

Gymnasium Bäumlihof, 5Bb

MATURAARBEIT

# Kann der Computer Werbung erkennen?

Bilderkennung mit einem Neuronalen Netzwerk

*Georg Schwan*

Betreuungsperson

*Test1*

Korreferent

*Test2*

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
1.1 Motivation . . . . .	3
1.2 Aufbau der Arbeit . . . . .	3
<b>2 Problemstellung</b>	<b>4</b>
2.1 Logo . . . . .	4
2.2 Neuronales Netzwerk . . . . .	5
<b>3 Neuronales Netzwerk</b>	<b>6</b>
3.1 Konzept . . . . .	6
3.2 Neuron . . . . .	6
3.3 Architektur . . . . .	7
3.3.1 Beschriftung . . . . .	8
3.4 Wie das Netzwerk lernt . . . . .	8
3.4.1 Kostenfunktion . . . . .	8
3.4.2 Gradient Descent . . . . .	9
3.4.3 Backpropagation . . . . .	10
3.5 Aktivierungsfunktionen . . . . .	12
3.5.1 Sigmoid . . . . .	12
3.5.2 Rectified Linear Units . . . . .	12
3.5.3 Softmax . . . . .	13
3.6 Convolution . . . . .	14
3.6.1 Architektur . . . . .	14
3.6.2 Max Pooling . . . . .	16
3.6.3 Convolution und Völlig Verbundene Schichten . . . . .	17
3.7 Regularization . . . . .	17
3.7.1 Dropout . . . . .	17
3.7.2 Batch Normalization . . . . .	17
<b>4 Lösungsansatz</b>	<b>19</b>
4.1 Logo . . . . .	19
4.2 Werbung erkennen mit Logo . . . . .	20
4.3 Werbung erkennen ohne Logo . . . . .	21

<b>5 Umsetzung</b>	<b>22</b>
5.1 Neuronales Netzwerk . . . . .	22
5.1.1 Aufbau . . . . .	22
5.1.2 Benutzung . . . . .	23
5.1.3 Grafikkarten unterstützung . . . . .	23
5.1.4 Bild bearbeitung . . . . .	23
5.2 Webserver . . . . .	24
<b>6 Ergebnisse</b>	<b>25</b>
6.1 Auswertung . . . . .	25
6.1.1 Datensatz . . . . .	25
6.1.2 Methoden . . . . .	25
6.2 Neuronales Netzwerk mithilfe des Logo's . . . . .	26
6.2.1 Netzwerk Architektur . . . . .	26
<b>7 Reflexion</b>	<b>28</b>
<b>Literaturverzeichnis</b>	<b>29</b>
<b>Abbildungsverzeichnis</b>	<b>30</b>

# Kapitel 1

## Einleitung

### 1.1 Motivation

Vor ein paar Jahren haben wir Fernsehen geschaut und immer wenn Werbung kam haben wir den Sender gewechselt, bis die Werbung vorbei war und das normale Programm weiter lief. Das Problem war nur, dass wir nie wussten wann die Werbung vorbei war. Meinem Bruder ist aufgefallen, dass bei Werbung nie das Logo vom Sender eingespielt wird. Daraufhin hat er probiert ein Algorithmus zu schreiben, der das Logo von einem Sender erkennen kann. Er versuchte das Logo mithilfe von Bedingungen und Schleifen auszudrücken, aber vergebens.

Als ich auf der Suche nach einer Idee für eine Maturaarbeit war erinnerte ich mich wieder an das Problem und an einen neuen Lösungsansatz, nämlich Neuronale Netzwerke, welche Heute überall verwendet werden und extrem Mächtig sind. Die Idee war aber nicht nur ein Logo zu erkennen sondern auch genau verstehen wie ein neuronales Netzwerk funktioniert und warum es so Mächtig ist.

### 1.2 Aufbau der Arbeit

# Kapitel 2

## Problemstellung

Das Ziel dieser Arbeit ist einen Algorithmus zu programmieren der Bilder als Werbung erkennen kann. Dafür wird ein neuronales Netzwerk benutzt, das sich auf die Bilderkennung beschränkt.

### 2.1 Logo

Wie schon gesagt wird bei Werbung das Senderlogo nicht eingeblendet und deswegen kann das Problem vereinfacht werden auf die Frage ob das Senderlogo eingeblendet ist oder nicht (siehe Abbildung 2.1). Man könnte meinen, dass das erkennen eines Logo's relativ simple ist. Zum



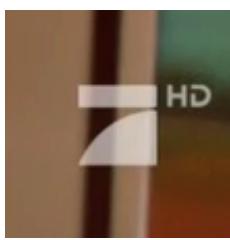
(a) Werbung



(b) Keine Werbung

Abbildung 2.1: Kein Senderlogo bei Werbung

Beispiel könnte man Schauen ob der Bereich, wo das Logo sein sollte, heller ist als ausserhalb.



(a)



(b)



(c)



(d)

Abbildung 2.2: Logo mit verschiedenen Hintergründen

Ein Problem des Logo's ist, dass es nicht einfach über das normale Bild eingespielt wird,

sondern man kann leicht hindurch sehen (siehe Abbildung 2.2a), dadurch kann man das Logo nicht an gleichen Pixel erkennen. Die grösste Schwierigkeit ist aber, dass bei manchen Hintergründen, vor allem bei Weissen, das Logo abgeschnitten oder kaum bis gar nicht sichtbar ist. Bei Abbildung 2.2 (b) ist das Logo abgeschnitten, bei (c) ist es kaum sichtbar und bei (d) ist komplett verschwunden.

Ein andere Schwierigkeit ist, dass das Logo nicht unbedingt immer am gleichen Ort sein muss. Zum Beispiel hat Prosieben 3 verschiedene Positionen (siehe Abbildung 2.3). Alle diese

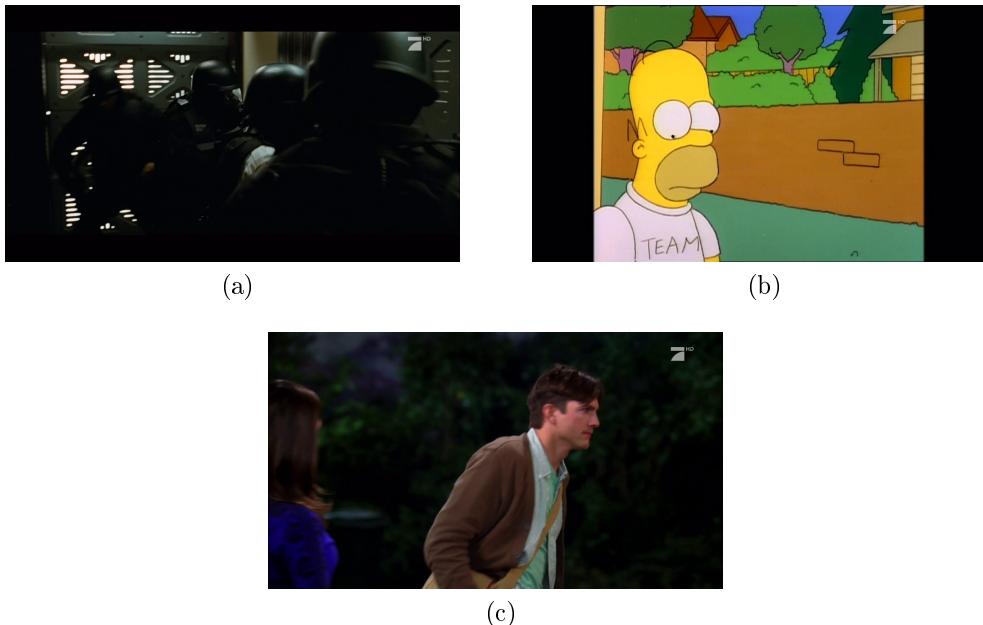


Abbildung 2.3: Logo an verschiedenen Positionen

Schwierigkeiten machen das erkennen eines Logo's ohne ein neuronales Netzwerk extrem schwer. Ein neuronales Netzwerk hingegen löst das Probleme ziemlich elegant.

Eine weiter interessante Frage auf die eingegangen wird ist, ob ein neuronales Netzwerk auch ohne ein Logo Werbung erkennen kann.

## 2.2 Neuronales Netzwerk

Neuronale Netzwerke sind ein sehr umfangreiches Thema und deswegen begrenze ich mich auf Netzwerke die für die Bilderkennung entscheidend sind. Darunter sind klassische feedforward und convolution Netzwerke. Auf recurrent neuronale Netzwerke<sup>1</sup> wird nicht näher eingegangen.

---

<sup>1</sup>Ein neuronales Netzwerk, das geeignet für Sequenzen ist

# Kapitel 3

## Neuronales Netzwerk

Dieses ganze Kapitel bezieht sich auf das Buch von Michael A. Nielsen[6], ausser es ist anders angegeben.

### 3.1 Konzept

Wenn man ein normales Programm schreiben will muss man das Problem in viele kleinere aufteilen, bis der Computer fähig ist, es zu lösen. In einem Neuronale Netzwerk wird dem Computer nicht gesagt wie es das Problem lösen kann, sondern ein neuronales Netzwerk versteht das Problem, indem es beispiel Daten bekommt und an ihnen lernen kann, bis es seine eigene Lösung gefunden hat. Zum Beispiel, wir wollen einem Netzwerk beibringen ob in einem Bild ein Auto vorkommt, dazu geben wir dem neuronalem Netzwerk viele Bilder, mit und ohne Auto. Mit jedem Bild, dass das neuronale Netzwerk bekommt, lernt es besser wie ein Auto ausschaut.

Das Konzept eines neuronales Netzwerk ist nicht etwas Neues. Im Jahre 1957 hat Frank Rosenblatt ein erste Idee eines neuronales Netzwerk vorgestellt. Die Idee war aus mathematischen Funktionen unser Gehirn zu modellieren. Indem man die biologischen Neuronen und Synapsen als mathematische Funktion ausdrückt.

Es ist aber erst in den letzten Jahren ist der grosse Hype für neuronale Netzwerke ausgebrochen, dies liegt daran, dass man erst jetzt die nötigen Daten und Rechenleistung zu Verfügung hat.

### 3.2 Neuron

User Gehirn kann Entscheidungen treffen, da wir Billionen von Neuronen haben, die miteinander verbunden sind und sich verständigen können. Ein Neuron an sich ist praktisch nutzlos, aber in grosser Anzahl können sie komplexeste Probleme lösen.

Nach dem gleichen Prinzip funktioniert ein neuronales Netzwerk, Es besteht aus vielen Neuronen (daher der Name) die miteinander verbunden sind.

Ein Neuron in einem neuronales Netzwerk wird als mathematische Funktion definiert wie Abbildung 3.1 verdeutlicht. Ein Neuron hat  $n$  verschiedene Eingaben, die als  $x_j$  bezeichnet

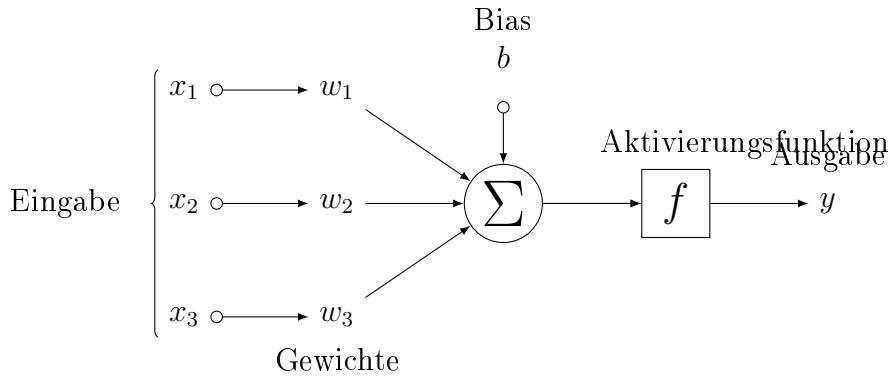


Abbildung 3.1: Einzelner Neuron in einem Neuronalen Netzwerks

werden und mit einem spezifischen Gewicht  $w_j$  multipliziert werden. Die Ausgabe erfolgt, indem man alle gewichteten Eingaben, mit einem Bias  $b$ , addiert und durch eine so genannte Aktivierungsfunktion  $f$  durchlaufen lässt. Eine klassische Aktivierungsfunktion ist die Sigmoid Funktion  $f(x) = \frac{1}{1+e^{-x}}$ , welche den Wert zwischen 0 und 1 normalisiert. Als Gleichung:

$$y = f \left( \sum_{j=1}^n x_j w_j + b \right)$$

Die Gewichte  $w_j$  und der Bias  $b$  des Neurons sind die Parameter, die angepasst werden und somit das Neuron lernfähig machen.

Eine Aktivierungsfunktion ist nötig, da ohne eine wäre ein neuronales Netzwerk eine komplett lineare funktion, welches nur lineare Probleme lösen könnte[1], da in einem Netzwerk nur multipliziert und addiert wird. Durch die Aktivierungsfunktion kommt eine nicht lineare Funktion hinzu, welche das Netzwerk komplizierter machen, aber auch mächtiger, da es so Verbindungen lernen kann, die man nicht vorher nicht herstellen konnte. Ohne diese Aktivierungsfunktion wäre das Netzwerk nicht in der Lage komplizierte Zusammenhänge wie auf Bilder oder Sprache zu erkennen.[1] Es wird näher auf die Aktivierungsfunktion eingegangen im Abschnitt 3.5

### 3.3 Architektur

Wie auch im biologischen Gehirn ist ein Neuron allein nutzlos. Erst wenn man die Neuronen miteinander verbindet kann es komplexe Zusammenhänge modellieren.

Eine mögliche Architektur kann wie in Abbildung 3.2 ausschauen. Ein Netzwerk wird generell immer in verschiedene Schichten unterteilt. Die linke Schicht wird als Eingabe Schicht bezeichnet und die Neuronen in dieser Schicht werden Eingabe Neuronen genannt. Analog dazu wird die rechte Schicht Ausgabe Schicht genannt, die die Ausgabe Neuronen beinhaltet. Die mittleren Schichten, die von der Anzahl her variieren können, werden versteckte Schichten genannt. Die Anzahl der Neuronen in jeder Schicht kann auch variieren. Abbildung 3.3 zeigt eine andere mögliche Architektur für ein Netzwerk, welches 2 versteckte Schichten hat. Jedes

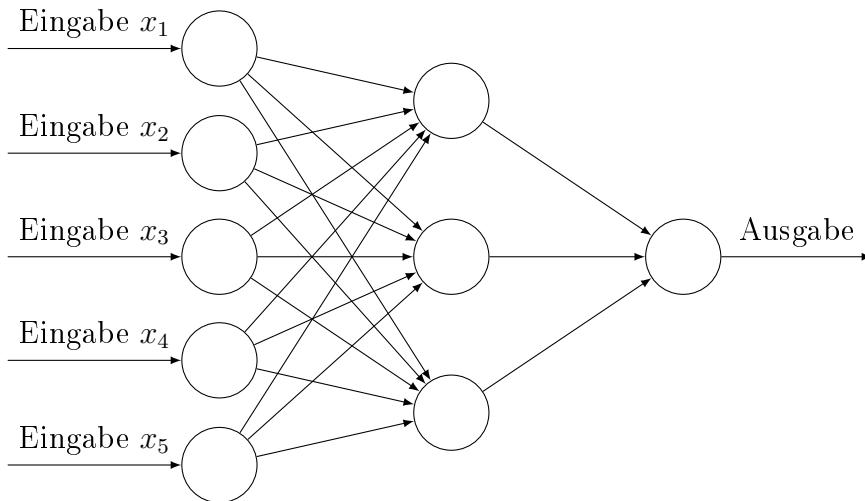


Abbildung 3.2: Mögliche Architektur eines Neuronalen Netzwerk

Neuron der vorigen Schicht ist mit jedem Neuron der nachfolgenden Schicht verbunden, welches als *völlig verbundene Schicht* bezeichnet werden. Dies ist ein klassisches feedforward Netzwerk, welches nur Verbindungen nach vorne hat und so keine Schleifen entstehen können<sup>1</sup>.

### 3.3.1 Beschriftung

Um eine allgemeine Gleichung zu bestimmen, muss man zuerst die Elemente des Netzwerks benennen. Wir bezeichnen das Gewicht  $w_{k,j}^l$  für die Verbindung des  $k^{ten}$  Neuron der  $(l-1)^{ten}$  Schicht zu dem  $j^{ten}$  Neuron der  $l^{ten}$  Schicht. Ähnlich dazu bezeichnen wir die Ausgabe des Neurons als  $a_j^l$  und der Bias des Neurons als  $b_j^l$  (siehe Abbildung 3.4).<sup>2</sup>

Mit dieser Notation kann eine Gleichung für das Netzwerk aufgestellt werden, welche der Gleichung einem Neuron ähnelt 3.2.

$$a_j^l = f \left( \sum_k a_k^{l-1} w_{k,j}^l + b_j^l \right)$$

## 3.4 Wie das Netzwerk lernt

Bis jetzt ging es nur darum wie ein neuronales Netzwerk aufgebaut ist. In dem Abschnitt geht wie ein neuronales Netzwerk, anhand von Daten, lernen kann

### 3.4.1 Kostenfunktion

Damit ein Netzwerk lernen kann muss man dem Netzwerk zuerst sagen können wie gut oder wie schlecht es gerade ist. Dazu definieren wir eine Kostenfunktion  $C$ , die von allen Gewichten  $w$  und allen Biases  $b$  abhängig ist. Der Ausgabewert des kompletten Netzwerk wird als  $y$  bezeichnet

---

<sup>1</sup>Es gibt auch Architekturen in denen Schleifen vorkommen, aber auf diese wird nicht näher eingegangen

<sup>2</sup>Das  $l$  dient nur zur Indexierung und nicht als Potenz

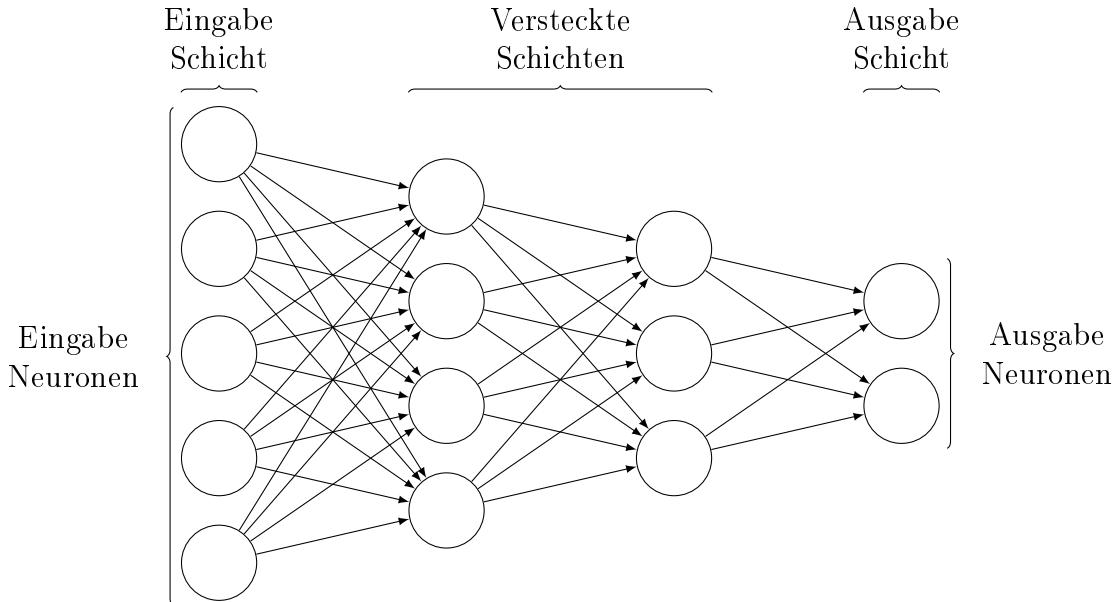


Abbildung 3.3: Neuronales Netzwerk mit 2 versteckten Schichten

und die entsprechende gewünschte Ausgabe als  $l$ . Beachte, dass  $y$  und  $l$  Vektoren sind. Zum Beispiel würde ein Bild, das 10x10 Pixel gross ist, einen  $(10 \times 10 =) 100$ -dimensionaler Vektor dargestellt werden, wobei jeder Eintrag im Vektor der Grauwert eines Pixels ist. Die Dimension vom ausgabe Vektor  $y$  und des gewünschten ausgabe Vektor  $l$  entspricht der Anzahl Neuronen in der letzten Schicht des Netzwerks, wobei jedes Neuron etwas bestimmten Aussagt. Zum Beispiel könnte ein Neuron für das vorkommen eines Auto, im Bild, stehen, wobei 0 für kein Auto und 1 für ein Auto steht.

$$C(w, b) = \sum_j (y_j - l_j)^2$$

Das Ziel des Netzwerkes ist diese Kostenfunktion zu minimieren, bis so viele beispiel Daten wie möglich  $C \approx 0$  entsprechen, dies geschieht wenn die Ausgabe des Netzwerks und die gewünschte Ausgabe ähnlich ist.

### 3.4.2 Gradient Descent

Um diese Kostenfunktion zu minimieren wird ein Algorithmus names *gradient descent* benutzt. Das Konzept basiert darauf, dass man eine Funktion, in Abhängigkeit einer Variablen, ableiten kann und so die Steigung (eng. gradient), an diesem Punkt, berechnen kann und die Variable richtung Minimum anpasst.

Zum Beispiel hat man eine Kostenfunktion  $C(x)$  die von  $x$  abhängig ist (siehe Abbildung 3.5).

Die Variable  $x$  wird am Anfang einen zufällige Werte zugewiesen, welches dem Orangen Punkt auf der Abbildung 3.5 entspricht und das Ziel ist  $s$  so anzupassen, dass man ein Minimum der Kostenfunktion findet. Um ein Minimum zu finden kann man sich einen Ball vorstellen, der in ein Minimum herunterrollt. Um dies zu berechnen muss man die Steigung, mithilfe einer

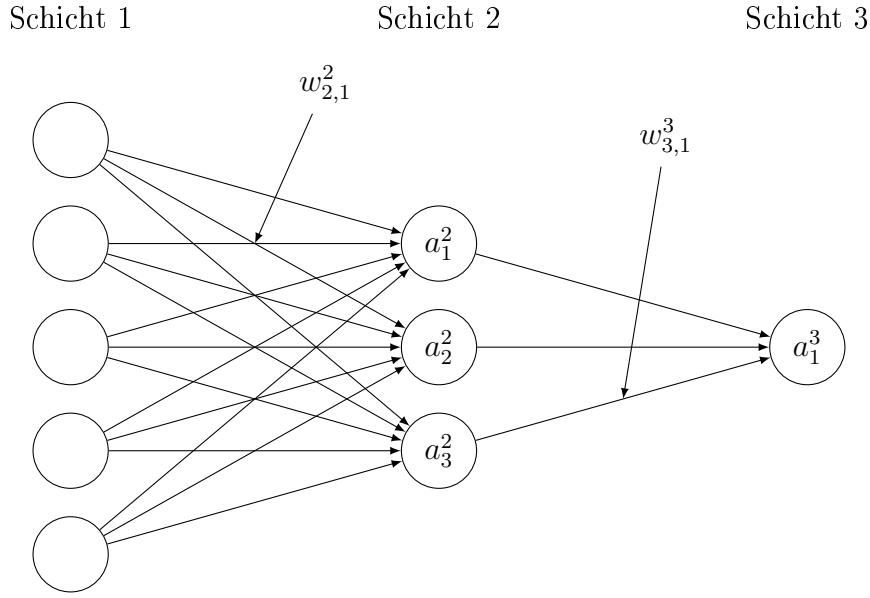


Abbildung 3.4: Bezeichnung der Parameter

Ableitung, herausfinden und die Variable in die gegensätzliche Richtung bewegen.

$$a \rightarrow a' = a - \eta \frac{\partial C}{\partial a}$$

wobei  $\mu$  eine kleine positive Zahl (learning rate genannt) ist, die die Geschwindigkeit der Bewegung steuert. Außerdem beachte, dass der Ball keine Beschleunigung hat. Wenn man diese Gleichung iterativ anwendet gelangt man früher oder später zum lokalen Minimum der Kostenfunktion (siehe Abbildung 3.6).

Der Algorithmus funktioniert auch bei mehr als nur einer Variable und lässt sich für die Gewichte und Biases des Netzwerkes genau gleich berechnen.

$$\begin{aligned} w_{k,j}^l \rightarrow w_{k,j}^{l'} &= w_{k,j}^l - \eta \frac{\partial C}{\partial w_{k,j}^l} \\ b_j^l \rightarrow b_j^{l'} &= b_j^l - \eta \frac{\partial C}{\partial b_j^l} \end{aligned}$$

Durch dieses Verfahren kann zwar relativ einfach ein Minimum gefunden werden, dabei ist aber zu beachten, dass es sich um ein lokales Minimum handelt und kein globales.

### 3.4.3 Backpropagation

Der Algorithmus um  $\frac{\partial C}{\partial w_{k,j}^l}$  und  $\frac{\partial C}{\partial b_j^l}$  zu berechnen wird als Backpropagation bezeichnet und ist der mathematisch schwerste Teil dieser Arbeit. Es ist aber nicht unbedingt nötig für das Verständnis eines Neuronale Netzwerkes. Es wird auch nicht näher auf die Beweise der Glei-

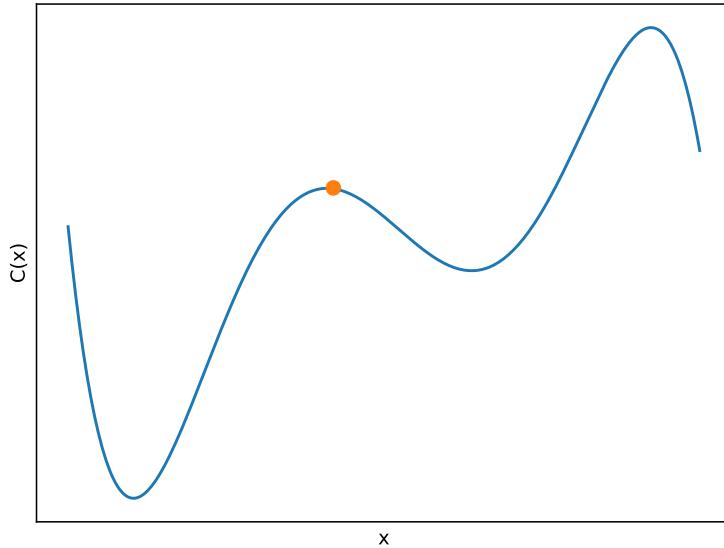


Abbildung 3.5: Kostenfunktion in abhängigkeit von  $x$

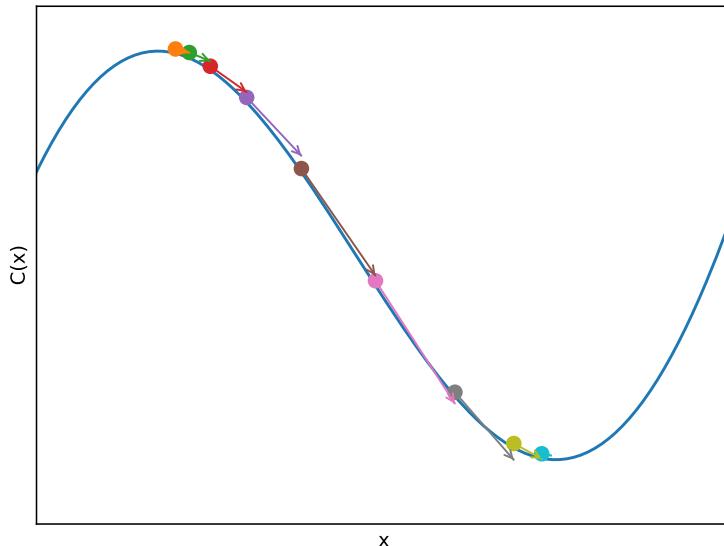


Abbildung 3.6: 2-dimensionaler Verlauf des gradient descent

chungen eingegangen, da es sonst komplizierter wird und im Grunde ist es nur die Anwendung der Kettenregel.

Um die Übersicht zu behalten wird eine Zwischenmenge  $\delta_j^l$  eingeführt, welches als *Fehler* bezeichnet wird. Der Fehler sagt aus wie gut oder schlecht ein Neuron ist und ist definiert als:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

wobei  $z_j^l$  die Ausgabe von einem Neuron ohne die Aktivierungsfunktion ist, also  $a_j^l = f(z_j^l)$ . Mit dieser Definition kann man den Fehler in der letzten Schicht  $L$  bestimmen:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} f'(z_j^L)$$

In unserem Fall benutzen wir eine quadratische Kostenfunktion  $C = \sum_j (a_j^L - y_j)^2$  bei der die Ableitung  $\frac{\partial C}{\partial a_j^L} = 2(a_j^L - y_j)$  ist und können  $\delta_j^L$  einfacher definieren als:

$$\delta_j^L = 2(a_j^L - y_j)f'(z_j^L)$$

Bei der Berechnung des Fehlers  $\delta_j^l$  Abhängig von  $\delta_j^{l+1}$  bekommt man:

$$\delta_j^l = \sum_k w_{j,k}^{l+1} \delta_k^{l+1} f'(z_j^l)$$

Beachte, dass bei  $w_{j,k}^{l+1}$  das  $j$  und  $k$  vertauscht sind, so dass man durch alle Neuronen der  $(l+1)^{ten}$  Schicht durch iteriert. Mit dieser Gleichung kann man jeden Fehler von jeder Schicht berechnen, indem man von hinten durch das Netzwerk durchläuft. Ähnlich wie wenn man sich beim Netzwerk nach vorne bewegt.

Die Gleichung für die Änderungsrate der Kosten in Bezug auf ein Bias im Netzwerk ist genau der Fehler:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

Die Gleichung für die Änderungsrate der Kosten in Bezug auf ein Gewicht im Netzwerk:

$$\frac{\partial C}{\partial w_{k,j}^l} = a_k^{l-1} \delta_j^l$$

## 3.5 Aktivierungsfunktionen

Das einzige was noch fehlt ist wie eine Aktivierungsfunktionen genau ausschaut. Wie schon gesagt darf eine Aktivierungsfunktionen nicht linear sein, da sie sonst nichts dem Netzwerk beiträgt.

### 3.5.1 Sigmoid

Ein Beispiel für eine Aktivierungsfunktion ist die Sigmoid Funktion  $f(x) = \frac{1}{1+e^{-x}}$  (siehe Abbildung 3.7). Besonders an dieser Funktion ist, dass sie den Ausgabewert zwischen 0 und 1 eingrenzt, was uns erlaubt den Ausgabewert des ganzen Netzwerkes besser zu deuten, als wenn der Wert zwischen  $-\infty$  und  $\infty$  liegt. Ein Problem der Sigmoid Funktion ist, dass wenn die Ausgabe nah bei 1 oder 0 ist, dann ist die Ableitung  $f'(x)$  davon auch nah bei 0, was den Fehler  $\delta_j^l$  sehr klein hält und so das Netzwerk nur noch langsam lernen lässt. Dieses Problem ist als *vanishing gradient problem* bekannt.

### 3.5.2 Rectified Linear Units

Eine andere populäre Aktivierungsfunktion ist die Rectified linear units Funktion oder kurz ReLu. Die Funktion  $f(x) = \max(0, x)$  (siehe Abbildung 3.8). löst das Problem des vanishing

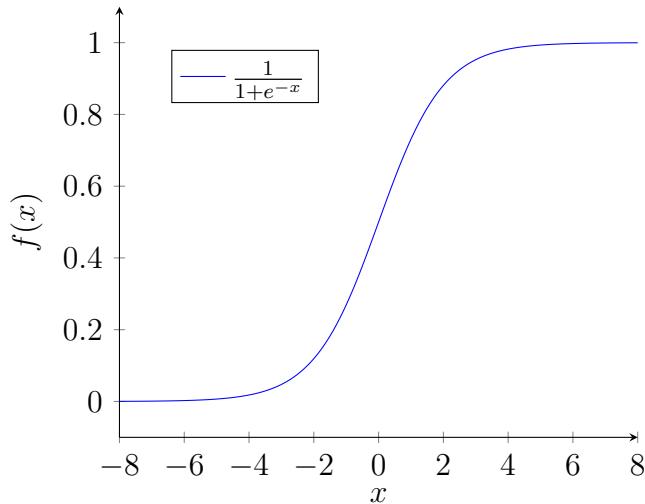


Abbildung 3.7: Sigmoid Aktivierungsfunktionen

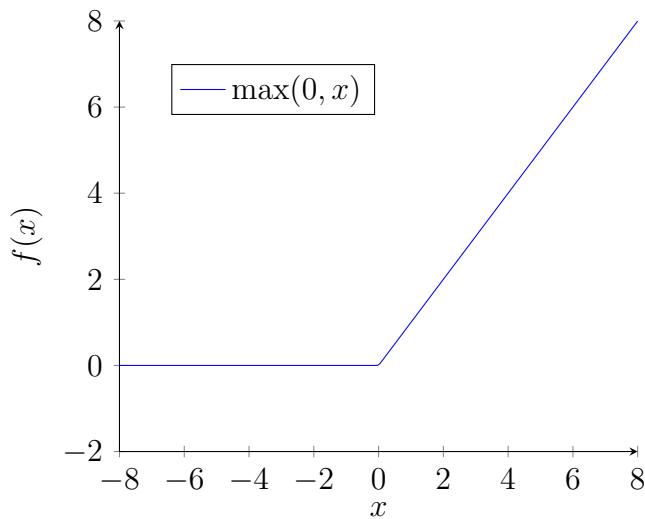


Abbildung 3.8: ReLu Aktivierungsfunktionen

gradient und praktisch alle Neuronalen Netzwerke benutzen Relu als ihre Aktivierungsfunktion, da es die besten Ergebnisse erbringt.[1] Ein Nachteil ist, dass sie nur in den Versteckten Schichten gut funktioniert, da der Ausgabewert der Funktion unendlich gross sein kann.

### 3.5.3 Softmax

Die softmax Funktion wird verwendet, um eindeutige Klassifikationen zu machen und ist definiert als:

$$f(x_i) = \frac{e^{x_i}}{\sum_k e^{z_k}}$$

Das besondere an dieser Aktivierungsfunktion ist, dass sie nicht nur einen Wert braucht, sondern alle Werte der ganzen Schicht, d.h nicht nur ein  $x_i$  sondern alle. Außerdem gibt die Summe aller Resultate  $\sum_i f(x_i) = 1$  und kann deswegen als eine Wahrscheinlichkeitsverteilung verstanden werden. Dies ist oft sehr hilfreich, da viele Probleme nur ein richtiges Resultat haben, zum Beispiel hat man Bilder von Zahlen, wo immer nur eine Zahl pro Bild zu sehen ist. Und durch

die softmax Funktion sieht man dann eine geschätzte Wahrscheinlichkeit vom Netzwerk für jede Zahl.

## 3.6 Convolution

Bis jetzt ging es nur um Schichten die völlig miteinander verbunden sind. Für die Bilderkennung kann das suboptimal sein, da bestimmte Eigenschaften eines Bildes nicht miteinbezogen werden, wie zum Beispiel die Beziehung von nebeneinander liegenden Pixel und das gesuchte Objekt in einem Bild an verschiedenen Orten vorkommen kann.

### 3.6.1 Architektur

Die Eingabe für einen convolutional Schicht ist nicht 1-Dimensional, sondern 2-Dimensional (siehe Abbildung 3.9). Die Neuronen werden normal verbunden, einfach mit dem Unterschied,

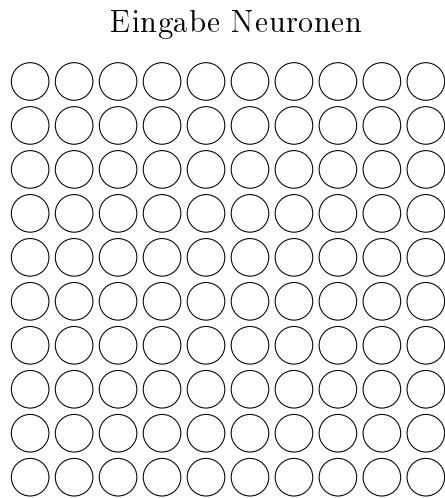


Abbildung 3.9: Eingabe Neuronen für eine convolution Schicht

dass nicht jeder Neuron mit jedem Neuron verbunden wird, sondern dass nur ein bestimmter Bereich zum nächsten Neuron verbunden ist. Dieser Bereich wird als *Filter* bezeichnet und in dem Beispiel auf Abbildung 3.10 wird ein 3x3 Filter benutzt. Der Filter wird dann auf den Eingabe Neuronen um ein Neuron verschoben, um den nächsten Neuron zu verbinden. Und so geht das weiter, auch nach unten, bis die ganze versteckte Schicht gemacht wurde. Dabei wird die versteckte Schicht auch kleiner, in dem Beispiel wird die 10x10 Schicht zu einer 8x8 Schicht, da der Filter irgendwann am anderen Rand anstösst. Abbildung 3.11 verdeutlicht das Prinzip noch einmal.

Der Filter kann auch um mehr als nur einen Neuronen verschoben werden, und man kann in den beiden Richtungen verschiedene Werte nehmen, zum Beispiel bewegt sich der Neuron nach links um zwei Neuronen und nach unten um drei.

Das entscheidende am Filter ist, dass er die gleichen Gewichte und Bias verwendet für die Verbindung, d.h bei einem Filter von 5x5 gibt es  $(5 * 5 =) 25$  verschiedene Gewichte und einen

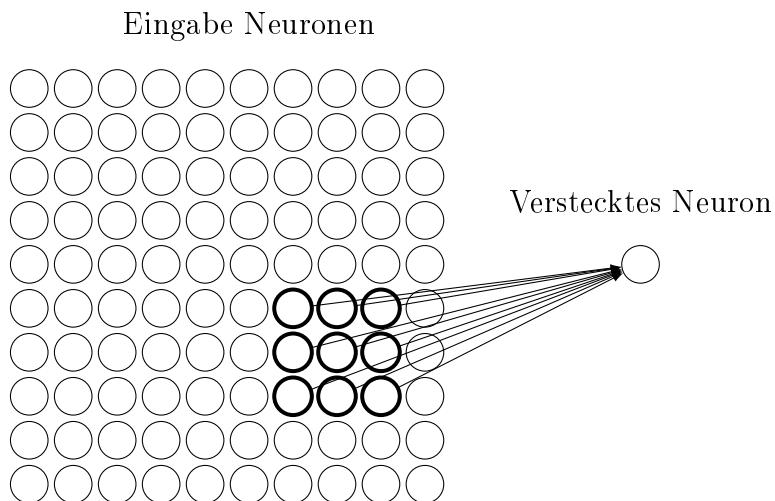


Abbildung 3.10: Verbindung eines versteckten Neurons in einem convolution Schicht

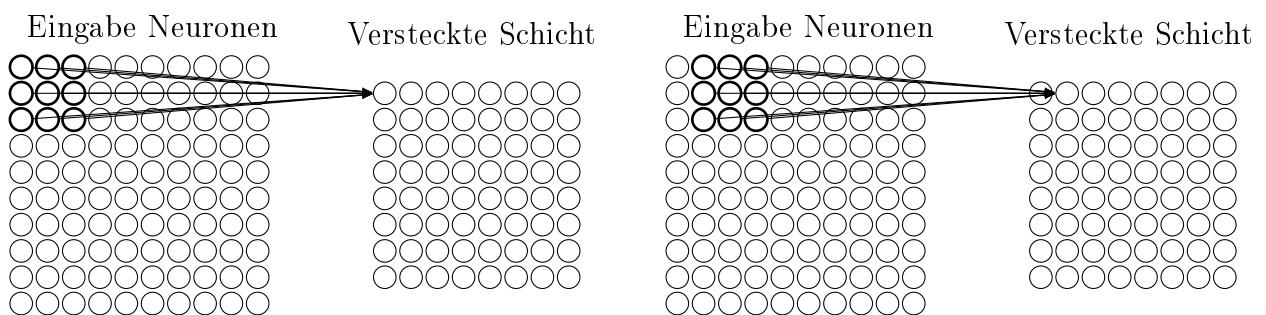


Abbildung 3.11: Bewegung eines Filters über eine convolution Schicht

Bias. Wenn der Filter bewegt wird werden immer die gleichen Gewichte und der gleiche Bias verwendet. Als Gleichung bei einem 3x3 Filter:

$$a_{j,k}^{l+1} = f \left( \sum_{p=0}^2 \sum_{m=0}^2 w_{p,m}^l a_{j+p,k+m}^l + b^l \right)$$

wobei  $a_{x,y}$  der Neuron an der Position  $x, y$  ist und  $f$  eine Aktivierungsfunktion. Dadurch dass immer die gleichen Gewichte und der gleiche Bias für jeden Filter benutzt werden, wird überall das gleiche Merkmal erkannt auch wenn es sich an einem anderen Ort befindet, deswegen wird die Ausgabe von dem Filter als *feature map* bezeichnet. Normalerweise will man mehr als nur ein Merkmal erkennen und deswegen werden mehrere Filter verwendet, wodurch mehrere feature maps entstehen (siehe Abbildung 3.12). Der Grund warum die Filter nicht alle das gleiche Merkmal erkennen, liegt an der zufälligen Initialisierung der Gewichte und Biases.

Die feature map kann noch grösser gemacht werden, indem ein Rand von Neuronen, mit dem Wert 0, hinzugefügt wird.[2] Das führt dazu, dass sich der Filter über die normale feature map hinweg bewegt, was in manchen Fällen erwünscht ist.

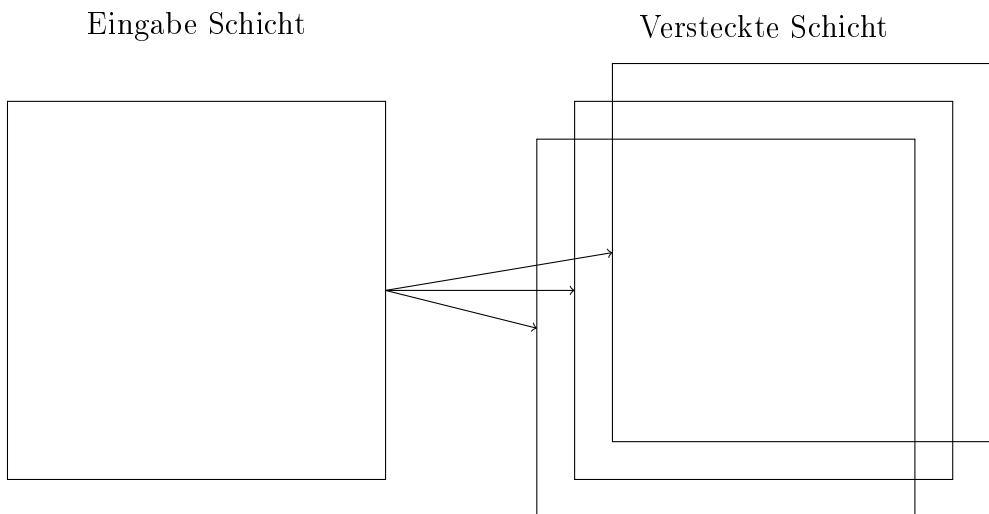


Abbildung 3.12: Convolution Schicht mit 3 Feature Maps

### 3.6.2 Max Pooling

Neben den Convolution Schichten gibt es auch die maxpool Schicht<sup>3</sup>, die Idee vom Pooling ist, dass es die Informationen zusammenfasst.

max polling nimmt eine Feature Map als Eingabe und lässt auch etwas ähnliches wie ein Filter darüber laufen. Der Filter bewegt sich genau gleich wie ein Normaler mit den Unterschied das es keine lernbaren Parameter hat und die Ausgabe des Filters der grösste Wert von den Eingaben ist (siehe Abbildung 3.13). Man kann max pooling verstehen als eine reduzierung der

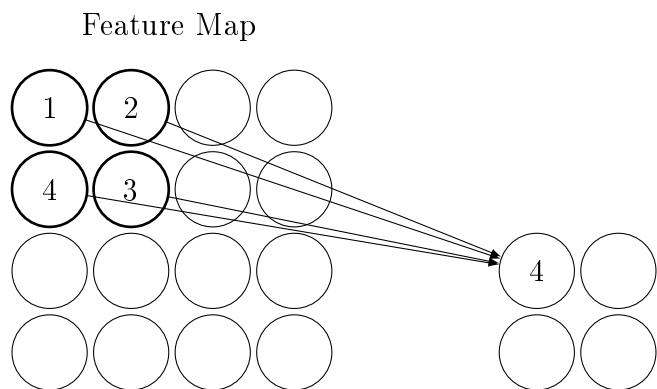


Abbildung 3.13: 2x2 maxpool Schicht mit einer Schrittweite von 2

vorhanden Informationen. Es nimmt das wichtigste Merkmal in einem gewissen Bereich und wirft die weniger wichtigeren Merkmale weg, so dass es weniger Neuronen gibt und die darauf folgenden Neuronen es einfacher haben.

---

<sup>3</sup>Es gibt noch andere pooling Schichten, die aber in dieser Arbeit nicht verwendet wurden

### 3.6.3 Convolution und Völlig Verbundene Schichten

Um das Netzwerk zu interpretieren muss man eine völlig verbundenen Schicht am Schluss haben, da man eine 2-Dimensionale Schicht nicht interpretieren kann.<sup>4</sup> Diese wird angehängt indem jedes Neuron von der convolution bzw maxpool Schicht mit jedem Neuron der völlig verbundenen Schicht verbunden wird. Es spielt keine Rolle, dass die Neuronen 2-Dimensional angeordnet sind.

Das trainieren des Netzwerk ist immer noch genau gleich. Es wird immer noch gradient descent und backpropagation benutzt, allerdings müssen die Gleichungen der backpropagation für die Convolution und max pooling angepasst werden.

## 3.7 Regularization

Um ein Netzwerk zu trainieren hat man meistens nur eine endliche Anzahl an Daten an denen das Netzwerk lernen kann. Aus dem Grund werden die gleichen Daten mehrmals zum trainieren verwendet. Dadurch kann ein Problem entstehen. Mit der Zeit kennt das Netzwerk die trainings Daten so gut, dass es die trainings Daten einfach auswendig lernt und die Daten nicht mehr an ihren gemeinsamen Merkmalen und Zusammenhängen erkennt, sondern an ihren ganz spezifischen Merkmalen die nur für die trainings Daten zutreffen, so das unbekannte Daten nicht mehr richtig erkannt werden. Dieses Phänomen ist als *overfitting* bekannt.

### 3.7.1 Dropout

Beim trainieren eines Netzwerkes werden Neuronen mit einer bestimmten Wahrscheinlichkeit temporär deaktiviert bzw ignoriert (siehe Abbildung 3.14). Bei jeder neuen Eingabe zum trainieren werden neue zufällige Neuronen ausgewählt, dabei sind die eingabe und ausgabe Neuronen davon ausgenommen.

Wenn das genug oft wiederholt wurde, wurden bestimmte Gewichte und Biases gelernt, unter der Bedingung das ein Teil der Neuronen deaktiviert sind. Das heist wenn man das Netzwerk dann benutzt werden mehr Neuronen gleichzeitig Aktiv sein als beim trainieren, um das Auszugleichen wird die Ausgabe des Neuron mit der Wahrscheinlichkeit, mit der es deaktiviert wird, multipliziert.

Dropout hilft gegen overfitting, da sich ein Neuron nicht auf bestimmte andere Neuronen verlassen kann, wodurch es gezwungen ist mit vielen zufälligen Verbindungen etwas nützliches anzufangen. Anders gesagt das Netzwerk wird robust gegen den Verlust von einzelnen Merkmalen und kann sich nicht auf ein einzelnes Merkmale verlassen.

### 3.7.2 Batch Normalization

Eine weitere Methode um overfitting zu vermeiden ist die *Batch Normalization*. Die Eingaben für ein Netzwerk werden normalisiert, damit die Wahl der learning rate nicht auch noch von den

---

<sup>4</sup>Es gibt auch Netzwerke, bei denen man eine convolution Schicht als Ausgabeschicht hat[9]

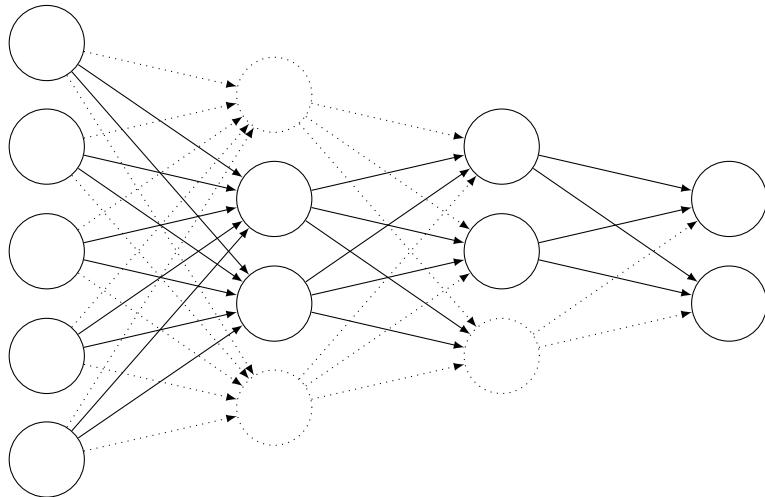


Abbildung 3.14: neuronales Netzwerk mit Dropout

Eingaben abhängig ist. Batch Normalization führt diese Idee weiter und normalisiert nicht die Eingabe, sondern die Ausgabe der versteckten Schichten so dass in den versteckten Schichten extrem hohen Werte vermieden werden.[3] Ähnlich wie bei Dropout bringt Batch Normalization eine leichte Störung in das Netzwerk, welches gegen overfitting hilft[3]

Die Schichten werden normalisiert indem beim trainieren mehrere Eingaben gleichzeitig durch das Netzwerk laufen und die Ausgabe eines Neurones mit dem Mittelwert der Eingaben subtrahieren und mit der Standardabweichung der Eingaben dividieren.[3]

$$\begin{aligned}\mu &= \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma &= \sqrt{\frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2} \\ \hat{x}_i &= \frac{x_i - \mu}{\sigma}\end{aligned}$$

wobei  $x_i$  die  $i^{te}$  Eingabe ist.

Ausserdem wird danach noch mit einem bestimmten Wert  $\gamma_i$  multipliziert und einem bestimmten Wert  $\beta_i$  addiert.

$$y_i = \gamma * \hat{x}_i + \beta$$

wobei  $\gamma$  und  $\beta$  Parameter sind die beim Netzwerk trainiert werden, wie ein normales Gewicht.[3] Die Parameter werden benutzt um dem Netzwerk die Option zu geben die Normalisierung zu verändern oder sogar rückgängig zu machen, wenn es meint, dass es ohne besser funktioniert.[4]

# Kapitel 4

## Lösungsansatz

### 4.1 Logo

Um das Logo zu erkennen muss man zuerst verstehen wie es auf das Bild gelangt. Man könnte meinen das Logo wird einfach über das ander Bild gelegt wird, aber das Logo wird eingespielt indem es mit dem Bild negativ multipliziert wird, d.h die das Bild und das Logo werden invertiert, multipliziert und dann wieder invertiert.[10]

$$f(a, b) = 1 - (1 - a)(1 - b)$$

wobei  $a$  ein Bild ist und  $b$  das Logo und die Werte von jedem Pixel von 0 (Schwarz) bis 1 (Weiss) gehen.

Das bewirkt, dass man leicht durch das Logo hindurchsehen kann, wodurch der Hintergrund hinter dem Logo auch eine Rolle spielt (siehe Abbildung 4.1a,b). Eine andere Eigenschaft ist,

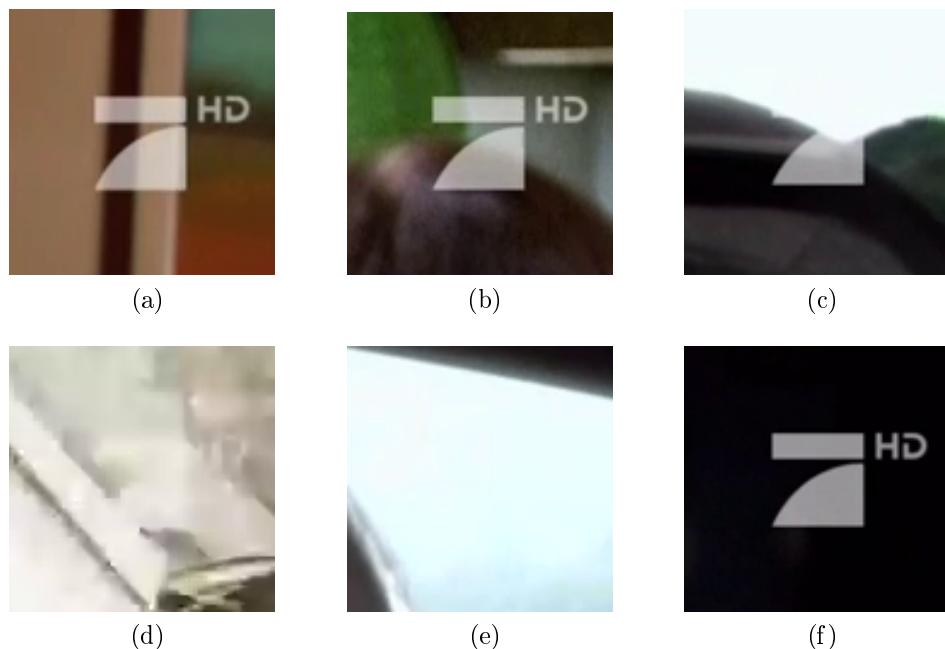


Abbildung 4.1: Logo mit verschiedenen Hintergründen

dass bei manchen Hintergründen, vor allem bei weissen, das Logo abgeschnitten oder kaum bis gar nicht sichtbar ist (siehe Abbildung 4.1c,d,e). Das ist auch einfach zu erklären. Wenn man für  $a = 1$  (Weiss) in die Formel oben einsetzt erhält man:  $f(1, b) = 1$ , was bedeutet, dass das ganze Bild nun Weiss ist und das Logo nicht mehr erkennbar ist. Analog dazu, wenn man für  $a = 0$  (Schwarz) einsetzt erhält man:  $f(0, b) = b$ , was bedeutet, dass das Logo, bei einem schwarzem Hintergrund, sich im ursprünglichen Zustand befindet. Aus dem Grund kann man das Logo komplett herausfiltern und wieder verwenden wenn man das Logo mit einem schwarzen Hintergrund findet (siehe Abbildung 4.1f).

## 4.2 Werbung erkennen mit Logo

Um ein Netzwerk zu trainieren braucht man sehr viele Daten, die schon kategorisiert sind. Eine Option, diese zu beschaffen, wäre die Bilder vom Sender selber zu kategorisieren, das Problem dabei, ist das es eine sehr lange und sehr langweilig arbeit wäre, da man um die millionen Bilder kategorisieren müsste. Die andere Möglichkeit ist die Bilder selber zu generieren, indem man das Logo, auf einen schwarzen Hintergrund, auf viele andere Bilder darauf multipliziert. Das Logo, auf einen schwarzen Hintergrund, findet man indem man lang genug im Sender darauf wartet. Die anderen Bilder bekommt man aus dem Open Images Dataset V3[5], welches um die 9 millionen Bilder enthält. Da diese Bilder des Open Images Dataset nicht in der richtigen Grösse vorhanden sind, werden sie zerteilt in die richtige Grösse. Die Bilder könnten auch auf die richtige Grösse skaliert werden. Das Problem dabei ist, dass dadurch viel weniger Bilder pro Sekunden erzeugt werden und das den ganzen Process des Lernen deutlich verlangsamen würde.

Das Logo kann nun mit dem Bild negativ multipliziert, so dass das Logo an einem zufälligen Ort auf dem Bild erscheint (siehe Abbildung 4.2). Man könnte sich auch überlegen, ob man das Logo nur an Position einspielt, wo es auch im Sender vorkommt. Aber für ein convolution Netzwerk spielt es keine grosse Rolle und der Algorithmus sollte so allgemein wie möglich sein. Die grösse von einem Bild ist 320x180 Pixel, da ein grösseres Bild unnötig viele Informationen

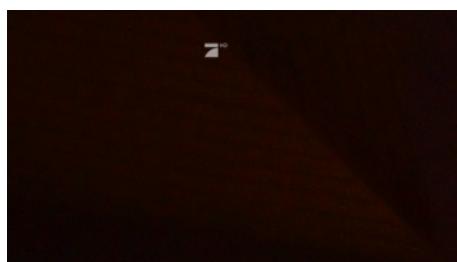


Abbildung 4.2: Selbst generiertes Bild mit einem Prosieben Logo

enthält, was dazu führt, dass das Netzwerk länger lernen müsste. Ein kleineres Bild, würde das Logo noch kleiner machen und kaum noch erkennbar (siehe Abbildung 4.3).

Mit diesen selber generierten Bildern kann man ohne Probleme ein neuronales Netzwerk trainieren, welches das Logo erkennen kann.



Abbildung 4.3: Prosieben Logo (17x11 Pixel) herausgenommen aus einem 320x180 Pixel Bild

### 4.3 Werbung erkennen ohne Logo

Um Werbung zu erkennen ohne ein Logo braucht es die richtige Senderbilder, da Werbebilder nicht generiert werden können. Da ich die Senderbilder nicht selber kategorisieren will benutze ich das neuronale Netzwerk das mithilfe des Logo die Werbung erkennt, um die Senderbilder zu kategorisieren. Ein Problem dabei ist, dass das Netzwerk nicht perfekt ist, um das ein bisschen auszugleichen wird die durchschnittliche Vorhersage des Netzwerk genommen, da Werbung bzw. das normale Programm immer ein weile am Stück läuft, bevor es wechselt

Damit sichergestellt ist, dass das neue Netzwerk nicht wieder das Logo erkennt, sondern die Werbung, wird das Bild so zugeschnitten, dass das Logo nicht mehr auf dem Bild vorhanden ist.

# Kapitel 5

## Umsetzung

Der ganze Code kann auf Github unter <https://github.com/GeorgOhneH/WerbeSkip> gefunden werden. Die verwendete Programmiersprache ist Python 3.

### 5.1 Neuronales Netzwerk

Der Code für diesen Teil befindet sich im Ordner *deepnet*.

Die Grundidee der Implementation für das neuronale Netzwerk ist, dass es Objekt Orientiert ist und man leicht neue Schichten und Funktionen hinzufügen kann. Die Benutzerschnittstelle ist angelegt an Keras, eine Bibliothek für neuronale Netzwerke. Als Grundbaustein wird Numpy, eine Bibliothek für wissenschaftliche Datenverarbeitung, benutzt. Da die Geschwindigkeit ein entscheidender Punkt ist, ist alles was möglich ist als eine Matrix Multiplikation implementiert.

#### 5.1.1 Aufbau

**Schichten** Jede Type von Schicht ist als eine Klasse implementiert, welche von einer Basisklasse erbt, damit jede Schicht gleich behandelt werden kann. Jede Schichten implementieren jeweils den vorwärts und den rückwärts (backpropagation) Gangs des Netzwerks. Die Schichten sind unabhängig, bekommen aber immer die Ausgabe der vorigen Schicht, bzw. der hintern Schicht bei der backpropagation. Die Werte die sie brauchen für die backpropagation, werden intern in jeder Schicht gespeichert.

**Kostenfunktionen** Jede Kostenfunktion ist auch eine Klasse und erbt auch von einer Basisklasse. Jede Kostenfunktion benötigt die Implementation von der Funktion und deren Ableitung.

**Optimierer** Ein Optimierer enthält die Funktionen für des gradient descent, bzw. eine Variante davon, da es gewisse Varianten des gradient descent gibt die den Prozess des Lernens noch beschleunigen können, z.B. wird noch eine simulierte Beschleunigung in den gradient descent mit einbezogen.[8] Auch die Optimierer Klasse erbt von einer Basisklasse. Der Optimierer wird an jede Schicht weiter geleitet um die Gewichte und Biases anzupassen.

**Netzwerk** Dies ist die Hauptklasse, die alle die Schichten, Kostenfunktionen und Optimierer zusammen setzt und sie sich so untereinander verständigen können und ist die Benutzerschnittstelle. Außerdem implementiert die Klasse noch diverse Funktionen, die zur Auswertung des neuronale Netzwerk hilfreich sind.

### 5.1.2 Benutzung

Beispiel Code kann unter *deepnet/examples* gefunden werden.

Um das Programm zu benutzen muss zuerst die eingabe Dimensionen von den Bildern bestimmen werden, welche der eingabe Schicht entspricht. Bei einer nur völlig verbundenen Netzwerk ist die Anzahl Neuronen, der eingabe Schicht. Wenn es ein convolution Netzwerk ist, dann besteht die eingabe Dimensionen aus 3 Zahlen. Die erste Zahl ist die Anzahl feature map, der eingabe Schicht, welche der Anzahl Farbkanäle im Bild entspricht, die Zweite die Höhe des Bildes und die Dritte die Breite des Bildes. Danach müssen die verschiedenen Schichten bestimmt werden, darunter sind auch die Aktivierungsfunktionen, da sie auch als eine Schicht implementiert sind. Als nächstes muss die Kostenfunktion und der Optimierer definiert werden. Am Schluss muss das neuronale Netzwerk noch trainiert werden, indem die trainings Daten dem Netzwerk gegeben werden. Die trainings Daten müssen ein numpy array sein mit der gleichen Form, wie die eingabe Dimensionen mit dem Unterschied, dass der numpy array noch an erster Stelle eine weiter Dimension hat, welcher die Anzahl der Bilder enthält. Die training Daten können auch als Generator übergeben werden.

### 5.1.3 Grafikkarten unterstützung

Der Code kann unter *numpywrapper* gefunden werden.

Die Geschwindigkeit des Programmes spielt eine entscheidende Rolle für ein neuronales Netzwerk und die Geschwindigkeit der CPU<sup>1</sup> ist nicht ausreichend. Um die GPU<sup>2</sup> zu benutzen, wird Cupy verwendet. Da Cupy genau die gleichen Funktionen wie Numpy hat, kann es einfach mit Numpy ausgetauscht werden, Dazu wird ein selbst geschriebenes Module verwendet um einfach zwischen beide hin und her zuschalten. Die GPU ist schneller als die CPU, da alles mit Hilfe von matrix Multiplikationen implementiert ist und die GPU auf matrix Multiplikationen spezialisiert ist.

### 5.1.4 Bild bearbeitung

Der Code kann unter *helperfunctions/image\_processing* gefunden werden.

Der Ordner enthält die Funktionen für die Beschaffung und Formatierung der Bilder, damit sie vom Netzwerk benutzt werden können. Um die Bilder zu bearbeiten wird OpenCV 2<sup>3</sup> und Numpy verwendet.

---

<sup>1</sup>Central processing unit

<sup>2</sup>Grafikkarte

<sup>3</sup>Bibliothek von funktionen um Bilder zu bearbeiten

Die grösste Herausforderung war die Beschaffung der livestream Bilder von einem Sender. Um die sender Bilder zu erhalten, wird der Teleboy Stream angezapft, nach einem Beispiel von Github.[7] und wird durch ffmpeg<sup>4</sup> dekodiert.

## 5.2 Webserver

Der Code kann unter *src* gefunden werden.

Um das fertige Programm auch zu benutzen wird eine Webseite verwendet, die unter der URL *werbeskip.com* erreichbar ist. Für das Backend des Servers wird Django<sup>5</sup> und Django Channels verwendet, wodurch ein WebSocket benutzt werden kann, so dass die Verbindung mit dem Server offen bleibt und die Seite immer auf dem aktuellen Stand ist..

Für das Frontend wird VueJs und VuetifyJS verwendet und es ist als eine Single Page Application implementiert.

---

<sup>4</sup>Programm für das Aufnehmen, Konvertieren und Streamen von Audio und Video

<sup>5</sup>Ein web framework

# Kapitel 6

## Ergebnisse

### 6.1 Auswertung

#### 6.1.1 Datensatz

Um die Leistung der Netzwerke richtig zu bewerten, wird ein selbst erstellter Datensatz verwendet, welches aus insgesamt 7830 Prosieben Bildern besteht. Aus den 7830 Bildern sind 4495 Bilder mit Logo und ohne Rand, 813 mit Logo und einem oberen und untern schwarzen Rand, 813 mit Logo und einem schwarzen Rand auf beiden Seiten und 2095 Bilder ohne Logo und ohne Rand. Außerdem enthält der Datensatz noch 400 Bilder, die nicht das klassische Prosieben Logo enthalten (siehe Abbildung 6.1), welche exkludiert sind vom Datensatz.



Abbildung 6.1: Kein klassisches Prosieben Logo

#### 6.1.2 Methoden

**loss** Die einfachste Methode das Netzwerk auszuwerten ist der Wert der Kostenfunktion anzuschauen, welcher als *loss* bezeichnet wird. So tiefer der *loss* ist umso besser ist das Netzwerk.

**Genauigkeit** Eine andere relativ einfache Methode ist die Genauigkeit des Netzwerks zu bestimmen, indem man die richtig erkannten Bilder durch die totale Anzahl Bilder teilt. Das Problem bei der Genauigkeit ist, dass sie nicht sehr Aussagekräftig ist, wenn das Datensatz nicht ausgeglichen ist. Zum Beispiel hat man ein Datensatz von 100 Bildern, 90 von den Bildern

haben ein Logo und 10 haben keins. Wenn jetzt ein neuronales Netzwerk immer sagt, dass ein Logo auf dem Bild ist, ergebe das eine Genauigkeit von 90%, was nach einem gutes Ergebnis ausschaut, aber das Netzwerk ist im Grunde nutzlos, da es das Logo nicht erkennt, sondern immer die gleiche Ausgabe ergibt.

**Matthews correlation coefficient** Der Matthews correlation coefficient[?] behebt genau dieses Problem. MCC<sup>1</sup> unterscheidet nicht nur zwischen Falschen und Wahren Vorhersagen des Netzwerk, sondern unterscheidet auch zwischen wahr positiv, wahr negativ, falsch positiv und falsch negativ (Siehe Tabelle 6.1). MCC gibt einen Wert von -1 bis +1 zurück. Der Wert +1

		Wahrer Zustand	
		Zustand positiv	Zustand negativ
Vorausgesagt Bedingung	Vorhergesagter Zustand positiv	wahr positiv	falsch positiv
	Vorhergesagter Zustand negativ	falsch negativ	wahr negativ

Tabelle 6.1: My table

repräsentiert eine perfekte Vorhersage, 0 nicht besser als eine zufällige und -1 eine komplette Unstimmigkeit zwischen Netzwerk und dem Datensatz. Der MCC wird nach folgender Formel berechnet[?]:

$$MCC = \frac{WP \cdot WN - FP \cdot FN}{\sqrt{(WP + FP)(WP + FN)(WN + FP)(WN + FN)}}$$

wobei WP die Anzahl der wahr positiven ist, WN die Anzahl der wahr negativen ist, FP die Anzahl der falsch positiven ist und FN die Anzahl der falsch negativen ist.

## 6.2 Neuronales Netzwerk mithilfe des Logo's

### 6.2.1 Netzwerk Architektur

- Convolution, Filter: 16, Filterbreite: 12, Filterhöhe: 8, Schrittweite 4
- BatchNorm
- ReLU
- Convolution, Filter: 64, Filterbreite: 6, Filterhöhe: 4, Schrittweite 1, Neuronenrand 2
- BatchNorm
- ReLU
- Maxpool, Filterbreite: 3 Filterhöhe: 3, Schrittweite 1
- Convolution, Filter: 128, Filterbreite: 4, Filterhöhe: 4, Schrittweite 1
- BatchNorm
- ReLU

---

<sup>1</sup>Matthews correlation coefficient

- Convolution, Filter: 128, Filterbreite: 3, Filterhöhe: 3, Schrittweite 2
- BatchNorm
- ReLU
- Convolution, Filter: 256, Filterbreite: 3, Filterhöhe: 3, Schrittweite 1
- BatchNorm
- Dropout 25%
- Völlig verbundene Schicht, Neuronen: 512
- BatchNorm
- ReLU
- Dropout 50%
- Völlig verbundene Schicht, Neuronen: 2
- Softmax

# Kapitel 7

## Reflexion

# Literaturverzeichnis

- [1] Anish Singh Walia. Activation functions and it's types- which is better? <https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f>, 2015. [Online; accessed 4. November 2018].
- [2] cs231n. Convolutional neural networks (cnns / convnets). <http://cs231n.github.io/convolutional-networks/>, 2018. [Online; accessed 4. November 2018].
- [3] Firdaouss Doukkali. Batch normalization in neural networks. <https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c>, 2017. [Online; accessed 4. November 2018].
- [4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.
- [5] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*, 2017.
- [6] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018. [Online; accessed 4. November 2018].
- [7] Roman Haefeli. watchteleboy. <https://github.com/reduzent>, 2018. [Online; accessed 4. November 2018].
- [8] Sebastian Ruder. An overview of gradient descent optimization algorithms. <http://ruder.io/optimizing-gradient-descent/>, 2016. [Online; accessed 4. November 2018].
- [9] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for Simplicity: The All Convolutional Net. *ArXiv e-prints*, December 2014.
- [10] Wikipedia contributors. Blend modes — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Blend\\_modes](https://en.wikipedia.org/wiki/Blend_modes), 2018. [Online; accessed 4. November 2018].

- [11] Wikipedia contributors. Matthews correlation coefficient — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Matthews\\_correlation\\_coefficient](https://en.wikipedia.org/wiki/Matthews_correlation_coefficient), 2018. [Online; accessed 4. November 2018].

# Abbildungsverzeichnis

2.1	Kein Senderlogo bei Werbung . . . . .	4
2.2	Logo mit verschiedenen Hintergründen . . . . .	4
2.3	Logo an verschiedenen Positionen . . . . .	5
3.1	Einzelner Neuron in einem Neuronalen Netzwerks . . . . .	7
3.2	Mögliche Architektur eines Neuronalen Netzwerk . . . . .	8
3.3	Neuronales Netzwerk mit 2 versteckten Schichten . . . . .	9
3.4	Bezeichnung der Parameter . . . . .	10
3.5	Kostenfunktion in abhängigkeit von $x$ . . . . .	11
3.6	2-dimensionaler Verlauf des gradient descent . . . . .	11
3.7	Sigmoid Aktivierungsfunktionen . . . . .	13
3.8	ReLU Aktivierungsfunktionen . . . . .	13
3.9	Eingabe Neuronen für eine convolution Schicht . . . . .	14
3.10	Verbindung eines versteckten Neurons in einem convolution Schicht . . . . .	15
3.11	Bewegung eines Filters über eine convolution Schicht . . . . .	15
3.12	Convolution Schicht mit 3 Feature Maps . . . . .	16
3.13	2x2 maxpool Schicht mit einer Schrittweite von 2 . . . . .	16
3.14	neuronales Netzwerk mit Dropout . . . . .	18
4.1	Logo mit verschiedenen Hintergründen . . . . .	19
4.2	Selbst generiertes Bild mit einem Prosieben Logo . . . . .	20
4.3	Prosieben Logo (17x11 Pixel) herausgenommen aus einem 320x180 Pixel Bild . .	21
6.1	Kein klassisches Prosieben Logo . . . . .	25

Abbildung aus dem Kapitel 3 sind von Michael A. Nielsen's Buch[6] inspiriert.

# Ehrlichkeitserklärung

Die eingereichte Arbeit ist das Resultat meiner persönlichen, selbstständigen Beschäftigung mit dem Thema. Ich habe für sie keine anderen Quellen benutzt als die in den Verzeichnissen aufgeführten. Sämtliche wörtlich übernommenen Texte (Sätze) sind als Zitate gekennzeichnet.

4. November 2018