis

1	Kreuzentropie	8
2	Vorhersagegenauigkeit	9
3	Recall	9
4	Precision	10
5	F1-Score	10
6	Mathematische Definition eines Künstlichen Neurons	13
7	Schwellenwertfunktion eines Perzeptrons	13
8	Matrizendarstellung eines einschichtigen Neuronalen Netzes	15
9	Mathematische Definition eines einschichtigen Neuronalen Netzes	15
10	Ziel des Gradientenabstiegsverfahren	15
11	ReLU	20
12	ELU	20
13	Softmax	20

1. Einleitung

2. Eingrenzung

3. Theoretische Grundlagen

In diesem Abschnitt sollen theoretische Grundlagen des Maschinellen Lernens vorgestellt werden. Es wird zunächst auf allgemeine Grundlagen eingegangen, um im folgenden auf die spezielleren Neuronalen Netze und Konvolutionellen Neuronale Netze einzugehen.

3.1. Allgemeine Grundlagen des Maschinellen Lernens

Maschinelles Lernen gibt einem Computer die Möglichkeit zu lernen ohne explizit von einem Entwickler für eine Aufgabe programmiert zu werden. Tom Mitchell beschreibt Maschinelles Lernen 1997 mit einem 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 [1, S. 4]. Erfahrungen sammelt ein System, indem Daten mit speziellen Algorithmen betrachtet werden. Auf Basis dieser Daten erstellt der Algorithmus dann eine Strukturbeschreibung, welche auch als Modell bezeichnet wird [2, S. 2].

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

3.1.1. Arten von Verfahren des Maschinellen Lernens

Verfahren des Maschinellen Lernens werden in der Literatur häufig mittels verschiedener Kategorien in verschiedene 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 betrachtet, die Verfahren einteilen [1, S.8-14][3, S.2].

Zunächst wird ein Verfahren danach eingeteilt, welche Informationen zu den Trainingsdaten anhand von Label nötig sind. Muss jeder Datenpunkt mit einem Label versehen sein, so handelt es sich um Überwachtes Lernen. Dem gegenüber steht das Unüberwachte Lernen, bei dem ein Algorithmus versucht Aussagen über mögliche Label aufgrund 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 [1, S.8-14][3, 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. Dem gegenüber steht das Online-Learning, welches nach und nach trainiert wird, und noch eine spätere Verfeinerung ermöglicht [1, S.14-17][3, S.2].

Mit einer weiteren Kategorie wird festgelegt wie ein Verfahren Daten verallgemeinert, um neue Aussagen treffen zu können. Mit 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 [1, S.14-17][3, S.2].

Ebenfalls lassen sich Verfahren nach der Art des zu lösenden Problems einteilen. Bei der Klassifikation werden Daten anhand ihrer Label 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 [1, S.8-9][3, 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 Folgenden weitere Bilder korrekt einem Zeichen in Gebärdensprache zuweisen können. Instanzbasiertes Lernen könnte bei Bilddaten zu Problemen führen. Da Bilder in der Regel relativ viele Daten beinhalten, könnte ein Merken der Beispiele zu sehr großen Modellen führen [1, S.18]. Aus diesem Grund sollte modellbasiertes Lernen vorgezogen werden. Es ist allerdings sowohl Batch-Learning als auch Online-Learning denkbar.

3.1.2. 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 [1, S.23][3, 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 Gelerte repräsentiert. Auch eine ungünstige Verteilung von Klassen in den Daten, kann zu Probleme führen, besonders wenn dieser Fall mit unpassenden Qualitätsmaßen ausgewertet wird [1, S.24-25][3, 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. [1, S.26][3, S.3]. Hilft dies nicht ist ein Sammeln von neuen Datensätzen sinnvoll.

Irrelevante oder schlecht gewählte Merkmale für das Antrainieren eines Datensat-

zes 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 [1, S.26][3, S.4].

Overfitting tritt 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 Performanz 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 auflegen, der sogenannten Regularisierung. Außerdem lässt es sich durch Sammeln neuer, sowie der Verringerung von Rauschen in vorhandenen, Daten reduzieren [1, S.27][3, S.13].

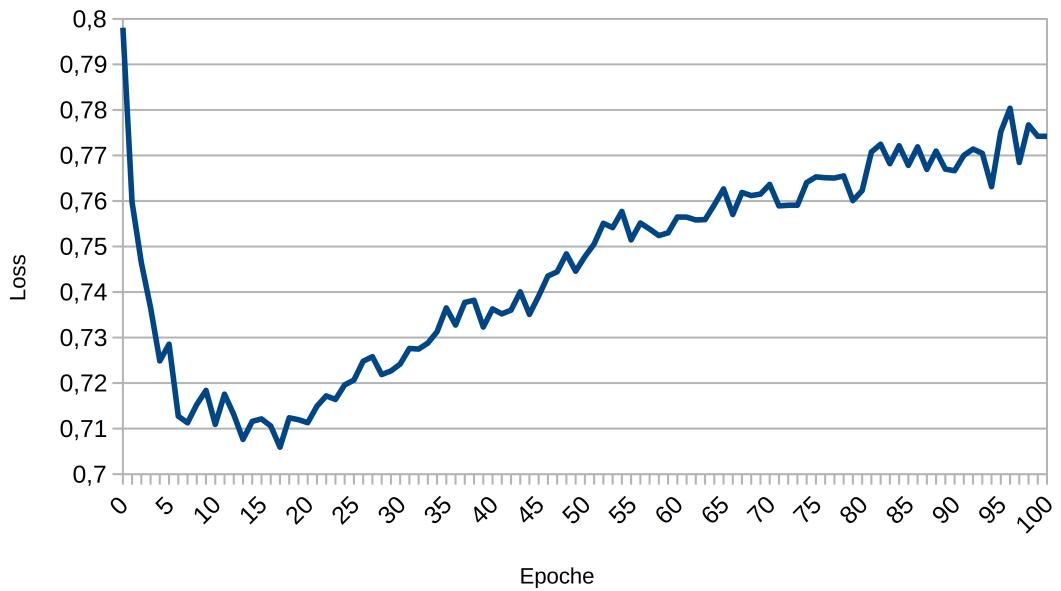


Abbildung 1: Nach der 20. Epoche, tritt in diesem mehrschichtigen Neuronalen Netz ein Overfitting auf [3, S.14].

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 Re-

gularisierung, reduzieren in der Regel ein Overfitting [1, S.29][3, S.14].

3.1.3. 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 in keiner Weise rauschfrei, wurde gezeigt, dass Verfahren zum Entfernen von Rauschen eine positiven Effekt haben können [4]. 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 zu einer 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 sinnvoll den relevanten Teil eines Bildes zu extrahieren. Dazu wurde im Praxisteil ein Machine-Learning Framework names MediaPipe genutzt [5].

Zuletzt müssen Bilder in der Regel vor der Nutzung skaliert werden. Ein Modell des Maschinellen Lernens erwartet in der Regel eine feste Eingabegröße. Dies gilt auch für Bilder, sodass als Beispiel das VGG19 Konvolutionelle Neuronale Netzwerk für Objekterkennung eine Eingabegröße von 224 x 224 Pixeln erwartet [6, S.2]. Auch mit Blick auf die Laufzeit ergibt eine Skalierung von Bildern Sinn. Moderne Webcams erreichen schon häufig 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 [6, S.2].

3.1.4. Qualitätsmaße

Um die Leistung eines Modells korrekt bewerten zu können, müssen abhängig vom Ziel Qualitätsmaße eingeführt werden, die es möglich machen Modelle 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 Strings 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 ähnlicher zueinander sind. Dies ist nicht für alle Problemstellungen ideal [3, S.12][1, 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 $\{a, b, c\}$ ergebe sich daher die Menge von Vektoren $\{(1 \ 0 \ 0), (0 \ 1 \ 0), (0 \ 0 \ 1)\}$. Korrekt klassifiziert wurde nun, wenn der vorliegende Klasse mit der höchsten Vorhersagen markiert wird. Da in diesem Fall keine Beziehung zwischen den Klassen erkannt wird, ist dieses Darstellung auch ideal für die Problemstellung [3, S.12][1, S.63].

Die bereits vorher erwähnte Kreuzentropie ist ein ideales Qualitätsmaß für Klassifikationsprobleme in One-Hot-Kodierung. Niedrige Vorhersagen bei der vorliegenden Klassen werden dabei abgestraft, während andere Vorhersagen ignoriert werden. Für einen Datensatz mit m Tupeln, seinen One-Hot-kodierten Labels y

mit K -Klassen und den One-Hot-kodierten Vorhersagen \mathbf{y}' mit K -Klassen gilt [3, S.13][1, 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 sich die Korrektheit einer Vorhersage bestimmen lassen 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 welche ein Beispiel in Abbildung 2 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 vertauscht 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 [1, S.86-87].

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

Abbildung 2: 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 [1, S.87]. Diese sind beispielsweise abhängig zu Klasse A in der Abbildung 3 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 3: 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, also ist eine Klasse 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 [1, 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 Klasse geeignet, da ein Fokus auf die korrekte Klassifikation dieser gesetzt wird [1, 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 [1, 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.

3.1.5. 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 typischen Verfahren etabliert, um damit umzugehen.

Zum einen kann ein Trainingsdatensatz in drei Teile, einem 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 eine 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 unbekannten 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 [1, S.30-31][3, 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 Teil 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 [2, S.22][3, S.44].

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

Abbildung 4: Beispiel für eine 5-fache Kreuzvalidierung [3, S.44].

3.2. Neuronale Netze

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

3.2.1. 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 [3, S.24][2, S.43-45][7, S.29-30].

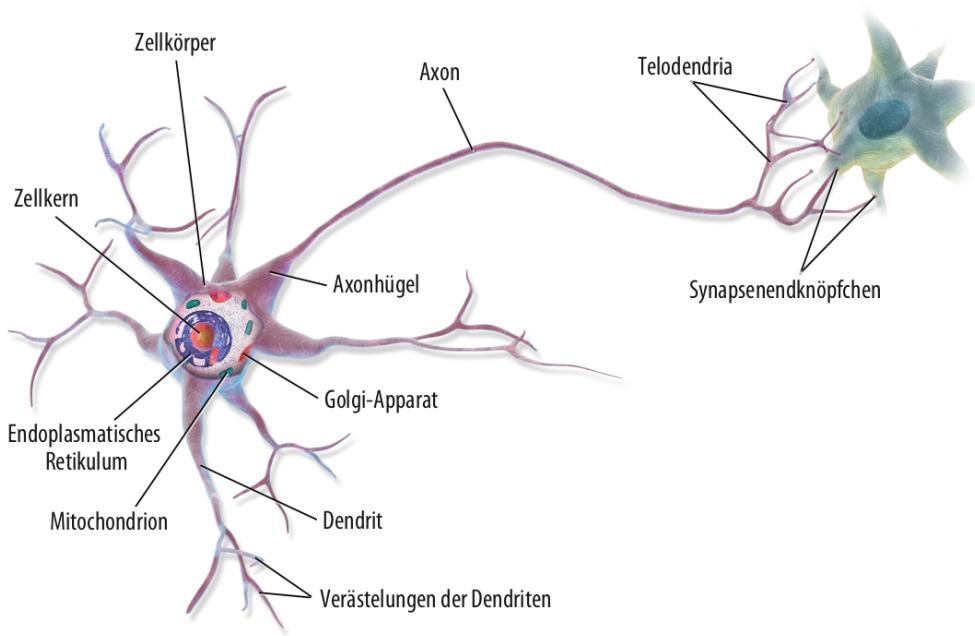


Abbildung 5: Aufbau eines biologischen Neurons [1, S.255].

3.2.2. 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 im Folgenden als ein Vektor \mathbf{x} mit n -Elementen dargestellt wird. Jede 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 eingegangen 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 geleitet werden, ähnlich zum ausgehenden Axon der biologischen Variante [3, S.25][7, S.30-31].

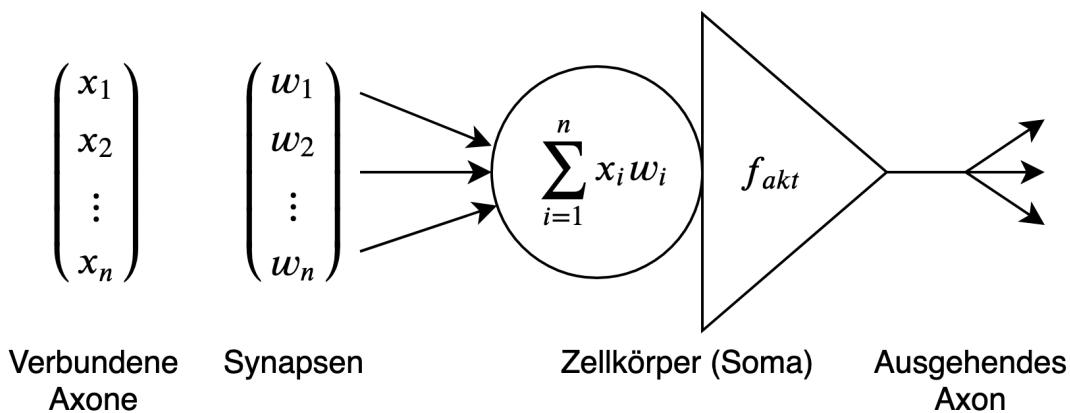


Abbildung 6: Aufbau eines künstlichen Neurons, angelehnt an das biologische Vorbild [3, S.25].

Für n -Eingänge lässt sich ein Künstliches Neuron nun folgendermaßen definieren [3, S.25][7, 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 [3, S.26][2, S.49].

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

3.2.3. Einschichtige Neuronale Netze

Liegen mehrere Künstliche Neuronen oder Perzeptrons in einer einzigen Schicht vor, entsteht ein erstes einschichtiges Neuronales Netz. Dabei wird jeder Eingang in das Neuronale Netz mit jedem Neuron in der einen Schicht des Netz 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 [3, S.26][1, S.258]. Ein einschichtiges Neuronale Netz mit einer vereinfachten Darstellung der Neuronen ist in der Abbildung 7 dargestellt.

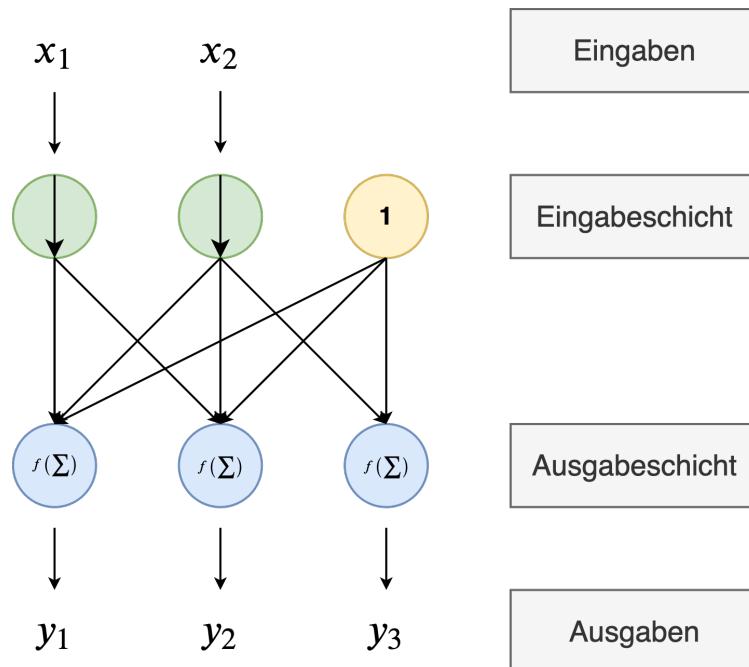


Abbildung 7: Aufbau eines einschichtigen Neuronalen Netzes [3, S.27][1, S.258].

Die Gewichtungen zwischen der Eingabeschicht und der Ausgabeschicht werden zur einfachen Berechnung typischerweise als Matrix \mathbf{W} dargestellt (8) [3, S.26][1,

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 [1, S.259].

3.2.4. 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) [3, 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 dem Fehler beschrieben ist, kann man sich die Arbeitsweise dieses Algorithmus leicht 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 8 erkennen [3, S.28][8, S.39].

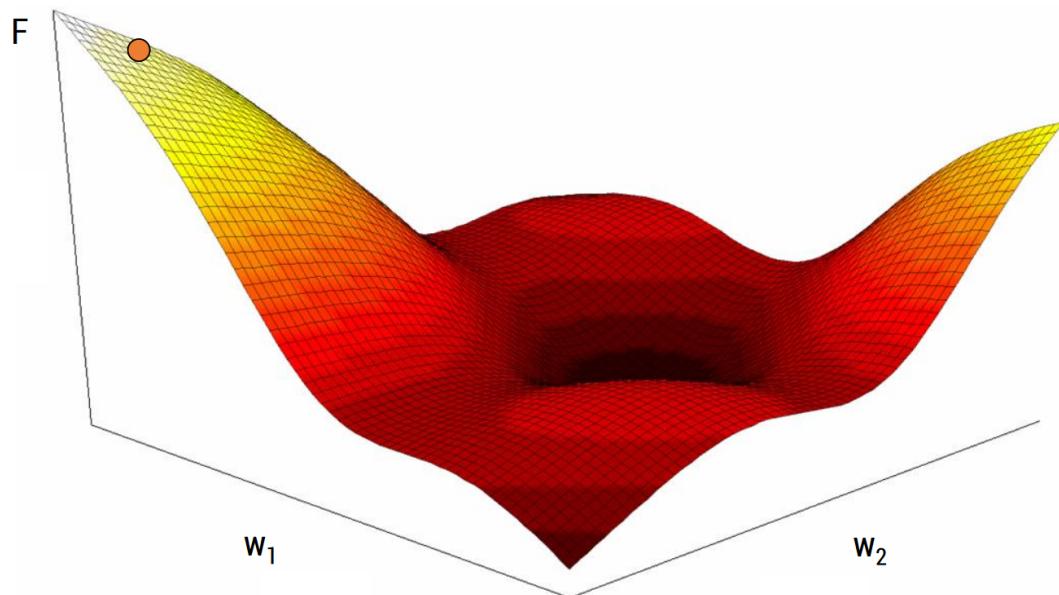


Abbildung 8: Gebirgslandschaft des Fehlers in einem Neuronalen Netz [9].

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 [3, S.29-30][8, S.41][2, S.31]. Häufig wird jede Epoche aber auch noch in kleinere Teile, den sogenannten Batches unterteilt.

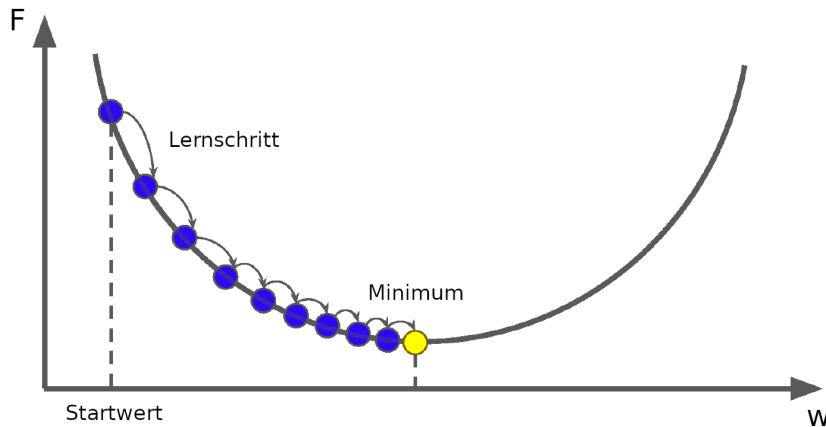


Abbildung 9: Das Gradientenabstiegsverfahren in zwei Dimensionen [1, S.113].

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 Wander durch die mehrdimensionale 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 ganz zum Überspringen des globalen Minimum führen [3, S.30][8, 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 [3, S.31-32][1, S.262].

3.2.5. 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 [3, S.31-32][1, S.261]. Ein Beispiel hierfür ist in der folgenden Abbildung zu erkennen:

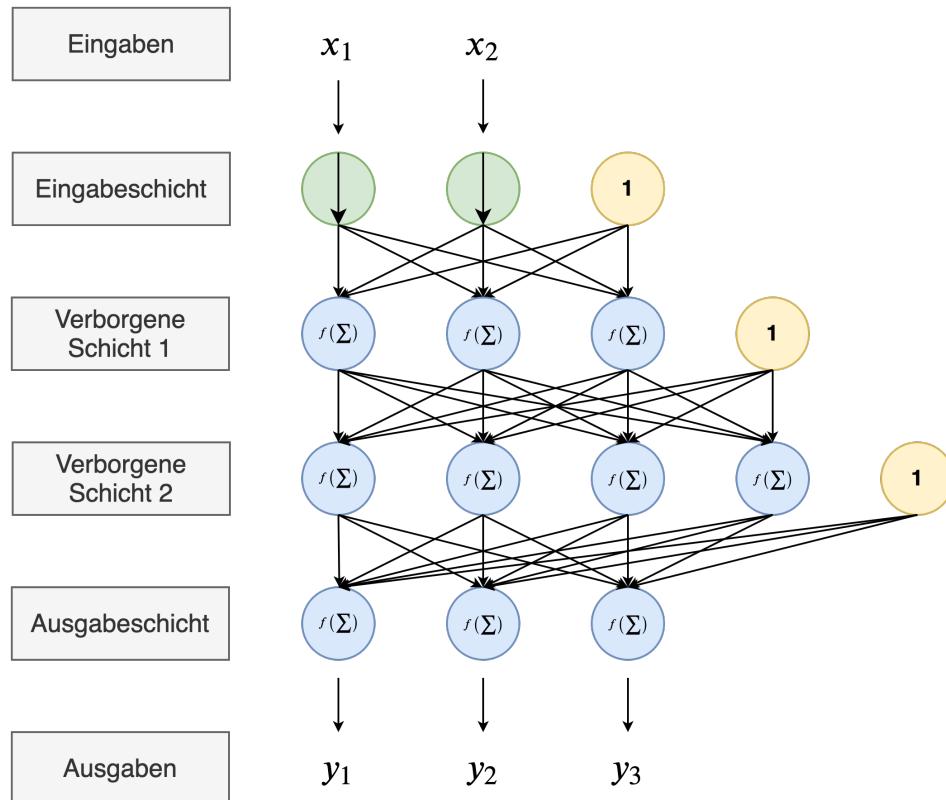


Abbildung 10: Ein mehrschichtiges Neuronales Netz mit zwei verborgenen Schichten [3, S.32][1, S.261].

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 [1, S.261].

3.2.6. 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 [8, 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 wurden [3, S.33][8, S.52-53]. Wurden die Gradienten errechnet, können die Gewichtungen mit einem vorher definierten Lernparameter angepasst werden [1, 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 [8, 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, die explodierenden Gradienten auftreten [3, S.33-34][1, S.275-276]. In der Regel lässt sich dies über sinnvollere Aktivierungsfunktionen und spezielle Regularisierungsverfahren lösen, die in den nächsten Kapiteln betrachtet werden.

3.2.7. 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, daher eine sehr kleine Ableitung, für positive Werte erfährt. Dies wirkt dem Problem der schwindenden Gradienten entgegen [3, S.34-35]. Für die Eingabe x ist die ReLU definiert als [1, S.245, S.279]:

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

Wie zu sehen 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 mich mehr zum Lernen bei. Tritt dies bei vielen Neuronen auf, kann dies ebenfalls den Lerneffekt beschränken [1, 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 wie eine negative Exponentialfunktion verhält die sich einem Parameter a annähert. Es gilt [3, 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 einen 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 [3, S.37][1, S.141-142, S.263]:

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

3.2.8. Regularisierungsverfahren

Zuletzt werden nun zwei typische Regularisierungsverfahren für Neuronale Netze betrachtet, mit denen die Performanz dieser auf bestimmte Weisen erhöht werden kann.

Die Batch-Normalisierung ist ein ideales Verfahren, um schwindenden Gradienten sowie Overfitting entgegen zu wirken 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. Schwindenden Gradienten werden dadurch so stark bekämpft, dass sogar der Einsatz der Sigmoidfunktion oder des Tangens hyperbolicus wieder denkbar wäre [3, S.37-38][1, 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 [3, S.38-39][7, S.205-206].

3.3. 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 eingesetzt und sind am Aufbau des visuellen Cortex des Menschen angelehnt. In diesem Arbeiten Neuronen in verschiedenen Schichten von Filtern, die aufeinander aufbauen. Die oberen Filter erkennen viele einfache Bildelemente, wie unterschiedliche Ausrichtungen von Linien. Diese werden weiter an untere Schichten geleitet, die das gefilterte immer weiter zu komplexen Mustern und Bildern kombinieren können [1, S.360].

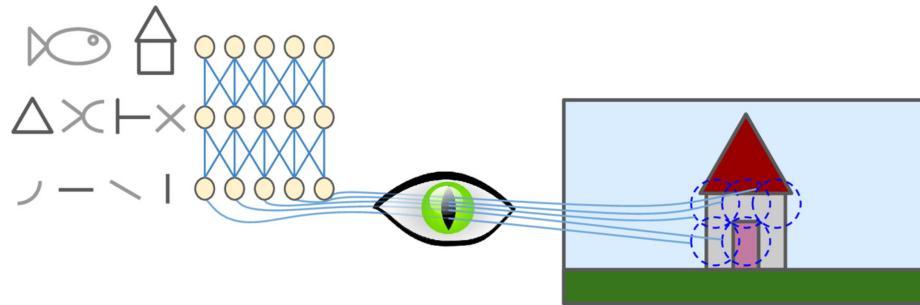


Abbildung 11: Aufbau des visuellen Cortex eines Menschen [1, S.360].

Diese Erkenntnisse wurden dann genutzt, um 1980 die ersten Konvolutionelle Neuronale 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 Neuronale Netz Schichten. Am Anfang befinden sich allerdings im Wechsel Convolutional und Pooling Layer, welche im folgenden betrachtet werden [1, S.361].

3.3.1. 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 einen Convolutional Layer betrachtetes Bild wandern. Der Filter multipliziert dabei in jedem durch die Schrittweite betrachtete 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 Rands mit Nullen, denkbar. In der Abbildung 12 ist ein Convolutional Layer mit einer Filtergröße von 3x3 Pixeln, einer Schrittweite von 2 und ein Zero-Padding erkennbar. Bei dieser Schrittweise ist klar zu erkennen, dass das Bild durch diese verkleinert wird. Dies ist nützlich, da hierdurch Bildauflösungen verringert werden können [1, S.361-363].

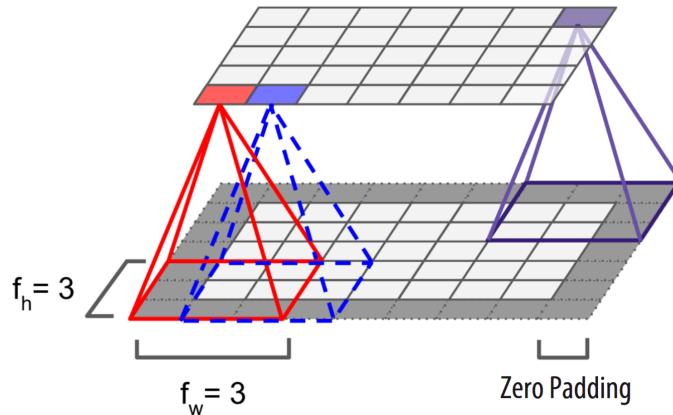


Abbildung 12: Ein einzelner Filter eines Convolutional Layers [1, S.362].

Die Arbeitsweise eines Filters ist sehr einfach zu visualisieren. Ein Filter, welcher in einer einzelnen Linie in einer vertikalen Ausrichtung hohe Gewichtungen aufweist, würde in einem Bild vertikale Linie hervorheben und horizontale Linie verschwimmen lassen. Dem gegenüber würde ein horizontaler Linienfilter, horizontale Linien hervorheben und vertikale verschwimmen lassen [1, S.363-364]. Ein Beispiel hierfür ist in folgender Abbildung zu erkennen:

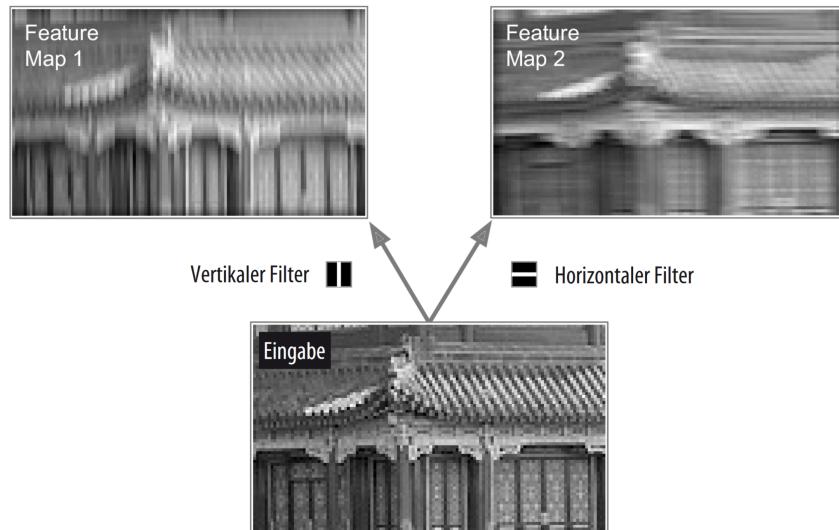


Abbildung 13: Ein Beispiel für horizontale und vertikale Filter [1, S.364].

In der Abbildung 14 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 [10, S.132].



Abbildung 14: Ein Beispiel für verschiedene visualisierte Filter im ersten Convolutional Layers eines CNNs [10, 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 mehrere Features von Grauwertbildern (ein Grauwert pro Bildpunkt) sehr gut extrahieren, jedoch gehen so viele Bildinformationen der Farbe verloren, deshalb sind tatsächliche Filter in modernen CNNs eher dreidimensionale Matrizen. Die Höhe und Breite dieser ist weiterhin die selbe 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 angewandt werden kann [1, S.364-365]. Dies ist in folgender Abbildung gut zu erkennen:

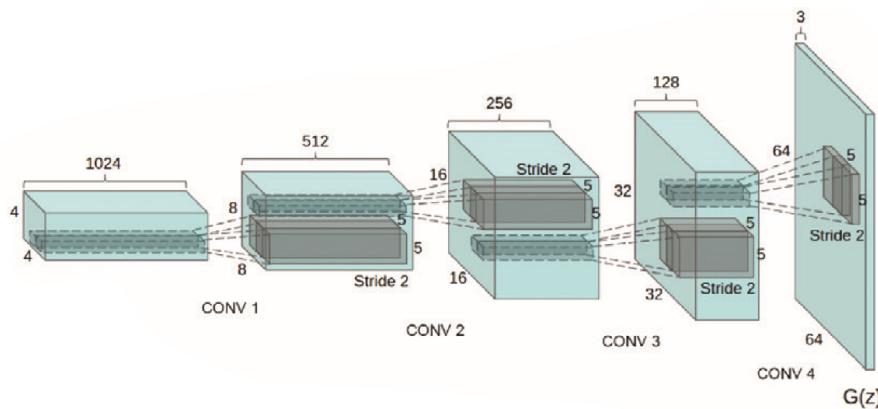


Abbildung 15: Ein Beispiel eines CNNs mit steigenden Feature Maps pro Layer [10, S.138].

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

3.3.2. Pooling Layer

Pooling Layer werden genutzt um die Größe von Convolutional Layern zu verringern, indem sie zwischen diesem 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 (Stride) haben, mit denen die Verringerung der Größe einer Feature Map erhöht werden kann [11, S.18-19][1, S.369-370]. In der Folgenden Abbildung wird die Arbeitsweise eines 2×2 Max Pooling mit einem Stride von 2 dargestellt:

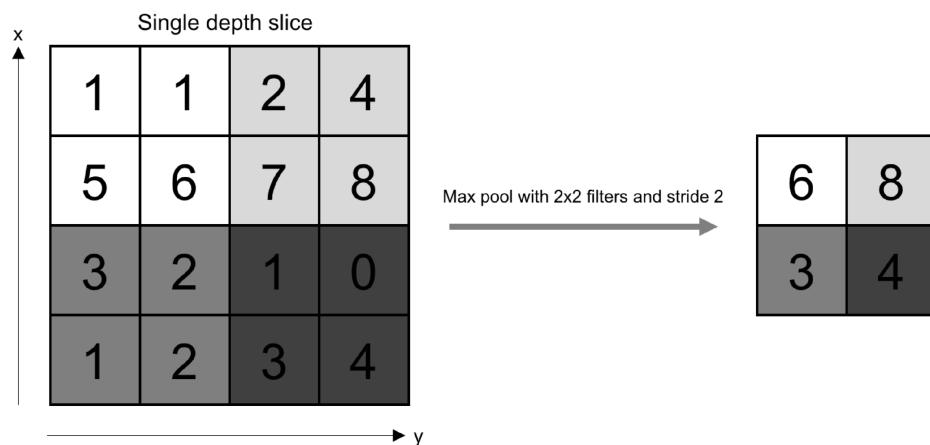


Abbildung 16: Die Arbeitsweise des Max Pooling mit einem 2×2 Kernel und Stride von 2 [11, S.19].

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

4. Datensätze

4.1. Sign Language MNIST Dataset

4.2. ASL Alphabet Dataset

4.3. Videodatensätze

5. Entwicklungsprozess

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

5.1. Datenvorverarbeitung

Da die beiden hauptsächlich verwendeten Datensätze in einer für ihre Auflösung ausreichende 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 [12]. 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 passenden Ordnerstrukturen umgewandelt werden, bei denen alle Bilder einer Klasse einen Ordner mit passender Bezeichnung nutzen. Außerdem ist eine Teilung in Test, sowie Trainings- und Validierungsdaten nötig, denn `image_dataset_from_directory` unterstützt lediglich eine Zweiteilung.

5.1.1. 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 entschieden die Werte zu extrahieren und als einzelne PNGs in die erforderlichen Ordnerstrukturen abzuspeichern. Der Kern der Skriptdatei zur Vorverarbeitung des Sign Language MNIST Dataset ist dabei folgende Funktion:

Listing 1: Kern der Sign Language MNIST Dataset Vorverarbeitung.

```

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")

```

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 in Integer der Daten 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 für Grauwertbilder entschieden, da so ein problemloses Verwenden von `image_dataset_from_directory` möglich wird.

5.1.2. 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 diese 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 umkopiert.

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

Listing 2: Kern der Sign Language MNIST Dataset Vorverarbeitung.

```

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)

```

5.1.3. 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 selber 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.

Listing 3: Beispielhafter Aufruf der Funktion zum Einlesen von Daten

```
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 )
```

Im Listing 3 wird ein Aufruf der Funktion `image_dataset_from_directory` be-

schrieben. 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`: Werden die Bilder nach dem Einlesen zufällig angeordnet.
- `validation_split`: Anteil der Validierungsdaten.
- `subset`: Das zurückgegebene Subset für dieses Verzeichnis. Dieses ist entweder `training` oder `validation`.
- `interpolation`: Genutztes Verfahren für die Skalierung.

Da die Datensätze für ihre Auflösung alle eine 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.

5.2. Entwickelte Modelle

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

5.2.1. 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 welchen ein Großteil der Datenvorverarbeitung schon erledigt werden kann. Danach folgen in der Regel 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 80-20 Validation Split unter einer in Kapitel 3.1.5. beschriebenen Validierung in drei Stufen verwendet. Am Ende der Vorbereitung wird in der Regel 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:

Listing 4: Beispielhafte Vorbereitung für das Training

```

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)
```

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 dieses 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 den Adam-Optimierer genutzt.

Listing 5: Kompilierung eines Modells

```

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'])

```

Dann wurde eine Funktion für das Loggen entwickelt. Diese kopiert das momentan aktive Skript in einen eigenen Ordner eines 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 dem 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 geringssten Fehler abspeichern.

Listing 6: Logging Funktion für Trainingsvorgänge

```

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

```

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

Listing 7: Start des Trainings mit Keras

```

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

```

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 auf den Testdaten zu bestimmen. Zum Schluss wird mittels Scikit-Learn in der eigens definierte evaluate Funktion eine Konfusionsmatrix und Auswertungstabelle mit Precision, Recall und F1-Score zu den Testdaten erstellt. Dies ist im folgenden Listing erkennbar:

Listing 8: Auswertung eines Modells mit Testdaten

```

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))

```

Abschließend sei erwähnt, dass die Auswertungsergebnisse der Testdaten nur in Betracht gezogen wurden, wenn bereits einige andere Modelle mit Validierungsdaten getestet wurden. Ein anderes Auswertungsverhalten würde die in Kapitel 3.1.5. genannten Grundsätze verletzen.

5.2.2. Neuronale Netze

5.2.3. Konvolutionelle Neuronale Netze

5.2.4. Transferlearning mittels VGG19

5.3. Backend der Webanwendung

5.4. Frontend der Webanwendung

5.5. Ergebnisse der Entwicklung

6. Fazit

7. Ausblick

8. Quellen

- [1] Aurelien Geron. *Praxiseinstieg: Machine Learning mit Scikit-Learn & TensorFlow*. Heidelberg: O'Reilly, 2018. isbn: 978-3-96009-061-8.
- [2] Josh Patterson und Adam Gibson. *Deep Learning: A Practitioner's Approach*. Sebastopol: O'Reilly, 2017. isbn: 978-1-491-91425-0.
- [3] Sebastian Schmidt. "Automatische Analyse von Stimmmerkmalen zur Vorhersage von Persönlichkeitsprofilen mittels Künstlicher Intelligenz". Iserlohn: Fachhochschule Südwestfalen, 25. Okt. 2019.
- [4] 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).
- [5] Google LLC. *Cross-platform ML solutions made simple*. url: <https://google.github.io/mediapipe/> (besucht am 05.07.2020).
- [6] 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).
- [7] Roland Schwaiger und Joachim Steinwendner. *Neuronale Netze programmieren mit Python*. Bonn: Rheinwerk, 2019. isbn: 978-3-8362-6142-5.
- [8] 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.
- [9] 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).
- [10] Terrence J. Sejnowski. *The Deep Learning Revolution*. MIT: The MIT Press, 2018. isbn: 978-0-262-03803-4.
- [11] Rajalingappa Shanmugamani. *Deep Learning for Computer Vision*. Birmingham: Packt, 2018. isbn: 978-1-78829-562-8.

- [12] keras. *Image data preprocessing*. url: <https://keras.io/api/preprocessing/image/>.