

Inhaltsverzeichnis

Abkürzungsverzeichnis	iv
Abbildungsverzeichnis	v
Tabellenverzeichnis	vii
Listingverzeichnis	vii
1 Einleitung	1
1.1 Begriffe in der Computervision	2
1.2 Lernende Algorithmen für die Computervision	3
1.3 Ziele, Aufbau und Methodik der Arbeit	5
2 Theoretische Grundlagen neuronaler Netze	7
2.1 Begriffsabgrenzungen im Deep-Learning	7
2.2 Neuronale Netze	10
2.2.1 <i>Single layer</i> -Perzeptron	11
2.2.2 <i>Multi layer</i> -Perzeptron	20
2.3 <i>convolutional</i> - und <i>pooling</i> -Schichten	28
2.3.1 Die <i>convolutional</i> -Schicht	28
2.3.2 Die <i>pooling</i> -Schicht	32
2.4 Weiteres zu neuronalen Netzen	33
2.4.1 <i>Generalization, overfitting</i> und <i>underfitting</i>	33
2.4.2 Die Varianten des <i>gradient descent</i> -Algorithmus	35
2.4.3 Adaptive Lernratenanpassung	36
3 Bildung der Datengrundlage	37
3.1 Datengewinnung für den Anwendungsfall	37
3.1.1 Geeignetes Bildmaterial finden	38
3.1.2 Bildmaterial vorverarbeiten mit YOLO	40
3.1.3 Abwandlung von YOLO	42
3.1.4 Bildinformationen extrahieren	44
3.1.5 Aufteilung in Trainings- und Testdaten	46
3.1.6 Konnotieren der Daten	48

4 Einführung in das Keras-Framework	51
4.1 Vorstellung des Deep-Learning-Frameworks Keras	51
4.2 Funktionen und Parameter in Keras	51
5 Aufbau des CNNs	55
5.1 Ziel des CNN	55
5.2 Umsetzung eines CNN	56
5.2.1 Praktische Implementierung des CNN	56
5.2.2 Beispielhafter Durchlauf durch das neuronale Netz	60
6 Hypothesengenerierung und -überprüfung	63
6.1 Generierung und Überprüfung der Hypothesen	63
6.2 Die Hypothesen	64
6.2.1 Hypothese zur Menge der verwendeten Trainingsdaten . .	64
6.2.2 Hypothese zur Anzahl der verwendeten Epochen in der Traingsphase	67
6.2.3 Hypothese zur Größe der verwendeten <i>batch size</i> pro Epoche in der Trainingsphase	69
6.2.4 Hypothese zur verwendeten Aktivierungsfunktion	71
6.2.5 Hypothese zur verwendeten Optimierungsfunktion	73
6.2.6 Hypothese zum <i>Shuffle</i> der Trainings- und Testdaten	75
6.3 Gesamtes Ergebnis der Hypothesen	76
7 Ausblick und Fazit	79
7.1 Ausblick	79
7.2 Fazit	82

Abkürzungsverzeichnis

- AI** artificial intelligence
- API** application programming interface
- ANN** artificial neural network
- CNN** convolutional neural network
- CNTK** Microsoft cognitive toolkit
- GAN** generative adversial network
- MLP** multi layer perceptron
- MSE** mean squared error
- NN** neural network
- RNN** recurrent neural network
- SGD** stochastic gradient descent

Abbildungsverzeichnis

Abb. 1: CNN für die Echtzeit-Objektverfolgung	1
Abb. 2: CNN-Bildsegmentation in der Medizin	1
Abb. 3: Auszug aus dem MNIST-Datensatz	4
Abb. 4: Übersicht von Algorithmen im <i>machine learning</i>	8
Abb. 5: <i>Feature extraction</i> durch <i>hidden layers</i> bei Bildern	9
Abb. 6: Gegenüberstellung vom biologischen und mathematischen Neuron	10
Abb. 7: Separation eines Vektorraums durch eine Hyperebene	12
Abb. 8: Darstellung des Gradientenabstiegsverfahrens	14
Abb. 9: Darstellung zur linearen Separierbarkeit	19
Abb. 10: Illustration eines mehrschichtigen neuronalen Netzes	21
Abb. 11: Darstellung eines <i>convolutional</i> -Kernels	30
Abb. 12: Dreidimensionale <i>convolutional</i> -Schicht	31
Abb. 13: Darstellung eines <i>pooling</i> -Prozesses	33
Abb. 14: Darstellung von <i>under-</i> und <i>overfitting</i>	34
Abb. 15: Daten- <i>Pipeline</i> für das CNN	37
Abb. 16: Vergleich Echtaufnahme mit Fifa 17	38
Abb. 17: Screenshot aus Fifa 17	40
Abb. 18: Fußballspielererkennung mit YOLO	41
Abb. 19: Darstellung eines ausgeschnittenen Spielers	42
Abb. 20: Darstellung $48px \times 48px$ Spielerbild	45
Abb. 21: Illustration eines RGB-Bildes	45
Abb. 22: Aufteilung in Trainings- und Testdaten	47
Abb. 23: Darstellung der <i>cross-validation</i> -Methode	48
Abb. 24: Gegenüberstellung TV-Bild(l.) und Simulation(r.)	49
Abb. 25: Bildliche Darstellung der aufgabenbezogenen Objektklassen	50
Abb. 26: Ziel des Neuronalen Netzes	55
Abb. 27: Vergleich Trainingsdatenmenge	65
Abb. 28: Vergleich der Epochengröße	68
Abb. 29: Vergleich der <i>batch size</i>	70
Abb. 30: Fehlermaß über Epochen des opt. CNNs	78
Abb. 31: Darstellung einer möglichen Weiterentwicklung	80
Abb. 32: Darstellung einer dynamischen Taktiktafel	80

Tabellenverzeichnis

Tab. 1 Vergleich der Aktivierungsfunktionen	72
Tab. 2 Vergleich von Optimierungsfunktionen	74
Tab. 3 Darstellung des Vergleichs der <i>shuffle</i> -Funktion	75

Listingverzeichnis

Listing 1: Methode boxtocropp	43
Listing 2: Ablauf der Klasse	44
Listing 3: Methode Load Image	46
Listing 4: Beispielhafte RGB-Werte	46
Listing 5: Deklarationen	56
Listing 6: Schichten des CNNs	57
Listing 7: compile-Methode	58
Listing 8: Fit-Funktion des CNNs	59
Listing 9: Trainingsdaten Auswahl und Vorverarbeitung	60
Listing 10: Array Transformierung	61
Listing 11: Datenformat von x	61
Listing 12: Datenformat von y	61
Listing 13: Ausgabe der Traingsphase	62
Listing 14: Initialisierung des CNN über eine .h5-Datei	62
Listing 15: Optimale Lösung des CNN	77

1 Einleitung

Deep-Learning-basierte Algorithmen haben im Feld der Computervision in den vergangenen Jahren enorme Erfolge erreicht. So ermöglichen Deep-Learning-Algorithmen heute etwa eine sehr akkurate Erkennung von Verkehrsschildern für selbstfahrende Fahrzeuge oder unterstützen Mediziner bei der Diagnose von Krankheiten auf MRT-Bildern, siehe Abbildungen 1 und 2.

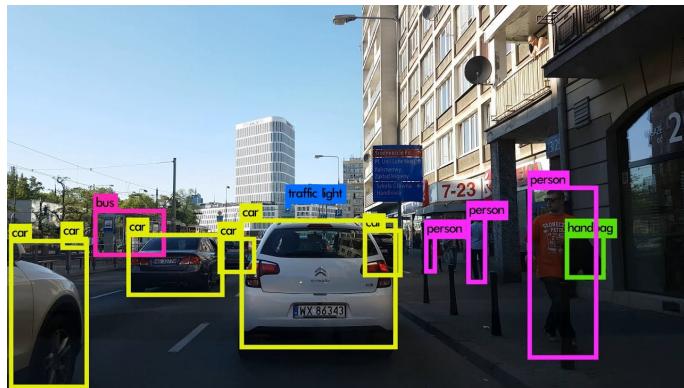


Abb. 1: Dargestellt ist die Echtzeit-Objektverfolgung von Verkehrsteilnehmern und Fußgängern mit dem YOLO-v3-Algorithmus.
Bildquelle: (Kathuria, Ayoosh, 2019).

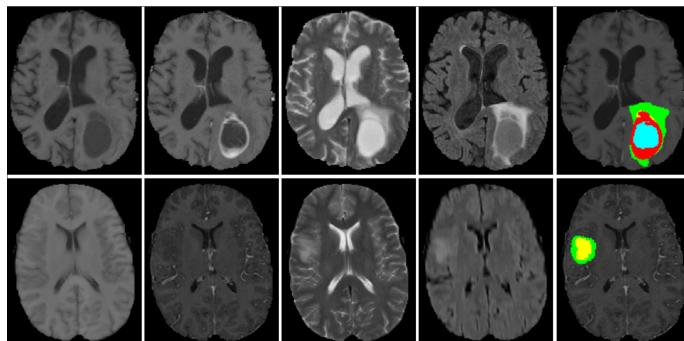


Abb. 2: Dargestellt ist die Bildsegmentation durch ein CNN von einem MRT-Scan zur Erkennung von Tumorzellen. Bildquelle: (Pereira, Sergio et. al., 2016).

Das vordergründige Ziel der vorliegenden Arbeit ist die Erkennung eines Fußballspielers in einem Bild durch einen Deep-Learning-basierten Algorithmus. Die Erkennung eines Fußballspielers in einem Bild kann in der Zukunft als Ausgangspunkt genutzt werden, um die Position eines Fußballspielers über mehrere Bilder

hinweg zu verfolgen, um Leistungsmaße wie Laufleistung oder Durchschnittsgeschwindigkeit zu ermitteln. Werden die Positionen aller Fußballspieler eines Fußballspiels verfolgt, ist zukünftig auch die Ermittlung von Taktikmaßen wie der *Packing-Quote* denkbar.

In diesem einleitenden Kapitel werden in Abschnitt 1.1 zunächst einige, in der Computervision häufig verwendete, Begriffe erklärt und abgegrenzt, um eine einheitliche Begriffsgrundlage zu schaffen. In Abschnitt 1.2 werden lernende Algorithmen, zu denen auch Deep-Learning-basierte Algorithmen zählen, intuitiv eingeführt und deren Verwendung in der vorliegenden Arbeit begründet. Dieses Kapitel abschließen wird eine präzise Definition der Ziele dieser Arbeit und der daraus folgende inhaltliche Aufbau mitsamt methodischer Vorgehensweise.

1.1 Begriffe in der Computervision

Wie im vorherigen Abschnitt erwähnt, ist das Ziel dieser Arbeit die Erkennung eines Fußballspielers in einem Bild. Eine solches Unterfangen fällt in den Aufgabenbereich der Computervision. Diese hat zum Ziel, Algorithmen zu entwickeln, die Aufgaben erfüllen können, die das menschliche visuelle System erfüllen kann (Huang, T. S., 1996) - bspw. die Fähigkeit durch visuelle Eingaben unterschiedliche Objekte voneinander zu unterscheiden. Im Folgenden sind einige populäre *High-Level*-Aufgabentypen der Computervision, die für die Arbeit relevant sind, erklärt:

Bildklassifikation Die Bildklassifikation ist der Vorgang, mit dem Instanzen von Objekten in Bildern identifiziert werden. (Mathworks.com, 2017) Typischerweise ist die Anzahl der zu identifizierenden Objekte auf einem Bild begrenzt, sodass in einem Bild häufig nur ein Objekt zu identifizieren ist. Das Objekt, das auf dem Bild identifiziert werden kann, kommt aus einer endlichen Menge an möglichen Objektausprägungen bzw. -klassen.

Bildsegmentation Die Bildsegmentation ist der Vorgang, bei dem hinreichend kleine Bildinformationen, häufig einzelne Pixel, als eine Objektklasse identifiziert werden. Im Beispiel aus Abbildung 2 wird jedes Pixel als eine der Objektklassen $\{Knochen, Gewebe, Wasser, Tumor, \dots\}$ identifiziert.

Objekterkennung Zu der Identifizierung und Zuweisung einer Objektklasse bei der Bildklassifikation kommt bei der Objekterkennung noch die Lokalisierung des Objektes im Bild hinzu. (Mathworks.com, 2017) Dies ermöglicht auch, dass mehrere Objekte im Bild identifiziert werden können. Die Position der Objekte im Bild wird dann häufig durch *bounding boxes*, siehe Abbildung 1, visualisiert.

Objektverfolgung Objekte in einer Abfolge von Bildern übergreifend zu verfolgen (engl. object tracking), ist Aufgabe der Objektverfolgung. (Mallick, Satya, 2017) Diese erweitert die Objekterkennung also auf Bildabfolgen, wobei sie zusätzliche Techniken nutzen muss. Beispiele für solche Objektverfolgungsalgorithmen sind „YOLO“ aus Abbildung 1, der Kalman-Filter oder der Mean- bzw. Camshift-Algorithmus. (Mallick, Satya, 2017)

Um das Ziel der Erkennung eines Fußballspielers in einem Bild zu erreichen, soll eine Bildklassifikation unternommen werden. Konkret muss der eingesetzte Bildklassifikationsalgorithmus einem Bild eine der zwei Objektklassen $\{player, noPlayer\}$ zuweisen können, d.h. auf dem Bild einen oder keinen Fußballspieler identifizieren können.

1.2 Lernende Algorithmen für die Computervision

Überlegt man sich nun, wie ein solcher Bildklassifikationsalgorithmus aussehen könnte, wird man feststellen, dass die Aufgabe, von Bildpixeln auf eine semantische Bedeutung dieser zu schließen, kein triviales Unterfangen ist. (Li, FeiFei et. al., 2018b)

Beispielsweise würde von einem Klassifikationsalgorithmus von handschriftlichen Ziffern gefordert werden, dass dieser eine gegebene Ziffer eine Objektklasse aus $\{0, 1, 2, \dots, 9\}$ zuweisen kann. Tritt man ohne jegliches Vorwissen an *Low-Level*- oder *High-Level*-Bildalgorithmen an diese Aufgabe heran, könnte man zunächst auf die Idee kommen, die Charakteristiken der einzelnen Ziffern direkt im Algorithmus zu beschreiben. Beispielsweise könnte die Ziffer „8“ durch zwei geschlossene Kreise definiert werden, die Ziffer „0“ durch einen einzelnen geschlossenen Kreis. Aber weder die Ziffer „8“, noch die Ziffer „0“ müssen perfekt geschlossene Kreise

haben, um als solche gemeint zu sein, da Handschriften höchst unterschiedlich sind. Deshalb müsste ein solcher Algorithmus dann auch die Distanz zwischen den offenen Enden eines Kreises erkennen und messen, etwa um die Ziffer „0“ nicht mit einer „6“ zu verwechseln.

Es stellt sich heraus, dass diese Herangehensweise, aufgrund der Vielzahl an verschiedenartigen Handschriften und der daraus folgenden Erforderlichkeit vieler, im Algorithmus definierter, Ziffercharakteristiken, enorm aufwendig ist. (Buduma & Lacascio, 2017, S. 3)

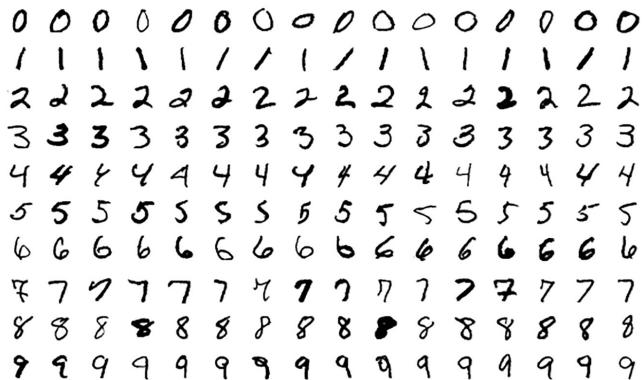


Abb. 3: Auszug aus dem MNIST-Datensatz, das aus handschriftlichen Ziffern besteht.

Erkennbar ist, dass etwa die Ziffer „7“ auf viele unterschiedliche Arten verschriftlicht wurde. Diese vielen handschriftlichen Nuancen als Ziffercharakteristiken in einem Algorithmus festzuhalten, um eine „7“ auch sicher zu erkennen, ist sehr aufwendig. Bildquelle: (Steppan, Josef, 2017).

Aus diesem Grund nutzt die Computervision als Herangehensweise für die Bildklassifikation häufig sogenannte lernende Algorithmen (engl. learning algorithms/models). (Buduma & Lacascio, 2017, S. 3 f.) In der vorliegenden Arbeit werden weiterhin ausschließlich sogenannte überwachte lernende Algorithmen (engl. supervised learning algorithms) beschrieben und genutzt.

Diese Modelle prägen sich bekannte, mit der Objektklasse konnotierte, Beispiele ein und versuchen das Gelernte dann auf unbekannte, nicht-konnotierte Beispiele zu transferieren. (Buduma & Lacascio, 2017, S. 3 f.) Ein solch lernender Algorithmus muss dann wenige bis keine (Eingabe ist dann das Bild selber) händisch vorverarbeitete Ziffercharakteristiken als Eingabe bekommen und es wird gehofft,

dass der Algorithmus durch die bekannten Beispiele eine Abbildung von Ziffercharakteristiken auf Ziffer erlernt. Aufgrund der Genauigkeit und zeitgenössischen Relevanz (Buduma & Lacascio, 2017, S. 7, 87 ff.) wird die Aufgabenstellung der Bildklassifikation von Fußballbildern in die Objektklassen $\{player, noPlayer\}$ in der vorliegenden Arbeit mit einem lernenden Algorithmus, einem Deep-Learning-Algorithmus, bewerkstelligt.

1.3 Ziele, Aufbau und Methodik der Arbeit

Die Ziele dieser Arbeit werden konkret wie folgt definiert:

- Theoretische Herleitung und praktische Entwicklung eines Deep-Learning-basierten Algorithmus zur Erkennung von Fußballspielern auf Bildern.
- Evaluation dieses Algorithmus mittels einer Hypothesengenerierung und -überprüfung zur Gewinnung von Erkenntnissen über die Hyperparameterkonfiguration.

Dazu werden in Kapitel 2 zunächst die theoretischen Grundlagen von Deep-Learning-basierten Algorithmen hergeleitet und erklärt. Danach wird in Kapitel 3 die Konstruktion der Daten-Pipeline beschrieben, die den Deep-Learning-Algorithmus mit Bildern versorgen wird. Als nächstes wird in Kapitel 4 auf das, für die Implementierung notwendige, Deep-Learning-Framework Keras eingegangen. Danach wird in Kapitel 5 die problemspezifische Implementierung des Deep-Learning-Algorithmus mit Keras dargestellt und erläutert. Eine Hypothesengenerierung und -überprüfung in Kapitel 6 dient dem Erkenntnisgewinn über das Verhalten der Hyperparameter des Deep-Learning-Algorithmus im vorliegenden Anwendungsfall und damit auch der systematischen Findung einer bestmöglichen Hyperparameterkonfiguration. Abschließend wird in Kapitel 7 ein Fazit über die gewonnenen Erkenntnisse hinsichtlich der hier definierten Ziele gezogen, sowie ein Ausblick auf Anwendungsmöglichkeiten des erstellten Algorithmus gegeben.

Im Vordergrund der vorliegenden Arbeit steht der Verständnisgewinn über die Funktionsweise und die praktische Implementierung eines Deep-Learning-Algorithmus

zur Bildklassifikation. Deshalb wird bei der Abfolge der oben beschriebenen Inhalte eine *Bottom-Up*-Methodik verfolgt. Dabei folgt auf die Theorie die praktische Umsetzung des Deep-Learning-Algorithmus. Aufgestellte Hypothesen zu den Hyperparametern des Algorithmus wollen wir in der Folge wissenschaftlich überprüfen. Für die theoretischen Ausführungen seien an dieser Stelle die folgenden Vereinbarungen getroffen:

Vektoren werden dick und klein geschrieben

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in V^n(\mathbb{K}).$$

Matrizen werden dick und groß geschrieben

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in M^{m \times n}(\mathbb{K}).$$

2 Theoretische Grundlagen neuronaler Netze

In der Einführung wurde definiert, was genau eine Bildklassifikation tut und es wurde begründet, warum in der vorliegenden Arbeit überwachte, lernende Algorithmen zur Bildklassifikation genutzt werden. Dieses Kapitel soll nun darüber Aufschluss geben, wie genau solche Algorithmen funktionieren.

Da es eine Vielzahl an unterschiedlichen lernenden Algorithmen gibt, wird in Kapitel 2.1 zunächst ein Überblick über die wichtigsten Begriffe und Algorithmen gegeben und es wird erklärt, warum ein Deep-Learning-Algorithmus für diese Arbeit ausgewählt wurde. In Kapitel 2.2 werden Deep-Learning-Algorithmen theoretisch beschrieben, wobei in Kapitel 2.2.1 zunächst Algorithmen, die einzelnen Neuronen nachempfunden sind, beschrieben. In Kapitel 2.2.2 wird dann zu den, im praktischen Teil der Arbeit verwendeten, neuronalen Netzen übergegangen. Um neuronale Netze für die Aufgaben der Computervision zu optimieren, wird das Roh-Bild zunächst durch sogenannte *convolutional layer* gereicht. Diese werden, aufgrund ihrer Relevanz für die Aufgabenstellung, separat in Kapitel 2.3 dargestellt und erklärt. Abschließend werden in Kapitel 2.4 kurz einige wichtige Implementierungskonzepte und -methodiken von neuronalen Netzen erklärt.

2.1 Begriffsabgrenzungen im Deep-Learning

Überwachte, lernende Algorithmen haben eine Ein- und Ausgabe. Während des sogenannten Trainings werden dem Algorithmus Beispiele von Eingabe-Ausgabe-Pärchen zugeführt. Der Algorithmus soll dann eine Abbildung modellieren, die Eingabe und Ausgabe in Beziehung zueinander setzt. (Brilliant.org, 2018a) Beispielsweise könnte so die Beziehung zwischen der Außentemperatur und Luftfeuchtigkeit (Eingaben) mit der Jahreszeit (Ausgabe) erlernt werden. Aufgabe der vorliegenden Arbeit ist es einen (Deep-Learning-)Algorithmus zu entwickeln, der die Beziehung zwischen einer Menge an $48px \times 48px$ -Fußballspielerbildern (Eingabe) mit $\{player, noPlayer\}$ (Ausgabe) erlernen kann.

Die Ausgabe kann auch numerisch skaliert sein, statt wie oben kategorial. Beispielsweise kann ein Algorithmus die Arm- und Beinlänge eines Kängurus (Ein-

gaben) in Beziehung zu der Sprungweite (Ausgabe) setzen. Bei einer kategorial skalierten Ausgabe wird von einer **Klassifikation** gesprochen, bei einer numerisch skalierten Ausgabe wird von einer **Regression** gesprochen. (Brilliant.org, 2018)

Neben überwachten lernenden Algorithmen gibt es auch unüberwachte lernende Algorithmen (engl. unsupervised learning algorithms/models). Unüberwachte Algorithmen versuchen Beziehungen zwischen verschiedenen Eingaben herzustellen. Eine Ausgabe wird somit nicht benötigt. Unüberwachte Algorithmen können durch sog. *cluster analysis algorithms* (dt. Clusteranalyse-Algorithmen) versteckte Muster in den Daten finden und somit insbesondere zur explorativen Datenanalyse beitragen. (Mathworks.com, 2018) Ein solcher Ansatz wird in der vorliegenden Arbeit nicht benötigt, dennoch spielen unüberwachte Algorithmen insbesondere für die Bildsegmentation eine wichtige Rolle. (Li, FeiFei, 2013)

Für überwachte und unüberwachte lernende Algorithmen gibt es eine Vielzahl an algorithmischen Varianten, die ihre Stärken, Schwächen und problembezogenen Anwendungsgebiete mitbringen. Eine Übersicht über diese Vielfalt ist in Abbildung 4 gegeben.

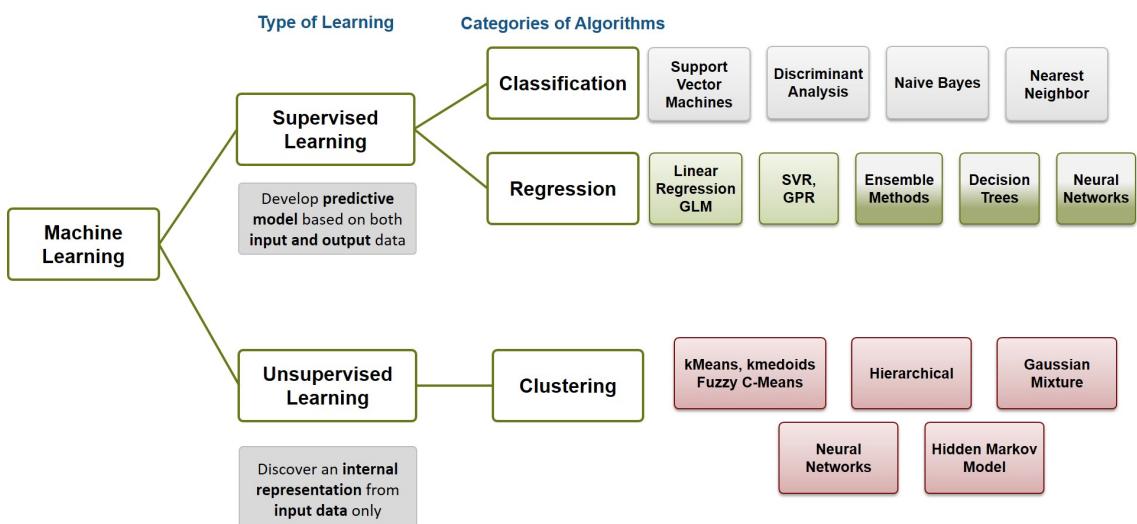


Abb. 4: Dargestellt sind verschiedene Algorithmen für das überwachte und unüberwachte Lernen. Ergänzend zur obigen Darstellung wird in der vorliegenden Arbeit ein neuronales Netz als Klassifikationsalgorithmus erklärt und implementiert.
Bildquelle: (Pilotte, Paul, 2016).

Überwachte und unüberwachte lernende Modelle zusammen bilden das Gebiet des maschinellen Lernens (engl. machine learning), das wiederum ein Teilgebiet der künstlichen Intelligenz (engl. artificial intelligence (AI)) ist. (Buduma & Lacascio, 2017, S. 4) Die vorliegende Arbeit wird als überwachte Algorithmus zur Bildklassifikation neuronale Netze, mit tiefem Netzaufbau, beschreiben, erklären und nutzen. Diese gehören zu einem Teilgebiet des maschinellen Lernens, das *deep learning* genannt wird. Der Vorteil dieser neuronalen Netze, im Vergleich zu anderen Algorithmen des maschinellen Lernens, liegt in ihrer hohen Abstraktionsfähigkeit durch die sogenannten *hidden layer* an Neuronen (Buduma & Lacascio, 2017, S. 29), dargestellt in Abbildung 5. Dies vereinfacht die in Kapitel 3.1 dargestellte Daten-Pipeline erheblich, da bei einem Deep-Learning-Algorithmus keine Bildcharakteristiken manuell definiert und zugeführt werden müssen.

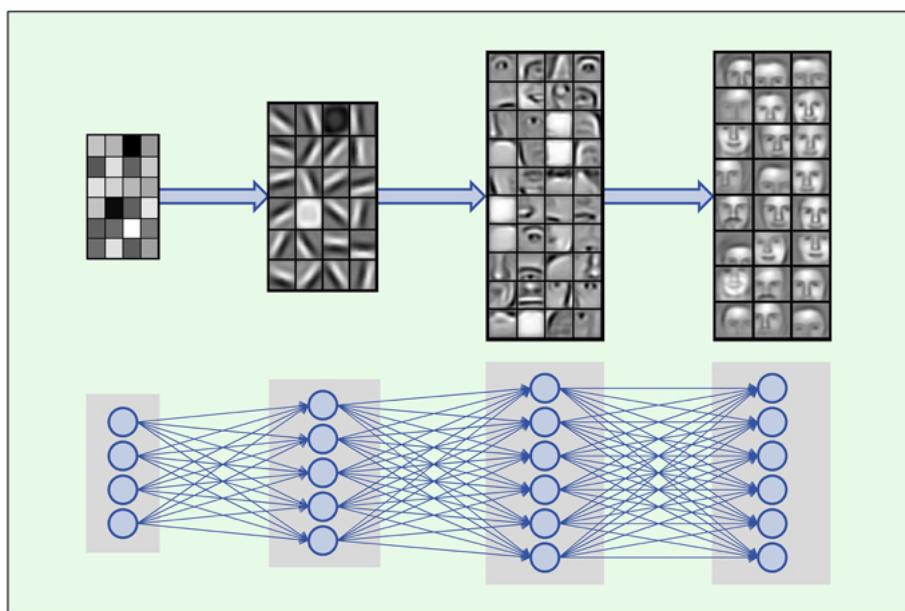


Abb. 5: Haben neuronale Netze *hidden layer* mit nicht-linearer Aktivierungsfunktion, dann sind sie in der Lage, Bildcharakteristiken, wie hier Gesichtsmerkmale, selbstständig aus dem Bild zu extrahieren. (Buduma & Lacascio, 2017, S. 12 f., 86 ff.) Bildquelle: (Liu, Wangxin, 2017).

In den folgenden Kapiteln wird die Inspiration, der Aufbau und die Funktionsweise von neuronalen Netzen erklärt und formalisiert, sodass in Kapitel 5 die praktische Implementierung eines solchen Netzes erfolgen kann.

2.2 Neuronale Netze

Künstliche neuronale Netze (engl. artificial neural network (ANN)) sind Berechnungsmodelle, die vom Aufbau des menschlichen Gehirns inspiriert sind (Buduma & Lacascio, 2017, S. 7 f.) und als Algorithmus für das überwachte Lernen in der vorliegenden Arbeit genutzt werden. Weiterhin wird in dieser Arbeit ein *feedforward neural network* genutzt, also ein neuronales Netz, das ein gerichteter azyklischer Graph ist. (Brilliant.org, 2018d) Diese Art neuronaler Netze werden primär in überwachten Lernszenarien genutzt, bei denen die Eingabedaten weder einer intrinsischen Reihenfolge folgen, wie z.B. Text, noch zeitabhängig sind, wie z.B. Messwerte. (Brilliant.org, 2018d) Beides ist für die $48px \times 48px$ -Fußballspielerbilder der Eingabe in dieser Arbeit nicht der Fall. Abbildung 6 stellt ein Neuron dar, das, im gerichteten und geschichteten Verbund mit weiteren Neuronen, ein *feedforward neural network* bilden kann.

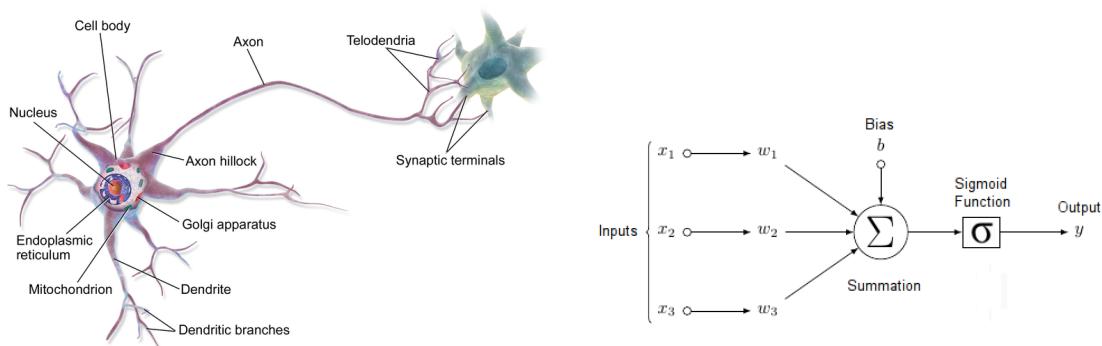


Abb. 6: Links zu sehen ist ein biologisches Neuron. Über Dendriten nimmt das Neuron Signale von anderen Neuronen entgegen und fasst diese im Zellkern zusammen. Übersteigt das gebündelte Signal einen Aktivierungswert, feuert das Neuron ein Signal über das Axon. (Buduma & Lacascio, 2017, S. 7) Diesem Prinzip folgt eine mathematische Nachbildung des Neurons, rechts zu sehen.

Bildquellen: (Blausen, Bruce, 2013)(l.) und (Brilliant.org, 2018a)(r.).

In obiger Abbildung ist die biologische und mathematische Darstellung eines Neurons zu sehen. Wie aus einem solch einzelnen Neuron ein lernender Algorithmus wird, wird in Kapitel 2.2.1 beschrieben. Wie aus einer geschichteten Anordnung von Neuronen (Buduma & Lacascio, 2017, S. 9 ff.) ein neuronales Netz entsteht, das die Fähigkeiten eines einzelnen Neurons weit übersteigt (Buduma & Lacascio, 2017, S. 11), wird in Kapitel 2.2.2 beschrieben.

2.2.1 Single layer-Perzeptron

Zunächst wird also der grundlegendste Fall eines neuronales Netzes betrachtet, nämlich ein neuronales Netz, das aus nur einem Neuron besteht. Dieses Netz wird (*single-layer*)-Perzeptron (engl. perceptron) genannt und wird als lernender Algorithmus seit den 1950er Jahren genutzt. (Buduma & Lacascio, 2017, S. 5)

Formell soll mit diesem Perzeptron-Algorithmus folgendes erreicht werden:

Sei

$$X = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\} \text{ mit } \mathbf{x}_i \in \mathbb{R}^n \text{ und } y_i \in \mathbb{R}$$

die (Trainings-)Menge aller Eingabe-Ausgabe-Tupel. Der Algorithmus soll auf dieser Menge X eine Abbildung $h : \mathbb{R}^n \rightarrow \mathbb{R}$ erlernen, sodass

$$h_{\mathbf{w}, b}(\mathbf{x}_i) \approx y_i$$

für jedes Eingabe-Ausgabe-Tupel $(\mathbf{x}_i, y_i) \in X$. (Brilliant.org, 2018a) Dabei sind $\mathbf{w} \in \mathbb{R}^n$ die Gewichte (engl. weights) und $b \in \mathbb{R}$ der sogenannten Bias des Neurons. Die Abbildungsvorschrift von h ist die gewichtete Summe aus dem Gewichtsvektor \mathbf{w} und dem Eingangsvektor \mathbf{x} , gemäß dem biologischen Vorbild aus Abbildung 6. Diese gewichtete Summe ist die Linearkombination aus den Gewichten und Eingaben und wird definiert als

$$z = h(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b = \sum_{i=1}^n w_i x_i + b = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b. \quad (1)$$

Dabei wird z als der *logit* bezeichnet. (Buduma & Lacascio, 2017, S. 13) Im Beispielneuron aus Abbildung 6 ist $\mathbf{w} = (w_1 \ w_2 \ w_3)^T$ der Gewichtsvektor und $\mathbf{x} = (x_1 \ x_2 \ x_3)^T$ eine Eingabe in das Neuron aus der Trainingsmenge, da es drei Eingänge und drei Gewichte gibt.

Der Grund für diese Funktionsvorschrift ist geometrisch leichter ersichtlich als durch Formeln, siehe Abbildung 7. Dabei spannt $z = h(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$ eine (Hyper-)Ebene auf, die auch Entscheidungsgrenze (engl. decision boundary) genannt wird. (Brilliant.org, 2018d) Hierbei definiert \mathbf{w} die Steigung der Entscheidungsgrenze

und ist der Normalvektor der Ebene, während der Bias b den Achsenabschnitt angibt. Angemerkt sei hier, dass w und b veränderbar sind, während x als fest und gegeben angesehen wird. Dadurch, dass w und b variabel sind, lässt sich auch die Entscheidungsgrenze anders in den Vektorraum legen. Die Entscheidungsgrenze soll möglichst so gelegt werden, dass, für gegebene Gewichte und Bias, $h(x_i) \approx y_i$ für alle $(x_i, y_i) \in X$ gilt. Die Entscheidungsgrenze so zu legen ist Aufgabe des Trainings, auch Optimierung genannt. (Buduma & Lacascio, 2017, S. 6)

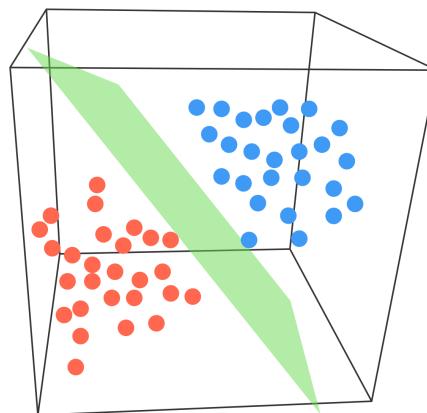


Abb. 7: Zu sehen ist eine Hyperebene $z = ax + by + c$ im Raum \mathbb{R}^3 , die zwei Ausgabeklassen linear separieren kann. Bildquelle: (Brilliant.org, 2018d).

Eine Entscheidungsgrenze, wie in Abbildung 7, ermöglicht eine Klassifikation oder Regression. Ob nun eine Klassifikation oder eine Regression umgesetzt wird, bestimmt die sogenannte Aktivierungsfunktion $g(z) = g(h(\mathbf{x})) = \hat{y}$. (Brilliant.org, 2018d) Im Falle einer Klassifikation, wie bei dem Perzeptronalgorithmus aus den 1950er Jahren (Buduma & Lacascio, 2017, S. 5), kann $g(x)$ die Heaviside-Funktion sein, sodass

$$\hat{y} = g(h(\mathbf{x})) = \begin{cases} 1 & \text{für } \mathbf{w}^T \mathbf{x} + b \geq 0, \\ 0 & \text{für } \mathbf{w}^T \mathbf{x} + b < 0. \end{cases} \quad (2)$$

Typische Aktivierungsfunktionen sind die Sigmoid-Funktion $\sigma(x) = (1 + e^{-x})^{-1}$, die hyperbolische Tangens-Funktion $\tanh(x) = (e^x + e^{-x})/(e^x - e^{-x})$, die *restricted linear unit*-Funktion, ReLU abgekürzt, $relu(x) = \max(0, x)$ oder die identische Abbildung $id(x) = x$. (Buduma & Lacascio, 2017, S. 13 ff.) Die Heaviside-, Sigmoid-

und hyperbolische Tangens-Funktion können für die Klassifikation eingesetzt werden, während die identische Abbildung bei der Regression zum Einsatz kommt. (Brownlee, Jason, 2018a, S. 44)

Wie oben erwähnt, lassen sich die Gewichte w und der Bias b anpassen, damit $g(h(\mathbf{x}_i)) \approx y_i$ für alle Elemente der Trainingsmenge gilt. Hierfür muss ein Maß für die Güte der Gewichte und des Bias eingeführt werden. Dieses Maß wird Fehlermaß genannt und durch eine Fehlerfunktion berechnet, die den Unterschied zwischen $g(h(\mathbf{x}_i)) = \hat{y}_i$ und der eigentlich gewünschten Ausgabe y_i aus $(\mathbf{x}_i, y_i) \in X$, bei gegebenem w und b , berechnet. (Buduma & Lacascio, 2017, S. 17 f.)

Der Optimierungsprozess

$$\min_{w,b} E(w, b, X)$$

wird dann auch Training des lernenden Algorithmus genannt, wobei $E(w, b, X)$ die Fehlerfunktion ist, die das Fehlermaß über die Trainingsmenge X ermittelt. Falls ein Neuron eine Regression durchführen soll, wird typischerweise das quadratische Fehlermaß (engl. mean squared error (MSE)) genutzt. Im Falle einer Klassifikation wird als Fehlerfunktion die binäre Kreuzentropie, für eine binäre Klassifikation, und die kategoriale Kreuzentropie, für eine Multi-Klassifikation, gewählt. (Brownlee, Jason, 2018a) An dieser Stelle wird das quadratische Fehlermaß als Fehlerfunktion angenommen. Dies sei folgendermaßen definiert:

$$E(w, b, X) = \frac{1}{2n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (3a)$$

$$= \frac{1}{2n} \sum_{i=1}^n (g(h(\mathbf{x}_i)) - y_i)^2 \quad (3b)$$

$$= \frac{1}{2n} \sum_{i=1}^n (g(\mathbf{w}^T \mathbf{x}_i + b) - y_i)^2. \quad (3c)$$

Ziel ist die Minimierung dieser Fehlerfunktion, sodass $E(w, b, X) = 0$, was genau dann der Fall ist, wenn $\hat{y}_i = y_i$ für alle (\mathbf{x}_i, y_i) aus X gilt. Es soll also versucht werden, die Gewichte w und den Bias b so anzupassen, dass das Fehlermaß gegen Null geht, was wiederum die Anforderung an den Algorithmus zum Erlernen einer

Abbildung die Eingabe und Ausgabe aus der Trainingsmenge in eine Beziehung setzen kann, ausdrückt.

Die Fehlerfunktion wird typischerweise durch das heuristische Gradientenabstiegsverfahren (engl. *gradient descent*) minimiert, da eine analytische Lösung der Fehlerfunktion, aufgrund ihrer Komplexität, insbesondere bei tiefen neuronalen Netzen wie sie in Kapitel 2.2.2 noch beschrieben werden, sehr aufwendig ist. (Brilliant.org, 2018d) Wenn die Fehlerfunktion das Fehlermaß über die gesamte Trainingsmenge X berechnet, um die Gewichte und den Bias anzupassen, wird dies *batch gradient descent* genannt. Wird nach jedem $(x_i, y_i) \in X$ das Fehlermaß zur Gewichts- und Bias-Anpassung genutzt, wird dies *stochastic gradient descent* genannt. (Brownlee, Jason, 2018a, S. 181 f.)

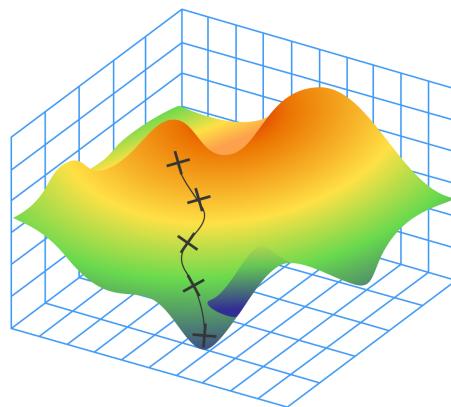


Abb. 8: Dargestellt ist das Gradientenabstiegsverfahren zur Minimierung der Fehlerfunktion $E(\mathbf{w}, b, X)$. Die drei Dimensionen werden hier etwa durch einen Gewichtsvektor $\mathbf{w} = (w_1 \ w_2)^T$ und der Fehlerfunktion E aufgespannt. Diese Fehlerfunktion gibt also, einfach formuliert, das Fehlermaß zu einer bestimmten Einstellung der Gewichte an, $(w_1 \ w_2)^T \rightarrow \mathbb{R}$. Ziel ist es, lokale Minima zu finden. Bildquelle: (Brilliant.org, 2018d).

Beim Gradientenabstiegsverfahren werden der Gewichtsvektor \mathbf{w} und der Bias b in Richtung des negativen Steigung der Fehlerfunktion adjustiert. Dies bedeutet, dass Werte für \mathbf{w} und b gefunden werden, die ein niedrigeres Fehlermaß erzeugen. Ziel des Verfahrens ist es, durch einen iterativen Prozess, das Fehlermaß konvergieren zu lassen, was sich häufig in der Findung eines lokalen Minima ausdrückt. (Brilliant.org, 2018d)

Zu Beginn des Gradientenabstiegsverfahrens werden \mathbf{w}^0 und b^0 auf einen zufälligen Wert gesetzt, wobei \mathbf{w}^i und b^i den Gewichtsvektor und den Bias nach der i -ten Iteration des Gradientenverfahrens bezeichnen. Gewichtsvektor und Bias werden durch die folgenden Formeln angepasst:

$$\mathbf{w}^{i+1} = \mathbf{w}^i + \Delta \mathbf{w} \quad (4)$$

$$b^{i+1} = b^i + \Delta b \quad (5)$$

Der Gewichtsvektor und der Bias werden also pro Iterationsschritt um ein $\Delta \mathbf{w}$ bzw. ein Δb angepasst. Die Berechnungen von $\Delta \mathbf{w}$ und Δb sind methodisch gleich (Brilliant.org, 2018d), daher wird häufig der sogenannte Bias-Trick genutzt, um die beiden Parameter \mathbf{w} und b als einen gemeinsamen auszudrücken. Dafür wird \mathbf{w} um eine Vektorkomponente erweitert, die fortan den Bias enthält. Da dieser nach Gleichung 1 nicht gewichtet wird, werden alle x_i aus $(x_i, y_i) \in X$ um eine Vektorkomponente 1 erweitert. (Li, FeiFei et. al., 2018c) Deshalb wird in der Folge nur die Berechnung der Gewichtsanpassung $\Delta \mathbf{w}$ dargestellt.

Das Gradientenabstiegsverfahren sieht einen Abstieg in die negative Richtung der Steigung der Fehlerfunktion vor. (Buduma & Lacascio, 2017, S. 20) Die Richtung der Steigung wird dabei durch den Gradienten

$$grad(E(\mathbf{w})) = \nabla E(\mathbf{w}) = \frac{\partial E}{\partial w_1} \mathbf{e}_1 + \cdots + \frac{\partial E}{\partial w_n} \mathbf{e}_n = \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{pmatrix}$$

angegeben, wobei $\mathbf{e}_1, \dots, \mathbf{e}_n$ die Einheitsvektoren der Koordinatenachsen sind.

Wie weit der Abstieg dabei in die negative Richtung der Steigung geschehen soll, wird durch eine Skalarmultiplikation des Gradienten mit der sogenannten Lernrate (engl. learning rate) α und der Richtungsumkehrung durch Multiplikation des Gradienten mit -1 , bestimmt. (Buduma & Lacascio, 2017, S. 21) Die Festlegung der Lernrate α wird in der Literatur häufig ausführlich erläutert und illustriert,

da eine zu kleine Lernrate die Trainingsdauer erhöhen kann und eine zu große Lernrate zu einem ungewollten Überspringen des Fehlerminimums im Suchbereich führen kann. (Buduma & Lacascio, 2017, S. 19 ff.) In Kapitel 2.4.3 wird kurz auf Optimierungstechniken eingegangen, die eine adaptive Lernrate während des Trainings ermöglichen. Der Gradienten und die Lernrate zusammen bestimmen, wie die Gewichte im Iterationsschritt geändert werden. (Buduma & Lacascio, 2017, S. 21) Daraus folgt die Gewichtsanpassung $\Delta\mathbf{w} = -\alpha \nabla E(\mathbf{w})$ für einen Iterationsschritt. (Brilliant.org, 2018a) Eingesetzt in Gleichung (4) folgt für die Anpassung des Gewichtsvektors

$$\mathbf{w}^{i+1} = \mathbf{w}^i + \Delta\mathbf{w} \quad (6a)$$

$$= \mathbf{w}^i + (-\alpha \nabla E(\mathbf{w}^i)) \quad (6b)$$

$$= \mathbf{w}^i - \alpha \nabla E(\mathbf{w}^i) \quad (6c)$$

$$= \mathbf{w}^i - \alpha \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{pmatrix}. \quad (6d)$$

Ein Abstieg lässt sich dann durch $E(\mathbf{w}^{i+1}) < E(\mathbf{w}^i)$ charakterisieren. (Brilliant.org, 2018a) Aus Gleichung 6d ist erkennlich, dass zur Gewichtsanpassung die partiellen Ableitungen von $E(\mathbf{w})$ nach w_j , für alle Gewichtsvektorkomponenten w_1, \dots, w_n mit $1 \leq j \leq n$ benötigt werden. Intuitiv gesehen soll eine Gewichtsvektorkomponente w_j stark geändert werden, wenn sie einen hohen Anteil am Fehler hat. Ein hoher Anteil eines Gewichtes w_j am Fehler liegt genau dann vor, wenn $\frac{\partial E}{\partial w_j}$, also die Steigung, groß ist. Wenn beim Gradientenabstieg einem Minimum nahe gekommen wird, wird die Steigung des Gradienten $|\nabla E(\mathbf{w})|$ immer kleiner, und somit auch die Gewichtsanpassungen $\Delta\mathbf{w}$.

Die partiellen Ableitungen von $E(\mathbf{w})$ nach w_j lassen sich, für einen *batch gradient descent*, folgendermaßen herleiten:

Aus Gleichung 3 ist erkennbar, dass die Fehlerfunktion aus einer Funktionenkomposition besteht, nämlich aus $E(\mathbf{w}, X)$, unter Annahme der Verwendung des Bias-Tricks, $\hat{y} = g(z)$ und $z = h(\mathbf{x})$. Deshalb wird die partielle Ableitung von

$E(\mathbf{w})$ nach w_j für die Trainingsmenge X in einem jeden Iterationsschritt durch Anwendung der Kettenregel

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j} \quad (7)$$

berechnet. Die Ableitung des ersten Faktors auf der rechten Seite von Gleichung 7 ist

$$\frac{\partial E}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left(\frac{1}{2n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \right) \quad (8a)$$

$$= \frac{1}{2n} \sum_{i=1}^n \frac{\partial}{\partial \hat{y}} ((\hat{y}_i - y_i)^2) \quad (8b)$$

$$= \frac{1}{2n} \sum_{i=1}^n 2(\hat{y}_i - y_i) \quad (8c)$$

$$= \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i). \quad (8d)$$

Eigentlich würde in 8a mit $\frac{\partial E}{\partial \hat{y}_i}$ die Summe wegfallen, aber hier soll *batch gradient descent* gezeigt werden. Die Ableitung des zweiten Faktors der rechten Seite von Gleichung 7 ist

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial}{\partial z} (g(z)) = g'(z). \quad (9)$$

Dabei wird $g'(z)$ hier nicht weiter ausgerechnet, da unterschiedliche Aktivierungsfunktionen genutzt werden können, die auch unterschiedliche Ableitungen haben. Wichtig aber ist die Feststellung, dass $g(z)$ differenzierbar sein muss. Die Heaviside-Funktion hat bei $z = 0$ eine Sprungstelle und ist dort nicht differenzierbar, kann hier also nicht angewendet werden. Weiterhin gilt $g(z) = z$ bei einer Regression, also gilt $\frac{\partial}{\partial z} (g(z)) = 1$. Außerdem ist in Gleichung 9 zu beachten, dass $z = h(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i$ ist.

Schließlich muss noch der dritte Faktor der Kettenregel aus 7 berechnet werden:

$$\frac{\partial z}{\partial w_j} = \frac{\partial}{\partial w_j} \left(\sum_{i=1}^n w_i x_i \right) = \frac{\partial}{\partial w_j} (w_j x_i) = x_i \quad (10a)$$

Weil nur nach w_j abgeleitet werden soll, muss nur der Term $w_j x_i$ abgeleitet werden. Konstante Koeffizienten bleiben stehen und so ergibt diese Ableitung x_i . Nun multiplizieren wir die Ableitungen 8d, 9 und 10 gemäß der Kettenregeln nach Gleichung 7 und erhalten

$$\frac{\partial E}{\partial w_j} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i) g'(z) x_i. \quad (11a)$$

So lässt sich nun der Gradient $\nabla E(\mathbf{w}^i)$ aus Gleichung 6c bestimmen und damit die Gewichtsanpassung $\mathbf{w}^{i+1} = \mathbf{w}^i - \alpha \nabla E(\mathbf{w}^i)$. Die Terminierungsbedingungsbedingungen für diese iterative Gewichtsanpassung beim *gradient descent* können sein:

Anzahl an Iterationen Nach n Iterationen wird der Abstieg gestoppt. Der Parameter n wird im Vorhinein festgelegt. Ein solches n kann praktisch durch eine Epochenzahl, wie in Kapitel 4 beschrieben, umgesetzt werden.

Erreichen eines Schwellwertes Der Abstieg wird gestoppt, falls $|E(\mathbf{w}^{i+1}) - E(\mathbf{w}^i)| \leq \varepsilon$ für n skizzative Iterationen gilt. Dabei werden die Parameter ε und n im Vorhinein festgelegt.

Early Stopping-Methoden Hier wird die Trainingsmenge X aufgeteilt in Trainings- und Validierungsdaten. So kann *over-* bzw. *underfitting* frühzeitig erkannt und der Abstieg automatisch gestoppt werden, auch wenn die n -te Iteration noch nicht erreicht wurde. (Buduma & Lacascio, 2017, S. 31 ff.)

Wie oben angesprochen, kann die Heaviside-Funktion aus Gleichung 2 für dieses Verfahren zur iterativen Gewichtsanpassung nicht genutzt werden, da die Heaviside-Funktion nicht vollständig differenzierbar ist. Um auch Neuronen mit Heaviside-Aktivierungsfunktion nutzen zu können, kann die Anpassungsregel $\Delta w_{ji} = \alpha(\hat{y}_i - y_i)x_{ji}$ genutzt werden. Dieses Verfahren nennt man Perzeptron-Algorithmus. Das oben vorgestellte Verfahren zur Gewichtsanpassung nennt man *delta rule* oder *adaline rule*. (Brilliant.org, 2018d)

In diesem Kapitel wurde ein neuronales Netz mit einem einzelnen Neuron, *single layer*-Perzeptron genannt, als lernender Algorithmus eingeführt. Das Lernen findet in Form des Trainings statt und sorgt dafür, dass die Gewichte des Neurons angepasst werden, sodass dieses eine Klassifikations- oder Regressionsaufgabe lösen kann. Einzelne Neuronen haben als lernender Algorithmus aber einen gewichtigen Nachteil: Sie können nur lineare separierbare Datenmuster klassifizieren bzw. extrapolieren. (Buduma & Lacascio, 2017, S. 12 f.) Linear heißt hier, dass jede Unbekannte einer Linearkombination höchstens in ihrer ersten Potenz vorkommen kann, was für das *single layer*-Perzeptron durch Gleichung 1 gilt. Illustrieren lässt sich diese Einschränkung durch Abbildung 9.

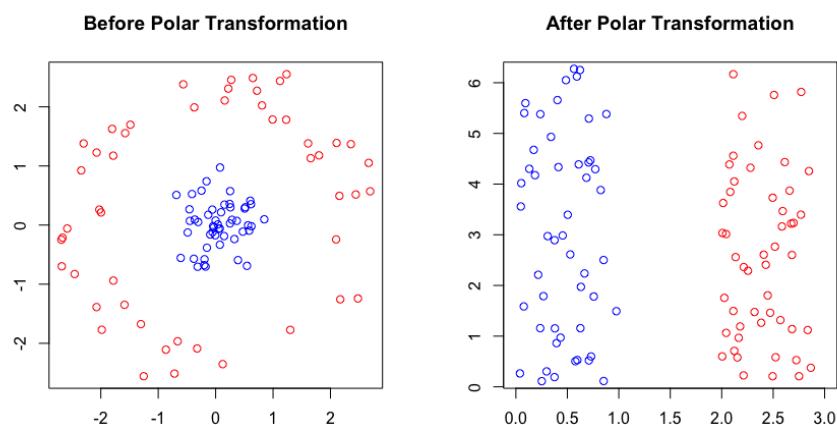


Abb. 9: Links dargestellt sind blaue und rote Datenpunkte, die nicht linear separierbar sind. Erst nach einer Transformation in ein Polarkoordinatensystem sind die Datenpunkte linear separierbar. Bildquelle: (Brilliant.org, 2018d).

Aus Abbildung 9 links ist erkennbar, dass es keine lineare Linie oder (Hyper-)Ebene geben kann, die die roten und blauen Datenpunkte, x , separieren kann. Erst nach

einer Polartransformation der Datenpunkte lassen sich diese linear separieren. Eine Polartransformation ist dabei nur ein Beispiel für eine Transformation, um Daten linear separierbar zu machen. Schön wäre es, wenn es eine Möglichkeit zur automatischen Transformation gäbe, sodass die Daten immer linear separierbar sind. Genau diese Möglichkeit wird durch *multi layer*-Perzeptron-Netzwerke, die im nächsten Kapitel 2.2.2 vorgestellt werden, eröffnet.

2.2.2 *Multi layer*-Perzeptron

Das *multi layer*-Perzeptron ist ein neuronales Netz, das aus einer Vielzahl von in Schichten angeordneten Neuronen, wie sie im vorherigen Kapitel 2.2.1 vorgestellt wurden, besteht. (Buduma & Lacascio, 2017, S. 10, 24) Anders als ein *single layer*-Perzeptron, kann ein *multi layer*-Perzeptron auch eine Klassifikation oder Regression mit nicht-linear separierbare Daten durchführen. (Brilliant.org, 2018d) Dies ist nach Kapitel 2.2.1 immer dann möglich, wenn sich die Daten so transformieren lassen, dass sie linear separierbar sind. Diese Transformationen kann ein *multi layer*-Perzeptron-Netz durchführen. (Brilliant.org, 2018d) Die Besonderheit von *multi layer*-Perzeptron-Netzen, die auch die automatische Transformation in einen linear separierbaren Raum ermöglicht, ist die Anordnung von Neuronen in Schichten, wie in Abbildung 10 dargestellt. Für ein neuronales Netz, das eine Bildklassifikation durchführen soll, bedeutet dies, dass das Netz alle, für die Klassifikation wichtigen, Bildmerkmale selber herausarbeiten kann. (Buduma & Lacascio, 2017, S. 11)

Kapitel 2.2.1 hat eine Einführung in die Funktionsweise eines (*single layer*)-Perzeptrons/Neurons gegeben. Dabei wurde, vereinfacht gesprochen, erklärt, dass zunächst eine gewichtete Summe aus Eingaben und Gewichten eines Perzeptrons berechnet wird. Diese Summe ist die Ausgabe des Perzeptrons. Es wird dann, durch eine Fehlerfunktion, überprüft, inwieweit die Ausgabe mit einer gewünschten Ausgabe übereinstimmt. Ist die Übereinstimmung für alle Eingaben einer Trainingsmenge noch nicht hoch, werden die Gewichte angepasst und der Ablauf beginnt von vorne. Ist die Übereinstimmung hoch, muss der Ablauf nicht weiter durchgeführt werden. Dieser Ablauf beschreibt das Lernen eines lernen-

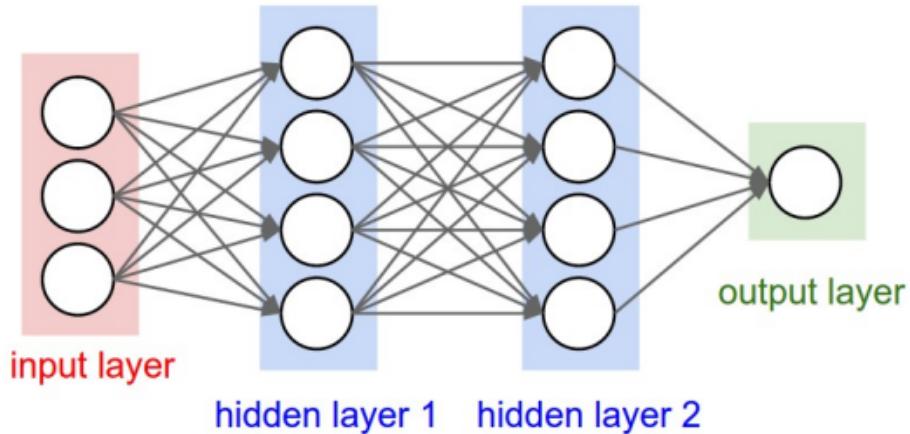


Abb. 10: Dargestellt ist ein neuronales Netz mit zwei *hidden*-Schichten mit jeweils vier Neuronen und einer Ausgabeschicht mit einem Neuron. Insgesamt besteht das neuronale Netz also aus neun Neuronen angeordnet in zwei Schichten. Die Eingabe zählt somit nicht als eigene Schicht. (Li, FeiFei et. al., 2018d) Jedes Neuron funktioniert wie ein *single layer*-Perzeptron aus Kapitel 2.2.1 und ist mit jedem Neuron aus der vorherigen Schicht verbunden, was *fully connected* genannt wird. Bildquelle: (Brilliant.org, 2018d).

den Algorithmus, wie *single layer*-Perzeptronen und *multi layer*-Perzeptronen sie sind.

Dieser Lernablauf wurde in Kapitel 2.2.1 für *single layer*-Perzeptronen beschrieben, lässt sich aber auch auf *multi layer*-Perzeptronen übertragen. Der Unterschied dabei ist, dass es nun mehrere, in Schichten angeordnete, Perzeptronen/Neuronen gibt, die zu unterschiedlichen Anteilen zum Fehlermaß des Netzes beitragen. Dies führt dazu, dass es bei *multi layer*-Perzeptronen-Netzen nun eine Möglichkeit geben muss, Gewichte in *hidden*-Schichten gemäß ihrem Anteil am Fehler anzupassen. Diese Möglichkeit wird durch den Backpropagation-Algorithmus gegeben. (Buduma & Lacascio, 2017, S. 23 f.)

Das Lernen und insbesondere der Backpropagation-Algorithmus sollen nun formal beschrieben werden, wobei die in Kapitel 2.2.1 hergeleitete *delta rule* nichts weiteres als eine nicht-rekursive Variante des Backpropagation-Algorithmus ist.

Sei w_{ij}^k das Gewicht, das in das j -te Neuron in der k -ten Schicht aus dem i -ten Neuron aus der $(k - 1)$ -ten Schicht einfließt. Sei o_j^k die Ausgabe des j -ten Neurons aus der k -ten Schicht. Dann wird die Linearkombination aus Gewichten

und Eingaben in das j -te Neuron aus der k -ten Schicht analog zu Gleichung 1 berechnet

$$a_j^k = \sum_{i=1}^{r_k} w_{ij}^k o_i^{k-1}, \quad (12a)$$

wobei r_k die Anzahl der Neuronen in der vorherigen Schicht meint und a_j^k den *logit* z aus dem vorherigen Unterkapitel ersetzt. Diese Berechnung kann auch in Vektor- bzw. Matrixschreibweise formuliert werden. Dabei repräsentiert eine Spalte der Matrix alle Gewichte des j -ten Neurons der k -ten Schicht. (Buduma & Lacascio, 2017, S. 11 f.) So lässt sich

$$\mathbf{a}^k = \mathbf{W}_k^T \mathbf{o}^{k-1} \quad (13a)$$

schreiben, was insbesondere für eine performante Hardware-Implementierung mit systolischen Arrays wichtig ist. Insbesondere lässt sich so das gesamte neuronale Netz als Tensor $(m - 1)$ -ter Ordnung darstellen, wobei m die Anzahl der Schichten eines neuronalen Netzes beziffert. Deep-Learning-Bibliotheken, wie Tensorflow, führen den Backpropagation-Algorithmus auch tatsächlich auf Tensoren aus. In der vorliegenden Arbeit wird aus Gründen der Übersichtlichkeit darauf aber verzichtet und auf (Johnson, Justin, 2018, S. 5 ff.) verwiesen.

Wie aus Gleichung 6 bekannt, wird die Gewichtsanpassung durch *gradient descent* durchgeführt. Die Gewichte zum j -ten Neuron der k -ten Schicht in der p -ten Iteration von *gradient descent* werden angepasst durch

$${}^{p+1}\mathbf{w}_j^k = {}^p\mathbf{w}_j^k + \Delta\mathbf{w}_j^k \quad (14a)$$

$$= {}^p\mathbf{w}_j^k - \alpha \nabla E({}^p\mathbf{w}_j^k) \quad (14b)$$

$$= {}^p\mathbf{w}_j^k - \alpha \begin{pmatrix} \frac{\partial E}{\partial w_{1j}^k} \\ \vdots \\ \frac{\partial E}{\partial w_{r_k j}^k} \end{pmatrix}. \quad (14c)$$

Auch hier müssen wieder alle partiellen Ableitungen $\frac{\partial E}{\partial w_{ij}^k}$ mit $1 \leq i \leq r_k$ zur Berechnung des Gradienten bestimmt werden. Diese Ableitungen können mit der Kettenregel

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \quad (15a)$$

bestimmt werden. Im Vergleich zu der Kettenregel des *single layer*-Perzeptrons aus Gleichung 7 fällt auf, dass ein Faktor fehlt. Der Grund dafür ist, dass es keine Fehlerfunktion für *hidden*-Schichten gibt. Da diese Kettenregel aber sowohl für die *hidden*-Schichten, als auch für die Ausgabeschicht gelten soll, wird eine Fallunterscheidung gemacht.

Dafür wird zunächst der erste Faktor der Gleichung 15 Error genannt (Brilliant.org, 2018b) und es soll gelten

$$\delta_j^k \equiv \frac{\partial E}{\partial a_j^k}. \quad (16a)$$

Der zweite Faktor aus Gleichung 15 kann mit Gleichung 12 direkt abgeleitet werden zu

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \left(\sum_{i=1}^{r_k} w_{ij}^k o_i^{k-1} \right) = o_j^{k-1}. \quad (17a)$$

Auch hier in Ableitung 17 ist wieder die Ableitung nach einem bestimmten Gewicht gesucht, was der Grund dafür ist, dass das Summe wegfallen kann. Da außerdem konstante Vorfaktoren beim Ableiten wegfallen, bleibt nur o_j^{k-1} stehen. Die Gleichungen 16 und 17 in Gleichung 15 eingesetzt ergeben dann

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}. \quad (18a)$$

Mithilfe dieser Gleichung 18 können wir nun die Fallunterscheidung durchführen. Im ersten Fall wird die Berechnung für die letzte Schicht m gezeigt, im zweiten Fall die Berechnung für eine jede *hidden*-Schicht.

Für den ersten Fall wird dieses Mal die binäre Kreuzentropie-Fehlerfunktion angenommen, wie sie für die Bildklassifikation in Kapitel 5 auch tatsächlich verwendet wird. Diese sei definiert als

$$E(\mathbf{w}, X) = \sum_{i=1}^n -y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i), \quad (19a)$$

siehe (Sadowski, Peter, 2018, S. 1 f.). Da nun die Fehlerfunktion bekannt ist, kann die folgende Gleichung, die den ersten Fall der letzten Schicht darstellt, gelöst werden. Diese Gleichung beschäftigt sich damit, wie groß der Einfluss von a_j^m auf das Fehlermaß des gesamten Netzes ist.

$$\delta_j^m = \frac{\partial E}{\partial a_j^m} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_j^m}. \quad (20a)$$

Für den ersten Faktor $\frac{\partial E}{\partial \hat{y}}$ ergibt sich

$$\frac{\partial E}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left(\sum_{i=1}^n -y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right) \quad (21a)$$

$$= \sum_{i=1}^n \frac{\partial}{\partial \hat{y}} (-y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)) \quad (21b)$$

$$= \sum_{i=1}^n \frac{-y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i} \quad (21c)$$

$$= \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)}. \quad (21d)$$

Für die zweite Ableitung der Kettenregel in Gleichung 20 entsteht

$$\frac{\partial \hat{y}}{\partial a_j^m} = \frac{\partial}{\partial a_j^m} (g(a_j^m)) = g'(a_j^m). \quad (22a)$$

Hierbei ist $g(a_j^m)$ eine beliebige Aktivierungsfunktion der Ausgabeschicht. Eingesetzt in Gleichung 20 bedeutet dies

$$\delta_j^m = \frac{\partial E}{\partial a_j^m} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_j^m} = \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} g'(a_j^m). \quad (23a)$$

So lässt sich nun konkludieren, dass für die Neuronen der Ausgabeschicht eines mehrschichtigen neuronalen Netzes die folgende Gleichung zur Berechnung der für den Gradienten benötigten partiellen Ableitungen genutzt werden kann:

$$\frac{\partial E}{\partial w_{ij}^m} = \delta_j^m o_i^{m-1} = \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} g'(a_j^m) o_i^{m-1}. \quad (24a)$$

In Kapitel 5 wird konkret die Sigmoid-Aktivierungsfunktion $\hat{y}_i = \sigma(a_j^m) = (1+e^{-a_j^m})^{-1}$ genutzt. Die erste Ableitung dieser Aktivierungsfunktion ist $\sigma'(a_j^m) = \sigma(a_j^m)(1 - \sigma(a_j^m))$, siehe (Sadowski, Peter, 2018, S. 2). Eingesetzt in Gleichung 24 ergibt dies

$$\frac{\partial E}{\partial w_{ij}^m} = \delta_j^m o_i^{m-1} = \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} g'(a_j^m) o_i^{m-1}. \quad (25a)$$

$$= \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \sigma(a_j^m)(1 - \sigma(a_j^m)) o_i^{m-1} \quad (25b)$$

$$= \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \hat{y}_i(1 - \hat{y}_i) o_i^{m-1} \quad (25c)$$

$$= \sum_{i=1}^n (\hat{y}_i - y_i) o_i^{m-1}. \quad (25d)$$

Da nun hergeleitet wurde, wie mit dem Backpropagation-Algorithmus die Gewichte hinzu der Ausgabeschicht berechnet werden, wird sich nun dem zweiten Fall gewidmet, nämlich der Berechnung der Gewichte in den *hidden*-Schichten. Dies bedeutet, dass statt δ_j^m nun δ_j^k mit $1 \leq k < m$ für die Gleichung 18 berechnet wird. Der Error δ_j^k beschäftigt sich damit, wie groß der Einfluss des Fehlermaßes, das erst nach der Ausgabeschicht bestimmt wird, auf die Berechnung von a_j^k ist. Dass dies notwendig ist, wird aus Gleichung 15 ersichtlich, denn über $\frac{\partial E}{\partial a_j^k}$ kann der Einfluss des Fehlermaßes auf das Gewicht w_{ij}^k bestimmt werden. Weiterhin muss festgestellt werden, dass $\frac{\partial E}{\partial a_j^k} = \delta_j^k$ abhängig ist von dem Fehleranteil der $(k+1)$ -ten Schicht, denn die Eingabe in die $(k+1)$ -te Schicht ist die Ausgabe der k -ten Schicht

$$a_l^{k+1} = \sum_{j=1}^{r_k} w_{jl}^{k+1} o_j^k = \sum_{j=1}^{r_k} w_{jl}^{k+1} g(a_j^k), \quad (26a)$$

wobei a_l^{k+1} die Ausgabe, noch ohne Aktivierungsfunktion, des l -ten Neurons der $(k+1)$ -te Schicht ist. Da aus Gleichung 15 hervorgeht, dass zunächst der Einfluss

des Fehlermaßes auf $\frac{\partial E}{\partial a_j^k}$ bestimmt werden muss, ergibt sich für die Berechnung des Errors δ_j^k somit folgende Gleichung

$$\delta_j^k = \frac{\partial E}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k} \quad (27a)$$

$$= \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}. \quad (27b)$$

Die Gleichung 27 ist ein Herzstück des Backpropagation-Algorithmus, ihr Verständnis deshalb wichtig. Gleichung 15 fordert, die Berechnung von $\frac{\partial E}{\partial w_{ij}^k}$, um ein Gewichtsupdate im *gradient descent* durchzuführen. Die Berechnungsvorschrift dafür ist durch Gleichung 18 gegeben. Gleichung 27 erklärt jetzt wie der Error δ_j^k für Neuronen in *hidden*-Schichten zu berechnen ist. Dieser Error δ_j^k hängt vom Error δ_l^{k+1} der nächsten Schicht ab. So lässt sich aus Gleichung 27 eine Rekursion erkennen, die den Error der Ausgabeschicht, über die *hidden*-Schichten, hinzu der Eingabeschicht rückwärts propagiert. (Buduma & Lacascio, 2017, S. 23 f.) Wird noch die verbliebene partielle Ableitung in Gleichung 27 aufgelöst, ergibt dies

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial}{\partial a_j^k} \left(\sum_{j=1}^{r_k} w_{jl}^{k+1} g(a_j^k) \right) \quad (28a)$$

$$= \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'(a_j^k) \quad (28b)$$

$$= g'(a_j^k) \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1}. \quad (28c)$$

Gleichung 28c eingesetzt in Gleichung 18 ergibt für $\frac{\partial E}{\partial w_{ij}^k}$ schlussendlich

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'(a_j^k) o_i^{k-1} \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1}. \quad (29a)$$

Zusammenfassend lässt sich sagen, dass zunächst *multi layer*-Perzeptronen-Netze motiviert wurden. Danach wurde das bereits in Kapitel 2.2 vorgestellte Gradientenabstiegsverfahren auf diese Art von neuronalen Netzen übertragen, wofür die Einführung des Backpropagation-Algorithmus notwendig ist. Dieser propagiert einen Error rückwärts-rekursiv durch das mehrschichtige neuronale Netz, sodass auch Gewichte angepasst werden können, deren Ausgabe nicht an einer Fehlerfunktion bemessen werden. Solche mehrschichtigen neuronalen Netze werden auch in Kapitel 5 für die Klassifikation von Fußballbildern genutzt. In der Computervision werden häufig noch sogenannte *convolutional*- und *pooling*-Schichten genutzt (Buduma & Lacascio, 2017, S. 89 ff.), die im nächsten Unterkapitel kurz erklärt werden.

2.3 *convolutional*- und *pooling*-Schichten

In diesem Unterkapitel soll nun die Funktionsweise von *convolutional*- und *pooling*-Schichten erklärt werden. Es könnte angenommen werden, dass mit dem in Kapitel 2.2.2 vorgestellten neuronalen Netz bereits eine Bildklassifikation durchgeführt werden könnte. Dazu müsste man lediglich jedes der Fußballspielerbilder aus Kapitel 3, die eine Auflösung von $48px \times 48px$ bei 3 Farbkanälen haben, als Eingabe in ein solches Netz einführen, was zu einer Eingabeschicht bestehend aus $48 \times 48 \times 3 = 6912$ Neuronen führen würde. Ein solches Vorgehen führt schon bei niedrig auflösenden Bildern zu einer immensen Vielzahl an Gewichten, die alle mit dem Backpropagation-Algorithmus trainiert werden müssen. (Buduma & Lacascio, 2017, S. 89 f.) Aus diesem Grund nutzen neuronale Netze in der Computervision zumeist sogenannte *convolutional*- und *pooling*-Schichten, die die wichtigsten Bildmuster erkennen und das Bild komprimieren, bevor ein neuronales Netz antrainiert wird. Diese beiden Schichten werden in diesem Unterkapitel dargestellt und erläutert.

2.3.1 Die *convolutional*-Schicht

Das Ziel einer *convolutional*-Schicht ist mit einem sogenannten *feature filter*, auch Kernel oder Filter genannt, ein Bild auf bestimmte Muster zu untersuchen. Die Idee

zur *convolution* ist dabei biologisch motiviert, da ein sehender Organismus visuelle Eingaben ebenfalls nur nach Muster untersucht. Dabei werden zunächst nur grobe Konturen von niedrigen Schichten des visuellen Cortex erkannt, während höhrere Schichten zunehmen abstrakte Objekte wahrnehmen können, siehe Abbildung 5. (Buduma & Lacascio, 2017, S. 87 f.)

Diese Herangehensweise zur visuellen Wahrnehmen wird von *convolutional*-Schichten imitiert. Dabei wird zunächst ein Kernel in Form einer $k \times q$ Matrix über ein Bild in Form einer $m \times n$ Matrix sukzessiv gleitend bewegt, wobei stets $k \leq m$ und $q \leq n$ gelten muss. Der Kernel führt eine Berechnung auf dem Bild aus und überträgt das Resultat in eine sogenannte *feature map*. Die Berechnung des Kernels auf dem Bild lässt sich wie folgt formalisieren:

Sei $\mathbf{A} \in M^{k \times q}(\mathbb{R})$ die Kernel-Matrix und $\mathbf{B} \in M^{m \times n}(\mathbb{R})$ die Bildmatrix. Dann nennen wir $\mathbf{A} \circ \mathbf{B}$ den *convolutional*-Operator oder das Hadamard-Produkt und definieren

$$(\mathbf{A} \circ \mathbf{B})_{ij} = \mathbf{A}_{ij} \mathbf{B}_{ij}, \quad (30a)$$

wobei \mathbf{A}_{ij} und \mathbf{B}_{ij} die i -te Zeile und j -te Spalte der Matrix meint. (Brilliant.org, 2018c) Für den Fall $k > m$ oder $q > n$, ist Gleichung 30 nicht sinnvoll definiert, jedoch ist die Bildmatrix meist größer als die Kernel-Matrix. (Brilliant.org, 2018c)

Zunächst wird also das Hadamard-Produkt von Kernel und Bild berechnet, sodass in der Folge die Summe über die berechnete $k \times q$ Matrix \mathbf{C} gebildet werden kann

$$s = \sum_{i=1}^k \sum_{j=1}^q c_{ij}. \quad (31a)$$

Auf diese Summe wird dann meist die *relu*-Funktion $relu(s) = \max(0, s)$, bekannt aus Kapitel 2.2.1, angewandt, um negative Werte in der *feature map* zu vermeiden. (Brilliant.org, 2018c)

Der Kernel gleitet dann über das gesamte Bild, entsprechend Abbildung 11, und berechnet eine *feature map*.

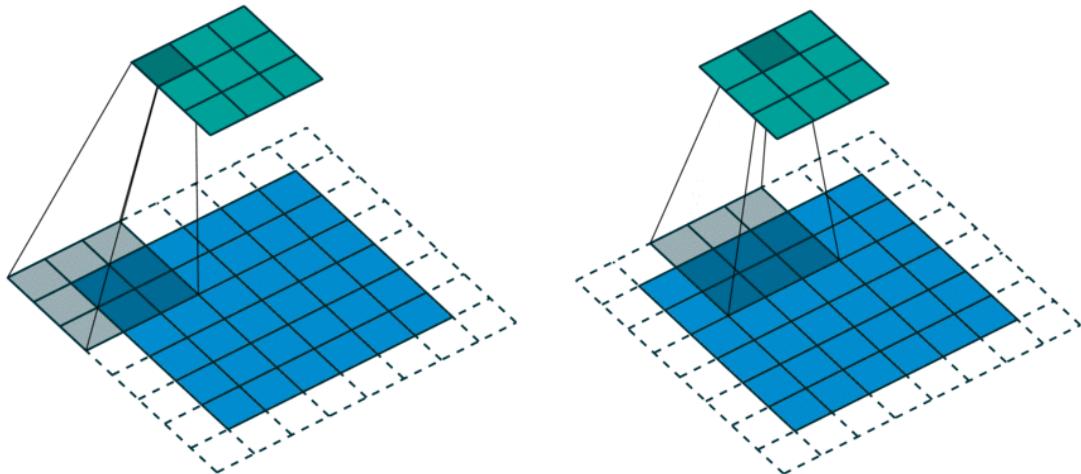


Abb. 11: Dargestellt ist der *convolutional*-Prozess. Dabei wird eine Bildmatrix (blau) mit einem *padding* von eins über die in den Gleichungen 30, 31 und $relu(s) = \max(0, s)$ definierten Berechnungen mit einem *stride* von zwei auf eine *feature map* (grün) abgebildet. Bildquelle: (deeplearning.net, 2017).

Stellt man nun die Frage, wie oft dieser *convolutional*-Prozess für ein gegebenen Kernel A und ein gegebenes Bild B durchgeführt werden muss, stellt man eigentlich die Frage nach der Größe der *feature map*. Um diese Frage zu beantworten, sollen zunächst noch zwei wichtige Parameter, die bei *convolutional*-Schichten stark Anwendung finden, erklärt werden.

Striding beschreibt den *offset*, mit dem der Kernel über das Bild wandert. (Buduma & Lacascio, 2017, S. 95) In Abbildung 11 ist der *stride* zwei, da der Kernel um zwei Kästchen nach rechts verschoben wird, bevor ein neuer Eintrag in die *feature map* erfolgt.

Padding beschreibt die Anzahl an zusätzlichen Bildpixeln, die zur Erkennung von am Bildrand befindlicher Objekte genutzt werden. In Abbildung 11 beträgt das *padding* somit einen Pixel.

Nun lässt sich eine Formel herleiten, um zu berechnen, wie groß die *feature map* bei gegebener Kernelgröße, *stride*- und *padding*-Anzahl und gegebener Bildgröße ist. (Brilliant.org, 2018c)

Sei $\mathbf{A} \in M^{k \times q}(\mathbb{R})$ die Kernel-Matrix, $\mathbf{B} \in M^{m \times n}(\mathbb{R})$ die Bildmatrix, $p \in \mathbb{N}$ das *padding* und $s \in \mathbb{N}$ der *stride*, dann ist die *feature map* eine

$$\left(\frac{m - k + 2p}{s} + 1 \right) \times \left(\frac{n - q + 2p}{s} + 1 \right) \text{ Matrix.} \quad (32a)$$

Bisher wurde immer davon ausgegangen, dass nur ein Kernel über die Bildmatrix gleitet, um eine *feature map* zu berechnen. Tatsächlich gleiten aber viele Kernel gleichzeitig über die Bildmatrix, da ein Kernel immer nur ein bestimmtes Muster im Bild entdecken kann. Für den häufigen Fall eines RGB-Bildes als Eingabebild, müssen drei Kernel über je einen Farbkanal gleiten, wie später in Kapitel 5 auch praktisch dargestellt. Es können bei einem RGB-Bild aber auch durchaus mehr Kernel als drei genutzt werden.

Da es nun offenbar $p k \times q$ -Kernel-Matrizen in einer *convolutional*-Schicht geben kann, muss es auch p *feature maps* geben. Somit lässt sich eine *convolutional*-Schicht als dreidimensional beschreiben, was sich auch auf die praktische Implementierung in Kapitel 5 auswirkt. In Abbildung 12 ist eine solche dreidimensionale Schicht abgebildet.

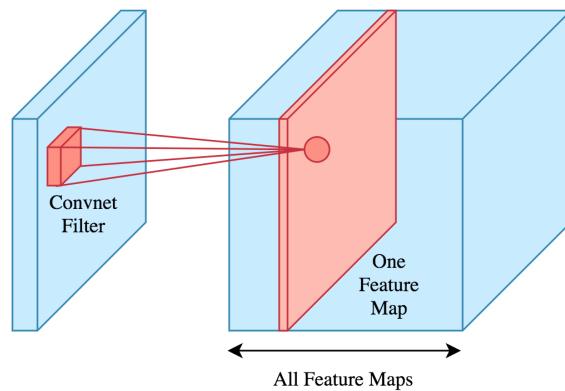


Abb. 12: Zu sehen ist, dass ein Kernel, hier Filter genannt, über die Bildmatrix wandert und in eine *feature map* schreibt. Dabei gibt es aber mehrere *feature maps*, die jede einen eigenen Kernel haben. Die *feature maps* lassen sich dann hintereinander aufgereiht veranschaulichen. Bildquelle: (Brilliant.org, 2018c).

Convolutional-Schichten sind sehr rechenintensiv. Unter anderem deshalb werden sogenannte *pooling*-Schichten genutzt, die die *feature maps* komprimieren.

2.3.2 Die *pooling*-Schicht

Eine *pooling*-Schicht wird nach einer *convolutional*-Schicht genutzt, um die berechneten *feature maps* zu komprimieren und aus diesen die wichtigsten Informationen zu extrahieren. (Buduma & Lacascio, 2017, S. 98 f.)

Formell lässt sich das *pooling* folgendermaßen beschreiben: Zunächst wird eine *feature map* in $n \times n$ Matrizen aufgeteilt, also Matrizen mit gleicher Anzahl an Zeilen und Spalten. Beim *pooling* soll für jede dieser $n \times n$ Matrizen eine $m \times m$ Matrix berechnet werden, für die $m < n$ gilt. Häufig, wie beim *average*- oder *max-pooling*, ist $m = 1$. In diesem Fall berechnet sich die Dimension aus der *pooling*-Schicht entstehende *feature map* durch

$$\left(\frac{k-n}{s} + 1 \right) \times \left(\frac{k-n}{s} + 1 \right), \quad (33a)$$

wobei eine $k \times k$ *feature map* als Eingabebildmatrix angenommen wird. Die $n \times n$ Matrix wird manchmal auch *pooling*-Filter genannt.

Nachdem eine *feature map* also in $n \times n$ Matrizen aufgeteilt wurde und mit $m = 1$ die Dimension *feature map*, die aus dem *pooling* entsteht, berechnet wurde, soll nun erklärt werden, wie die $n \times n$ Matrizen auf entsprechende $m \times m$ Matrizen abgebildet werden können.

Diese Abbildung kann durch das *average-pooling* oder das *max-pooling* geschehen. Das *average-pooling* errechnet sich aus

$$z = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n q_{ij} \quad (34a)$$

einer $n \times n$ Matrix \mathbf{Q} , also dem arithmetischen Mittel der Matrixeinträge.

Das *max-pooling* errechnet sich aus

$$z = \max(\mathbf{Q}) \quad (35a)$$

einer $n \times n$ Matrix \mathbf{Q} , also der Ermittlung des Maximalwertes aller Matrixeinträge. Das *max-pooling* wird in Kapitel 5 auch praktisch angewandt. Abschließen soll die Ausführungen zum *pooling* eine Abbildung, die die oben genannte Gleichung zum *max-pooling* visuell-intuitiv zusammenfasst.

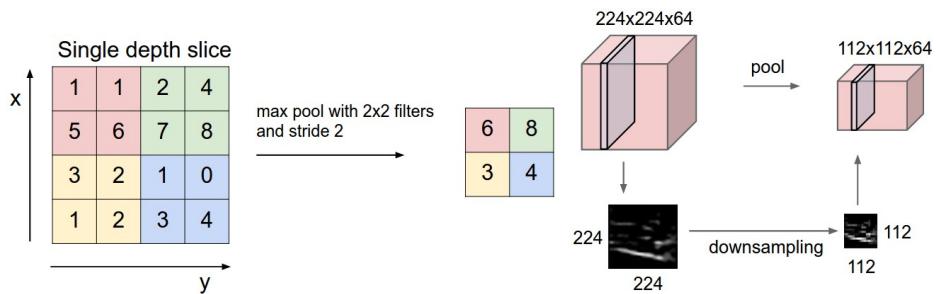


Abb. 13: Links zu sehen ist ein *max-pooling*-Prozess mit einem 2×2 *pooling*-Filter. Rechts zu sehen ist ein *pooling*-Prozess mit 64 *feature maps*. Bildquelle: (Li, FeiFei et. al., 2018a).

2.4 Weiteres zu neuronalen Netzen

Nachdem nun die theoretischen Grundlagen von neuronalen Netzen hergeleitet wurden, sollen an dieser Stelle noch einige Konzepte, die für die praktische Implementierung von neuronalen Netzen wichtig sind, kurz erklärt werden.

2.4.1 Generalization, overfitting und underfitting

Unter der *generalization* eines lernenden Algorithmus wird seine Fähigkeit zur erfolgreichen Adaption der auf der Trainingsmenge $X = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ erlernten Abbildung $h_{\mathbf{w}, b}(\mathbf{x}_i) \approx y_i$ auf neue, unbekannte Daten verstanden. (Buduma & Lacascio, 2017, S. 29 f.) Wird festgestellt, z.B. durch Genauigkeitsmetriken

auf Testdaten, dass die Fähigkeit zur Adaption nicht ausgeprägt vorliegt, kann der Grund dafür das *overfitting* oder das *underfitting* sein. Beim *overfitting* passt sich der Algorithmus so gut an die Trainingsdaten an, dass er die Abbildung $h_{w,b}(x_i) \approx y_i$ auswendig lernt, somit die wichtige Fähigkeit abstrahieren zu können, verliert. (Brilliant.org, 2018d) Beim *underfitting* passiert genau das Gegenteil: Die Abbildung wird nur schlecht erlernt, was ebenfalls zu schlechten Ergebnissen auf neuen Daten führt.

Bei neuronalen Netzen kann insbesondere die Anzahl an Neuronen und die Anzahl der Schichten erheblichen Einfluss auf das *over-* und *underfitting* haben. Dabei stehen benötigte Komplexität und Generalisierbarkeit der Abbildung in ständiger Abwägung. (Buduma & Lacascio, 2017, S. 30)

Under- and Over-fitting examples

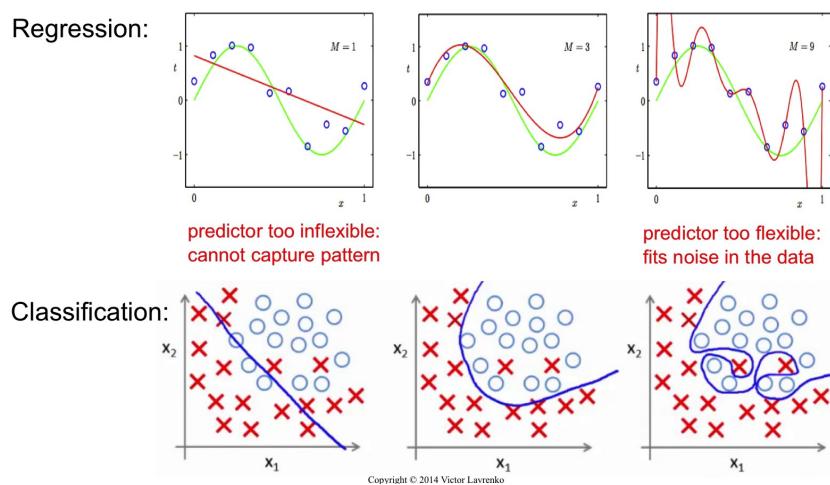


Abb. 14: Links dargestellt sind Beispiele von einem lernenden Algorithmus, der die Daten *underfitted* hat. Hier wäre angebracht, z.B. die Anzahl der Neuronen pro Schicht zu erhöhen, das neuronale Netz mit mehr Schichten auszustatten, oder das Netz mit mehr Epochen oder mehr/unterschiedlicheren Daten trainieren zu lassen. In der Mitte ist dargestellt, wie die Daten optimal *ge-fitted* sind. Rechts ist dargestellt, wie es aussieht, wenn die Daten *overfitted* sind. Hier wären Maßnahmen entgegen denen vom *underfitting* zu treffen. Die grüne Linie in der oberen Abbildungen gibt die echte Eingabe-Ausgabe-Beziehung an, während die rote Linie die angelernte Beziehung angibt. Bildquelle: (Lavrenko, Victor et. al., 2018).

Eine Möglichkeit *over-* und *underfitting* während des Trainings zu erkennen sind sogenannte Validierungsdaten. Auf diese wird in Kapitel 3.1.5 noch ausführlicher eingegangen. In Kapitel 6.2.1 und Kapitel 6.2.2 werden Hypothesen getestet, die für das *over-* und *underfitting* relevant sind.

2.4.2 Die Varianten des *gradient descent*-Algorithmus

Prinzipiell lässt sich der *gradient descent*-Algorithmus in drei Varianten einteilen. (Buduma & Lacascio, 2017, S. 25 ff.) Zwei davon wurden bereits in Kapitel 2.2.1 genannt, *batch gradient descent* und *stochastic gradient descent*. Hinzu kommt jetzt noch *mini batch gradient descent*.

Bei der *batch gradient descent*-Variante wird der Fehler über die gesamte Trainingsmenge X in die Gewichtsanpassung des i -ten Iterationsschrittes einbezogen. Die Anpassung des Gewichts w_j mit *batch gradient descent* ist mit Gleichung 11 hergeleitet und beschrieben worden. Nachteil von *batch gradient descent* ist die Anfälligkeit, Sattelpunkte der Fehlerfunktion als Minima zu deuten, da die Steigung des Gradienten im Sattelpunkt ebenfalls null ist. (Buduma & Lacascio, 2017, S. 26)

Die *stochastic gradient descent*-Variante betrachtet immer nur ein $(x_i, y_i) \in X$ für die Gewichtsanpassung in einer Iteration. Dadurch wird keine starre Fehlerebene, wie in Abbildung 8 dargestellt, betrachtet und Sattelpunkte können vermieden werden. (Buduma & Lacascio, 2017, S. 26 f.) Die *stochastic gradient descent*-Variante von Gleichung 11 ist $\frac{\partial E}{\partial w_{ji}} = (\hat{y}_i - y_i)g'(z)x_i$, da das Gewicht immer nur im Bezug zu einem Trainingstupel $(x_i, y_i) \in X$ angepasst wird, die Summe also wegfällt. Nachteil dieser Variante ist, dass weniger Informationen über die Fehlerebene pro Iteration zur Verfügung stehen, also das Finden eines Minima länger andauern kann. (Buduma & Lacascio, 2017, S. 27)

Einen Mittelweg dieser beiden Varianten ist der *mini batch gradient descent*. Diese Variante passt die Gewichte nach k Trainingstupel aus der Trainingsmenge an, verbindet damit die Vor- und Nachteile von *batch gradient descent* und *stochastic gradient descent*. Formell bedeutet dies, dass ein Fehlermaß über k Trainingstupel

summiert wird mit $1 < k < n$. Aus Gleichung 11 wird beim *mini batch gradient descent* dann

$$\frac{\partial E}{\partial w_j} = \frac{1}{k} \sum_{i=1}^k (\hat{y}_i - y_i) g'(z) x_i. \quad (36a)$$

Ist $k = 1$, dann wird ein *stochastic gradient descent* durchgeführt, ist $k = |X|$, dann wird ein *batch gradient descent* durchgeführt. (Buduma & Lacascio, 2017, S. 27)

Einen bestmöglichen Wert für k im *mini batch gradient descent*-Verfahren für den Anwendungsfall der Bildklassifikation zu finden, ist Gegenstand einer Hypothese in Kapitel 6.2.3.

2.4.3 Adaptive Lernratenanpassung

In den Gleichungen 4 oder 14 wird stets die Lernrate α genannt, die den negativen Gradienten im Gradientenabstiegsverfahren skalieren kann. Ein Gradientenabstieg mit einer zu kleinen Lernrate sorgt dafür, dass ein neuronales Netz nur langsam lernt und viele Iterationen zur Findung eines Minimas benötigt. Ein Gradientenabstieg, bei dem die Lernrate zu groß gewählt ist, wird Minima nicht systematisch ansteuern können oder gar überspringen. Aus diesem Grund wird eine adaptive Lernrate gefordert, die sich während des Trainings verändern kann, sodass der Gradientenabstieg möglichst gut zu Minima findet. (Buduma & Lacascio, 2017, S. 23 f.) Zwei dieser sogenannten Optimierungsalgorithmen (engl. optimizer) zur Berechnung einer adaptiven Lernrate sind für den vorliegenden Anwendungsfall relevant: *Adadelta* und *Adam*. In Kapitel 6.2.5 sind die empirischen Ergebnisse von *Adadelta*- und *Adam*-Optimierungsalgorithmen in einer Hypothese gegenübergestellt. Aus Platzgründen wird auf eine mathematische Herleitung dieser Algorithmen an dieser Stelle verzichtet und auf die umfangreichen Ausführungen in (Buduma & Lacascio, 2017, S. 78 ff.) verwiesen.

3 Bildung der Datengrundlage

Nachdem in Kapitel 2 die theoretischen Grundlagen von neuronalen Netzen vorgestellt wurden, wird in diesem Kapitel die praxisorientierte Vorgehensweise zur Gewinnung von Bilddaten für die Fußballspielerklassifikation dargestellt. Dabei wird in der Folge erklärt, wie die Daten gewonnen und passend aufbereitet werden.

3.1 Datengewinnung für den Anwendungsfall

Bevor ein neuronales Netz seiner eigentlichen Kernaufgabe, in diesem Fall eine Bildklassifikation, nachgehen kann, muss zunächst eine Daten-*Pipeline* aufgestellt werden. Diese sorgt dafür, dass ausreichend und vor allem richtige Daten für das Netz zum Lernen zur Verfügung stehen. Speziell im vorliegenden Anwendungsfall war es äußerst schwer, valide Daten zu erhalten, da es keinen vorgefertigten Datensatz gibt, der fertig-formatierte Bilder von Fußballspielern zur Verfügung stellt. Dies ist für typische Klassifikationsbeispiele, wie der Erkennung von Verkehrsschildern, anders. Somit war zunächst die Herausforderung gegeben, dass die Daten-*Pipeline* nicht wie üblich bei der Transformierung oder dem Abändern von verfügbaren Daten startet, sondern zuerst das Gewinnen von validen und verwertbaren Daten nötig ist. Bevor jedoch auf die einzelnen Schritte der verwendeten Daten-*Pipeline* eingegangen wird, soll diese zuerst kurz vorgestellt werden.



Abb. 15: Dargestellt ist die verwendete Daten-*Pipeline* für die Generierung der benötigten Daten.

Die verwendete *Pipeline* besteht aus sechs Schritten. Die ersten vier Schritte stellen die Grundlage für die Gewinnung der benötigten Bilddaten dar. Die letzten beiden Schritte bilden das Aufteilen und Konnotieren der Daten, damit ein überwachtes Lern-Szenario gebildet werden kann. Da, wie in den folgenden Unterkapiteln noch dargestellt, die Bilddaten aus einem anderen Algorithmus beziehen,

entfallen die Schritte der Datensäuberung und der Datenreduktion. In der Praxis ist oftmals der umgekehrte Fall der Standard. Das bedeutet, dass eine große Menge an Daten für einen noch nicht scharf formulierten Anwendungsfall zur Verfügung steht. Daraus resultiert, dass die große Menge an Daten zuerst untersucht, reduziert oder transformiert werden muss. Im vorliegenden Fall mussten die Daten ebenfalls weiterverarbeitet werden, jedoch war keine Reduktion oder ein komplettes Verändern der Daten notwendig. Deshalb wird in diesem Beispiel der Begriff *data preprocessing* allgemein für den Prozess der Datenverarbeitung verwendet.

3.1.1 Geeignetes Bildmaterial finden

Wie in der Pipeline dargestellt wurde, ist die notwendige Bedingung, um ein neuronales Netz zu erstellen, das Fußballspieler klassifizieren kann, genügend qualitativ hochwertiges Bildmaterial zur Verfügung zu haben. In diesem konkreten Fall standen für die Gewinnung des benötigten Bildmaterials zwei Optionen zur Verfügung: Zum einen das Herunterladen von Fußball-TV-Übertragungen vom Videoportal Youtube, zum anderen die Generierung durch das Simulieren von Fußballspielen mit dem Computerspiel Fifa 17 von Electronic Arts. Bevor darauf eingegangen wird, welche Variante für den vorliegenden Anwendungsfall verwendet wird und welche Gründe dafür ausschlaggebend waren, wird zunächst ein direkter Vergleich zwischen den beiden Varianten dargestellt.

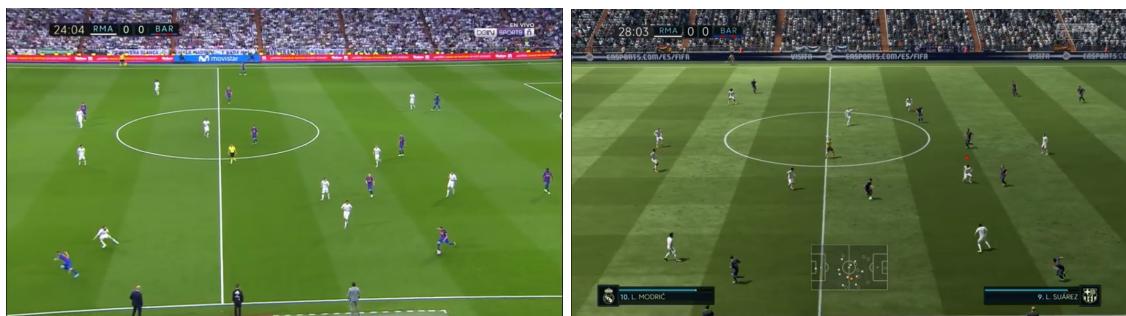


Abb. 16: Dargestellt ist eine Spielsituation aus dem Spiel Real Madrid gegen den FC Barcelona im Santiago Bernabéu (TV-Bild(l.) und Simulation(r.)).

Werden die beide Bilder in Abbildung 16 betrachtet, so fallen dem neutralen Betrachter keine wirklichen gravierenden Unterschiede auf. Lediglich die Farbstärke

und Belichtung ist in der Realität etwas kräftiger als auf dem gegenübergestellten Bild der Simulation. Wobei angemerkt werden muss, dass dies bei der Simulation auch oft von der verwendeten Hardware (CPU, GPU, RAM, usw.) abhängt. Wird von diesen marginalen Unterschieden abgesehen, so ist die Ähnlichkeit zwischen beiden Bildern sehr auffällig und direkt ersichtlich. Aus dieser Gegenüberstellung resultieren die folgenden Kriterien, die für die Auswahl der Variante ausschlaggebend waren:

Verfügbarkeit	Menge in der das Bildmaterial verfügbar ist
Reproduzierbarkeit	Möglichkeit das Bildmaterial zu vervielfältigen
Variabilität	Möglichkeit das Bildmaterial zu verändern

Bezieht man diese Kriterien in die Entscheidung zwischen beiden Varianten mit ein, so ergibt sich eine klare Tendenz zur Verwendung der Daten der Fußballsimulation Fifa 17. Dies resultiert daraus, dass durch Fifa 17 die Möglichkeit besteht, sowohl den Kamerawinkel als auch die Darstellung der Spielsituationen realitätsnah nachzustellen. Ein weiterer Vorteil durch Fifa 17 ist der hohe Grad an Variabilität, der sich durch die unterschiedlichen Einstellungsmöglichkeiten ergibt. Das bedeutet, dass es in Fifa 17 möglich ist, die unterschiedlichen Mannschaften in unterschiedlichen Trikots, Stadien, bei unterschiedlichen Lichtverhältnissen und unter Verwendung verschiedener Kamerawinkel antreten zu lassen. Dazu ist es durch die ständige Verfügbarkeit möglich, beliebig viele Aufnahmen in Fifa 17 zu erstellen. Dem gegenüber stehen die wenigen frei zur Verfügung stehenden Spielaufzeichnungen auf Youtube. Dies resultiert vor allem daraus, dass die Bilder rechtlich geschützt sind, durch die für den Vertrieb zuständigen Pay-TV-Sender wie z.B. SKY oder Eurosport. Außerdem ist es auch nicht möglich Spielaufzeichnungen variabel zu gestalten, da die oben genannten Parameter wie Kamerawinkel, Trikots, Stadien usw. nicht anpassbar sind. Auf Basis der Variabilität, Reproduzierbarkeit, Verfügbarkeit und der Ähnlichkeit zur Realität wurde sich für dieses Experiment für die Fifa 17-Variante als Grundlage für das Bildmaterial entschieden.

3.1.2 Bildmaterial vorverarbeiten mit YOLO

Nachdem die Grundlage der Datenerhebung dargestellt und begründet wurde, wird in diesem Punkt auf die Vorverarbeitung der generierten Bilder aus Fifa 17 eingegangen. Die Basis der verwendeten Bilddaten bilden in Fifa 17 aufgenommene Screenshots wie in Abbildung 17 dargestellt.



Abb. 17: Dargestellt ist ein beispielhafter Screenshot aus Fifa 17 als Datengrundlage.

Von dieser Art Screenshot wurden mehrere hundert Stück aufgenommen. Jedoch fällt auf, dass die Bilder in diesem Format für das spätere neuronale Netz nutzlos sind, da es später die Aufgabe hat zu bestimmen, ob auf einem Bild genau ein Fußballspieler vorhanden ist oder nicht. Dies ist, wie in Kapitel 1.1 erklärt, eine Bildklassifikation. Das bedeutet im Umkehrschluss, dass nur Bilder genutzt werden können, auf denen entweder genau ein Spieler vorhanden ist oder kein Spieler. Daraus resultiert die Frage, wie es möglich ist, den vorhandenen, manuell gesammelten Datensatz so zu verarbeiten, dass nach der Verarbeitung aus einem großen Bild, wie bspw. in Abbildung 17, alle darin enthaltenen Spieler als ausgeschnittene, separate Bilder extrahiert werden können.

Auch für diese Aufgabenstellung standen zwei Varianten zur Auswahl: Zum einen kann der Ausschnitt der Spieler händisch über ein Bildverarbeitungsprogramm erfolgen. Da dies jedoch zeitlich sehr aufwendig ist, wäre dies nicht zielführend. Dabei ist eine Bedingung zur Nutzung von mehrschichtigen neuronalen Netzen, wie sie Kapitel 2.2.2 erläutert wurden, gerade eine möglichst große Ansammlung von Trainingsdaten zu haben. Dies wäre mit dieser Art von manuellen Aufwand

zwar möglich gewesen, jedoch nicht praktikabel. Die zweite Variante bestand darin, einen bereits vorhandenen Objekterkennungsalgorithmen als Datensatz-Generator zu verwenden. Die Idee die sich dahinter verbirgt, ist, dass der Algorithmus alle Spieler auf dem Bild erkennt und mit einer sogenannten *bounding box* umgrenzt, wie in Kapitel 1.1 auch erklärt. Anschließend wird der Algorithmus so modifiziert, dass die angegebenen *bounding boxen* als Rahmen für das auszuschneidende Objekt, hier einen Spieler, genutzt werden. Das bedeutet in diesem Fall, dass durch den Objekterkennungsalgorithmen eine valide Datengrundlage zum Trainieren des neuronalen Netzes automatisiert sichergestellt und bereitgestellt werden kann.

Damit diese Idee auch in die Realität übertragen werden kann, muss zunächst ein leistungsstarker, lauffähiger Algorithmus für die Objekterkennung gefunden werden. Für dieses Experiment wird der Algorithmus You Only Look Once (YOLO) verwendet. YOLO ermöglicht eine Objekterkennung in Echtzeit und ist ein vortrainierter Algorithmus der auf mehr als 20 Objektklassen antrainiert wurde. Dies umfasst sowohl Personen, als auch andere Lebewesen oder Gegenstände. Jedoch ist die Erkennung von Fußballspielern, also Personen, in der vorliegenden Arbeit gefordert, wodurch YOLO sich als optimaler Lösungsweg zur automatischen Generierung von den geforderten Daten herausstellt. Wird das in Abbildung 17 dargestellte Bild als exemplarischer Screenshot, für die zur Verfügung stehenden Bilder aus Fifa 17 angesehen, so würde nach der Bearbeitung des Bildes mit YOLO das folgende Bild resultieren:



Abb. 18: Dargestellt ist ein beispielhafter Screenshot aus Fifa 17 nach einem Durchlauf von YOLO.

Somit liefert YOLO die Grundlage, um den geforderten Datensatz, bestehend aus Bildern mit einem Fußballspieler pro Bild, zu generieren. Dies wird durch die von YOLO erstellten *bounding boxen* deutlich, die somit die für den Datensatz relevanten Bildinhalte markieren und eingrenzen. Dadurch sind diese Koordinaten der *bounding box* Anhaltspunkt für die Lokalisierung eines Fußballspielers im Bild. Im folgenden Unterkapitel wird dargestellt, wie diese *bounding box* zu extrahieren und zu speichern ist.

3.1.3 Abwandlung von YOLO

Wie im vorherigen Unterkapitel dargestellt, wurde YOLO zur Erkennung und Eingrenzung von Spielern auf einem Fifa 17-Screenshot verwendet. Die Herausforderung, die es nach der Erkennung und Eingrenzung aller auf dem Bild befindlicher Spieler zu lösen galt, war es, die nun erkannten Spieler in dem richtigen Format und der richtigen Größe auszuschneiden, sodass aus den von YOLO erkannten Spielern Einzelbilder der Fußballspieler entstehen. Als Veranschaulichung kann Abbildung 19 betrachtet werden. So soll nach einer Anpassung des YOLO-Algorithmus ein Spieler als solcher erkannt, ausgeschnitten und abgespeichert werden.



Abb. 19: Dargestellt ist ein beispielhafter ausgeschnittener Spieler aus einem kompletten Bild nach der Abwandlung von YOLO.

Damit YOLO diese Funktionalitäten abbilden kann, muss der zur Verfügung stehende YOLO-Source-Code angepasst werden. Diese Anpassung geschieht in der Datei `yolo.py` und maßgeblich in der Methode `detectimage()`. Innerhalb der Methode `detectimage()` bearbeitet YOLO jedes einzelne Bild, dass zur Verarbeitung

übergeben wurde. Anschließend generiert YOLO für jedes erkannte Objekt innerhalb der Methode ein Array mit den *bounding box*-Koordinaten für die erkannten Objekte. Die erstellten Koordinaten werden anschließend genutzt, um diese auf das Bild in Form eines roten Rechtecks zu übertragen. Genau hier ist der Punkt an dem YOLO modifiziert werden muss. Dadurch, dass in einer `for`-Schleife jede *bounding box* separat bearbeitet wird, eignet sich dieser Punkt im Source-Code, um die einzelnen *bounding boxen* und deren Inhalt zu bearbeiten. Dies geschieht vorrangig mit der Methode `boxtocropp()` die in Listing 1 dargestellt ist.

```

1 top, left, bottom, right = box
2 top = max(0, np.floor(top + 0.5).astype('int32'))
3 left = max(0, np.floor(left + 0.5).astype('int32'))
4 bottom = min(image.size[1], np.floor(bottom + 0.5).astype('
    int32'))
5 right = min(image.size[0], np.floor(right + 0.5).astype('int32'
    ))
6 ims.box_to_cropp(image, top, left, bottom, right, counter)

```

Listing 1: Methode `boxtocropp`

Diese Methode bekommt das jeweilige komplette Bild zum Beschneiden und die für das jeweilige Objekt erkannten *bounding box* Koordinaten in Form von den Tupellementen `top`, `left`, `bottom` und `right` übergeben. Dazu wird noch ein hochgezählter Index als Zählervariable übergeben, um die Spielerbilder fortlaufend zu nummerieren. Die aufgerufene Methode befindet sich in einer separaten Datei `ImageSampler.py`, in der die Logik für das Ausschneiden und das Abspeichern der Bilder abgelegt ist. Wurden die Werte von dem zu bearbeitenden erkannten Objekt übergeben, wird zuerst das übergebene Bild eingelesen, um somit eine Bearbeitung und Beschneidung des Bildes zu ermöglichen. Wurde das Bild erfolgreich eingelesen, werden die übergebenen Koordinaten der *bounding box* als Vorlage verwendet und ermöglichen somit ein genaues Ausschneiden des erkannten Objektes.

Jedoch muss beachtet werden, dass die ausgeschnittenen Bilder nicht alle in der gleichen Auflösung (*Bildbreite* \times *Bildhoehe*) ausgeschnitten werden, da sich die Größe der von YOLO erstellten *bounding boxen* unterscheiden kann. Deshalb müssen die ausgeschnittenen Bilder auf die gleiche Größe zugeschnitten werden. Deshalb wurden für den Anwendungsfall dieser Arbeit die Spielerbilder in das

```

1 img = cv2.imread(image.filename)
2 ...
3 crop_img = img[top:bottom, left:right]
4 ...
5 crop_img = cv2.resize(crop_img, (48, 48))
6 ...
7 cv2.imwrite(os.path.join(result_path, 'player' + str(counter) +
    '.jpg'), crop_img)

```

Listing 2: Ablauf der Klasse

Auflösungsformat $48px \times 48px$ zugeschnitten. Diese Größe wurde gewählt, da diese garantiert, dass alle erkannten Spieler aus den *bounding boxes* erfolgreich ausgeschnitten werden. Würde die Größe noch kleiner gewählt werden, so würde dies dazu führen, dass die Spieler nicht vollständig ausgeschnitten worden wären und bei einer zu großen Größe würden unnötige Informationen oder Überschneidungen mit nah angrenzenden Mitspielern die Folge sein. Die festgelegte Größe ist vor allem für das später erstellte neuronale Netz notwendig, da die Bilder in einer einheitlichen Auflösung vorhanden sein müssen. Wurde das Bild erfolgreich ausgeschnitten und auf die Auflösung $48px \times 48px$ gebracht, wird dieses in einem für die ausgeschnittenen Spieler angelegten Ordner abgelegt. In diesem Ordner befinden sich alle erkannten und ausgeschnittenen Spieler.

3.1.4 Bildinformationen extrahieren

In den vorherigen Punkten wurde dargelegt, auf welche Weise die benötigte Datengrundlage beschafft und generiert wird. Abschließend müssen die über den YOLO-Algorithmus erkannten und ausgeschnittenen Spielerbilder jedoch noch weiter verarbeitet werden. Doch bevor auf die Weiterverarbeitung detailliert eingegangen wird, wird zuerst ein einzelnes Spielerbild und die Informationen, die dieses für das neuronale Netz liefert, näher betrachtet.

Die Auflösung des Bildes gibt in erster Linie Aufschluss über die enthaltene Menge an Informationen. In diesem Kontext sind Informationen gleichzusetzen mit der pro Bild zur Verfügung stehenden Anzahl von Pixeln. Diese ergibt sich durch die Multiplikation der Bildhöhe mit der Bildbreite. Das bedeutet, dass für die

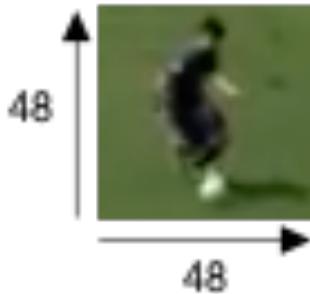


Abb. 20: Dargestellt ist ein Beispielhaftes Spielerbild in den Maßen $48px \times 48px$.

vorhandenen Spielerbilder pro Bild 2304 Pixel an Informationen über das Bild zur Verfügung. Dies ist vor allem deshalb wichtig, da diese Pixel die Eingabe des neuronalen Netzes sind. Dies resultiert daraus, da ein neuronales Netz nicht in der Lage ist, ein Bild wie ein Mensch anzusehen. Im Gegenteil, ein neuronales Netz sieht wie in diesem Fall ein Bild durch die übergebenen Pixel und die damit verbundenen Farbwerte in dem RGB-Modell. In dieser Arbeit werden die gerade beschriebenen Pixel als Eingabe verwendet. Wie oben beschrieben liefert jeder der 2304 Pixel eines Spielerbildes Informationen über das Bild. Dabei liefert jeder Pixel drei Farbwerte aus dem RGB-Modell wie in Abbildung 21 dargestellt.

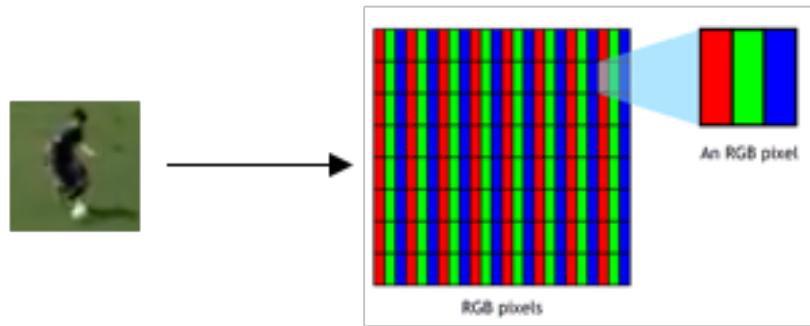


Abb. 21: Dargestellt ist eine beispielhafte Umwandlung eines Spielerbildes in RGB-Werte

Jeder Wert in dem RGB-Modell kann einen Wert zwischen 0 – 255 annehmen, da RGB-Farben mit 8-Bit kodiert sind. Somit ergeben sich für das neuronale Netz als

Eingabe jeweils 2304 unterschiedliche RGB-Kombinationen, die durch die unterschiedlichen Farbwerte und der Pixelposition den Computer das Bild wahrnehmen lassen. Für das neuronale Netz bedeutet dies, dass es eine Kombination von 2304 unterschiedlichen RGB-Werten als Eingabe erhält und auf Basis dieser unterschiedlichen Werte erlernen soll, ob sich auf dem Bild ein Spieler befindet oder nicht. Damit die Informationen der einzelnen RGB-Werte der Pixel in einem Bild genutzt werden können, muss jedes Bild zuvor mit der folgenden Python Methode `loadImage()` aufgerufen werden.

```
1 loadImage(image)
```

Listing 3: Methode Load Image

Der Rückgabewert dieser Methode sind die einzelnen RGB-Werte für jeden Pixel in einem Array. Die komplette Rückgabe für ein einzelnes Bild ist in dem Listing 4 beispielhaft dargestellt.

```
1 [ 54.  73.  17.]
2 [ 54.  73.  17.]
3 [ 55.  74.  18.]
4 ...
5 [ 57.  71.  20.]
6 [ 53.  70.  18.]
7 [ 52.  69.  17.]
```

Listing 4: Beispielhafte RGB-Werte

Nachdem diese Schritte erfolgt sind, kann anschließend das Konnotieren und das Aufteilen der Bilddaten in Trainings- und Testdaten betrachtet werden.

3.1.5 Aufteilung in Trainings- und Testdaten

Der letzte Schritt, bevor das neuronale Netz trainiert und getestet werden kann, bildet das Aufteilen der generierten Daten in einen Trainings- und Testdatensatz. Das bedeutet, dass der komplette Datenbestand in zwei Mengen aufgeteilt wird. Die erste Menge sind die Trainingsdaten. Diese werden genutzt, um dem neuronalen Netz in der Trainingsphase die geforderte Funktionalität, in diesem Fall das

Erkennen von Spielern auf einem Bild beizubringen, siehe Kapitel 2.2.2. Die zweite Menge sind die Testdaten. Die Testdaten dienen der Validierung der erlernten Fähigkeit. In diesem Fall wird, nachdem das neuronale Netz mit den Trainingsdaten angelernt wurde, überprüft, ob es wirklich in der Lage ist zu erkennen, ob sich auf einem Bild ein Spieler befindet oder nicht. Durch die dem Netz unbekannten Testdaten kann somit festgestellt werden, ob das neuronale Netz erfolgreich die Fähigkeit zur Erkennung von Fußballspielern erlernt hat oder ob es diese Fähigkeit nur auf bereits bekannten Daten hat. Dies wäre ein klassisches Beispiel von *underfitting*, siehe Kapitel 2.4.1. Die generelle Aufteilung des kompletten Datensatzes kann in Abbildung 22 betrachtet werden.

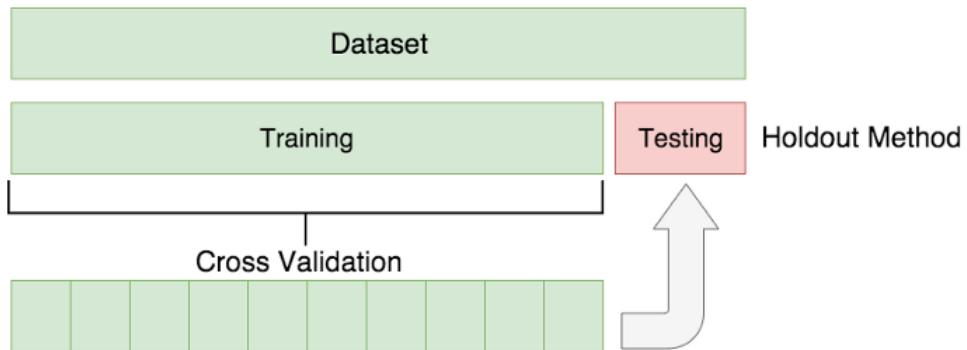


Abb. 22: Dargestellt ist eine beispielhafte Aufteilung des Datensatzes in Trainigs- und Testdaten. Bildquelle: (Bronshtein, Adi, 2017).

Dieser sogenannte Trainings-Test-*split* bedeutet, dass pro von YOLO bearbeiteten Bild eine festgelegte Anzahl von erkannten Spielerbildern in den Trainingsdatensatz abgelegt werden und eine kleinere Anzahl von Spielerbildern in den Testdatensatz abgelegt werden. Dadurch wird die Generierung des Trainings- und Testdatensatzes gewährleistet und automatisch mit der Spielerbild Erstellung über YOLO gekoppelt. Außerdem wird so die Einhaltung einer angemessenen Verteilung der Daten auf den Trainings- und Testdatensatz gewährleistet. Wie aus Abbildung 22 entnommen werden kann, sollte die Trainingsmenge immer größer als die Testmenge sein. In dem Trainingsprozess werden die Testdaten in gleichgroße Untermengen unterteilt und validiert.

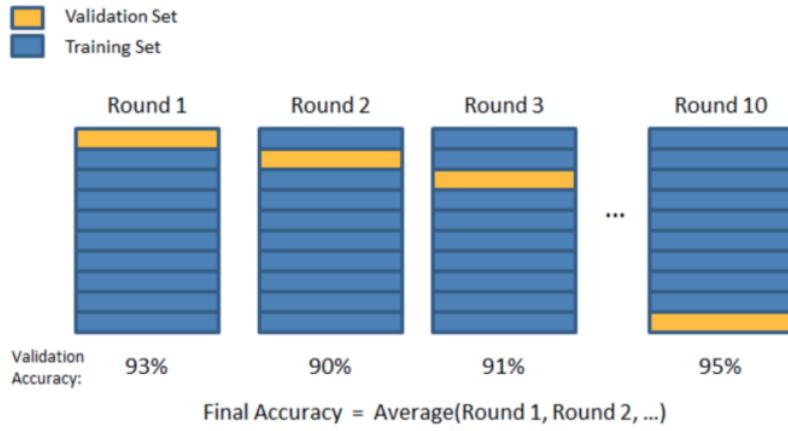


Abb. 23: Dargestellt ist ein beispielhafter Ablauf der *cross-validation*-Methode. Bildquelle: (McCormick, Chris, 2013).

3.1.6 Konnotieren der Daten

Abschließend soll in diesem Unterkapitel das Konnotieren, also die Zuweisung von einer Objektklasse zu einem Bild, des gesammelten Datensatzes dargelegt werden. Wie in den obigen Punkten vorgestellt, besteht der verwendete Datensatz aus mehreren tausend Spielerbildern der Auflösung $48px \times 48px$. Wird dazu das binäre Klassifikationsmodell von 1 für Spieler und 0 für kein Spieler betrachtet, so fällt auf, dass für die ordnungsgemäße Vorverarbeitung der Daten, noch das benötigte Konnotieren der Daten fehlt. Das Konnotieren der Daten wurde für die Spielerbilder fortlaufend und mit der Konvention PlayerXXX.jpg gehandhabt. Das bedeutet, dass jeder Spieler den Namen und einen Index in dem Dateinamen trägt. Das ermöglicht die spätere Generierung der Objektklassenzuweisung für die Trainingsdaten. Wird jedoch das gerade erwähnte binäre Klassifikationsmodell betrachtet so fällt auf, dass zwar ausreichend Bilder von Fußballspielern vorhanden sind und diese trainiert und getestet werden können, jedoch bis jetzt noch keine Möglichkeit besteht, die Komplementärklasse (hier: 0 = Kein Spieler) zu trainieren. Um dieses Problem zu lösen wird der Anwendungsfall nochmal betrachtet. Das spätere neuronale Netz soll die Anforderung erfüllen, auf einem Bild einen oder keinen Fußballspieler zu erkennen. Dazu muss es aber auch in der Lage sein und gelernt haben, wie das Gegenteil eines korrekten Beispiels aussieht. Betrachtet man das Komplementärobject, so würde dies ein Bild sein, dass nur Rasen enthält

oder die Kombination aus Rasen und Linien. Das bedeutet, dass die Klasse 0 (kein Spieler) aus allen Bildern besteht die entweder nur Rasen, Rasen + Linien, Rasen + Banden oder Zuschauern enthält. In Abbildung 24 sind zwei Bilder dargestellt. Zum einen die Erkennung für die benötigte Klasse 1 und die Erkennung für die benötigte Klasse 0.

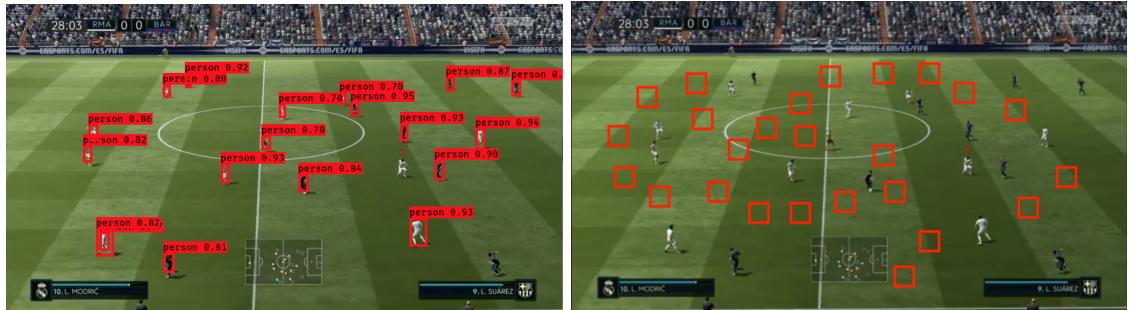


Abb. 24: Gegenüberstellung TV-Bild(l.) und Simulation(r.).

Das Problem das sich für die Generierung der Klasse 0-Bilder herausgestellt hat war, dass die in Punkt 3.1.2 und 3.1.3 vorgestellte Methode für die Generierung der Klasse 1-Bilder in Kombination mit YOLO für diesen Fall nicht anwendbar war. Dies resultiert aus der Verwendung von YOLO zum Erkennen von bestimmten Objekten, da Rasen oder Linien aber keine vortrainierten Objekte in dem YOLO Algorithmus darstellten, musste die Generierung der Klasse 0 Bilder händisch erfolgen. Es wäre zwar möglich gewesen auf Basis der erkannten Spieler, die Komplementärklasse durch verschieben der *bounding boxes* zu generieren, jedoch führte dies oftmals dazu, dass Teile von Spielern, Banden oder Zuschauern mit in den Bildern enthalten waren und somit keine saubere Trennung der Bildinhalte ermöglichten. Für die generierten Bilder wurde die Namenskonvention NoPlayer-XXX.jpg festgelegt. Dazu folgen die Klasse-0-Bilder dem Format 48px × 48px. Daraus ergibt sich somit der komplette Datensatz bestehend aus Klasse-1-Bilder für Spieler und Klasse-0-Bilder für keine Spieler. Beispielhaft wurde eine Auswahl von mehreren Spieler- und nicht-Spieler-Bildern in Abbildung 25 dargestellt.

Abschließend müssen die unterschiedlichen Klassen noch auf den Trainingsdatensatz verteilt werden. Für den Trainingsdatensatz wurde eine 50/50-Aufteilung zwischen Objektklasse 1 und Objektklasse 0 vorgenommen.



Abb. 25: Dargestellt ist in der oberen Reihe Bilder mit Fußballspieler und in der unteren Reihe Bilder ohne Fußballspieler. Diese sollen jeweils mit einer Klasse aus $\{player, noPlayer\}$ identifiziert werden.

In diesem Kapitel sollte die Überlegungen und die Methodik aufgezeigt werden, die nötig waren, um für diesen Anwendungsfall für den es keinen beispielhaften Datensatz gibt, eine funktionierende Daten-*Pipeline* aufzustellen. Dies erfolgt über mehrere Schritte und reicht von der generellen Entscheidung zwischen dem grundlegenden Bildmaterial, über die Generierung der Spieler- und kein-Spieler-Bilder mittels YOLO bis hin zu der Informationsgewinnung aus den einzelnen Pixel Werten eines jeden Bildes und mündet schlussendlich in der Generierung eines Datensates, der in Trainings- und Testdaten gesplitten werden kann. Somit wurde in diesem Kapitel die Datengrundlage gebildet, um das spätere neuronale Netz auf den in dieser Arbeit bearbeitet Anwendungsfall trainieren zu können. Bevor jedoch das neuronale Netz und die Struktur dieses dargelegt wird, wird in dem nächsten Kapitel zuerst das verwendete Framework Keras zur Erstellung des neuronalen Netzes vorgestellt.

4 Einführung in das Keras-Framework

In diesem Kapitel wird Keras, ein Framework für neuronale Netze vorgestellt. Dazu ist das Kapitel in zwei Teile aufgeteilt. Im ersten Teil wird Keras allgemein beschrieben, im zweiten Teil werden einige Hyperparameter für Keras vorgestellt.

4.1 Vorstellung des Deep-Learning-Frameworks Keras

Keras ist ein in Python geschriebenes Framework und benutzt Funktionen entweder von TensorFlow, Microsoft cognitive toolkit (CNTK) oder Theano. Es wird empfohlen TensorFolw als Basis zu verwenden (Keras Webseite, 2015). Deswegen ist TensorFlow auch als Standard ausgewählt. Im Fokus von Keras steht die einfache und schnelle Entwicklung von neuronalen Netzen zum Prototyping für KI-Entwickler und Wissenschaftler. Neben einfachen neuronalen Netzen unterstützt Keras auch convolutional neural network (CNN)s, recurrent neural network (RNN)s und erlaubt auch eine Kombination verschiedener Netze, beispielsweise um ein generative adversial network (GAN) zu erstellen (Keras Webseite, 2015). Es kann von der Python-Version 2.7 bis 3.6 verwendet werden.

Keras ist modular aufgebaut. So sind zum Beispiel die Schichten eines neuronalen Netzes, die Fehlerfunktion, die Optimierungsfunktionen, das Initialisierungsschemata, die Aktivierungsfunktionen und das Regularisierungsfunktionen eigenständige Module und können schnell und einfach miteinander kombiniert werden. Dadurch, das Keras komplett in Python geschrieben ist, können die Module einheitlich debuggt werden. Außerdem kann Keras direkt in ein Tensorflow-Programm eingebunden werden, um auf erweiternde *low-level*-Funktionen zuzugreifen.

4.2 Funktionen und Parameter in Keras

In diesem Unterkapitel werden die einzelnen Parameter und Funktionen von Keras erläutert. Dabei wird sich auf die Funktionen von Keras beschränkt, die für das

CNN in Kapitel 5 benötigt werden. Keras bietet über die hier beschriebenen Funktionen also noch einige weitere an. Diesen können in der Keras-Dokumentation nachgelesen werden. (Keras Webseite, 2015)

Dense	Repräsentiert eine <i>fully connected</i> -Neuronenschicht in Keras. Diese Funktion erhält als Übergabeparameter die Anzahl der Neuronen in der Schicht und die Aktivierungsfunktion der Neuronen.
Conv2D	Diese Funktion beschreibt die <i>convolutional</i> -Schicht in Keras, wie sie in Kapitel 2.3.1 erklärt wurde. Die Funktion erwartet die folgenden Parameter: Erstens die Anzahl der Kernel, zweitens die Größe des Kernels, drittens die Aktivierungsfunktion, was fast immer die <i>relu</i> -Funktion ist, und viertens die <i>input_shape</i> der Bildmatrix.
MaxPooling2D	Entspricht der <i>max-pooling</i> -Schicht aus Kapitel 2.3.2. verworfen. Als Übergabeparameter bekommt diese Funktion nur die <i>pool_size</i> , die der $n \times n$ Filtermatrix aus Kapitel 2.3.2 entspricht.
Flatten	Diese Funktion bildet einen Tensor d -ter Ordnung auf einen Tensor erster Ordnung ab, formell also $f : \mathbb{R}^{m_1 \times \dots \times m_d} \rightarrow \mathbb{R}^{(\prod_i^d m_i) \times 1}$. Dies ist notwendig, da die Ausgabe von <i>convolutional</i> - und <i>pooling</i> -Schichten noch in ein <i>feedforward neural network</i> überführt wird.
batch_size	Gibt nach Kapitel 2.4.2 die Anzahl k an Trainingstupeln aus X an, nach denen eine Gewichtsanpassung durchgeführt wird. Praktisch bedeutet dies, dass eine kleinere <i>batch size</i> weniger Arbeitsspeicher benötigt, während bei einer großen <i>batch size</i> mehr Trainingsdaten in den Arbeitsspeicher geladen werden müssen. Dies kann gerade bei großen Datenmengen problematisch werden. Ist $k = 1$ wird ein <i>stochastic gradient descent</i> durchgeführt, ist $k = X $, wird ein <i>batch gradient descent</i> durchgeführt, ist $1 < k < X $, wird ein <i>minibatch gradient descent</i> durchgeführt.

epochs	Gibt die Anzahl der Iterationen über die Trainingsmenge beim Training an. Wird, wie in Kapitel 6.2.3 später, <i>minibatch gradient descent</i> genutzt, werden $\frac{ X }{k}$ Gewichtsanpassungen pro Iteration durchgeführt, wobei k für die Größe des <i>minibatch</i> stehen. (Buduma & Lacascio, 2017, S. 31)
activation	Gibt für die jeweilige Neuronen-Schicht die Aktivierungsfunktion an. Beispiele für Aktivierungsfunktionen sind die <i>relu</i> -, <i>sigmoid</i> - und <i>softmax</i> -Funktion.
optimizer	Gibt den Optimierungsalgorithmus zur adaptiven Lernrate an. Bekannt aus Kapitel 2.4.3. Der Optimierungsalgorithmus wird in der <i>compile</i> -Funktion angegeben, also nur während des Trainings genutzt. Kapitel 6.2.5 enthält mit Hypothese 5 einen empirischen Vergleich der unterschiedlichen Optimierungsalgorithmen.
compile	In der <i>compile</i> -Methode wird der Trainingsprozess konfiguriert. Für die Methode gibt es drei Parameter: 1. Der <i>optimizer</i> gibt die Optimierungsfunktion an, 2. der <i>loss</i> gibt die Fehlerfunktion, bekannt aus Kapitel 2, an und 3. Die <i>metric</i> gibt an, welche Performanzmetriken während des Trainings aufgenommen werden sollen.
fit	Ist in Keras die Methode zum Trainieren eines neuronalen Netzes. Dazu werden folgende Parameter übergeben: 1. Die <i>batch size</i> , 2. Die Anzahl der Epochen, 3. Der <i>validation_split</i> und zuletzt mögliche <i>callbacks</i> der Methode.
validation_split	Ist eine Gleitkommazahl zwischen 0 und 1 und gibt die Anzahl an Testdaten an. Wenn der <i>validation_split</i> zum Beispiel bei 0.2 liegt, hält das Modell 20% der Daten zurück mit denen nicht trainiert wird, sondern am Ende getestet wird, um während des Trainings <i>underfitting</i> oder <i>overfitting</i> zu erkennen.
kernel_size	Ist Übergabeparameter in der <i>Conv2D</i> -Funktion. Gibt die Größe des verwendeten Kernels an, siehe Kapitel 2.3.1.

loss	Gibt die Fehlerfunktion für den Trainingsprozess an.
callbacks	Werden dafür verwendet, um einen besseren Blick in den Trainingsprozess zu erhalten. Zum Beispiel kann man sich über die <i>callbacks</i> einen Verlauf des Fehlermaßes über die Epochen anzeigen lassen.
input_shape	Ist Übergabeparameter in der Conv2D-Funktion und beinhaltet die Größe der Bildmatrix, siehe Kapitel 2.3.1. Dabei umfasst die <i>input shape</i> hier auch die Tiefe der Conv2D-Schicht, die für RGB-Bilder, aufgrund der drei Farbkanäle und damit der drei notwendigen Kernel, in Kapitel 5 auf drei gesetzt wird.

Die verwendeten Informationen für die oben stehende Auflistung stammen allesamt aus der Dokumentation von Keras.(Keras Webseite, 2015)

Nachdem in diesem Kapitel das Framework Keras kurz betrachtet wurde, wird im nächsten Kapitel genauer auf ein die Implementierung eines *convolutional neuronal networks* eingegangen.

5 Aufbau des CNNs

Nachdem im vorherigen Kapitel das verwendete Framework Keras zur Erstellung eines neuronalen Netzes und die für dieses verwendeten notwendigen Hyperparameter vorgestellt wurden, wird in diesem Kapitel der Fokus auf das neuronale Netz selbst gelegt. Ziel dieses Kapitels ist es den Aufbau des neuronalen Netzes anschaulich und verständlich darzulegen. Für dieses Kapitel und das neuronale Netz selbst werden die in Kapitel 3 beschriebenen Datengrundlagen bestehend aus Spielerbildern und nicht Spielerbildern verwendet.

5.1 Ziel des CNN

Das generelle Ziel des neuronalen Netzes ist es, wie bereits vorgestellt, das Erkennen von Fußballspielern auf einem Bild und das Erkennen von Bildern ohne Fußballspieler.

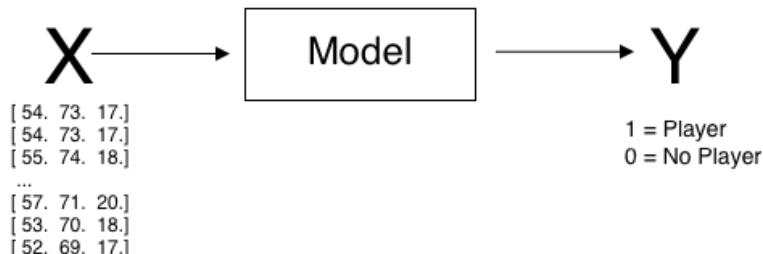


Abb. 26: Ziel des Neuronalen Netzes

Betrachtet man ein bereits trainiertes Netz und legt x als die Eingabe für das trainierte Netz fest, so soll am Ende dieses Kapitels das neuronale Netz in der Lage sein, die Eingabe x in eine der beiden Objektklassen $\{player, noPlayer\}$

bzw. $\{0, 1\}$ umzuwandeln und somit eine Aussage darüber zu treffen, ob es sich um einen Fußballspieler handelt oder nicht. Wie in Abbildung 26 dargestellt, orientiert sich der spätere Ablauf des Netzes nur noch daran, bisher die unbekannte Daten zu verarbeiten und deren Zugehörigkeit zu einer der beiden Klassen zu bestimmen.

5.2 Umsetzung eines CNN

In diesem Unterkapitel wird das CNN und die genutzte Struktur genauer betrachtet. Dazu wird im ersten Teil der genaue Aufbau des Netzes beschrieben. Im zweiten Teil wird auf das Trainieren und Testen von CNNs mit der in Kapitel 3 beschriebenen Datengrundlage eingegangen.

5.2.1 Praktische Implementierung des CNN

Bevor die richtige Struktur des CNNs betrachtet wird, werden zunächst für das Netz wichtige Hyperparameter betrachtet. Für das genutzte CNN wurden die in Listing 5 dargestellten Hyperparameter ausgewählt.

```
1 NUM_CLASSES = 2
2 IMG_SIZE = 48
3 batch_size = 64
4 epochs = 5
```

Listing 5: Deklarationen

Die Auswahl umfasst als erstes die Anzahl der Klassen für die Klassifikation welche in der Variablen `NUM_CLASSES` festgelegt wird. In diesem Anwendungsfall findet eine binäre Klassifikation statt, da nur zwischen den zwei Klassen Fußballspieler (1) und kein Fußballspieler (0) unterschieden wird. Als nächster Parameter folgt die Größe des Bildes, die in der Variablen `IMG_SIZE` angegeben ist. Diese resultiert aus der in Kapitel 3 dargestellten einheitlichen Beschneidung der Bilder auf eine Größe von $48px \times 48px$ Pixeln. Aus diesem Grund wird nur eine Bildgröße verwendet, da die zu verwertenden Bilder quadratisch sind. Des Weiteren wird noch die *batch size*

und Epochen festgelegt, jedoch sollten diese beiden Parameter und ihre Belegung zu diesem Zeitpunkt als beispielhaft angesehen werden.

```
1 def cnn_model():
2     model = Sequential()
3     model.add(Conv2D(32, kernel_size=(3, 3),
4                      activation='relu',
5                      input_shape=(IMG_SIZE, IMG_SIZE, 3)))
6     model.add(MaxPooling2D(pool_size=(2, 2)))
7     model.add(Conv2D(64, (3, 3), activation='relu'))
8     model.add(MaxPooling2D(pool_size=(2, 2)))
9     model.add(Conv2D(256, (3, 3), activation='relu'))
10    model.add(MaxPooling2D(pool_size=(2, 2)))
11    model.add(Flatten())
12    model.add(Dense(32, activation='relu'))
13    model.add(Dense(NUM_CLASSES, activation='sigmoid'))
14
15    return model
```

Listing 6: Schichten des CNNs

Das Listing 6 beschreibt die Schichten, die für das CNN verwendet werden. In Zeile 2 wird das Model, welches erstellt wird, mit der Methode `Sequential` initialisiert. Die `Sequential`-Methode ermöglicht eine lineare Anordnung von unterschiedlichen Schichten. Anschließend folgt die Definition der ersten Schicht des Netzes. In diesem Fall ist dies ein `Conv2D`-Schicht, da das CNN zweidimensionale Bilder verarbeiten muss. Der erste Parameter stellt die Anzahl der verwendeten Kernel-Filter gemäß Kapitel 2.3.1 dar. In diesem Fall wurden 32 Kernel festgelegt, die das Eingabebild gleitend nach Mustern absuchen. Anschließend folgt die Deklarierung der `kernel size`. Die `kernel size` ergibt sich, nach Kapitel 2.3.1, aus der gewünschten Größe des Kernels, die in jedem Fall kleiner als das Bild sein sollte. Als vorletzter Parameter wird die Aktivierungsfunktion für die erstellte `convolutional`-Schicht festgelegt. Nach Kapitel 2.3.1, wird typischerweise die `relu`-Funktion für `convolutional`-Schichten genutzt.

Abschließend muss noch die `input_shape` der Eingabebilder festgelegt werden. Diese wird durch die Auflösung der Bilder und ihrer Farbkanalanzahl bestimmt. Im vorliegenden Fall beträgt die Auflösung $48px \times 48px$ bei 3 Farbkanälen. Die zweite Schicht eine `MaxPooling2D`-Schicht. Diese Schicht folgt i.d.R. auf eine `Conv2D`-Schicht nach Kapitel 2.3.1. Durch die Verwendung der `MaxPooling`-Schicht werden die `feature maps` der `Conv2D`-Schicht auf eine reduziert. Dabei wird ein

pooling-Filter der Größe 2×2 genutzt, was die *feature map* entsprechend der Gleichung 33 reduziert.

Werden die Zeilen 7 bis 10 des Listings 6 betrachtet, so fällt auf, dass die gerade vorgestellten Schichten wiederholt eingesetzt werden und sich lediglich die Anzahl der verwendeten Kernel erhöht (32, 64, 256). Die verwendeten MaxPooling2D-Schichten sind identisch zu dem vorgestellten obigen Beispiel. In Zeile 11 folgt die Funktion Flatten. Diese ist definiert als $f : \mathbb{R}^{m_1 \times \dots \times m_d} \rightarrow \mathbb{R}^{(\prod_i^d m_i) \times 1}$, bildet einen Tensor d -ter Ordnung also auf einen Spaltenvektor ab, was für die Eingabe in ein neuronales Netz, wie es in Kapitel 2.2.2 erklärt wurde, notwendig ist. In Zeile 12 wird eine *fully connected feedforward Dense*-Schicht mit 32 Neuronen festgelegt. Diese verwendet ebenfalls die Aktivierungsfunktion *relu*. Die letzte Schicht in dem Netz ist ebenfalls ein *Dense*-Schicht. Diese bekommt die Anzahl der zu klassifizierenden Objektklassen in der Variable NUM_CLASSES übergeben. Diese Schicht unterscheidet sich außerdem in der verwendeten Aktivierungsfunktion von den anderen Schichten. Für die letzte Schicht wurde die *sigmoid*-Aktivierungsfunktion ausgewählt. (Zhang, 2017)

Nachdem das CNN in seiner Struktur festgelegt wurde, kann dieses nun kompiliert und trainiert werden. Damit das CNN kompiliert werden kann, muss zunächst das CNN erzeugt werden. Dies geschieht wie in Listing 7 dargestellt.

```
1 model = cnn_model()
2 model.compile(loss='binary_crossentropy',
3                 optimizer=keras.optimizers.Adadelta(),
4                 metrics=['accuracy'])
```

Listing 7: compile-Methode

Dazu wird als erstes das Model, das in Listing 6 beschrieben wurde, initialisiert. Anschließend folgt der Aufruf der `compile`-Funktion. Dabei wird zunächst eine Fehlerfunktion, wie aus Kapitel 2 bekannt, festgelegt. In diesem Fall wird die Funktion `binary_crossentropy` als Fehlerfunktion festgelegt. Dies resultiert aus der speziellen Eignung der `binary_crossentropy` für eine binäre Klassifikation. Für mehr als zwei Klassifizierungen wäre die `binary_crossentropy` nicht geeignet. Stattdessen sollte dann die `categorical_crossentropy` verwendet werden. Der zweite Parameter ist die Optimierungsfunktion, hier ist diese `Adadelta`. Als letztes

wird der `compile`-Funktion die gewünschte Ausgabemetrik übergeben. Bei diesem CNN ist die Metrik auf `accuracy` gesetzt. Diese gibt nach jedem Epochendurchlauf die Genauigkeit für die jeweilige Epoche an und sorgt somit für eine transparente Darstellung des Trainingsverlaufes. Nachdem das Modell angelegt und kompiliert wurde, ist dieses bereit, um trainiert zu werden. Im nachfolgenden Listing 8 wird der verwendete Aufruf der `fit`-Funktion für das CNN dargestellt.

```
1 history = model.fit(X, y_train,
2                      batch_size=batch_size,
3                      epochs=epochs,
4                      validation_split=0.2,
5                      callbacks=[LearningRateScheduler(lr_schedule),
6                                 ModelCheckpoint('model.h5', save_best_only
7                                     =True)]
```

Listing 8: Fit-Funktion des CNNs

Die `fit`-Funktion ermöglicht das Trainieren und das Testen des erstellten neuronalen Netzes. Diese Funktion bekommt als ersten Parameter ein Array mit den relevanten Bildinformationen in Form von RGB-Werten übergeben. Der zweite Parameter ist `y_train`. Dieses Array beinhaltet die Kategorien für die in `x` übergebenen Bilder. Dies ist notwendig, damit das Netz die Bilder mit der richtigen Klasse referenziert und somit erfolgreich lernen kann.

Die nächsten beiden Parameter sind die `batch_size` und die Epochen. Die Werte dieser beiden Parameter wurden bereits in Listing 5 definiert und werden hier verwendet. Der vorletzte Parameter `validation` hat die Aufgabe, einen prozentualen Anteil festzulegen in der der übergebene Trainingsdatensatz aufgeteilt werden soll. In diesem Fall werden 20 Prozent der Trainingsdaten zurückgehalten und es wird ausschließlich auf den anderen 80 Prozent trainiert wird. Anschließend werden die zurückgehaltenen 20 Prozent zur Validierung des gelernten Verhaltens genutzt, um *overfitting* oder *underfitting* gemäß Kapitel 2.4.1, zu erkennen. Der letzte Parameter der `fit`-Funktion ist das `callback`-Parameter. Dieses gibt dem Benutzer etwa die Möglichkeiten, die Lernrate des Netzes benutzerdefiniert anzupassen oder aus dem trainierten Netz eine `.h5`-Datei zu erstellen. Diese speichert die gelernten Gewichte des Netzes und ermöglicht somit eine Initialisierung des Netzes unabhängig von einer erneuten Trainingsphase.

5.2.2 Beispielhafter Durchlauf durch das neuronale Netz

Nachdem in dem Kapitel 3 die Grundlage zur Gewinnung der Daten und die Datenaufbereitung dargelegt und definiert wurde und anschließend in Kapitel 5.1.2 der verwendete strukturelle Aufbau des Netzes beschrieben wurde, soll in diesem Kapitel ein beispielhafter Durchlauf durch das neuronale Netz erfolgen. Dies soll vor allem der Veranschaulichung und der praxisnahen Zusammenführung von Daten und Netz dienen.

Der erste Schritt ist in Listing 9 mit der Auswahl und der Vorverarbeitung der Trainingsdaten dargestellt. Zuerst werden alle benötigten Daten für das Trainieren

```
1  imgs = []
2  labels = []
3  train_path = os.path.dirname(os.path.realpath(__file__)) + "/trainings_daten/t_2500"
4  all_img_paths = glob.glob(train_path + '/*.jpg')
5  np.random.shuffle(all_img_paths)
6  for img_path in all_img_paths:
7      if img_path.endswith(".jpg"):
8          img = io.imread(img_path)
9          pixel_image = load_image(img_path)
10         label = get_label(img_path)
11         imgs.append(pixel_image)
12         labels.append(label)
```

Listing 9: Trainingsdaten Auswahl und Vorverarbeitung

des Netzes aus dem jeweiligen Verzeichnis ausgewählt. Anschließend werden alle, aus dem Verzeichnis ausgewählten Bilder mit der Funktion `shuffle` durchmischt. Dies hat den Vorteil, dass sichergestellt wird, dass das Netz beim späteren Trainieren keine saubere Trennung zwischen den Bildern der beiden Objektklassen hat, sondern durchgemischte und keinem Muster in der Anordnung folgende Daten. Anschließend werden für jedes sich in dem Verzeichnis befindliche Bild die RGB-Werte ausgelesen und die dazugehörige Objektklasse für das Bild bestimmt. Die beiden benötigten Informationen werden in zwei unterschiedlichen Arrays gespeichert. Die Bildinformationen werden in dem Array `imgs` und die Objektklasse des Bildes in dem Array `labels` gespeichert. Wurde der Prozess für alle Bilder erfolgreich durchgeführt, müssen die beiden gefüllten Arrays `imgs` und `labels` noch mit den in Listing 10 dargestellten Methoden weiterverarbeitet werden.

```
1 X = np.array(imgs, dtype='float32')
2 y_train = keras.utils.to_categorical(labels, 2)
```

Listing 10: Array Transformierung

Dies ist notwendig, um die Daten in das für das neuronale Netz benötigte Format zu transformieren. Wurden diese Schritte erfolgreich ausgeführt stehen zwei Mengen zur Verfügung. Zum einen ein Array `X` mit den Bildinformationen aller eingelesenen Bilder und zum anderen ein Array `y_train`. Das zu jedem Bild die richtige Klasse bereitstellt. Das Datenformat der beiden Arrays ist nachfolgend in den Listings 11 und 12 dargestellt.

```
1 [138. 169. 93.]
2 [138. 169. 93.]
3 [138. 169. 93.]
4 ...
5 [132. 163. 87.]
6 [132. 163. 87.]
7 [132. 163. 87.]
```

Listing 11: Datenformat von `x`

```
1 [[0. 1.]
2 [1. 0.]
3 [0. 1.]
4 ...
5 [1. 0.]
6 [0. 1.]
7 [0. 1.]]
```

Listing 12: Datenformat von `y`

Sind die Daten erfolgreich in dieses Format transformiert worden, kann anschließend das neuronale Netz initialisiert und trainiert werden. Ist das neuronale Netz in der Trainingsphase, wird für jede durchlaufende Epoche die in Listing 13 dargestellte Ausgabe erzeugt.

Diese gibt Aufschluss darüber, wie hoch die Genauigkeit und das Fehlermaß pro Epoche ist und ermöglicht somit eine transparente Verfolgung des Trainings. Wurde das Netz erfolgreich trainiert, ist dieses nun in der Lage Vorhersagen auf neuen Daten zu tätigen. Damit dies möglich ist, kann die ebenfalls in Kapitel 5.2.1

```

1 2008/2008 [=====] - 5s 3ms/step - loss
      : 4.6547 - acc: 0.5989 - val_loss: 0.2222 - val_acc: 0.9263
2 Epoch 2/3
3 2008/2008 [=====] - 5s 2ms/step - loss
      : 0.1351 - acc: 0.9664 - val_loss: 0.1038 - val_acc: 0.9711
4 Epoch 3/3
5 2008/2008 [=====] - 5s 2ms/step - loss
      : 0.0691 - acc: 0.9871 - val_loss: 0.0662 - val_acc: 0.9781

```

Listing 13: Ausgabe der Traingsphase

generierte .h5-Datei, welche am Ende jedes Trainings erstellt wird, verwendet werden, um das gelernte Modell unabhängig von der Trainingsphase zu initialisieren. Dies ist notwendig, um wie in Listing 14 dargestellt das Modell über die .h5-Datei zu initialisieren.

```

1 model = load_model('model.h5')
2 X = np.array(imgs, dtype='float32')
3 ynew = model.predict_classes(X)

```

Listing 14: Initialisierung des CNN über eine .h5-Datei

Anschließend können über das initialisierte Modell Vorhersagen auf neue Daten ausgeführt werden. Damit dies möglich ist, müssen die neuen Daten auf das gleiche Format gebracht werden wie in Listing 8 bereits dargestellt wurde. Der einzige Unterschied besteht darin, dass für die Vorhersage von neuen Daten nur die Bildinformationen (RGB-Werte) vorhanden sein müssen, da die Aufgabe des Netzes darin besteht, diese neuen Bildinformationen erfolgreich zu klassifizieren.

6 Hypothesengenerierung und -überprüfung

In den letzten Kapiteln wurde der Ablauf, der für die Umsetzung des benötigten neuronalen Netzes zur Klassifikation von Fußballspielern nötig ist, behandelt. Es wurde dargelegt, wie für den speziellen Anwendungsfall die benötigten Daten gesammelt und anschließend aufbereitet werden können, sodass eine Bildklassifikation möglich wird. Dazu wurde das Framework Keras zur Implementierung des neuronalen Netzes mitsamt den wichtigsten Hyperparametern vorgestellt. Abschließend wurde die Struktur des verwendeten Netzes dargelegt und ein beispielhafter Durchlauf durch das Netz dargestellt. Abschließend soll nun durch das Überprüfen von Hypothesen die beste Netz-Struktur herausgearbeitet werden. Jedoch sollte beachtet werden, dass die Parameter, die in den Hypothesen überprüft werden, begrenzt sind. Dies resultiert aus der hohen Anzahl von Parametern und deren Kombinationsmöglichkeiten wie z.B. die Anzahl der Trainingsdaten, die unterschiedlichen *batch sizes* und Epochengrößen, unterschiedliche Optimierungsalgorithmen und Aktivierungsfunktionen und die generelle Struktur der unterschiedlichen *hidden*-Schichten. Werden alleine diese Parameter betrachtet und deren unterschiedlichen möglichen Werte, so würde die Anzahl an Anordnungen schnell auf über 1000 unterschiedliche Netzeinstellungen steigen können. Daraus folgt, dass nur bestimmte Parameter und dazu festgelegte Parameterwerte getestet werden, um die beste Konfiguration des neuronalen Netzes zu finden und, gemäß Kapitel 1.3, einen Erkenntnisgewinn herbeizuführen.

6.1 Generierung und Überprüfung der Hypothesen

In diesem Kapitel werden verschiedene Hypothesen aufgestellt und überprüft. Das Ziel ist es, durch die Überprüfung der Hypothesen die bestmögliche Kombination von unterschiedlichen Parametern für das Netz zu finden. Jedoch werden durch die aufgestellten Hypothesen auch generelle Aussagen und Richtlinien für neuronale Netze überprüft. Der generelle Aufbau einer Hypothese teilt sich in zwei Teile auf. Zum einen wird zu Beginn die Hypothese aufgestellt und dargelegt. Anschließend werden nach jedem Experiment die herausgearbeiteten Ergebnisse dargestellt und eine Entscheidung aus den Ergebnissen getroffen. Diese Entscheidung beeinflusst

die nächsten Hypothesen in dem Maße, als dass die Entscheidung Einfluss auf die Struktur des Netzes nimmt und somit maßgeblich die Erkenntnisse mit einfließen lässt. Abschließend kann festgehalten werden, dass sich durch diese Art des iterativen Testen nach jeder Hypothese die betroffenen Parameter ändern und somit eine aktualisierte Grundlage für die weiteren Hypothesen bildet.

6.2 Die Hypothesen

In diesem Teil werden die verschiedenen Hypothesen aufgestellt und beurteilt. Dazu werden einige Definitionen festgelegt. Generell wird in diesem Kapitel oft das Wort Genauigkeit im Zusammenhang mit dem Netz erwähnt. Das bedeutet für diesen Anwendungsfall nichts anders als die Genauigkeit des Netzes auf den Testdaten. Da für jede Hypothese die Genauigkeit nach dem Training und dem Testen betrachtet wird, wurde letztendlich die Genauigkeit auf den Testdaten als am wichtigsten befunden. Außerdem ist in den folgenden Hypothesen das Wort Netz gleichzusetzen mit dem erstellten CNN aus Kapitel 5.2.

In den folgenden Hypothesen werden Diagramme dargestellt, die die unterschiedlichen Genauigkeiten visualisieren. Die in den Diagrammen dargestellten Linien haben die folgende Bedeutung: Die erste Linie mit dem Label „acc_training“ bedeutet die Genauigkeit der Erkennung von Bildern während des Trainings. Die zweite Linie „acc_test“ gibt die Genauigkeit der Bilder nach einem Trainingsdurchlauf an. Dies spiegelt den *validation split* wieder. Dieser bedeutet, dass eine bestimmte Anzahl von Daten zurückgehalten wird und diese nach dem Training genutzt werden um das gelernte Klassifikationsmodell auf *overfitting* oder *underfitting* zu überprüfen, siehe Kapitel 2.4.1.

6.2.1 Hypothese zur Menge der verwendeten Trainingsdaten

Die erste Hypothese die behandelt wird, überprüft die Menge der verwendeten Trainingsdaten. Allgemein ist es im maschinellen Lernen schwierig konkrete Aussagen über die optimale Größe der Trainingsmenge zu treffen. Dies resultiert aus den unterschiedlichen Anwendungsfällen und der damit verbundenen Komplexität des behandelten Problems. (Brownlee, Jason, 2017) Beispielsweise ist

die benötigte Menge an Trainingsdaten für eine Multiklassifikation wie die Verkehrsschilderkennung höher einzustufen, als für eine binäre Klassifikation. Daraus resultiert die für diesen Anwendungsfall relevante Frage, ab welcher Menge von Trainingsdaten das Netz hohe bis sehr hohe Genauigkeiten liefert. Hohe Genauigkeit bedeutet in diesem Fall, dass das Netz auf die Testdaten eine hohe korrekte Klassifizierungsquote aufweist.

Hypothese 1 Die Genauigkeit des Netzes steht im Zusammenhang mit der Menge der verwendeten Trainings- und Testdaten.

Für die aufgestellte Hypothese wurden fünf unterschiedliche Große Traingsdatensätze erstellt. Diese wurden zu jeweils gleichen Teilen in die Klassen $\{player, noPlayer\}$ unterteilt. Die fünf verwendeten Datensätze beinhalten die folgenden Größen: 250, 500, 1000, 2500, 5000.

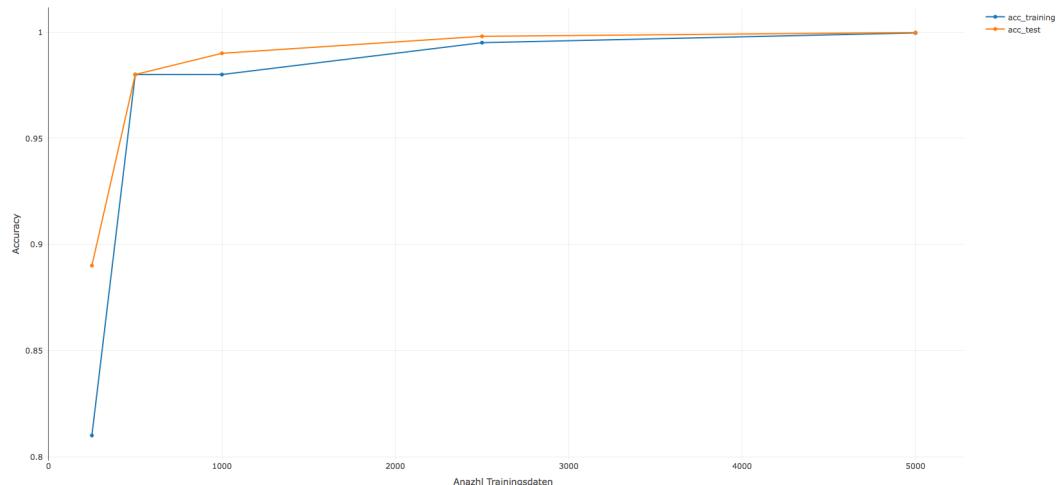


Abb. 27: Dargestellt ist der Verlauf in der Genauigkeit für die unterschiedlichen getesteten Trainingsmengen

In Abbildung 27 sind die Ergebnisse der Trainings und Testphase dargestellt. Daraus wird ersichtlich, dass die ersten drei Trainingsmengen 250, 500 und 1000 stark, in der Genauigkeit auf den Trainings- und Testdaten schwanken und schlechtere Ergebnisse erzielen als die folgenden Trainingsmengen. Dies stützt auch die Beobachtung, dass mit jeder Erhöhung der Trainingsmenge die Genauigkeit auf die Testdaten steigt. Ab einer Trainingsmenge von 2500 Daten tritt eine Sättigung

in der Genauigkeit auf den Testdaten ein. Allgemein kann festgehalten werden, dass die Erhöhung der Traingsmenge bis zur Menge von 2500 einen Unterschied ausmacht und ab dieser Menge keine weitaus besseren Ergebnisse erzielt werden. Der einzige Unterschied in beiden Mengen zeigt sich in einer kleinen Differenz in der Genauigkeit nach dem Training. In dieser Metrik schneidet die 5000 Menge leicht besser ab als die 2500 Menge.

Ergebnis 1 Die Genauigkeit steigt mit den ersten drei Trainingsmengen an, jedoch wird ab der Trainingsmenge von 2500 eine Sättigung erreicht und es sind nur noch marginale Änderungen bei der Verdopplung der Trainingsdaten möglich. Jedoch muss beachtet werden, dass das Ergebnis keine Allgemeingültigkeit und nur speziell für den getesteten Anwendungsfall sich so verhält. Es kann sein, dass beispielsweise bei einer Multivarianten Klassifikation mehr Trainingsdaten für zufriedenstellende Genauigkeitswerte benötigt werden. Außerdem kann es auch bei einer binären Klassifikation wie in diesem Fall dazu kommen, dass mehr Trainingsdaten benötigt werden, da die Erkennung von Fußballspielern auf einem Bild eine geringe Komplexität von der Anordnung und der Fülle der Farbwerte in diesem getesteten Datensatz enthält. Als Beispiel können die unterschiedlichen Feldlinien dienen. Diese führen nämlich dazu, dass das Netz diese manchmal als Spieler erkennt. Hier könnte auch darüber nachgedacht werden, den Trainingsdatensatz weiter aufzuspalten in bspw. drei Klassen: Spieler, Rasen, Rasen + Linie, um so eine Interpretation von einem Linien Muster als Spieler zu verhindern. Diese Änderung alleine würde jedoch wieder zu einer Erhöhung der Trainingsdaten Menge führen. Daraus resultiert, dass die folgende Hypothese bestätigt wurde, da durch eine Erhöhung der Trainingsdaten eine merkliche Verbesserung in der Genauigkeit erreicht wurde. Jedoch muss festgehalten werden, dass dies nur bis zu einem gewissen Grad funktioniert und somit ab einem bestimmten Punkt eine Sättigung in den Werten erreicht wird.

Die Entscheidung die aus diesen Ergebnissen resultiert ist, dass fortan für das Netz die Traingsmenge von 2500 verwendet wird. Dies resultiert aus den zum

einen höchsten Genauigkeitswerten für das Testen gegenüber allen anderen Mengen. Zum anderen besteht zwischen der 2500 und 5000 Menge nur marginale Unterschiede in der Genauigkeit nach dem Training. Außerdem können mit dem Einsatz von 50 Prozent der Daten nahezu identische Ergebnisse erreicht werden und somit auch eine Sparsamkeit in der verwendeten Rechenleistung.

6.2.2 Hypothese zur Anzahl der verwendeten Epochen in der Traingsphase

Die zweite Hypothese behandelt die Anzahl der verwendeten Epochen in der Trainingsphase. Eine Epoche beschreibt einen vollen Trainingszyklus inklusive Validierung über den kompletten Traingsdatensatz. Generell beschreibt die verwendete Anzahl von Epochen die Anzahl von Zyklen in der der angegebene Traingsdatensatz verwendet wird (Brownlee, Jason, 2018b). In der Praxis wird oft eine höhere Anzahl an Epochen gewählt, um die Gewichtsparameter eines neuronalen Netzes so anzupassen, dass das Fehlermaß gering ist.

Hypothese 2 Je höher die Anzahl der Epochen, desto höher ist die Genauigkeit auf den Testdaten, da das neuronale Netz genügend Iterations-schritte hatte, die Gewichte zu optimieren.

Für die aufgestellte Hypothese wurden sieben unterschiedliche Anzahlen an Epochen verwendet. Die Epochen für diese Hypothese wurden in der folgenden Reihenfolge getestet: 1, 3, 5, 8, 10, 15, 20.

Wird das Ergebnis in Abbildung 28 betrachtet, so fällt auf, dass sich die Genauigkeit ab einer Epochenzahl von zehn nicht signifikant verbessert. Hierbei kann auch beobachtet werden, dass die Genauigkeit für das Training und das Testen, bis auf Epoche eins, für alle anderen Epochen fast den selben Verlauf haben, mit marginalen Unterschieden in den Werten.

Ergebnis 2 Betrachtet man den Verlauf der Genauigkeiten kann festgehalten werden, dass die Genauigkeitsmetrik ab Epoche Zehn konstant hoch ist. Die Genauigkeit nach dem Testen erreicht ihren Höhepunkt bei Epoche zehn. Ein Grund wieso die Genauigkeit schon nach drei Epochen fast genauso hoch ist wie nach zehn Epochen

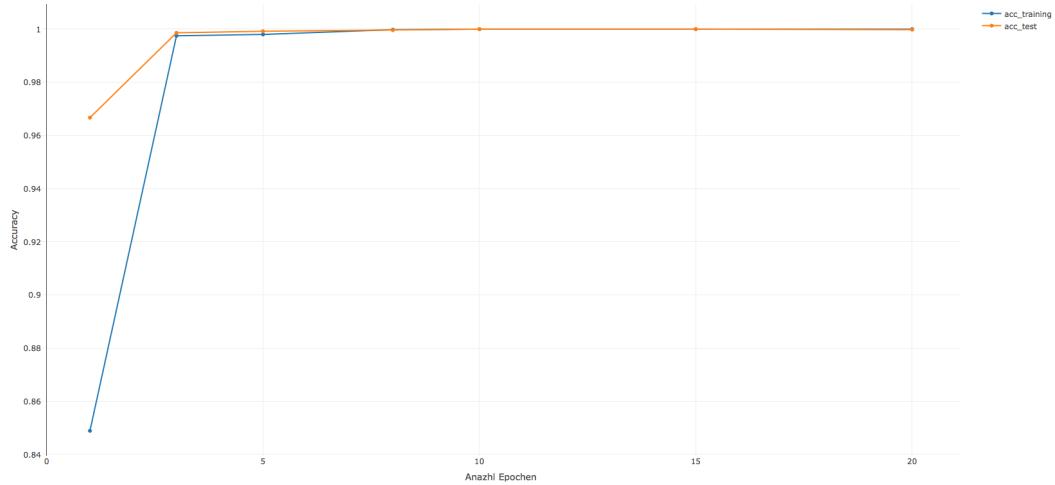


Abb. 28: Dargestellt ist der Verlauf in der Genauigkeit für die unterschiedlichen getesteten Epochen

Könnte in der Einfachheit des Anwendungsfalls liegen. Das bedeutet, da das Netz nur auf eine Unterscheidung zwischen zwei möglichen Klassen trainiert wurde und die Traingsdaten dazu strikt getrennt wurden, dies dazu führt, dass sich schnelle Lernerfolge einstellen. Aber auch hier sollte bedacht werden, dass dieses Ergebnis nur für diesen Anwendungsfall Anwendung findet. Bei komplexeren Klassifikations Beispielen oder auch numerischen Vorhersagen ist das Fehlerpotenzial weitaus höher als bei einer binären Klassifikation, da dort die Anzahl der unterschiedlichen Werte und deren Differenz deutlich höher ausfällt. Daraus folgt, dass die aufgestellte Hypothese bestätigt wurde. Dies folgt aus der laufenden Verbesserung in der Genauigkeit bis zum Höhepunkt bei Epoche Zehn. Generell sind die hohen Genauigkeitswerte bei geringen Epochen wie oben angeführt auf die Einfachheit des Klassifizierungsproblems zurückzuführen.

Die Entscheidung die aus diesen Ergebnissen resultiert, ist, dass die Epochenzahl Zehn das beste Ergebnis liefert, bezogen auf die Effizienz. Dies resultiert zum einen daraus, dass sie die höchste Genauigkeit von 100% vorweist und außerdem auch in Bezug auf die Rechenleistung ein guten Kompromiss darstellt, da sich der mehr Aufwand für die Epochen größer als Zehn nicht in einer höheren Genauigkeit

auszahlen kann, da bereits die höchste Genauigkeit erreicht wurde. Aus dieser Hypothesen lässt sich aber keinesfalls sofort folgern, dass eine hohe Anzahl an Epochen gleichzusetzen ist mit einer hohen Genauigkeit durch langwierig optimierte Gewichte. Gerade bei einer hohen Anzahl an Epochen kann ein neuronales Netz *overfitten*, da es oft über die Trainingsmenge iteriert. Die Genauigkeit eines Netzes setzt sich immer zusammen aus allen Hyperparametern, hier wurde eines dieser Hyperparameter untersucht.

6.2.3 Hypothese zur Größe der verwendeten *batch size* pro Epoche in der Trainingsphase

Die dritte Hypothese untersucht die verwendete Größe der innerhalb der Epochen angewandten *batch size*. Die *batch size* beschreibt die Anzahl der Trainingsdaten, die zu einer Gewichtsanpassung genutzt werden, siehe Gleichung 36. (Brownlee, Jason, 2018b). Das bedeutet, desto kleiner die *batch size* gesetzt wird, desto öfter werden die Gewichte angepasst. Daraus resultiert die folgende Hypothese:

Hypothese 3 Desto kleiner die *batch size* ist, desto höher ist die Genauigkeit des Netzes, weil die Gewichte häufiger angepasst werden.

Für die vorgestellte Hypothese wurden sieben unterschiedliche *batch sizes* überprüft und getestet. Diese starten bei der Größe von eins, was einem *stochastic gradient descent* entspricht, und erhöhen sich danach jeweils um den Faktor Zwei. Da die restlichen *batch sizes* k nie genauso groß sind wie der Trainingsdatensatz, $|X| > k$, entsprechen die restlichen *batch sizes* dem *mini batch gradient descent*. So ergibt sich die folgende Testreihenfolge für die *batch size*: 1, 4, 8, 16, 32, 64, 128.

Werden die in Abbildung 29 dargestellten Ergebnisse betrachtet so fällt direkt auf, dass ein großer Unterschied zwischen der Genauigkeit nach dem Training und dem Testen besteht. Zu Beginn liegen die Genauigkeiten für das Training und das Testen für die ersten vier *batch sizes* (1, 4, 8, 16) auf einer Höhe. Anschließend fällt für die *batch size* 32 die Genauigkeit nach dem Testen marginal ab und gleichzeitig bleibt die Genauigkeit nach dem Training auf einem stabil hohen

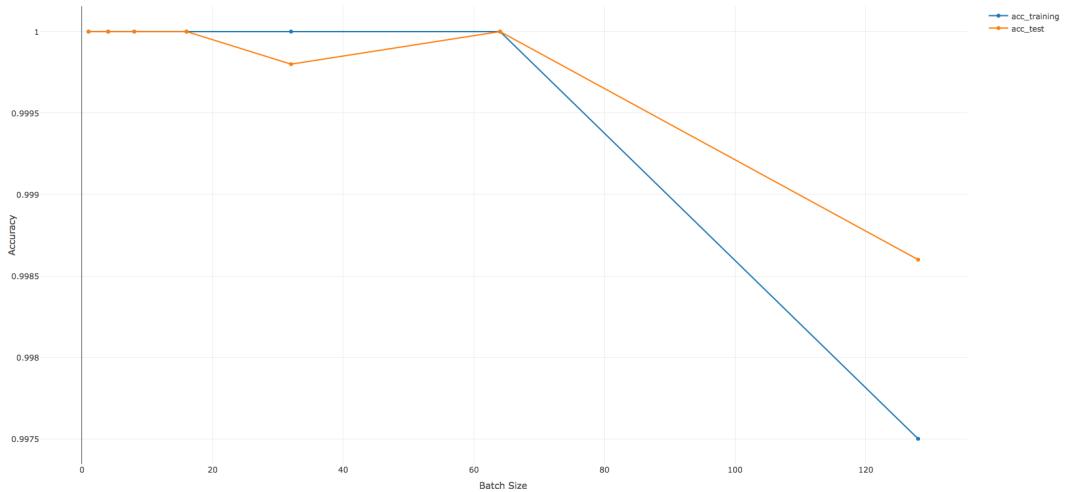


Abb. 29: Dargestellt ist der Verlauf in der Genauigkeit für die unterschiedlichen getesteten *batch sizes*

Niveau. Für die nächste *batch size* von 64 erhalten beide wieder die gleichen Genauigkeitswerte und abschließend stürzen beide bei der *batch size* von 128 stark in den Genauigkeiten ab.

Ergebnis 3 Werden die Ergebnisse aus Abbildung 29 betrachtet so kann festgehalten werden, dass die Ergebnisse für das Training und das Testen stabil auf einem hohen Niveau liegen. Lediglich für die *batch size* von 32 gibt es Schwankungen in der Genauigkeit der Testdaten, jedoch können diese als marginal eingestuft werden. Gravierender ist der Abfall für die Genauigkeit von den Trainings und Testdaten bei der *batch size* von 128 im Vergleich zu den anderen Werten. Dieser gravierende Abfall kann mit der geringeren Anzahl der Gewichtsanpassungen während der Trainingsphase erklärt werden. Dies ist auch nur logisch, da die Anzahl der Gewichtsanpassungen pro Epoche bei einer *batch size* von 16 bei knapp 125 liegen und bei einer *batch size* von 128 lediglich bei 15. Werden diese Werte weiter betrachtet so kann festgehalten werden, dass bei einer großen *batch size* die Gewichte 110 mal weniger angepasst werden als bei der kleinen *batch size*. Dies könnte als Begründung für die abweichende Genauigkeit gelten, da somit das Netz weniger

Möglichkeiten besitzt sich feinfühlig auf die Daten einzustellen. Die ursprüngliche Hypothese kann als bestätigt angesehen werden, da die Genauigkeit im Zusammenhang mit der verwendeten *batch size* steht. Dies ist in diesem Fall deutlich an der starken Abweichung in der Genauigkeit ab einer *batch size* von 128 zu sehen. Daraus resultiert die bestätigte Hypothese, dass kleinere *batch sizes* besser für eine hohe Genauigkeit sind.

Aus diesen Ergebnissen resultiert, dass für die nächsten Hypothesen die *batch size* von 16 verwendet wird. Dies ergibt sich aus dem folgenden Grund. Die *batch size* von 16 weißt die höchste Genauigkeit auf den Testdaten auf und ist ein ebenfalls ein guter Kompromiss zwischen einer zu kleinen Betrachtung (1, 4, 8) und einer zu großen Betrachtung (128). Bei einer zu kleinen Betrachtungsgröße würde die Rechenzeit unnötig bei gleichbleibenden Ergebnissen erhöht werden und es würde die Gefahr des Overfittings gefördert werden. Bei einer zu großen Betrachtungsgröße würde die Gefahr des Underfittings bestehen. Deshalb wird die *batch size* von 16 als optimaler Mittelweg gewählt und fortan verwendet.

6.2.4 Hypothese zur verwendeten Aktivierungsfunktion

Die vierte Hypothese behandelt die Frage nach der Art Verwendungen derivierungsfunktion die der Ausgabeschicht In Kapitel 2 wurden einige Aktivierungsfunktionen vorgestellt und es wurde in Kapitel 2.2.2 konkret gezeigt, wie eine *sigmoid*-Funktion nutzbar gemacht werden kann. Auch die *tanh*-Funktion wurde kurz vorgestellt, die auf dem Intervall $(-1; 1)$ definiert ist. Schuldig geblieben sind wir noch der *softmax*-Funktion, die die Besonderheit aufweist, eine Wahrscheinlichkeitsverteilung auf der Ausgabeschicht zu berechnen. Die *softmax*-Formel dazu ist

$$g(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}},$$

aus der ersichtlich wird, dass *softmax*, im Gegensatz zu *sigmoid*, eine eigene Schicht darstellt, bei der alle Ausgaben zusammen eine Wahrscheinlichkeit von eins ergeben. (Buduma & Lacascio, 2017, S. 15) Hieraus wird folgende Hypothese abgeleitet:

Hypothese 4 Für die binäre Klassifikation eignen sich laut der Literatur die *sigmoid*, *softmax* und *tanh*-Aktivierungsfunktionen (Gupta & Dishashree, 2017), siehe auch Kapitel 2.2.2. Aufgrund der eindeutigen Wahrscheinlichkeitsverteilung der *softmax*-Funktion wird mit dieser, als Aktivierungsfunktion auf der Ausgabeschicht, die höchste Genauigkeit erzielt.

	<i>sigmoid</i>	<i>softmax</i>	<i>tanh</i>
acc nach training	1	1	0.5032
loss nach training	1.7287e-04	8.0762e-06	3.9817
acc nach test	1	1	0.50996
loss nach test	0.000149	1.3050e-05	3.9697

Erklärungen:

acc – accuracy

nop – no player

p – player

Tab. 1: Dargestellt ist der Vergleich in der Genauigkeit von den getesteten Aktivierungsfunktionen

Ergebnis 4 Wird das Ergebnis der Auswertung für die drei Aktivierungsfunktionen betrachtet, so ist eine leicht bessere Performanz der *softmax*-Aktivierungsfunktion zu erkennen. Außerdem fällt auf, dass die *tanh*-Funktion gegenüber den beiden anderen Funktionen stark abfällt und diese nur eine Genauigkeit von 50 Prozent nach dem Training und nach dem Testen aufweist. Der Grund dafür könnte die Abbildung in den negative Wertebereich durch die *tanh*-Funktion sein. Werden die Genauigkeiten der *softmax*- und *sigmoid*-Funktion nach dem Training und dem Testen betrachtet, so fällt auf, dass die *softmax* besser abschneidet als *sigmoid*. Dies lässt sich dadurch erklären, dass die *softmax*-Funktion besser für eindeutige Klassifizierungsaufgaben geeignet ist, wie sie hier mit der harten Einteilung in $\{player, noPlayer\}$ vorliegt, während die *sigmoid*-Funktion eher für Klassifizierungsaufgaben gedacht ist, bei denen multiple Fälle gleichzeitig eintreten können, z.B. die gleichzeitige Erkennung mehrerer Objekte in einem einzelnen Bild. Dieser Umstand ist auch

aus der Formel von *softmax* ersichtlich, da eine Ausgabe durch die Summe aller Ausgaben geteilt wird, somit die eine Ausgabe bezüglich aller anderen normiert wird. Da die Aktivierungsfunktion der Ausgabeschicht auch unmittelbar Auswirkung auf das Fehlermaß hat, siehe etwa Gleichung 20, kommt es wahrscheinlich auch zu den aus Tabelle 1 ersichtlichen Unterschieden. Somit wurde die Hypothese betätigt.

Wie in der Hypothese schon gemutmaßt, hat sich empirisch bewiesen, dass die *softmax*-Funktion für den vorliegenden Anwendungsfall der Bildklassifikation die besten Ergebnisse liefert. Somit wird diese zukünftig als Aktivierungsfunktion für die letzte Neuronenschicht übernommen.

6.2.5 Hypothese zur verwendeten Optimierungsfunktion

Als vorletzte Hypothese wird wie schon in der Hypothese 4 zuvor eine Funktion überprüft. In diesem Fall wird jedoch nicht die Aktivierungsfunktion sondern die Optimierungsfunktion überprüft. Auch für diese Hypothese wird sich auf wenige Funktionen beschränkt. Konkret werden die Optimierungsfunktionen: Adam, Adadelta und ein reiner *stochastic gradient descent*, zu Vergleichszwecken, untersucht.

Hypothese 5 Die Adam-Optimierung ist eine der meist genutzten Optimierungsfunktionen für eine adaptive Lernrate bei CNNs und ist daher am besten für das Netz geeignet. (Brownlee, Jason, 2018a, S. 181)

	adadelta	adam	SGD
acc nach training	1	0.4826	0.4875
loss nach training	8.0762e-06	8.2945	8.2147
acc nach test	1	0.49801	0.50199
loss nach test	1.30501e-05	8.04705	7.9831

Erklärungen:

acc – accuracy

nop – no player

p – player

SGD – Stochastic gradient descent

Tab. 2: Dargestellt ist der Vergleich in der Genauigkeit von den getesteten Optimierungsfunktionen.

Ergebnis 5 Betrachtet man die Ergebnisse der unterschiedlichen Optimierungsfunktionen so kann festgehalten werden, dass die Hypothese falsifiziert wurde. Sowohl Adam als auch SGD schneiden in diesem Versuch gleich schlecht ab. Beide Optimizer legen sich auf die kein Spieler Klasse fest und sagen für jeden neuen Validierungsfall die besagte Klasse vorher. Anders ist dies bei dem Adadelta Optimizer, dieser ist in der Lage valide und genaue vorhersagen zu treffen. Betrachtet man das Ergebnis in der Gesamtheit ist es verwunderlich warum der Adam Optimizer so schlechte Resultate liefert und gleichzeitig der Adadelta Optimizer so gute.

Wird das Ergebnis der unterschiedlichen Optimierungsfunktionen betrachtet, so bleibt als Resultat nur die Adadelta Optimierungsfunktion als Alternative übrig, da die anderen beiden Optimierungsfunktionen sich auf eine Klasse festlegen und somit eine schlechte Genauigkeit erreichen, da diese nicht wirklich gelernt haben zwei Klassen vorherzusagen sondern einfach nur eine Klasse statisch vorhersagen.

6.2.6 Hypothese zum *Shuffle* der Trainings- und Testdaten

Wird der Aufbau und die Anordnung der Trainingsdaten betrachtet, so fällt auf das diese in einem 50/50 Split angeordnet wurden. Das bedeutet, dass die Daten auf den ersten 50 Prozent reine Spielerbilder sind und auf den letzten 50 Prozent reine kein Spielerbilder. Damit das Netz diese Werte aber nicht strikt in dieser Reihenfolge anlernt und auch dazu eine diverse Anordnung der Trainingsdaten sinnvoll ist, wurden die Daten geshuffelt. Das bedeutet, dass die Daten nicht in ihrer Ursprungsanordnung (50/50) in das Netz gegeben wurden sondern dass die Positionen durch den *Shuffle* Befehl durchgemischt wurden. Dadurch wird gewährleistet, dass die Daten keinem strikten Muster folgen und das Netz so von Bild zu Bild lernen muss. Daraus resultiert folgende Hypothese:

Hypothese 6 Das *Shuffling* der Daten ist notwendig, um hohe Ergebnisse in der Genauigkeit zu erzielen.

	mit <i>shuffle</i> -Funktion	ohne <i>shuffle</i> -Funktion
acc nach training	1	1
loss nach training	8.0762e-06	5.6105e-05
acc nach test	1	1
loss nach test	1.30501e-05	4.81920e-05

Tab. 3: Dargestellt ist der Vergleich in der Genauigkeit des Netzes mit und ohne *shuffle*-Funktion.

Ergebnis 6 Betrachtet man die Ergebnisse der beiden Anordnungen der Trainingsdaten so fällt auf, dass beide die höchst mögliche Genauigkeitsrate von 1 aufweisen. Lediglich in den unterschiedlichen Fehlermaßen nach dem Training und nach dem Testen haben beide marginale Unterschiede zueinander. Am resultierenden Ergebnis in Form der Genauigkeit ändert sich bei beiden dadurch nichts. Ganz im Gegenteil beide sind trotz einer unterschiedlichen Anordnung der Trainingsdaten in der Lage die gleichen Genauigkeitswerte zu erzielen. Dies ist überraschend, da angenommen wurde, dass die gemischte Anordnung der Trainingsdaten wichtig für das erfolgreiche Lernen

des Netzes ist. Dies wurde jedoch mit den nicht geshuffelten Daten widerlegt und somit wurde die Hypothese widerlegt.

Daraus resultiert keine notwendige Änderung für das Netz, da das *Shuffling* der Daten keine erheblichen Genauigkeitseinbußen darstellt im Vergleich zu den Daten ohne *Shuffling* und somit die Ausgangseinstellung beibehalten werden kann.

6.3 Gesamtes Ergebnis der Hypothesen

Nachdem alle Hypothesen vorgestellt und abgearbeitet wurden und die Erkenntnisse dieser Hypothesen in die Struktur des Netzes mit eingeflossen sind, ergibt sich im Rahmen dieser Arbeit die folgende Netzstruktur, welche in Listing 15 als optimale Lösung zu sehen ist. Trotzdem muss bei dieser optimalen Netzstruktur berücksichtigt werden, dass diese unter den Voraussetzungen der festgelegten Standardstruktur in Kapitel 5 und der getesteten und variierten Parameter entstanden ist. Das bedeutet nicht, dass diese Lösung die einzige und die beste mögliche Lösung ist, jedoch dass diese Lösung der im Rahmen dieser Arbeit festgelegten und getesteten Parameter die beste Lösung für den bearbeiteten Anwendungsfall darstellt.

Wird der Aufbau des Netzes in Listing 15 betrachtet, so fällt auf dass diese sich nicht grundlegend von der Struktur in Kapitel 5 unterscheidet. Dies erscheint auf den ersten Blick logisch da keine Strukturen wie Layer oder Anzahl der Neuronen getestet wurden. Wird jedoch auf die unterschiedlichen Anzahl von Variationen der getesteten Parameter zurückgeblickt fällt auf, das lediglich die *batch size* und die Anzahl der Epochen angepasst wurden. Die Aktivierungsfunktion und Optimierungsfunktion hingegen entsprechen der Ausgangseinstellung.

Wird zusätzlich die sich aus der Struktur des Netzes und den eingestellten Hyperparametern ergebende Fehlermaßkurve betrachtet, so fällt auf, dass diese mit jeder Epoche deutlich abnimmt und einen ebenfalls gewünschten Lernverlauf darstellt. Würde die Fehlermaßkurve zu früh abflachen oder sogar steigen anstatt zu fallen, wäre dies ein eindeutiger Indikator dafür, dass das Netz nicht richtig lernt.

```

1 model = Sequential()
2 model.add(Conv2D(32, kernel_size=(3, 3),
3                 activation='relu',
4                 input_shape=(IMG_SIZE, IMG_SIZE, 3)))
5 model.add(MaxPooling2D(pool_size=(2, 2)))
6 model.add(Conv2D(64, (3, 3), activation='relu'))
7 model.add(MaxPooling2D(pool_size=(2, 2)))
8 model.add(Conv2D(256, (3, 3), activation='relu'))
9 model.add(MaxPooling2D(pool_size=(2, 2)))
10 model.add(Flatten())
11 model.add(Dense(32, activation='relu'))
12 model.add(Dense(NUM_CLASSES, activation='softmax'))
13
14 batch_size = 16
15 epochs = 10
16
17 model.compile(loss='binary_crossentropy',
18                 optimizer=keras.optimizers.Adadelta(),
19                 metrics=['accuracy'])
20
21 history = model.fit(X, y_train,
22                       batch_size=batch_size,
23                       epochs=epochs,
24                       validation_split=0.2,
25                       callbacks=[LearningRateScheduler(lr_schedule),
26                                  ModelCheckpoint('model.h5', save_best_only
27                                     =True)])

```

Listing 15: Optimale Lösung des CNN

Abschließend bleibt festzuhalten, dass die Hypothesen nicht bestätigt oder falsifiziert wurden, um möglichst viel an den Grundeinstellungen zu verändern, sondern neues Wissen für das CNN, insbesondere die Hyperparameter, zu generieren. Dies wurde erfolgreich in diesem Kapitel und mit den bearbeiteten Hypothesen geschafft und somit wurde der Zielsetzung aus Kapitel 1.3 gerecht.

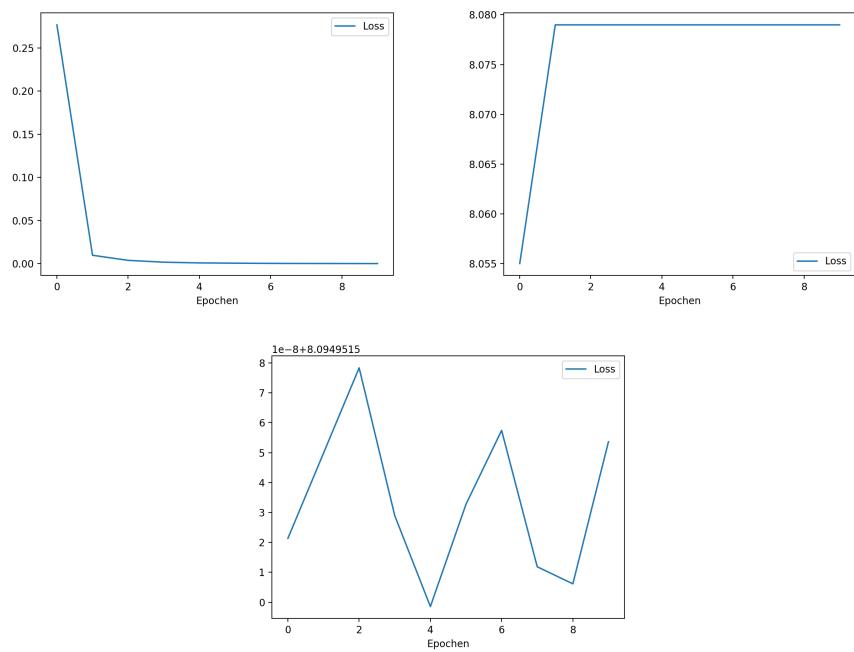


Abb. 30: Dargestellt ist das Fehlermaß der optimalen Lösung für das neuronale Netz für die Epochen hinweg.

7 Ausblick und Fazit

In diesem Kapitel wird ein Ausblick für mögliche Anwendungsfelder des implementierten CNNs gegeben. Am Ende dieses Kapitels wird ein Fazit über die ganze Arbeit gezogen.

7.1 Ausblick

Betrachtet man die Ergebnisse dieser Projektarbeit, so legen diese den Grundstein für eine über Bilddaten mögliche Performanz- und Taktikanalyse im Fußball. Wie diese beiden Möglichkeiten aufzeigen, kann die Erkennung von Spielern grundlegend für zwei Ausrichtungen genutzt werden. Zum einen, kann durch die Lokalisierung von Spielern und ein damit ermöglichtes Tracking der erkannten Spieler, eine mögliche Performanz-Analyse erfolgen. Diese würde Kennzahlen wie z.B. die Laufleistung, die Intensität der Läufe, die Anzahl von geführten Zweikämpfen, Passstatistiken und vieles mehr ermöglichen. Auf der anderen Seite, könnte die ganzheitliche Analyse von Spielsituationen für das Erkennen von Taktikmustern und Schwachstellen bei Gegnern genutzt werden. Beispielsweise könnte man die lokalisierten Spieler in einer 2D-Karte eintragen und somit Staffelungen, Abstände und gewisse Verhaltensmuster von Fußballmannschaften analysieren.

Gerade da diese Arbeit den Grundstein für die oben erwähnten Anwendungsfälle legt, sollte vor allem auch aus technologischer Sicht betrachtet werden, was nicht nur in der Zukunft aus Produktsicht möglich sein kann, sondern auch welche Schritte unternommen werden müssen, um diese Nutzen wirklich realisieren zu können. Betrachtet man dazu das vorhandene System, so ist es absolut notwendig einzelne Spieler zu erkennen. Jedoch sollte dies nicht nur auf perfekt ausgeschnittenen Spielerbildern mit maximal einem Spieler erfolgen, sondern es sollte generell möglich sein, ein Bild in seiner totalen Bildeinstellung einzulesen und wie der YOLO-Algorithmus dies heute schon bereitstellt, direkt auf einem Bild alle Objekte zu erkennen und diese mit *bounding boxen* zu umranden. Beispielhaft ist dies in Abbildung 31 dargestellt.



Abb. 31: Dargestellt ist die erforderliche Weiterentwicklung des in dieser Arbeit vorgestellten Klassifizierungsmodells: Bild links: ohne YOLO, Bild Rechts: mit YOLO.

Betrachtet man anschließend die gleiche Spielsituation in einer transformierten 2D-Darstellung und nimmt die Spielszene als Grundlage, so bieten sich mit der automatischen Generierung dieser Darstellungen neue automatisierte Möglichkeiten in der Spielanalyse und -Betrachtung. Wie in Abbildung 32 dargestellt, ermöglicht die folgende Grafik das Erkennen von Abständen und gefährlichen Räumen für die jeweilige Situation. Außerdem ist somit eine qualitative Bewertung von Spielsituationen und Angriffen möglich.

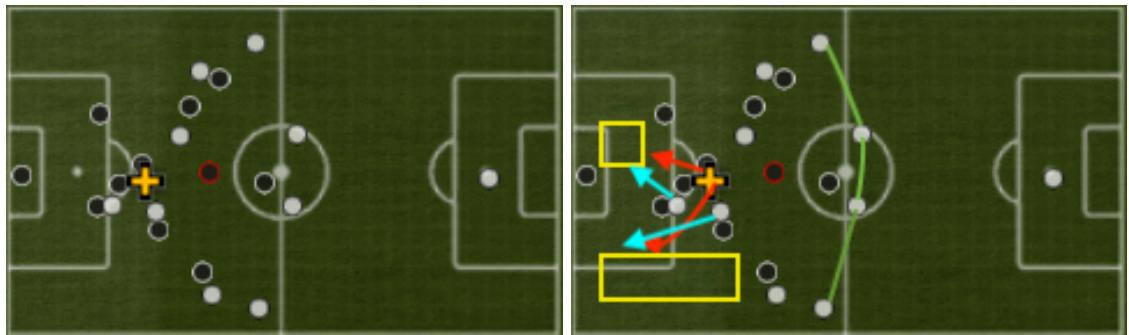


Abb. 32: Dargestellt ist die 2D-Karte, die durch das Erkennen der Spieler entsteht und somit Spielsituationen Analyse ermöglicht.

Erklärungen:

- | | |
|--------------|--------------------------------------|
| Rot Pfeil | – Passweg Angreifer |
| Blauer Pfeil | – Laufweg Angreifer |
| Gelbe Box | – Gefährlicher Raum in der Situation |
| Grüne Kanten | – Staffelung der einzelnen Ketten |
| Kreuz | – Ball + ball-führender Spieler |

Dieser Ansatz fokussiert sich somit auf die Taktik bzw. die Mustererkennung in Spielsituationen. Eine für den Performanz-Bereich in den letzten Jahren immer wichtiger gewordene Kennzahl stellt das sogenannte *Packing* dar. Mit der sogenannten *Packing*-Quote kann die Spielstärke von Spielern und Mannschaften angegeben werden. Die Berechnung der *Packing*-Quote erfolgt wie folgt: Zum bestimmten Zeitpunkt t eines Spiels stehen zwischen dem ballführenden Spieler (dem Passgeber) und dem gegnerischen Tor eine gewisse Anzahl von Gegenspielern, die in diese Situation eingreifen können. Spielt der ballführende Spieler zu dem Zeitpunkt einen Pass und der angezielte Mitspieler (Passemppfänger) nimmt diesen Ball erfolgreich an, bestimmt die *Packing*-Quote die durch diesen Pass überspielte Anzahl von Gegnerischen Spielern zum Zeitpunkt t' . Die Visualisierung dieses Prozesses kann beispielsweise in Abbildung 33 betrachtet werden.

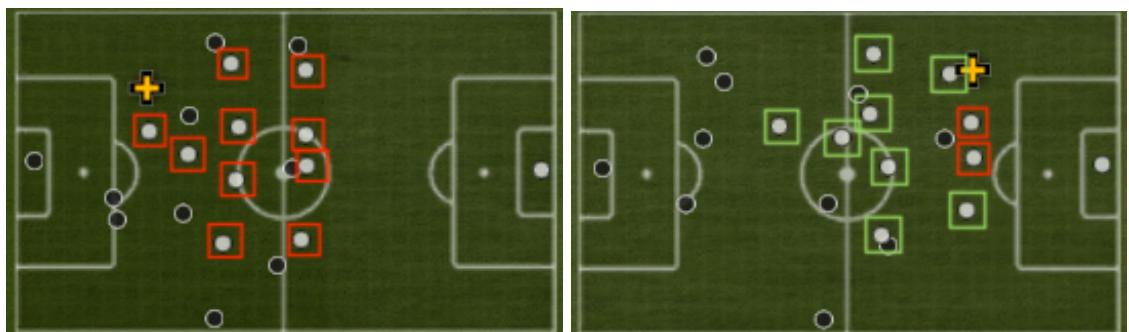


Abb. 33: Dargestellt ist ein beispielhafter *Packing*-Vorgang zum Zeitpunkt t (links: vor dem Pass) und t' (rechts: nach dem Pass).

Erklärungen:

- Rote Box – Gegenspieler die noch eingreifen können
- Grüne Box – Gegenspieler die aus dem Spiel genommen wurden
- Kreuz – Ball + ball-führender Spieler

Wird die obere Abbildung betrachtet, so ergeben sich zwei Kennzahlen. Für den Passgeber ergibt somit die Anzahl der aus dem Spiel genommenen Gegenspieler. Das bedeutet, vor dem Pass zum Zeitpunkt t konnte eine Anzahl n von Gegenspielern die Situation noch verteidigen. Kommt der Ball zum Zeitpunkt t' bei dem Passemppfänger an, besagt die Differenz der zum Zeitpunkt t und t' gezählten Gegenspieler die noch in die Situation eingreifen können. Genauso ist dies aber auch eine Kennzahl für den Passemppfänger, da diesem ebenfalls die aus dem Spiel genommenen Gegenspieler zugerechnet werden und diese somit einen Indika-

tor darstellen, wie oft und wie erfolgreich sich ein Spieler in Räumen oder freien Räumen anbietet. In dem oben aufgezeigten Beispiel wurden mit einem Pass acht Gegenspieler aus dem Spiel genommen.

Dieser Ausblick und somit auch eine Vision zeigt deutlich, wohin der Weg für diese Art der Technologie in dem Profifußball gehen kann und wie viel Potenzial in diesem Bereich steckt. Dies wird durch die Ergebnisse und Modelle der Universität Mainz bestätigt die solche Performanz- und Spielsituationsbewertungen schon mit einem System ermöglichen und dieses bereits unter Bundesliga Bedingungen eingesetzt wird. (Gutenberg, 2014) Dazu wurde seit der Saison 2018/19 den Vereinen und den Nationalmannschaften erlaubt technische Hilfsmittel zu nutzen (Sportschau, 2018) und die Erkenntnisse dieser direkt im Spiel nutzen zu können. Betrachtet man dieses Potenzial so bleibt dieses Gebiet sehr spannend. Jedoch muss unweigerlich darauf hingewiesen werden, dass die in dieser Arbeit erarbeiteten Erkenntnisse den ersten Schritt darstellen, um eine wie oben Beschriebene Analyse auch nur ansatzweise möglich zu machen.

7.2 Fazit

Wird die Zielsetzung aus Kapitel 1.3 rückwirkend betrachtet, so kann festgehalten werden, dass es gelungen ist, ein CNN zu erstellen, welches erfolgreich einzelne Fußballspieler auf Bildern erkennen kann. Zu Beginn der praktischen Implementation des CNN-Algorithmus wurde ersichtlich, dass die Beschaffung von verwendbaren Eingabedaten für das CNN kompliziert ist, da die Generierung von einem ausreichend großen Bilddatensatz einen kreativen und hohen Aufwand vereinnahmte. Dieser Mehraufwand wurde jedoch durch die steile Lernkurve des Keras-Frameworks ausgeglichen. Durch Keras wird jedem Entwickler ein ausreichendes und gut erklärtes Framework bereitgestellt. Dieses verfügt über alle notwendigen Komponenten, um ein CNN nicht nur für diesen Anwendungsfall einfach und schnell zu erstellen. Die letztendliche Modifizierung der festgelegten CNN-Struktur durch die überprüften Hypothesen in Kapitel 6 trug schlussendlich zu einem Erkenntnisgewinn bei den überprüften Hyperparametern bei. Abschließend kann festgehalten werden, dass das CNN die Zielsetzungen erfolgreich erreicht hat. Jedoch muss auch festgehalten werden, dass die erste Ausgangsidee

einer Objektverfolgung zur Metrikbestimmung zu umfangreich für den gestellten Zeitraum war. Wird dazu der verwendete YOLO-Algorithmus betrachtet, so kommt dieser Algorithmus der Ausgangsidee am nächsten. Dies ist jedoch hauptsächlich der Komplexität des Themas geschuldet und letztendlich auch, dass es nicht einfach ist, ein so komplexes System innerhalb von drei Monaten zu programmieren. Da außerdem die Zielsetzung für diese Arbeit nicht darin bestand, eine Blackbox für ein Produkt zu verwenden, sondern in diese Blackbox hineinzuschauen und diese zu verstehen, entspricht das Ergebnis der Arbeit der anfänglich beschriebenen Zielsetzung. Außerdem wurde durch die aufgestellten und überprüften Hypothesen in Kapitel 6 ein wesentlicher Erkenntnisgewinn erzielt und somit eine weitere Zielsetzung der Projektarbeit erfolgreich bearbeitet. Durch die überprüften Hypothesen konnte die für das CNN benötigten Hyperparameter überprüft werden und es wurden für die unter den Bedingungen festgelegten Versuche die besten Einstellungen für die Trainingsmenge, die Epochengröße, die *batch size*, die Aktivierungsfunktion, die Optimierungsfunktion und die Anordnung der Daten (*shuffling*) ermittelt. Somit wurde im Rahmen dieser Projektarbeit nicht nur die erste Zielsetzung zur Erstellung eines CNN bezogen auf den Anwendungsfall erfüllt, sondern auch die zweite Zielsetzung zur Wissensgenerierung über die verwendeten Hyperparameter erfolgreich bearbeitet. Daraus folgt, dass abschließend alle in der Einleitung vorgestellten Zielsetzungen als erfolgreich bearbeitet angesehen werden können.

Literaturverzeichnis

- Blausen, Bruce (2013) Multipolar Neuron, Verfügbar unter: https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png, abgerufen am 04.09.2018.
- Brilliant.org (2018a) Artikel „Artificial Neural Network“, Abschnitt „ANNs as Graphs“, Verfügbar unter: <https://brilliant.org/wiki/artificial-neural-network/>, abgerufen am 04.09.2018.
- Brilliant.org (2018b) Artikel „Backpropagation“, Verfügbar unter: <https://brilliant.org/wiki/backpropagation/>, abgerufen am 13.09.2018.
- Brilliant.org (2018c) Artikel „Convolutional Neural Networks“, Verfügbar unter: <https://brilliant.org/wiki/convolutional-neural-network/>, abgerufen am 14.09.2018.
- Brilliant.org (2018d) Artikel „Feedforward Neural Networks“, Verfügbar unter: <https://brilliant.org/wiki/feedforward-neural-networks/>, abgerufen am 04.09.2018.
- Brilliant.org (2018) Artikel „Machine Learning“, Verfügbar unter: <https://brilliant.org/wiki/machine-learning/>, abgerufen am 03.09.2018.
- Bronshtein, Adi (2017) Train/Test Split and Cross Validation in Python – Towards Data Science, Verfügbar unter: <https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6>, abgerufen am 04.09.2018.
- Brownlee, Jason (2017) How Much Training Data is Required for Machine Learning?, Verfügbar unter: <https://machinelearningmastery.com/much-training-data-required-machine-learning/>, abgerufen am 19.08.2018.
- Brownlee, Jason (2018a) Long Short-Term Memory Networks With Python, 1. Auflage, Jason Brownlee (Eigenpublikation).
- Brownlee, Jason (2018b) What is the Difference Between a Batch and an Epoch in a Neural Network?, Verfügbar unter: <https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/>, abgerufen am 04.09.2018.

Buduma, N. & N. Lacascio (2017) Fundamentals of Deep Learning, 2. Auflage, O'Reilly Media.

deeplearning.net (2017) Artikel „Convolution arithmetic tutorial“, Verfügbar unter: http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html, abgerufen am 14.09.2018.

Gupta, D. & Dishashree (2017) Fundamentals of Deep Learning - Activation Functions and their use, Verfügbar unter: <https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/>, abgerufen am 11.09.2018.

Gutenberg, J. (2014) Soccer – new software developed, Verfügbar unter: <https://www.cs.uni-mainz.de/soccer-new-software/>, abgerufen am 03.09.2018.

Huang, T. S. (1996) Computer Vision: Evolution and Promise, Verfügbar unter: <http://cds.cern.ch/record/400313/files/p21.pdf>, abgerufen am 03.09.2018.

Johnson, Justin (2018) CS231n Convolutional Neural Networks for Visual Recognition - Derivatives, Backpropagation, and Vectorization, Verfügbar unter: <http://cs231n.stanford.edu/handouts/derivatives.pdf>, abgerufen am 11.09.2018.

Kathuria, Ayoosh (2019) What's new in YOLO v3?, erste Abbildung, Verfügbar unter: <https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>, abgerufen am 03.09.2018.

Keras Webseite (2015) Keras: The Python Deep Learning library, Verfügbar unter: <https://keras.io/>, abgerufen am 19.08.2018.

Lavrenko, Victor et. al. (2018) Introductory Applied Machine Learning, Verfügbar unter: <https://www.inf.ed.ac.uk/teaching/courses/iaml/slides/eval-2x2.pdf>, abgerufen am 13.09.2018.

Li, FeiFei (2013) CS 131 Computer Vision: Foundations and Applications, Lecture 13, Verfügbar unter: http://vision.stanford.edu/teaching/cs131_fall1314_nope/lectures/lecture13_kmeans_cs131.pdf, abgerufen am 03.09.2018.

- Li, FeiFei et. al. (2018a) CS231n Convolutional Neural Networks for Visual Recognition - Convolutional Neural Networks Lecture Notes, Verfügbar unter: <http://cs231n.github.io/convolutional-networks/>, abgerufen am 14.09.2018.
- Li, FeiFei et. al. (2018b) CS231n Convolutional Neural Networks for Visual Recognition - Image Classification Lecture Notes, Verfügbar unter: <http://cs231n.github.io/classification/>, abgerufen am 04.09.2018.
- Li, FeiFei et. al. (2018c) CS231n Convolutional Neural Networks for Visual Recognition - Linear Classification Lecture Notes, Verfügbar unter: <http://cs231n.github.io/linear-classify/>, abgerufen am 08.09.2018.
- Li, FeiFei et. al. (2018d) CS231n Convolutional Neural Networks for Visual Recognition - Neural Network Lecture Notes, Verfügbar unter: <http://cs231n.github.io/neural-networks-1/>, abgerufen am 10.09.2018.
- Liu, Wangxin (2017) Lecture notes of LSTM's Inventor Sepp Hochreiter, Verfügbar unter: http://wangxinliu.com/machine+learning/machine+learning+basic/deep+learning/research&study/LSTM_Lecture-en/, abgerufen am 04.09.2018.
- Mallick, Satya (2017) Object Tracking using OpenCV (C++/Python), Verfügbar unter: <https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>, abgerufen am 04.09.2018.
- Mathworks.com (2017) Was ist die Objekterkennung?, Verfügbar unter: <https://de.mathworks.com/solutions/deep-learning/object-recognition.html>, abgerufen am 04.09.2018.
- Mathworks.com (2018) Unsupervised Learning, Verfügbar unter: <https://de.mathworks.com/discovery/unsupervised-learning.html>, abgerufen am 03.09.2018.
- McCormick, Chris (2013) K-Fold Cross-Validation, With MATLAB Code, Verfügbar unter: <http://mccormickml.com/2013/08/01/k-fold-cross-validation-with-matlab-code/>, abgerufen am 04.09.2018.

Pereira, Sergio et. al. (2016) Brain Tumor Segmentation using Convolutional Neural Networks in MRI Images, Verfügbar unter: http://dei-s2.dei.uminho.pt/pesosas/csilva/brats_cnn/, abgerufen am 03.09.2018.

Pilotte, Paul (2016) Analytics-driven embedded systems, erste Abbildung, Verfügbar unter: <http://www.embedded-computing.com/embedded-computing-design/analytics-driven-embedded-systems-part-2-developing-analytics-and-prescriptive-controls>, abgerufen am 03.09.2018.

Sadowski, Peter (2018) Notes on Backpropagation, Verfügbar unter: <https://www.ics.uci.edu/~pjsadows/notes.pdf>, abgerufen am 13.09.2018.

Sportschau (2018) DFL gibt elektronische Kommunikation für die Bundesliga frei, Verfügbar unter: <https://www.sportschau.de/fussball/bundesliga/bundesliga-technische-hilfsmittel-erlaubt-100.html>, abgerufen am 03.09.2018.

Steppan, Josef (2017) A few samples from the MNIST test dataset., Verfügbar unter: <https://commons.wikimedia.org/wiki/File:MnistExamples.png>, abgerufen am 03.09.2018.

Zhang, C. (2017) How to choose Last-layer activation and loss function, Verfügbar unter: <https://www.dlogy.com/blog/how-to-choose-last-layer-activation-and-loss-function/>, abgerufen am 03.09.2018.