Typen Künstlicher Neuronaler Netze

Darstellung, Vergleich und Untersuchung der Anwendungsmöglichkeiten

Lars Erber

Phoenix Gymnasium Dortmund, Informatik LK Herr Disselkamp, Q1

5. März 2018

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit werde ich mich ausführlich mit dem Thema Neuronale Netze auseinandersetzen. Ich finde das Thema spannend, da erst seit wenigen Jahren bekannt ist, wie viele Möglichkeiten sich durch deren Verwendung ergeben, obwohl die ersten Ansätze schon fast ein Jahrhundert alt sind, weil aufgrund von mittlerweile vorhandener hoher Rechenkraft und großen Datenmengen dementsprechend viel geforscht wird. Besonders faszinierend ist die Tatsache, dass Neuronale Netze im Gegensatz zu üblichen Algorithmen die Lösung eines Problems selbst finden können und im Gegensatz zu anderen Arten des Maschinellen Lernens, wie beispielsweise bei genetischer Programmierung, nicht zu großen Teilen auf Zufall basieren.
Mein Ziel ist es, die Funktionsweise verschiedener Typen von Netzen zu erklären, sowie diese in unterschiedlichen und relevanten Kategorien zu vergleichen und mich damit auseinanderzusetzen, welche Anwendungsmöglichkeiten sich bieten. Einige der Netze möchte ich auch implementieren, um eigenständige Test durchzuführen und mich auch nach dieser Arbeit weiter mit dem Thema beschäftigen sowie Neuronale Netze anwenden zu können. Ein Teil der Anwendungsbeispiele wird sich damit befassen, Daten entsprechend so zu kodieren, dass die Netze diese verarbeiten können und gute Ergebnisse produzieren. Hauptsächlich werde ich dort aber untersuchen, wie gut sich die unterschiedlichen Typen für entsprechende Aufgaben eignen.

Kapitel 2

Darstellung

Um sich mit dem Thema auseinandersetzen zu können, benötigt man zunächst einen Überblick. Man muss wissen, was künstliche Neuronale Netze sind und welche Prinzipien diese verfolgen, um zu verstehen, wie und warum sie funktionieren. Daher möchte ich kurz beschreiben, was ein solches Netz ausmacht.
Neuronale Netze sind vernetze Strukturen, bestehend aus Neuronen, welche die Knoten darstellen und Axonen, die diese untereinander verbinden. Dieses Prinzip basiert auf einer Entdeckung der Biologie, dass der Denkapparat von Mensch und Tier als Verknüpfung von Nervenzellen aufgebaut ist und durch Weiterleitung von Signalen mithilfe dieser Verknüpfungen funktioniert. Die Neuronen verarbeiten hierbei Signale, welche sie über einige synaptische Verbindungen erhalten und über weitere übertragen. Bei diesem Vorgang können die Signale sowohl erregend als auch hemmend wirken [vgl. 1, S. 18f.].
Trotz dieser sehr einheitlichen Grundidee für den Aufbaus eines solchen Netzes, gibt es einige Möglichkeiten, wie sich entsprechende künstliche Neuronale Netze unterscheiden können. Zum einen differenzieren sich die Netze in der Verarbeitung und der Art der Daten, welche übertragen werden. Hierbei kann es sich beispielsweise um einfache Werte, aber auch um Werteansammlungen wie Bilder handeln. Zum anderen können sie sich auch in der Art und Anordnung der Neuronen und Verknüpfungen unterscheiden. Durch diese Verschiedenheiten bekommen die Netze unterschiedlichste Eigenschaften, deren Vor- und Nachteile ich in dieser Arbeit untersuchen möchte.

Künstliche Neuronale Netze haben sehr viele Anwendungsmöglichkeiten, welche durch diese vielen Eigenschaften sowohl eingeschränkt, als auch erweitert werden. Sie machen sich das biologische Analogon zu Nutze, um komplexe Vorgänge zu lernen, für die der Aufwand zu hoch wäre, einen Algorithmus zu entwickeln [vgl. 1, S. 3.]. Welche Typen sich für welche Aufgabe besonders eignen, möchte ich in dieser Arbeit herausfinden. Dazu werde ich zunächst einmal feststellen, welche Typen sich für einen solchen Test überhaupt eignen, da einige Varianten sehr speziell und wenig untersucht sowie dementsprechend schwach entwickelt sind, sodass es sich kaum lohnt, diese zu betrachten. Ich habe daher nur solche Typen recherchiert, die bisher häufig Anwendung gefunden haben und daher gut dokumentiert und nachvollziehbar darzustellen sind, wofür ich dieses Kapitel nutzen werde.

2.1 Multi Layer Perceptron (MLP)

Das Multi Layer Perceptron, kurz MLP, gehört zu den bedeutendsten Typen künstlicher neuronaler Netze. Es basiert auf dem schon 1958 von Frank Rosenblatt entwickelten [1, vgl S. 73] Perceptron und zeichnet sich durch seine einfache Struktur aus.

2.1.1 Das einfache Perceptron

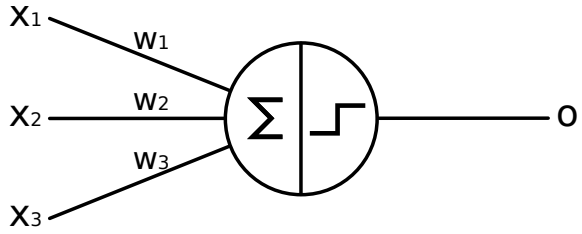

Abbildung 2.1: Darstellung eines Perceptrons und seiner Teile. Man erkennt die Eingaben x_i, ihre Gewichte w_i, die Summe und Aktivierungsfunktion, sowie die Ausgabe o.

Ein Perceptron lässt sich als stark vereinfachtes Modell eines biologischen Neurons betrachten. [vgl. 2, S. 41.] Die Funktionsweise eines solchen lässt sich relativ einfach erläutern, da es prinzipiell aus zwei Komponenten besteht: der sogenannten gewichteten Summe, also der Summe von gewichteten Eingabewerten, deren

Gewichte durch einen Lernalgorithmus kontrollierten werden, sowie einer Aktivierungsfunktion. [vgl. 1, S. 73]
Nehmen wir x_i als Eingabewerte wobei $i \in \{0, ..., n\}$, dann haben diese für ein Perceptron j (Man kann auch mehrere Perceptrons mit den gleichen Inputs verarbeiten, hier spricht man dann von einem Single Layer Perceptron, kurz SLP, siehe Abbildung 2.6) Gewichtungen w_{ji}, und es berechnet sich die gewichtete Summe s des Perceptrons j wie folgt:

$$s = \sum_{i=0}^{n} x_i * w_{ji}$$

Diese gewichtete Summe wird anschließend von einer Aktivierungsfunktion a verarbeitet, welche den Ausgabewert je nach gewünschtem Ergebnis anpasst. Oft wird hierbei eine binäre Klassifizierungsfunktion beziehungsweise Schrittfunktion genutzt, welche die Summe als einen negativen oder positiven Wert einstuft. Somit ist

$$o = a(s)$$

allgemein die Formel für den Ausgabewert o eines Perceptrons. Damit auch dann, wenn alle $x_i = 0$ sind ein brauchbarer Wert bestimmt werden kann, wird zusätzlich ein Bias, auch on-Neuron genannt, hinzugefügt, welcher einen zusätzlichen Eingabewert liefert, der immer 1 beträgt (das heißt $x_0 = 1$). Dieser ist natürlich ebenfalls gewichtet und sorgt so dafür, dass eine Leereingabe einen anderen Wert als 0 haben kann. Mit diesem Prinzip lassen sich bereits Räume der Dimension der Zahl der Eingabewerte durch eine Hyperebene teilen. Bei einer zweidimensionalen Eingabedimension wäre die teilende Hyperebene beispielsweise eine Gerade [vgl. 1, S. 86]. Das Teilen durch eine Hyperebene entspricht insofern einer Klassifikation, als dass ein Ausgabeneuron je nachdem, ob der Punkt in der Eingabedimension ober- oder unterhalb dieser Hyperebene liegt, festlegen kann, zu welcher Kategorie ein

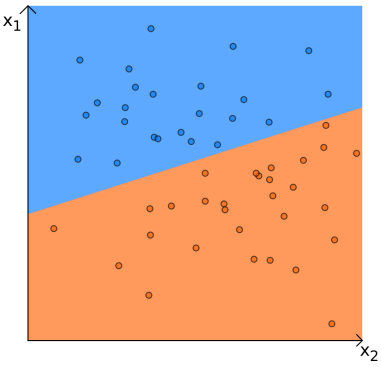

Abbildung 2.2: So könnte ein Perceptron zwei Dimensionen nach entsprechendem Lernvorgang mithilfe der Datenpunkte teilen.

Neuron gehört. Zu Problemen kommt das Verfahren erst, sobald das zu lösende Problem nicht mehr linear separierbar, also durch eine Hyperebene teilbar ist, was beispielsweise bei der XOR-Funktion zu beobachten ist [vgl. 3, S. 85f.].
Der Lernvorgang eines solchen Neurons arbeitet mithilfe des Fehlers beziehungsweise der Abweichung des Ausgabewertes von dem gewünschten Wert d. Es handelt sich also um einen Algorithmus zum überwachten Lernen. Dieser rechnet nach Rosenblatt [2, S. 53] mit folgender Formel:

$$\begin{aligned} w_{ji}^{k+1} &= w_{ji}^{k} + \alpha * e_j^k * x_i \\ e_j^k &= d_j^k - o_j^k \end{aligned}$$

wobei k die jeweilige Iteration ist und α ein Faktor, welcher die Lernrate angibt. Dieser ist meist ein Wert zwischen 0 und 1, damit das gelernte nicht durch Ausnahmefälle oder neue Daten überschrieben wird, beziehungsweise, um einen sauberen Gradientenabstieg zu gewährleisten. Dies werde ich in Abschnitt 2.1.3 näher erläutern.

2.1.2 Mehrere Neuronenebenen

Eine Lösung des Problems der linearen Separierbarkeit bietet das Multi Layer Perceptron [vgl. 1, S. 86f.]. Obwohl es verschiedene Unterarten des MLP gibt, ist meist die simpelste Variante, ein vollständig verbundenes feed-forward-Netz gemeint [vgl. 2, S. 67]. Hierbei werden einfache Zahlenwerte über gewichtete Verbindungen vorwärts über mehrere Perceptronenebenen übertragen. Streng genommen besteht das einfache Perceptron bereits aus mehreren – nämlich zwei – Ebenen: Die Eingabeebene, über die die Eingabewerte eingegeben wer-

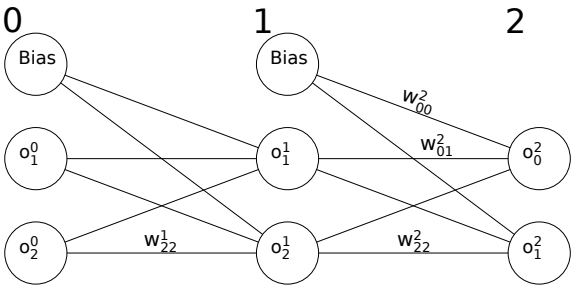

Abbildung 2.3: Darstellung eines MLPs. Die Ebenen sind mit Zahlen beschriftet, die Neuronen entweder mit Bias oder dem Formelzeichen für seinen Ausgabewert. Beispielhaft wurden auch einige Verbindungen mit dem jeweiligen Formelzeichen ihres Gewichts beschriftet.

den, und die Ausgabeebene, in welcher das eigentliche Perceptron liegt und den Wert seiner Aktivierungsfunktion ausgibt. Daher bezeichnet man im allgemeinen ein Perceptronennetz erst ab drei Ebenen als MLP oder sieht die Eingabeebene nicht als Ebene an [vgl. 1, S. 87]. Die Eingänge der Neuronen/Perceptronen jeder Ebene, welche nicht die Eingabeebene ist, sind mit den Ausgängen der vorhergehenden Ebene verbunden. Jedes Neuron hat also genau so viele gewichtete Eingänge, wie die vorhergehende Ebene an Neuronen besitzt. Wie beim einfachen und Single Layer Perceptron auch enthält jede Ebene einen Bias ($o_0^e = 1, e \in E$, wobei E die Menge der Ebenen des MLP darstellt). Der Ausgabewert eines Neurons außerhalb der Eingabeebene ($e > 0$) berechnet sich somit über die Formel

$$o_j^e = a\left(\sum_{i=0}^{n_{(e-1)}} o_i^{e-1} * w_{ji}^e\right)$$

wobei a die Aktivierungsfunktion, e eine Ebene, n_e die Zahl der Neuronen in einer Ebene e, o_i^e der Ausgabewert des i-ten Neurons in einer Ebene e und w_{ji}^e das Gewicht des i-ten Eingabewertes des j-ten Neurons einer Ebene e ist.
Netze dieses Typs heißen daher feed-forward, da sie in der Eingabeebene mit Werten „gefüttert“ werden, welche nach Verarbeitung dann an die nächsten Ebenen vorwärts – und wirklich nur vorwärts; daher der Name – „weiterverfüttert“ werden.

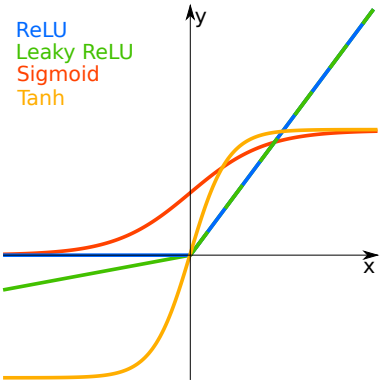

Abbildung 2.4: Verschiedene Aktivierungsfunktionen

Der übliche Lernalgorithmus nennt sich passenderweise Backpropagation, „Rückausbreitung“, da der Fehler anders als beim SLP sich über die hier vorhan-

denen Ebenen verteilt und sich eine Korrektur der Gewichtungen von hinten über die Neuronen ausbreiten muss. Der Algorithmus arbeitet auch hier wieder nach dem Verfahren des Gradientenabstiegs, doch hier möchte ich näher darauf eingehen, da bei MLPs als Aktivierungsfunktion in den Hidden Layers eine sogenannte rectifier-Funktion (ReLU) genutzt wird (Siehe Abbildung 2.4), bei welcher die Ableitung, anders als bei Stufenfunktionen, beachtet werden muss. Eine nichtlineare Aktivierungsfunktion hilft, sich bei anderen Problemen, beispielsweise Vorhersagen, exakteren Werten anzunähern [vgl. 3, S. 126]. Dies ist bei der Stufenfunktion nicht notwendig, da die Ableitung dieser an jeder Stelle 0 ist. Früher wurde oft eine Sigmoidfunktion verwendet, diese hat allerdings diverse Nachteile, wie ihre zu 0 konvergierende Ableitung bei Werten mit hohem Betrag. Sie kann aber in der Ausgabeebene weiter genutzt werden [vgl. 4]. Durch die Verwendung einer Sigmoidfunktion wird die Ausgabe einem Wert zwischen 0 und 1 zugewiesen, was hilfreich sein kann, möchte man sehen, für wie wahrscheinlich das Netz es hält, dass es richtig liegt.

2.1.3 Lernvorgang

Der Prozess des überwachten Lernens findet statt, indem man das Netz so lange mit einer Menge von Trainingsdaten, welche aus Eingabewerten und gewünschter Ausgabe besteht, trainiert, bis es auch auf einer Testmenge korrekte Ergebnisse liefert. Nach jedem Trainingsdurchlauf mit einem Werteset aus der Trainingsmenge werden Gewichtungen angepasst, bevor der nächste Schritt

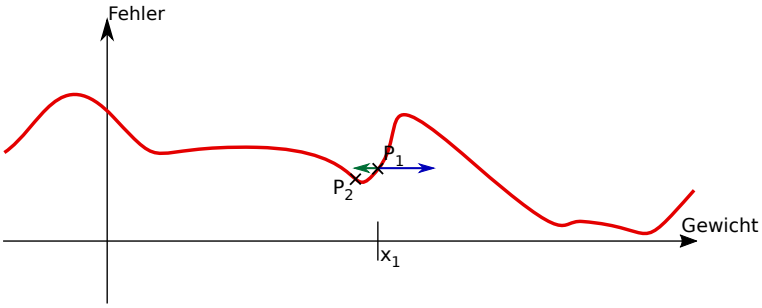

Abbildung 2.5: Beispiel eines Gradientenabstiegs. Zunächst hat das Gewicht den Wert x_1 und bildet mit der Fehlerkurve den Punkt P_1. Der Gradient ist als blauer Vektor eingezeichnet und zeigt in Richtung des größeren Anstiegs. Er entspricht der Ableitung an der Stelle x_1. Wird dieser mit einer negativen Lernrate multipliziert, erhält man den grünen Vektor. Addiert man diesen auf das Gewicht, so liegt es durch den Fehler auf P_2. Dieser Punkt liegt hier (wie meistens) tiefer als P_1.

durchlaufen wird. Sobald der Fehler beim Training ausreichend gering ist, prüft man, ob dies auch bei der Testmenge der Fall ist, um festzustellen, dass das MLP gut gelernt hat [vgl. 1, S. 59], oder man eine schlechte Trainingsmenge gewählt hat. Der Backpropagation-Algorithmus korrigiert die Gewichte der Neuronenverbindungen mithilfe des Gradientenabstiegverfahrens. Es wird also für alle Gewichte derjenige Vektor in der Gewichtsdimension berechnet, der in Richtung des am schnellsten steigenden Fehlers zeigt, damit man die Gewichte in die entgegengesetzte Richtung anpassen kann und sich so einem niedrigeren Fehler nähert. Bildlich vorstellen kann man sich das, indem man sich einen beliebigen Graphen anschaut, wobei auf der X-Achse der Wert eines Gewichts steht und auf der Y-Achse der Fehler des Neurons abgebildet wird. Befindet man sich nun an einem beliebigen Punkt des Graphen lässt sich über eine Formel der Gradient, also der Vektor, welcher in Richtung der höchsten Steigung zeigt bestimmen. Im zweidimensionalen Raum ist der Gradient bloß die Ableitung, bei einer dreidimensionalen Funktion berechnet er sich wie folgt:

$$\nabla f(x,y) = (\frac{\delta f(x,y)}{\delta x}, \frac{\delta f(x,y)}{\delta y})$$

Hierbei ist ∇f der Gradient und $\frac{\delta f(x,y)}{\delta x}$ sowie $\frac{\delta f(x,y)}{\delta y}$ stehen für die partiellen Ableitungen von f nach x beziehungsweise y. Analog ist der Gradient im n-dimensionalen Raum in jeder Dimension i die Ableitung der Funktion nach x_i. Mithilfe des negierten Gradienten lässt sich herausfinden, in welcher Richtung ein nahes, lokales Minimum liegen könnte. Dieses versucht man dann zu erreichen, indem man einen Schritt definierter Länge in die entsprechende Richtung macht. Nun befindet man sich im Optimalfall bei einem niedrigeren Fehlerwert als vorher. Man muss hierbei allerdings zwei Dinge feststellen: zum einen, dass es unmöglich ist zu sagen, ob man sich auf dem Weg zu einem guten lokalen oder vielleicht sogar dem globalen Minimum befindet, oder wie man sich einem besseren Minimum, das heißt einem mit geringerem Fehler annähern kann, und zum anderen, dass es nicht gerade trivial ist, dieses Verfahren mit mehreren Eingängen – und somit Gewichten – gleichzeitig durchzuführen, da mit jedem Gewicht, welches man zeitgleich betrachtet der Graph um eine weitere Dimension wächst.

Es stellt sich nun allerdings die Frage, was die Funktion f ist, von der man den Gradienten bilden möchte. Diese Frage zu beantworten kann einfach sein, möchte

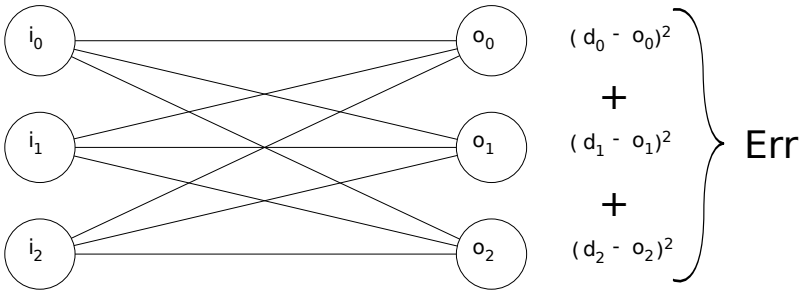

Abbildung 2.6: Grafische Darstellung der Fehlerberechnung in einem SLP (äquivalent zur Ausgabeschicht eines MLPs). Nachdem die Eingabewerte i_n über die Gewichte w_{mn} (nicht beschriftet) übertragen und von den Ausgabeneuronen zu den Werten o_m verarbeitet wurden, werden diese von den Zielwerten abgezogen, diese Differenzen werden einzeln quadriert und dann alle addiert, um den Fehler zu bestimmen.

man hierbei allerdings eine nutzbare Formel haben, wird es schon etwas schwieriger. Fange ich also mit dem einfach Teil an: Da man den Fehler minimieren will, sollte klar sein, dass die Funktion den Fehler definieren muss. Ebenfalls muss die Funktion abhängig der Gewichtungen sein, weil ebendiese angepasst werden sollen. Daher nennt man die Funktion üblicherweise nicht $f(x)$, sondern $Err(W)$. Damit ist klar, was die Funktion ist. Nicht klar ist allerdings, was sie bedeutet und wie man damit umgeht. Zunächst einmal ist W hierbei der Vektor, welcher alle Gewichte der Eingänge zu den Neuronen enthält. Somit definiert die Funktion die n-dimensionale Fehlerfläche im Hyperraum der Gewichte. Um sich auf dieser Fläche zu bewegen muss man für jedes Gewicht den Wert anpassen. Da man hier jedes Gewicht einzeln betrachten muss, muss die Funktion praktischerweise auch nach jedem Gewicht einzeln abgeleitet werden. Das Ganze ist deshalb praktisch, weil jedes Gewicht genau eine Dimension darstellt und es deshalb reicht, für den Gradienten dieses einen Gewichts die Ableitung nach seiner Dimension zu betrachten. Man kennt mit den Trainingsdaten zwar nur den Zielwert für die letzte Ebene, kann aber bereits die Formel

$$\Delta w_{ji} = -\alpha * \frac{\delta Err(W)}{\delta w_{ji}}$$

bezüglich dieser für die Veränderung eines Gewichts w_{ji} (j-tes Gewicht für Neuron i) der Ausgabeebene aufstellen, wobei α weiterhin der Proportionalitätsfaktor oder die Lernrate ist. Wenn die Fehlerfunktion den Fehler angeben soll, so muss sie für einen Eingabewert die Summe der Fehler jedes Ausgabeneurons sein. Um auch bei starken Ausreißern in der Testmenge gute Ergebnisse, das heißt, einen kleineren Fehler zu erzielen, berechnet man den Euklidischen Abstand (Länge der Strecke zwischen zwei Punkten, $\sqrt{(x_1)^2 + (x_2)^2 + (x_3)^2 + ... + (x_n)^2}$) zwischen der gewünschten Ausgabe und der tatsächlichen. Um den Rechenaufwand zu minimieren, zieht man nach der Summe der Fehler nicht die Wurzel. Stattdessen multipliziert man diese mit $\frac{1}{2}$, um später eine einfachere Formel zu erhalten. Diese Anpassungen verhindern nicht den Zweck der Rechnung, sich besseren Gewichtungen anzunähern [1, S. 81].

$$Err(W) = \frac{1}{2} * \sum_{i=0}^{n} (d_i - o_i)^2$$

d_i und o_i sind die gewünschten und die tatsächlichen Ausgabewerte der Neuronen i in der Ausgabeebene. n ist hierbei die Zahl der Neuronen. Mithilfe der Kettenregel lässt sich die Formel der Ableitung erweitern durch die gewichtete Summe s_i des Neurons, was später helfen wird.

$$\frac{\delta Err(W)}{\delta w_{ji}} = \frac{\delta Err(W)}{\delta s_i} * \frac{\delta s_i}{\delta w_{ji}}$$

um den ersten Bruch zu bestimmen, wendet man wieder die Kettenregel an:

$$\frac{\delta Err(W)}{\delta s_i} = \frac{\delta Err(W)}{\delta o_i} * \frac{\delta o_i}{\delta s_i}$$

Alle diese Brüche fasse ich nun noch einmal in einer Gleichung zusammen, um sie der Reihe nach zu betrachten:

$$\frac{\delta Err(W)}{\delta w_{ji}} = \frac{\delta Err(W)}{\delta o_i} * \frac{\delta o_i}{\delta s_i} * \frac{\delta s_i}{\delta w_{ji}}$$

Da bisher nur die Ausgabeebene betrachtet wurde, ist die Ableitung von $Err(W)$ nach o_i nur noch abhängig von der Differenz der gewünschten Ausgabe. Aufgrund

des Vorfaktors der Fehlerfunktion ergibt sich dann

$$\frac{\delta Err(W)}{\delta o_i} = -(d_i - o_i)$$

als Teilformel. Für den zweiten Bruch ist der Wert offensichtlicher. Da die Ausgabe des Neurons bloß die Aktivierungsfunktion über der gewichteten Summe s_i ist, ist jene nach dieser abgeleitet natürlich die Ableitung g' der Aktivierungsfunktion g:

$$\frac{\delta o_i}{\delta s_i} = g'(s_i)$$

Nun fehlt nur noch der letzte Bruch, damit man eine vollständige Formel hat. Dieser stellt die Ableitung der gewichteten Summe des Neurons i nach dem Gewicht j ebendieses dar. Da die gewichtete Summe von w_{ji} abhängt, kürzt sich das Gewicht heraus und so ergibt sich für den zweiten Bruch o_i. Die Formel für die Bestimmung einer Gewichtsänderung in der letzten Ebene ist nun also folgende:

$$\Delta w_{ji} = -\alpha * (\delta d_i - \delta o_i) * g'(s_i) * o_i$$

Man bezeichnet den Teil $(\delta d_i - \delta o_i) * g'(s_i)$ auch als δ_i.
Haben wir die Korrekturwerte für die Gewichte der Neuronen der letzten Ebene berechnet (welche übrigens noch nicht angewendet werden dürfen), kann der Fehler nun rückpropagiert werden. Hierzu bildet man bei jedem Neuron einer Ebene quasi die gewichtete Summe der δs der Neuronen der Nachfolgeebene (also der, deren *delta*s wir vorher berechnet haben) und multipliziert diese wie gewohnt mit $g'(s_i)$, um das δ des aktuellen Neurons zu erhalten. Der Rest unserer Formel bleibt erhalten. Somit sieht die Formel

$$\Delta w_{ji}^k = -\alpha * \sum_{h=0}^{n_{k+1}} (\delta_h^{k+1} * w_{hi}^{k+1}) * g'(s_i) * o_i$$

für die Veränderung Eingabegewichte der Neuronen, die weder Eingabe- noch Ausgabeneuronen sind, der für die Ausgabeneuronen sehr ähnlich. Wichtig anzumerken ist, dass δ hier $\sum_{h=0}^{n_{k+1}} (\delta_h^{k+1} * w_{hi}^{k+1})$ ist. In der Formel ist w_{ji}^k das i-te Gewicht des j-ten Neurons in Ebene k und n_k die Zahl der Neuronen in dieser, sowie δ_h^k das δ des

h-ten Neurons in Ebene k und w_{hi}^{k} das Gewicht. Man muss für die Eingabeneuronen kein δ berechnen, da diese keine Gewichte haben, welche angepasst werden müssen. Dies ist nämlich der letzte Schritt im Lernprozess einer Trainingseinheit: Auf jedes Gewicht im Netz addieren wir nun das errechnete Δw.
Zur Herleitung dieser Regel vergleiche [1, S. 79-84 und S. 89-94].

2.2 Convolutional Neural Network (CNN)

Ein Convolutional Neural Network behandelt statt einfacher Zahlenwerte mehrdimensionale Matrizen von Werten zwischen 0 und 1, meistens Bilder. Bei Bildern sind die Matrizen häufig zwei- oder dreidimensional, da die einzelnen Pixel eine X und eine Y Koordinate, sowie eventuell Werte der verschiedenen Farbkanäle (wie zum Beispiel Rot, Grün und Blau) besitzen. Das Prinzip eines CNN ähnelt dem eines MLP zunächst sehr, durch zwei einfache Überlegungen lässt sich der bei erster Betrachtung entstehende enorme Berechnungsaufwand allerdings deutlich verringern. Dazu nutzen CNNs neue Prinzipien, welche sich von denen des MLP deutlich unterscheiden. Bei der Herleitung eines CNN orientiere ich mich an den Ausführungen von *Aghdam et al.[5]* .

2.2.1 Convolution

Um gute Ergebnisse zu erzielen, müsste man bei einem MLP deutlich mehr Neuronen in der ersten versteckten Schicht haben, als das Bild an Werten hat. Bei einem relativ kleinen RGB-Bild (3 Kanäle) mit den Abmaßen $16*16$ Pixel würde man zum Beispiel als erste Idee $16*16*3*50 = 38400$ Neuronen nutzen, wobei 50 ein beliebiger Faktor ist, um höhere Genauigkeiten in der Verarbeitung hervorzurufen. Dies resultiert natürlich in immensen Anzahlen von Gewichten von der Eingangsschicht zum Hidden Layer und somit hohem Berechnungsaufwand. Ich lasse den Faktor 50 zunächst außen vor, um die nachfolgenden Überlegungen zu vereinfachen, führe ihn später aber wieder ein. Man hat also nur ein Neuron pro Pixel in der ersten versteckten Ebene, sodass dort $16*16*3 = 768$ Neuronen sind. Um die Zahl der Gewichte zu dieser Ebene zu verringern, überlegt man sich nun, dass nahe beieinander liegende Pixel in einem Bild korrespondieren und es daher reicht, jedes Neuron im Hidden

Layer mit einem kleinen Bereich und nicht mit allen Pixeln aus dem Bild zu verbinden und so dann das Bild mithilfe der Neuronen quasi abzutasten. Wählt man zum Beispiel einen $5 * 5$ Pixel großen Bereich, fallen die meisten Gewichte weg, da jedes Neuron nur noch 25 Eingabegewichte hat.
Zusätzlich ist die Anzahl der Neuronen geringer, da die Pixel am Rand des Bildes nicht mehr einzeln betrachtet werden können. Dazu kann man sich vorstellen, dass man den Bereich an der linken oberen Ecke beginnt und man je fünf Pixel nach rechts und nach unten betrachtet. Dies ist für die letzten vier Reihen rechts und unten nicht mehr möglich. Da diese Randbereiche aber bereits abgetastet wurden, ist dies nicht weiter relevant und es sind in dieser Betrachtung nur noch $12*12*3 = 432$ Neuronen.
Um weiter Gewichte zu sparen, nutzen die Neuronen dieselben Gewichte, was sinnvoll ist, da die entsprechenden Gewichtsmatrizen zum Beispiel senkrechte Linien klassifizieren könnten und dies nicht nur in einem Bereich des Bildes hilfreich sein kann. Da natürlich auch andere Klassen nötig sind, führe ich jetzt den Faktor wieder ein, wobei jedes Duplikat des soeben gestalteten Neuronenblocks sich andere Gewichte teilt. So hat man bei Miteinbeziehung des Faktors 50 vom Anfang eine Zahl von $12 * 12 * 3 * 50 = 21600$ Neuronen und nur noch $5 * 5 * 3 * 50 = 3750$ Gewichtungen statt $16 * 16 * 3 * 38400 = 29491200$. Das entspricht einer Reduktion um $99,987\%$. Man erhält hierdurch quasi 50 neue Bilder, wenn man die Neuronen im Hidden Layer als Pixel sieht, welche abhängig von den jeweiligen Gewichten aus dem Originalbild generiert wurden.
Dieses Prinzip entspricht der Faltung von Funktionen in der Mathematik, auch Konvolution (auf Englisch „Convolution“) genannt, daher der Name des Netztyps. Ich werde im Folgenden hauptsächlich den englischen Begriff verwenden. Nach Anwendung der Faltungsmatrizen (jeweils die Gewichte einer Neuronengruppe) erhält man mehrere Graubilder. Auf diese werden dann Pixel für Pixel, beziehungsweise Neuron für Neuron (da die gewichteten Summen die Pixel eines Bildes nach Anwendung der Faltung darstellen) wie vom MLP gewohnt die Aktivierungsfunktion angewendet.

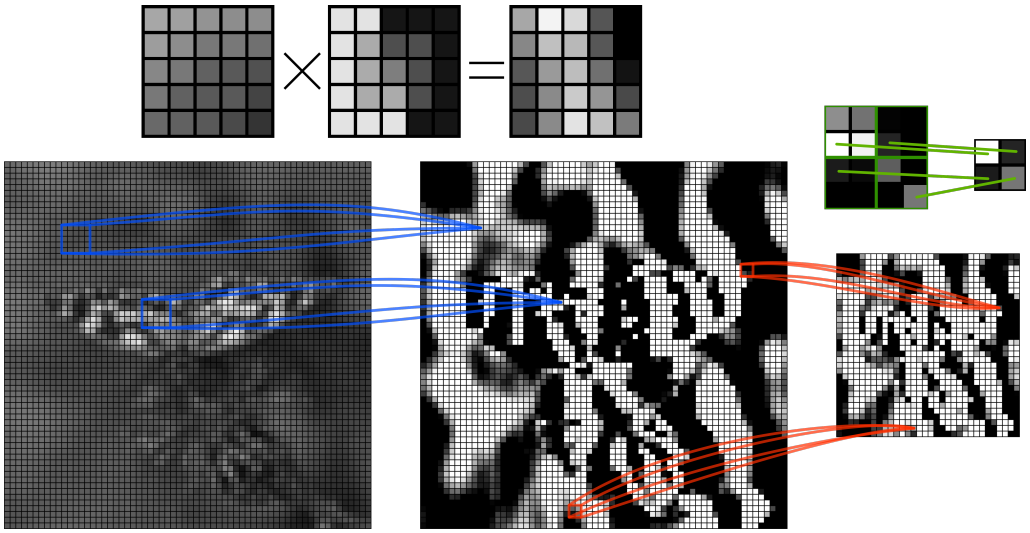

Abbildung 2.7: Grafische Darstellung von Convolution und Pooling. Oben sieht man einen Ausschnitt des Bildes, über den die nebenstehende Matrix gefaltet wird. Jeder Pixel in dem mittleren Bild ist das Ergebnis einer Multiplikation eines Bildausschnitts und der Matrix an der gleichen Stelle im Original (links). Beim Pooling wird das Bild in diesem Beispiel in $2*2$ Pixel große Bereiche eingeteilt. Aus jedem Bereich wird der nur hellste Pixel weitergegeben. Die Anwendung der Aktivierungsfunktion vor dem Pooling wurde zur Vereinfachung der Darstellung ausgelassen.

2.2.2 Pooling

Da eine Klassifikation in einer beliebigen Anzahl von Möglichkeiten nicht über Convolution möglich ist, nutzt man im letzten Abschnitt ein normales MLP. Damit dieses nicht zu viele Eingabeneuronen hat, werden nach jeder Convolution die resultierenden Bilder verkleinert. Um möglichst wenige Informationen zu verlieren, entspricht meist jeder Pixel im verkleinerten Bild dem Pixel mit dem höchsten Wert in einem Quadrat mit einer Seitenlänge von circa 2 bis 4 aus einem gefalteten Original (Siehe Abbildung 2.7).

2.2.3 Backpropagation

Ein CNN benutzt abwechselnd eine oder mehrere Konvolutionsebenen mit Aktivierungsfunktion und Poolingebenen. Dies baut man so lange auf, bis eine Poolingebene Bilder ergibt, die klein genug sind, um sinnvoll von einem MLP verarbeitet werden

zu können (zum Beispiel 8 Bilder à $4 * 4$). Dass trotz dieser nicht unerheblichen Datenreduktion noch genügend Informationen vorhanden sind, kann man sich an einem Beispiel verdeutlichen: Markiert eine der Faltungsoperationen der Ersten Ebene beispielsweise senkrechte Linien im Bild und eine weitere waagerechte, so kann die nächste Ebene dieses „Wissen“ nutzen, um Ecken zu erkennen. Dieses Muster setzt sich immer weiter fort bis zur Erkennung komplexer Formen wie zum Beispiel Gesichter. Ähnlich einem RGB-Bild werden die bei der Faltung mit verschiedenen Filtern entstehenden unterschiedlichen Bilder als Kanäle in einer weiteren Dimension behandelt.
Es kann einem auffallen, dass das Prinzip bis auf das Pooling einem MLP immer noch sehr ähnelt. Daher lässt sich das CNN auch mit Backpropagation trainieren. Ich werde die Herleitung der Regeln hier nicht weiter erläutern, da die meisten Berechnungen analog zum MLP verlaufen. Es reicht zu wissen, dass die Konvolutionsebenen im Prinzip die Faltung rückwärts auf den berechneten Fehler anwenden [5, S. 94].
Das Pooling wird, sofern man hier Bereiche auf den Pixel/das Neuron mit dem höchsten Wert komprimiert, behandelt, indem man den Fehler ausschließlich an das entsprechende Neuron beziehungsweise den Pixel weitergibt (oder zurück, je nach Betrachtungsweise). Da über die übrigen Werte keine Informationen für Fehler vorhanden sind, bekommen sie auch keinen Fehlerwert, der zur Anpassung der Gewichte dienen könnte. Benutzt man ungewöhnlicherweise andere Pooling-Algorithmen, muss man eine Formel entwickeln, die den Fehler entsprechend verteilt [5, S. 98].

2.2.4 Upsampling Layer

Es ist auch möglich mit einem Convolutional Neural Network Bilder (oder andere Mehrdimensionale Daten) zu generieren. Hierzu benötigt man anstelle des Poolings einen weiteren Ebenentyp, der das Bild vergrößert, anstatt es zu verkleinern. Dies geschieht meist mithilfe von Interpolation. Des weiteren kann hiernach nun mittels Konvolution und erneuter Anwendung des Verfahrens ein Bild erzeugt statt klassifiziert werden. Oft nutzt man dieses Verfahren, um Bilder auf Basis von anderen Bildern zu erzeugen, wie beispielsweise in *Larsen et al.[6]*.

2.3 Recurrent Neural Network (RNN)

Ein weiterer, wichtiger Typ Neuronaler Netze ist das Recurrent Neural Network. Auch wenn dies eine große Menge von verschiedenen Typen umfasst, ist meistens das Elman Modell gemeint. Mit diesem ist es möglich, sequenzielle Daten beliebiger Länge zu verarbeiten. Die Daten werden dazu in einer Sequenz x mit Zeitschritten gespeichert. So erhält ein RNN in einem Zeitschritt t nur die Daten $x[t]$, wobei die Zeitschritte in chronologischer Reihenfolge eingegeben werden. Damit das funktionieren kann, müssen allerdings einige große Veränderungen an dem bei diesem Typ wieder zugrundeliegenden MLP vorgenommen werden.

2.3.1 Struktur

Der größte Unterschied im Aufbau zum MLP besteht bei einem RNN darin, dass die versteckten Ebenen nicht nur untereinander, sondern auch mit sich selbst verknüpft sind. Die Daten fließen in einem RNN also nacheinander von der Eingabeebene über Gewichte in einen hidden Layer. Von da aus werden sie in die nächste Ebene und zusätzlich wieder zurück in den hidden Layer übertragen, jeweils mit unterschiedlichen Gewichten. Bei der Rückleitung in die aktuelle Ebene gibt es allerdings eine Verzögerung von einem Zeitschritt, selten auch mehreren. So können auch vergangene Zeitschritte noch beachtet werden. Durch diese zusätzlichen Verbindungen erhält das RNN auch seinen Namen, da die Daten innerhalb des Netzes wiederkehrend (engl. recurrent) sind. Beim ersten Zeitschritt kann der vorherige natürlich nicht beachtet werden, daher geht man davon aus, dass alle Werte mit 0 belegt sind, sodass der erste Schritt keinerlei Beeinflussungen erhält, außer der ihm zugeteilten Daten. Bei den gewichteten Summen gibt es, aus dem gleichen Grund wie beim MLP auch einen Bias, der wieder wie eine gewichtete Eingabe der Größe 1 behandelt wird. [vgl. 7, S. 23f]

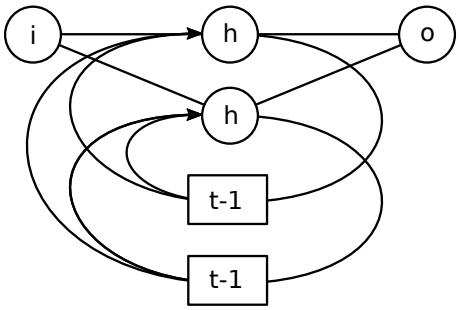

Abbildung 2.8: Darstellung eines RNN. t-1 stellt die Verzögerung von einem Zeitschritt dar.

2.3.2 Backpropagation

Wenn man diese Struktur kennt, stellt man sich natürlich die Frage, wie diese Netztypen trainiert werden. Dies ist aufgrund der rekurrenten Verbindungen nicht einfach. Daher löst man diese auf, indem man für das Training eine Abwicklung des Netzes erzeugt und so in ein feed-forward Netz umwandelt, welches nach bekannten Verfahren trainiert werden kann [vgl. 7, S. 11]. Diese Abwicklung hat im Normalfall ein Tiefe, welche der Anzahl der Zeitschritte entspricht, mit denen man trainiert. Da sich herausgestellt hat, dass die Ergebnisse bei Anwendung von Backpropagation nur nach der letzten Ausgabe keine guten Ergebnisse bewirkt, Training nach jedem Zeitschritt aber zu rechenintensiv wäre, berechnet man die Gradienten nach der Hälfe der Abwicklung und ein zweites mal am Ende. Dies erzeugt kaum schlechtere Ergebnisse als für jeden Schritt eine Berechnung durchzuführen, benötigt aber deutlich weniger Rechenzeit, wie *Bianchi et al.* herausgefunden hat [vgl. 7, S. 12f].

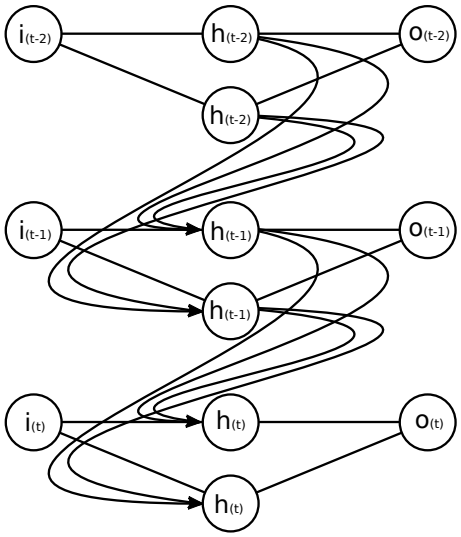

Abbildung 2.9: Abwicklung eines RNN.

2.3.3 Long Short-Term Memory (LSTM)

Eine wichtige Variante des RNN ist das LSTM-Netzwerk, welches in vielen Bereichen die besten Ergebnisse liefert. Dieses ersetzt die Neuronen des hidden Layers, welche auf die gewichtete Summe nur eine Aktivierungsfunktion anwenden mit Zellen, de-

ren Ausgabe aus 5 Funktionen zusammengesetzt wird. Zum einen hat die Zelle drei Tore oder Ventile (engl. Gates), über die reguliert wird, wie viel die Zelle von ihren Daten vergessen (Forget Gate), erneuern (Update Gate) und weitergeben (Output Gate) kann. Zum anderen hat die Zelle einen Status und einen Statuskandidaten. Der Statuskandidat ist im Prinzip, wie beim RNN, die normale Ausgabe des Neurons, also die gewichtete Summe von Eingabe und vorheriger Ausgabe der Ebene unter einer Aktivierungsfunktion. Der Zellenstatus setzt sich zum einen zusammen aus dem Wert des Statuskandidaten, welcher mit dem Wert des Update Gates, der zwischen 0 und 1 liegt, multipliziert wird. Er sorgt damit für die Regulierung des Kandidaten. Zum anderen besteht er aus dem vorherige Status, der, reguliert durch das Forget Gate, auf den regulierten Statuskandidaten addiert wird.
Der Wert eines Ventils g einer Zelle j zu einem Zeitpunkt t setzt sich nach folgender Formel zusammen. Diese ist für alle Ventile gleich.

$$\sigma_{gj}[t] = \sigma\left(\sum_{i=0}^{n_{x^e}} \left(x[t]_i^e * w_{jig}^{x^e}\right) + \sum_{i=0}^{n_{y^e}} \left(y[t-1]_i^e * w_{jig}^{y^e}\right)\right)$$

Diese Formel sieht sehr komplex aus, lässt sich aber recht gut nachvollziehen. Hierbei ist $x[t]_i^e$ der i-te Eingabewert für die Zelle in der Ebene e zum Zeitpunkt t und $y[t-1]_i^e$ analog der i-te Ausgabewert der Zellen der Ebene e zum Zeitpunkt $t-1$. Die Bezeichnung $w_{jig}^{x^e}$ bezeichnet das Gewicht i des Ventils g der Zelle j für die Eingabewerte x der Ebene e, $w_{jig}^{y^e}$ ist analog definiert. n_{x^e} und n_{y^e} sind die Anzahlen der Ein- beziehungsweise Ausgabewerte der Ebene e. σ ist die Sigmoidfunktion.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Zusammenfassend hat ein Ventil einer Zelle den Wert der Sigmoidfunktion über die gewichtete Summe der Eingaben und der vorherigen Ausgaben der Ebene, wobei beispielsweise das forget gate andere Gewichte nutzt als das update gate. Die tatsächliche Ausgabe der Zelle ist dann der Zellstatus unter einer zweiten Aktivierungsfunktion, reguliert durch das output gate [vgl. 7, S. 25f].
Über diese Modellierung ist es möglich, das Verhalten von Lang- und Kurzzeitgedächtnis zu simulieren. Zusätzlich kommt es seltener zu Problemen, falls es nötig

ist, lang vergangene Zeitschritte in der Backpropagation zu beachten, da diese Informationen nicht mit der Zeit verschwinden müssen und für extreme Gradienten sorgen [vgl. 7, S. 17].

2.4 Weitere Typen

Es gibt noch viele weitere Typen Neuronaler Netze, welche sich zum Teil grundlegend von den hier Betrachteten unterscheiden. Hierzu zählen beispielsweise Self-Organizing Maps, Pulse Coupled Neural Networks, Boltzmann-Maschinen und viele andere, sowie Verbindungen und Erweiterungen anderer Typen, wie etwa Growing Neural Gas. Diese werde ich, um den Umfang der Arbeit zu erhalten, nicht weiter erläutern.

Kapitel 3

Vergleich

Auch wenn ich nur einige der unzähligen existierenden Typen beschrieben habe, lohnt sich ein Vergleich zwischen diesen. Es gibt einige Typen, wie zum Beispiel selbstorganisierende Karten, die viele Besonderheiten in Struktur und Lernverhalten aufweisen, allerdings werde ich diese nicht betrachten. Neben der bloßen Funktionsweise ist es besonders in alltäglichen Anwendungen wichtig, wie gut sich die Typen für eine Aufgabe eignen. Hierbei ist vor allem zu beachten, wie schnell und mit welcher Qualität diese ausgeführt werden. Daher behandle ich die Performance in diesem Kapitel kurz theoretisch, um dann in Kapitel 4 gezielt auf die Akkuratesse und Geschwindigkeit in verschiedenen Anwendungen einzugehen.

3.1 Struktur und Verarbeitung

CNN und MLP sind auf den ersten Blick sehr verschieden: Ein CNN kategorisiert hauptsächlich Bilder oder kann bei Bedarf auch umgekehrt Bilder erzeugen, während ein MLP dazu geeignet ist, beliebige Daten zu klassifizieren oder vorherzusagen. Auf den zweiten Blick bemerkt man, dass das CNN auf dem MLP basiert und beide Modelle auf ähnliche Weise mit Ebenen arbeiten, auch wenn diese beim CNN oft noch einmal in Blöcke unterteilt werden. Die Faltungsoperationen sind, wie in Kapitel 2.2.1 erklärt, äquivalent zur Nutzung von Neuronen, welche gruppiert und als Bildpixel dargestellt werden. Bei ganz genauer Betrachtung stellt man allerdings fest, dass die Unterschiede doch recht groß sind, da mehrere Neuronen sich die Gewich-

te bei der Faltung teilen, was dazu führt, dass exakte Positionen im Bild schnell an Wichtigkeit verlieren können. So können selbst bei gleichem Training MLP und CNN zu unterschiedlichen Ergebnissen kommen. Gleichzeitig hat man hierdurch einen signifikanten Unterschied im Aufbau der Netze, da die Neuronen nicht mehr vollständig über Gewichte verbunden sind, sondern nur durch Matrizen kleinerer Größe, welche ebenfalls Gewichte repräsentieren, allerdings deutlich weniger. Also hat man dort auch weniger Verbindungen pro Neuron und ebenfalls weniger unterschiedliche Gewichte.
Ein entscheidender Faktor für die Unterscheidung der Modelle ist ebenfalls der Fakt, dass es in einem CNN Pooling gibt, wozu es keine direkte Äquivalenz im MLP gibt. Dennoch ist es vergleichbar mit der Reduktion von Neuronen von einer Ebene zur nächsten, wobei dort aufgrund der diese verbindenden Gewichte eine gewichtete Informationsreduktion stattfindet. Betrachtet man die Tatsache, dass beim Pooling immer der höchste Wert in einem Bereich übernommen wird, fällt einem auf, dass dies in einem MLP ähnlich verläuft, da Neuronenausgaben desselben Neurons mit höherem Betrag automatisch mehr ins Gewicht fallen, als solche mit niedrigerem Betrag. Falls eine weitere Gewichtung beim Pooling dennoch gewünscht ist, so ist es möglich vor dem Pooling eine zweite Convolution Layer anzuwenden.

3.2 Lernen

Auch im Lernprozess sind sich MLP und CNN recht ähnlich. Beide Modelle trainieren, indem sie versuchen anhand von Trainingsbeispielen mit vorgegebenen Ergebnissen den Fehler zu minimieren, welcher sich aus der Differenz der Schätzung des Netzes mit dem erwarteten Ergebnis ergibt. Diese Fehlerminimierung findet meist mithilfe eines Gradientenabstiegs statt, es gibt allerdings auch optimierte Varianten, wie Adam[8]. Allerdings hat auch das Pooling hier einen Effekt, der (wie ein Kapitel 2.2.3 beschrieben) dafür sorgt, dass weniger Neuronen gleichzeitig trainiert werden. Dies wirkt sich zum Teil sogar positiv auf den Lernprozess aus, weshalb in MLPs und anderen Typen manchmal während des Trainings verschiedene Neuronen quasi ausgeschaltet werden. Diese Technik nennt sich Dropout [vgl. 9, S. 13] und [10].
In beiden Modellen spielt auch die Lernrate eine große Rolle. Bei geringer Lernrate ist die Schrittweite sehr gering, weswegen das Netz auch bei nicht optimalen

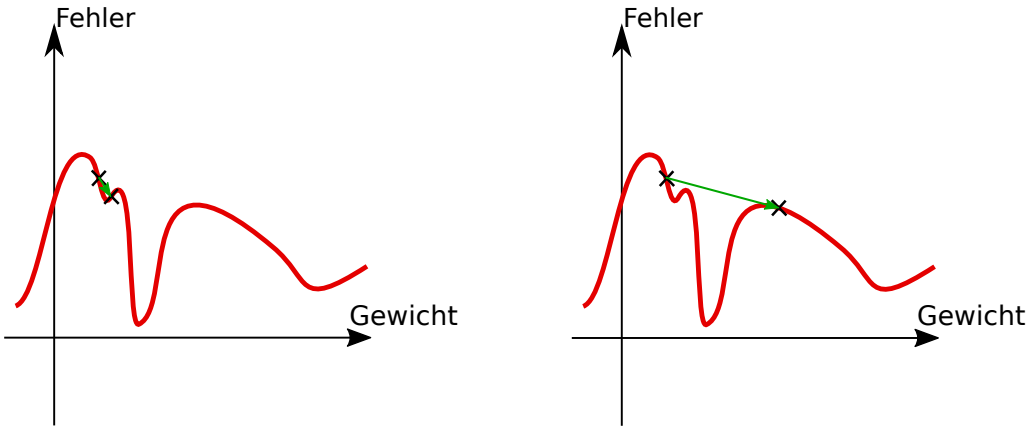

Abbildung 3.1: Veranschaulichung der Probleme bei falscher Lernrate. Links ist die Lernrate so klein, dass das Netz vor einem guten Minimum quasi stecken bleibt, rechts ist sie so groß, dass das Minimum übersprungen wurde.

Minima nicht weiter optimiert. Bei großer Schrittweite kann ein gutes Minimum übersprungen werden. Welche Schrittweite die richtige ist, lässt sich nur schwierig berechnen, vor allem deshalb, da es möglich ist, dass ein brauchbares Minimum bei einem Schritt übersprungen werden kann. Ein Schritt der gleichen Größe kann aber an anderer Stelle zum Stillstand der Optimierung führen, also zu klein sein. Daher wird die Schrittweite bei vielen Anwendungen zunächst groß eingestellt, sodass die Gewichte stark angepasst werden und schnell für einen relativ geringen Fehler sorgen und dann schrittweise verringert, um ein besseres Minimum zu erreichen.

> „In fact, a large learning rate provides a high amount of kinetic energy in the gradient descent, which causes the parameter vector to bounce, preventing the access to narrow area of the search space, where the loss function is lower. On the other hand, a strong decay can excessively slow the training procedure, resulting in a waste of computational time.“
> [7, S. 15]

Bei der Wahl einer Lernrate spielt es hypothetisch zunächst keine Rolle, welches Modell verwendet wird, es kommt allein auf die Komplexität an. Da mit mehr Gewichten die Fehlerebene extremer und komplexer wird, werden viele Minima zwangsläufig übersprungen. Es gibt aber zusätzlich mehr Minima, welche dafür sorgen, dass das Netz bei kleiner Lernrate nicht weiter oder nur sehr langsam optimiert. Damit lernt

ein MLP bei gleicher Lernrate also langsamer als ein CNN mit äquivalenter Größe (sozusagen die optimierte Variante), da ein MLP dann deutlich komplexer ist und deutlich mehr Gewichte angepasst werden müssen.
Das Problem der hohen Variation der Gewichte, also dass bei zu hoher Lernrate viele Minima übersprungen werden können und sich der Fehler teilweise sogar erhöht, ergibt sich bei MLP bereits bei geringerer Lernrate als im CNN, fällt allerdings nicht so stark auf, da die Fehler der Gewichte sich ausgleichen können.
Ich habe ein Experiment durchgeführt, welches diese Hypothese bestätigt. Bei diesem Experiment habe ich das in Kapitel 4.2.1 beschriebene MNIST-Szenario mit CNNs und MLPs verschiedener Lernraten getestet. Hierbei hat sich nach gleicher Schrittanzahl (20000) gezeigt, dass das CNN mit einer Lernrate von $0,001$ bereits eine Genauigkeit von durchschnittlich $97,01\%$ erreicht, während ein in etwa äquivalentes MLP bei gleicher Lernrate nur $15,24\%$ erreicht. Wählt man eine Lernrate von $0,2$, so erreicht das CNN eine Genauigkeit von $99,3\%$, das MLP liegt bei $93,61\%$. Bei einer Lernrate von $0,45$ stellt man fest, dass ein CNN bei einem von drei Versuchen nur noch eine Genauigkeit von $10,28\%$ aufweist, während sich das MLP noch einmal auf $95,17\%$ steigern konnte. Bei einer Lernrate von 1 verlieren beide Modelle an Akkuratesse, das MLP hält sich mit $94,79\%$ allerdings noch bei nutzbaren Werten, bei einer Lernrate von 2 kann aber auch dieses keine nützlichen Aussagen mehr treffen, die Korrektheit liegt dort im Mittel nur noch bei $10,23\%$. Die vollständige Tabelle der Messreihe findet sich im Anhang 7.2.

3.3 Performance

Ein Vergleich der Geschwindigkeit von Neuronalen Netztypen ist wichtige, weil es zeitkritische Aufgaben gibt. Bei selbstfahrenden Autos zum Beispiel ist es wichtig, dass mögliche Hindernisse oder Fußgänger schnell erkannt werden, um rechtzeitig zu bremsen. Da sich die Berechnungen in einer Ebene parallelisieren lassen, lässt sich zwischen CNN und MLP kaum ein Unterschied in der Berechnungsdauer feststellen. Da aber alle Werte zwischengespeichert werden müssen, ist der Zeitaufwand bei einem MLP für Bildrechenaufgaben höher, wie ich während der Experimente zu Kapitel 4 festgestellt habe. Es gibt jedoch auch Aufgaben, wie die Vorhersage von Börsenkursen (welche zwar nicht zeitkritisch ist, aber in dieser Arbeit behandelt

wird), deren Daten zunächst nicht einem Bild entsprechen. Hier ist es manchmal sinnvoll, die Daten in ein Bild umzuwandeln und dann mit einem Neuronalen Netz zu behandeln, da dies zu besseren Ergebnissen führen kann. Dieser Aufwand lohnt sich allerdings nicht immer, da einige Daten sich nicht gut genug als Bild darstellen lassen oder es hierfür zu hohen Rechenaufwand benötigt. Ein MLP kann bei richtiger Nutzung nahezu alle Daten klassifizieren, benötigt hierfür aber unter Umständen mehr Zeit als ein anderer Typ.

3.4 Realitätsnähe

Da künstliche Neuronale Netze auf einer biologischen Vorlage basieren, könnte es hilfreich sein, die hierbei vorhandene Realitätsnähe zu betrachten, um zu evaluieren, ob sich diese Modelle auch für Neurobiologische Forschung eignen könnten.
Künstliche Neuronale Netze sind statistische Modelle, welche universell Funktionen approximieren können [vgl. 1, S. 87]. Dennoch kommen sie einem echten Gehirn recht nahe. Echte Neuronen haben genau wie die Modellneuronen Eingänge, welche sich unterschiedlich stark auswirken und sowohl erregend als auch hemmend wirken können. Auch zur Aktivierungsfunktion gibt es eine gewisse Äquivalenz, denn die biologischen Neuronen „entscheiden“ anhand der erhaltenen Reize, ob und ein wie starkes Signal sie weiterleiten [vgl. 1, S. 23]. Der hier behandelte Backpropagation-Algorithmus hat keine bekannte biologische Entsprechung, solche Lernvorgänge lassen sich also nicht direkt nachvollziehen. Das Gelernte kommt allerdings auf mindestens eine Ebene mit menschlicher Leistung, sodass sich hieran eventuell Prozesse nachvollziehen lassen. Google hat beispielsweise ein gelehrtes CNN genutzt, um Halluzinationen zu simulieren, indem einzelnen Neuronen der letzten Schicht ein hohen soll-Wert zugewiesen wurde, um von da aus den „Fehler“ des Eingabebildes zu korrigieren. Diese sogenannten Deep Dreams sind faszinierende Bilder, in denen von einem CNN viele Fremdobjekte hineininterpretiert wurden.
Die Tatsache, dass Neuronen im Hirn zeitlich nacheinander aktiv sein können, wird in vielen Modellen nicht beachtet, eben sowenig, wie dass es Rückkopplungen geben kann. Es gibt aber auch Modelle wie ein Pulse Coupled Neural Networks oder Recurrent Neural Networks, die solche Komplexitäten miteinbeziehen. Während im Hirn längst nicht jedes Neuron mit jedem Verbunden ist, sind nahe beieinanderlie-

gende Bereiche oft stark untereinander verknüpft. So könnte man ein künstliches Neuronales Netz mit vielen Verbindungen wie ein MLP zumindest bezüglich seines Verhaltens als Teil eines Gehirns ansehen [vgl. 11, S. 80f].

Kapitel 4

Anwendungsmöglichkeiten

Die Spanne der Einsatzmöglichkeiten von künstlichen Neuronalen Netzen ist enorm. Je nach Anwendungszweck werden häufig sehr unterschiedliche Modelle angewendet. Mein Ziel in diesem Kapitel ist es, herauszufinden, wie gut sich verschiedene klassische Netztypen für typische Aufgaben eignen. In echten Anwendungen werden auch oft Kombinationen von verschiedenen Typen verwendet, um Vorteile aller Arten zu nutzen. Aktuell gibt es fast wöchentlich Fortschritte in der Videobearbeitung mit Neuronalen Netzen. Während die „FakeApp“, über die in vielen Medien aufgrund ihren unmoralischen Ursprüngen berichtet wurde noch eine Kombination von zwei Convolutional Neural Networks nutzt, welche wiederum aus einem typischen CNN und einem mit Upsampling bestehen, nutzt Facebooks „DensePose“ eine Mischung aus CNN und RNN, um zeitliche Verläufe in Videos mitzubeachten[12, S.2].
Ich möchte in diesem Kapitel herausfinden, ob sich die unterschiedlichen Typen Neuronaler Netze besonders für die Datenmengen eignen, für die sie gemacht wurden, oder ob durch entsprechende Anpassungen andere Typen eventuell besser sind. Testen werde ich dies in möglichst alltäglichen Szenarien, da ich so realistische Ergebnisse erhalten kann, sowie diese mit anderen Arbeiten, welche meist einzelne Alltagsszenarien behandeln, vergleichen.

4.1 Implementierung

Ich habe in Kapitel 2 beim Rechnen mit Gewichten ab und zu auch von Vektoren und Matrizen gesprochen, da man alle Operationen auf den Gewichten auch als Matrixoperationen schreiben kann. Beide gehören zur Gruppe der Tensore. Diese sind also sozusagen mehrdimensionale Matrizen. Daher passt der Name der Bibliothek, die ich zum implementieren der Tests nutze: TensorFlow. TensorFlow wird von Google entwickelt und bereits in vielen bekannten Plattformen und Geräten verwendet. Da das Rechnen mit Tensoren für mehrere Werte parallel geschehen kann, ist von TensorFlow eine für Grafikkarten optimierte Version verfügbar. Diese Installation benötigt allerdings zusätzliche Treiber, was die Verwendung erschwert. Der Code, um TensorFlow zu nutzen ist allerdings für Grafikkarten und CPUs derselbe. Ich schreibe die TensorFlow Anwendungen in Python, da die Bibliothek nur Interfaces für diese Sprache, sowie C anbietet, in Python allerdings einfacher zu installieren und verwenden ist.
Die Datensätze, welche ich in den Tests nutze, stammen zu großen Teilen aus Internetdatenbanken, in denen Datensätze für diverse Szenarien und Modelle verfügbar sind. Details zu den Datensätzen und Parametern der Netze beschreibe ich in den einzelnen Kapiteln.

4.2 Klassifizierung

Die Klassifizierung von Daten gehört zu den wichtigsten Anwendungen von künstlichen neuronalen Netzen, da sie ein wichtiger Bestandteil im Leben ist. Wenn ein Mensch Auto fährt, muss er fast sekündlich Schilder lesen, und wenn man ein Dokument digitalisiert, muss man jedes Zeichen erkennen. Diese Aufgaben sind auf Dauer anstrengend und könnten daher von einer Maschine übernommen werden. Gerade bei kritischen Aufgaben sollten einem Computer aber wesentlich weniger Fehler als einem Menschen unterlaufen. Daher ist es sinnvoll zu prüfen, wie gut sich die verschiedenen Typen für bestimmte Aufgaben eignen.

4.2.1 Zahlen

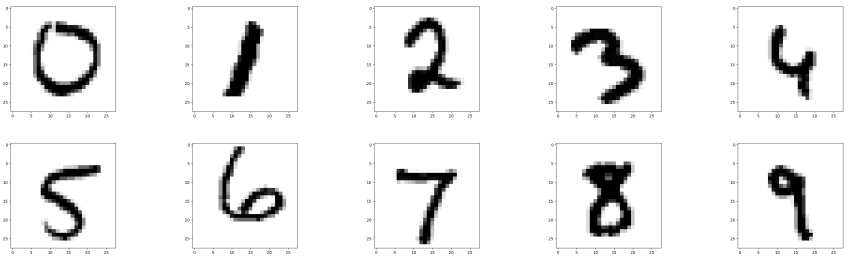

Abbildung 4.1: Einige Bilder von Ziffern aus dem MNIST Datensatz.

Die Erkennung von handgeschriebenen Ziffern gehört zu den Standardaufgaben eines neuronalen Netzes. Hier teste ich die Eignung der beiden Typen CNN und MLP anhand des MNIST Datensatzes, welcher aus rund 60000 Bildern von Zahlen der Auflösung $28 * 28$ besteht. Da das MLP das gesamte Bild auf einmal betrachtet, gehe ich davon aus, dass mit diesem eine höhere Korrektheit erreicht werden kann. Allerdings muss hierfür eine weit größere Zahl an Neuronen verwendet werden als bei den anderen Typen, sodass der Rechenaufwand bei größeren Aufgaben nicht mehr akzeptabel ist. Ein CNN sollte fast die gleiche Qualität erreichen, allerdings in sehr viel kürzerer Trainings- und Laufzeit.
Wirklich vergleichbare Ergebnisse zu erhalten ist schwierig, da es einige Parameter gibt, die sowohl das Ergebnis, als auch den Rechenaufwand beeinflussen, sowie solche, die nur das Ergebnis verändern, wie beispielsweise die Lernrate. Ist diese zu gering, kann das Netz im Lernprozess stecken bleiben, ist sie aber zu hoch, variieren die Gewichte sehr stark, sodass in keinem Fall ein Optimalergebnis erzielt werden kann. Mein Vorgehen ist es daher, Test mit verschiedenen Lernraten durchzuführen und diese bei guten Ergebnissen auf Konsistenz zu überprüfen. Meine Tests basieren auf dem von TensorFlow bereitgestellten Beispiel (`https://www.tensorflow.org/tutorials/layers`). Dieses stellt ein einfaches CNN dar, welches ich durch wenige Änderungen (siehe Anhang 7.1) in ein MLP umwandeln konnte.
Mit einem Training von je 100 Bildern auf einmal hat sich über 20000 Trainingsiterationen eine durchschnittliche Genauigkeit von $97,24\%$ beim MLP und $97,02\%$ beim CNN eingestellt. Allerdings hat das MLP mit der Lernrate $0,005$ 1391 Sekun-

den für das Training benötigt, während das CNN mit einer Lernrate von $0,001$ nur 370 Sekunden benötigte. Mit der gleichen Lernrate konnte ein MLP, wie in Kapitel 3.2 gezeigt, kaum etwas erreichen. Beide Netzwerke nutzen ReLU zur Aktivierung. Das genutzte CNN hat folgenden Aufbau: Zunächst beginnt das Netz mit der Verarbeitung der Eingabe durch eine Konvolutionsebene mit 32 Filtern der Auflösung $5 * 5$. Danach wird ein Pooling mit $2 * 2$ großen Bereichen angewendet, welche einen Abstand von 2 haben, sodass weder Pixel doppelt gepoolt werden können, noch welche ausgelassen werden. Hiernach gibt es eine zweite Konvolutionsebene mit wieder $5 * 5$ großen Filtern, allerdings 64 von diesen. Das MLP am Ende des CNN (die „Fully Connected Layer“) hat 1024 Neuronen in der Hidden Layer mit einer Dropoutrate von 40%. Diese modelliert die Wahrscheinlichkeit für die Deaktivierung eines Neurons (siehe Kapitel 3.2).

4.2.2 Akustische Worterkennung

Gesprochene Worte zu erkennen ist etwas alltägliches. Es gibt bereits einige Systeme, die auf Basis von Sprachbefehlen arbeiten. Beispiele hierfür sind etwa die Smart Speaker von Amazon oder Google. Diese Systeme müssen mit hoher Sicherheit Wörter wie „Ja“ und „Nein“ unterscheiden können. Daher habe ich einen Datensatz von Google [13] genutzt, in dem unter anderem einige tausend Sprachaufnahmen der Wörter „yes“ und „no“ enthalten sind. Ich habe dann ein Python-Script zum Laden dieser Daten entwickelt und mit einem CNN und einem MLP getestet. Da sich herausgestellt hat, dass das CNN trotz vieler Ebenen und verschiedenen Filtergrößen nur durch etwas Glück während des Tests besser war als pures Raten, habe ich die Sprachaufnahmen in Spektrogramme umgewandelt, was für noch bessere Ergebnisse als bei einem MLP gesorgt hat. Spektrogramme sind Bilder, auf denen die Lautstärke unterschiedlicher Frequenzen in einem Zeitraum dargestellt werden. Die waagerechte

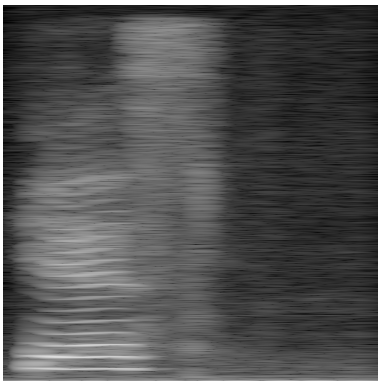

Abbildung 4.2: Ein Spektrogramm einer meiner „yes“-Aufnahmen. Hier kann man anhand des verrauschten Bereichs in der Mitte des Bildes sehr gut den Zischlaut erkennen.

Achse stellt hierbei meist die Zeit da, die horizontale die Frequenz/Tonhöhe, und die Helligkeit der Pixel entspricht der Lautstärke.
Das MLP hat drei versteckten Ebenen, welche die Neuronenanzahlen 4096, 4096 und 1024 haben. Zunächst wollte ich in der ersten versteckten Ebene mehr Neuronen haben, dies war aufgrund des begrenzten Arbeitsspeichers meiner Grafikkarte aber nicht möglich und wie sich herausstellte nicht einmal nötig, da das Netz auch so mit einem Training von 5000 Schritten der Lernrate $0,05$ eine Genauigkeit von durchschnittlich $98,80\%$ erreichte.
Das CNN, welches aufgrund der eindimensionalen Sounddaten eindimensionale Faltungen anstelle der sonst zweidimensionalen vornimmt, hat folgende versteckte Ebenen in dieser Reihenfolge: Convolution mit 32 Filtern der Länge 5, Convolution mit 64 Filtern der Länge 10, Pooling der Länge 2 mit Schrittgröße 2, Convolution mit 32 Filtern der Länge 10, Convolution mit 128 Filtern der Länge 20, Pooling der Länge 2 mit Schrittgröße 2, Convolution mit 64 Filtern der Länge 20, Convolution mit 32 Filtern der Länge 40, Fully Connected Layer mit 1024 Neuronen und 40% Dropout. Dass die Länge der Filter mit der Tiefe der Ebenen zunimmt, habe ich eingefügt, da die Audiodaten sehr kleinschrittig sind und daher große Bereiche abgedeckt werden müssen, um nutzbare Ergebnisse zu erhalten. Dennoch wurde bei 2000 Trainingsschritten nur eine durchschnittliche Akkuratesse von 55% erreicht.
Das Spektrogramm-CNN hat den gleichen Aufbau mit dem Unterschied, dass die Filter alle eine Größe von $5 * 5$ haben und die Poolings ebenfalls wieder zweidimensional mit einer Größe von $2 * 2$ sind. So viele Filter wirken sich auch auf die Trainingszeit aus, welche mit ca. $360s$ für 2000 Schritte sehr hoch ist. Im Vergleich dazu hat das MLP nur etwas mehr als 5 Minuten für mehr als Doppelt so viele Schritte gebraucht. Andererseits hat das CNN mit der Lernrate $0,01$ eine Genauigkeit von durchschnittlich $99,07\%$ erreicht, was minimal höher ist als das MLP, deren bestes Ergebnis sogar über dem des CNN liegt. Die vollständigen Tabellen finden sich im Anhang.
Da mich die gute Performance des CNNs trotz kürzeren Trainings sehr gewundert hat, habe ich selbst ein paar Sprachaufnahmen erstellt und vom Netzwerk klassifizieren lassen. Diese wurden zu 100% richtig identifiziert. Da ich vermutet habe, dass das Netzwerk die Aufnahmen anhand des ‚s‘ unterscheidet, habe ich es auch mit Aufnahmen von „jo“ und „nose“ getestet. Es hat wie erwartet „jo“ als „no“ und

„nose“ als „yes“ klassifiziert.
Es wäre sicherlich noch von großer Bedeutung, ein RNN in dieser Aufgabe zu bemühen, da es für genau solch eine Art von Daten modelliert wurde. Die Implementierung eines solchen würde aber den Rahmen dieser Arbeit sprengen.

4.3 Vorhersagen

Da Vorhersagen in vielen Bereichen oft hilfreich sind, möchte ich mich mit diesen auseinandersetzen. Viele Ereignisse in der Zukunft hängen von solchen in der Vergangenheit ab. Das erinnert an des Prinzip eines RNNs, welches sich die vergangenen Schritte merkt. Vielleicht reicht es aber auch aus, einem anderen Netztypen immer mehrere Werte aus der Vergangenheit zu zeigen, um einen Wert für den nächsten Zeitschritt in der Zukunft zu erhalten.

4.3.1 Börse

Aktienkurse schwanken sehr stark und es scheint selbst für Profis unmöglich, diese annähernd vorherzusehen. Möglicherweise schafft es also ein Neuronales Netz eben so gut, ohne weitere Daten Aktienkurse vorauszusagen, wie Menschen, die Zugang zum Weltgeschehen haben.
Ich habe hier selbst einen Test implementiert, allerdings nur mit einem MLP. Den Datensatz (in meinem Test von Ölpreisen) habe ich von Yahoo herunterladen können. Das MLP mit drei hidden Layers der Größen 500, 200 und 50 hat jeweils 100 Eröffnungspreise von hintereinanderliegenden Tagen ausgehend von einem zufälligen Datum bekommen und sollte den nächsten Bestimmen. Es hat sich herausgestellt, dass hierfür eine geringere Lernrate als sonst (nämlich $0,0000001$) nötig ist, und auch so konnte das MLP nach 20000 Trainingsschritten nur eine Genauigkeit erreichen, die durchschnittlich 1,3$ vom tatsächlichen Preis abgewichen ist. Ein RNN hätte dies vermutlich etwas besser geschafft, dies kann ich im Umfang dieser Arbeit allerdings nicht prüfen. Der Code für den Datenimport und das Netz finden sich im Anhang.

4.4 Erzeugung

Zur Generierung von realistischen Daten hat sich eine Technik namens Generative Adversarial Network, kurz GAN, durchgesetzt. Ein GAN arbeitet mit zwei Neuronalen Netzwerken, die sich gegenseitig verbessern sollen. Eines dieser Netzwerke generiert Daten auf Basis von Rauschen oder zugewiesenen Attributen. Dieses wird Generator genannt. Das andere Netzwerk, der Diskriminator, versucht nun festzustellen, ob die Daten realistisch sind. Hierzu wird es sowohl mit echten Daten gefüttert, als auch mit generierten und soll diese unterteilen in generierte und echte Daten. Das Generatornetz kann sich nun anhand der Einschätzung des Diskriminators verbessern. Da das Netzwerk nicht mit beschrifteten Daten versorgt wird, sondern anhand von Beispielen eigenständig lernt, diese zu replizieren, zählt man GANs zu den unüberwacht lernenden Netzparadigmen.

4.4.1 Portraits

Ein Beispiel für das Erzeugen von Gesichtsbildern ist das Werk von *Larsen et al.[6]*. Dort hat man zusätzlich zu einem GAN einen Autoencoder genutzt, welcher die Datenmengen in einem Bild über ein Convolutional Neural Network (Encoder) verringert und dann mit einem Deconvolutional Neural Network (Decoder) wieder in ein Bild umwandelt, welches vom Diskriminator überprüft wird.
Dieses Konzept ließ sich nutzen, um durch Vorgeben von Daten an das Decodernetz synthetische Bilder zu erzeugen, aber auch, um die Daten des Encoders vor der Weitergabe zu manipulieren und so auf Basis eines Portraits eines mit veränderten Attributen zu erstellen, beispielsweise einer anderen Haut- oder Haarfarbe oder auch einer anderen Frisur. Obwohl die so erzeugten Bilder vom Original zum Teil sehr abwichen, sahen sie größtenteils jedoch realistisch aus.
Es wurde außerdem gezeigt, dass sich eine GAN-Struktur hierfür besser eignet als ein einzelner Autoencoder, welcher die Bilder nur anhand der Abweichung rekonstruiert.

4.5 Weitere Anwendungen

Da es im Rahmen dieser Arbeit unmöglich ist, alle Anwendungsmöglichkeiten zu betrachten, habe ich hier nun einige weiterführende Literatur zu anderen, interessanten Anwendungen Neuronaler Netze zusammengestellt. Diese beinhalten auch weitere Typen, welche ich in dieser Arbeit nicht behandelt habe.

- Reading Text in the Wild with Convolutional Neural Networks [9]
- Dense Human Pose Estimation In The Wild [12]
- Multi-Layer Perceptron (MLP)-Based [...] Stock Forecasting Model [14]
- Facial expression recognition based on a mlp [...] [15]
- Sparse Multilayer Perceptron for Phoneme Recognition [16]
- A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction [17]
- Composition-aided Sketch-realistic Portrait Generation [18]
- Global-Local Face Upsampling Network [19]
- Convolutional Neural Networks for Distant Speech Recognition [20]
- Convolutional Neural Networks for Speech Recognition [21]
- 3D Convolutional Neural Networks for Human Action Recognition [22]
- Convolutional Neural Network Based Automatic Object Detection on Aerial Images [23]
- Recurrent Neural Networks for Short-Term Load Forecasting [7]
- Human localization in the video stream using [...] growing neural gas and fuzzy inference [24]
- Writing in the Air Using Kinect and Growing Neural Gas Network [25]

- Neural Gas and Growing Neural Gas Networks for Selective 3D Sensing [26]
- A Novel Word Spotting Method Based on Recurrent Neural Networks [27]
- Applications of Pulse-Coupled Neural Networks [28]
- Image processing using pulse-coupled neural networks [29]
- Background Subtraction Based on Pulse Coupled Neural Network [30]

Kapitel 5

Fazit

Die Menge der Anwendungsmöglichkeiten von Neuronalen Netzen ist enorm, daher sind sie nicht ohne Grund das aktuell populärste Feld der Informatik. Auch aufgrund der Vielseitigkeit, mit der sie eingesetzt werden können, haben sie eine große Bedeutung für die Menschheit. Ich denke, dass künstliche Neuronale Netze eine ähnlich prägende Errungenschaft wie der Computer sind und daher noch viel Forschungsarbeit in diese investiert werden muss. Zur Zeit erfordert ein Arbeiten mit ihnen noch einiges an Einarbeitung, doch das wird sich in den nächsten Jahren vermutlich ändern. Ich habe mich während dieser Arbeit viel mit dem Aufbau von Neuronalen Netzen beschäftigt und kann sagen, dass das Verständnis des Themas sehr komplex ist, aber ungemein hilft bei der Implementierung von künstlichen Neuronalen Netzen. Vermutlich wird sich auch das in der nächsten Zeit ändern, sodass die Entwicklung solcher Netze ähnlich simpel wie einfaches Programmieren sein wird. Produkte, welche auf künstlichen Neuronalen Netzen basieren, gibt es jetzt schon zuhauf. Neuronale Netze bieten eine solche Fülle an Möglichkeiten, dass es für nahezu jeden möglich sein sollte, diese für eigene Anwendungen zu implementieren.

Ich habe während dieser Arbeit viel gelernt, was mich auch auf neue Forschungsfragen und Ideen gebracht hat, zu denen ich noch keine Veröffentlichungen gefunden habe, wie etwa die vollsynthetische Erzeugung von Videomaterial. Es kann eigentlich jede beliebige Aufgabe mit einem Neuronalen Netz gelöst werden. Auch wenn ein Typ einer Aufgabe zunächst nicht gewachsen scheint, so gibt es nahezu immer

eine Möglichkeit, die Daten in nutzbare Form zu bringen oder verschiedene Typen zu kombinieren.
Bezogen auf meine Untersuchungen lässt sich abschließend sagen, dass sowohl die erweiterten Varianten einfacher MLPs, als auch die speziellsten Typen ihre Daseinsberechtigung haben, da sie den Trainingsprozess, genauso wie die Nutzung beschleunigen und zum Teil sogar einfacher zu trainieren sind. Beispielsweise lässt sich anhand des MNIST-Beispiels aus Kapitel 4.2.1 zeigen, dass das CNN, welches auf Bilder ausgelegt ist, sich auch besser eignet, um diese zu klassifizieren.
Ich freue mich darauf, die Arbeit an diesem Thema auch nach Abschluss der Facharbeit fortzuführen.

Kapitel 6

Literaturverzeichnis

[1] Kriesel, David: *Ein kleiner Überblick Über Neuronale Netze*. 2007. http://www.dkriesel.com/science/neural_networks.

[2] Cichocki, Andrzej und Rolf Unbehauen: *Neural networks for optimization and signal processing*. Wiley, Chichester, reprinted with corr Auflage, 1994, ISBN 3519064448. https://www.ub.tu-dortmund.de/katalog/titel/HT006268875.

[3] Bishop, Christopher M.: *Neural networks for pattern recognition*. Oxford University Press and Clarendon Press, Oxford and Oxford, 1995, ISBN 0198538499. https://www.ub.tu-dortmund.de/katalog/titel/HT006707577.

[4] Siraj Raval: *Which Activation Function Should I Use?*, 2017. https://www.youtube.com/watch?v=-7scQpJT7uo, besucht: 25.2.2018.

[5] Habibi Aghdam, Hamed und Elnaz Jahani Heravi: *Guide to Convolutional Neural Networks*. Springer International Publishing, Cham, 2017, ISBN 978-3-319-57549-0.

[6] Larsen, Anders Boesen Lindbo, Søren Kaae Sønderby, Hugo Larochelle und Ole Winther: *Autoencoding beyond pixels using a learned similarity metric*. http://arxiv.org/pdf/1512.09300v2.

[7] Bianchi, Filippo Maria, Enrico Maiorino, Michael C. Kampffmeyer, Antonello Rizzi und Robert Jenssen: *Recurrent Neural Networks for Short-*

Term Load Forecasting. Springer International Publishing, Cham, 2017, ISBN 978-3-319-70337-4.

[8] Kingma, Diederik P. und Jimmy Ba: *Adam: A Method for Stochastic Optimization*. http://arxiv.org/pdf/1412.6980v9.

[9] Jaderberg, Max, Karen Simonyan, Andrea Vedaldi und Andrew Zisserman: *Reading Text in the Wild with Convolutional Neural Networks*. International Journal of Computer Vision, 116(1):1–20, 2016, ISSN 09205691. http://search.ebscohost.com/login.aspx?direct=true&db=egs&AN=112155640&site=ehost-live.

[10] Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever und Ruslan R. Salakhutdinov: *Improving neural networks by preventing co-adaptation of feature detectors*. http://arxiv.org/pdf/1207.0580v1.

[11] Kohonen, Teuvo: *Self-Organizing Maps*, Band 30 der Reihe *Springer Series in Information Sciences*. Springer, Berlin and Heidelberg, third edition Auflage, 2001, ISBN 9783540679219. http://dx.doi.org/10.1007/978-3-642-56927-2.

[12] Güler, Rıza Alp, Natalia Neverova und Iasonas Kokkinos: *DensePose: Dense Human Pose Estimation In The Wild*. http://arxiv.org/pdf/1802.00434v1.

[13] Warden, Pete: *Speech Commands: A public dataset for single-word speech recognition.* Dataset available from http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz, 2017.

[14] Yassin, I. M., M. F. Abdul Khalid, S. H. Herman, I. Pasya, N. Ab Wahab und Z. Awang: *Multi-Layer Perceptron (MLP)-Based Nonlinear Auto-Regressive with Exogenous Inputs (NARX) Stock Forecasting Model*. International Journal on Advanced Science, Engineering and Information Technology, 7(3):1098–1103, 2017, ISSN 2088-5334.

[15] Boughrara, Hayet, Mohamed Chtourou, Chokri Ben Amar und Liming Chen: *Facial expression recognition based on a mlp neural network using constructive*

training algorithm. Multimedia Tools and Applications, 75(2):709–731, 2016, ISSN 1380-7501.

[16] Sivaram, G. S. V. S. und H. Hermansky: *Sparse Multilayer Perceptron for Phoneme Recognition*. IEEE Transactions on Audio, Speech, and Language Processing, 20(1):23–29, 2012, ISSN 1558-7916.

[17] Chollampatt, Shamil und Hwee Tou Ng: *A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction*. `http://arxiv.org/pdf/1801.08831v1`.

[18] Gao, Fei, Shengjie Shi, Jun Yu und Qingming Huang: *Composition-aided Sketch-realistic Portrait Generation*. `http://arxiv.org/pdf/1712.00899v1`.

[19] Tuzel, Oncel, Yuichi Taguchi und John R. Hershey: *Global-Local Face Upsampling Network*. `http://arxiv.org/pdf/1603.07235v2`.

[20] Swietojanski, P., A. Ghoshal und S. Renals: *Convolutional Neural Networks for Distant Speech Recognition*. IEEE Signal Processing Letters, 21(9):1120–1124, 2014, ISSN 1070-9908.

[21] Abdel-Hamid, O., A. r. Mohamed, H. Jiang, L. Deng, G. Penn und D. Yu: *Convolutional Neural Networks for Speech Recognition*. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 22(10):1533–1545, 2014, ISSN 2329-9290.

[22] Ji, S., W. Xu, M. Yang und K. Yu: *3D Convolutional Neural Networks for Human Action Recognition*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(1):221–231, 2013, ISSN 0162-8828.

[23] Ševo, I. und A. Avramović: *Convolutional Neural Network Based Automatic Object Detection on Aerial Images*. IEEE Geoscience and Remote Sensing Letters, 13(5):740–744, 2016.

[24] Amosov, O. S., Y. S. Ivanov und S. V. Zhiganov: *Human Localization in the Video Stream Using the Algorithm Based on Growing Neural Gas and Fuzzy Inference*. Procedia Computer Science, 103(Supplement C):403–409, 2017, ISSN 1877-0509.

[25] Heidari, Mohammad Reza Aminian, Azrulhizam Shapi'i und Riza Sulaiman: *Writing in the Air Using Kinect and Growing Neural Gas Network.* Jurnal Teknologi, 72(5), 2015.

[26] Petriu, Emil M., Ana Maria Cretu und Pierre Payeur: *Neural Gas and Growing Neural Gas Networks for Selective 3D Sensing: a Comparative Study.* Sensors & Transducers Journal, (5), 2009.

[27] Frinken, V., A. Fischer, R. Manmatha und H. Bunke: *A Novel Word Spotting Method Based on Recurrent Neural Networks.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 34(2):211–224, 2012, ISSN 0162-8828.

[28] Ma, Yide, Kun Zhan und Zhaobin Wang: *Applications of Pulse-Coupled Neural Networks.* Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2010. Auflage, 2011, ISBN 9783642137440. `http://dx.doi.org/10.1007/978-3-642-13745-7`.

[29] Lindblad, Thomas und Jason M. Kinser: *Image processing using pulse-coupled neural networks: Applications in Python.* Biological and medical physics, biomedical engineering. Springer, Berlin, 3. ed. Auflage, 2013, ISBN 9783642368769. `http://dx.doi.org/10.1007/978-3-642-36877-6`.

[30] Cai, Xi, Guang Han und Jin Kuan Wang: *Background Subtraction Based on Pulse Coupled Neural Network.* In: Industrial Engineering, Computation and Information Technologies (Herausgeber): *Industrial Engineering, Computation and Information Technologies*, Band 701 der Reihe *Applied Mechanics and Materials*, Seiten 293–296. Trans Tech Publications, 2015.

Kapitel 7

Anhang

7.1 Code MLP MNIST

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

def mlp_model_fn(features, labels, mode):
  """Model function for MLP."""
  # Input Layer
  # Reshape X to flat tensor: [batch_size, size]
```

```
# MNIST images are 28x28 pixels, and have one color channel
input_layer = tf.reshape(features["x"], [-1, 28 * 28 * 1])

# Hidden Layer #1
# 16384 Neurons
hidd1 = tf.layers.dense(
    inputs=input_layer,
    units=16384,
    activation=tf.nn.sigmoid)

# Hidden Layer #2
# 4096 Neurons
hidd2 = tf.layers.dense(
    inputs=hidd1,
    units=4096,
    activation=tf.nn.sigmoid)

# Hidden Layer #3
# 1024 Neurons
hidd3 = tf.layers.dense(
    inputs=hidd2,
    units=1024,
    activation=tf.nn.sigmoid)

# Logits layer
# Input Tensor Shape: [batch_size, 1024]
# Output Tensor Shape: [batch_size, 10]
logits = tf.layers.dense(inputs=hidd3, units=10)

predictions = {
    # Generate predictions (for PREDICT and EVAL mode)
    "classes": tf.argmax(input=logits, axis=1),
    # Add `softmax_tensor` to the graph. It is used for PREDICT and by the `logging_hook`.
    "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
  return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.005)
  train_op = optimizer.minimize(
      loss=loss,
      global_step=tf.train.get_global_step())
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
```

```
    eval_metric_ops = {
        "accuracy": tf.metrics.accuracy(
            labels=labels, predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):
    # Load training and eval data
    mnist = tf.contrib.learn.datasets.load_dataset("mnist")
    train_data = mnist.train.images  # Returns np.array
    train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
    eval_data = mnist.test.images  # Returns np.array
    eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

    # Create the Estimator
    mnist_classifier = tf.estimator.Estimator(
        model_fn=mlp_model_fn, model_dir="/tmp/mnist_mlp_model")

    # Set up logging for predictions
    # Log the values in the "Softmax" tensor with label "probabilities"
    tensors_to_log = {"probabilities": "softmax_tensor"}
    logging_hook = tf.train.LoggingTensorHook(
        tensors=tensors_to_log, every_n_iter=500)

    begin_time = time.clock()

    # Train the model
    train_input_fn = tf.estimator.inputs.numpy_input_fn(
        x={"x": train_data},
        y=train_labels,
        batch_size=100,
        num_epochs=None,
        shuffle=True)
    mnist_classifier.train(
        input_fn=train_input_fn,
        steps=20000,
        hooks=[logging_hook])

    end_time = time.clock()

    # Evaluate the model and print results
    eval_input_fn = tf.estimator.inputs.numpy_input_fn(
        x={"x": eval_data},
        y=eval_labels,
        num_epochs=1,
        shuffle=False)
    eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
    print(eval_results)
    print("Training took ", end_time - begin_time," seconds")
```

```
if __name__ == "__main__":
  tf.app.run()
```

7.2 Code MLP MNIST (ReLU)

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  # Reshape X to 4-D tensor: [batch_size, width, height, channels]
  # MNIST images are 28x28 pixels, and have one color channel
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Convolutional Layer #1
  # Computes 32 features using a 5x5 filter with ReLU activation.
  # Padding is added to preserve width and height.
  # Input Tensor Shape: [batch_size, 28, 28, 1]
  # Output Tensor Shape: [batch_size, 28, 28, 32]
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
```

```
    activation=tf.nn.relu)

# Pooling Layer #1
# First max pooling layer with a 2x2 filter and stride of 2
# Input Tensor Shape: [batch_size, 28, 28, 32]
# Output Tensor Shape: [batch_size, 14, 14, 32]
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

# Convolutional Layer #2
# Computes 64 features using a 5x5 filter.
# Padding is added to preserve width and height.
# Input Tensor Shape: [batch_size, 14, 14, 32]
# Output Tensor Shape: [batch_size, 14, 14, 64]
conv2 = tf.layers.conv2d(
    inputs=pool1,
    filters=64,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu)

# Pooling Layer #2
# Second max pooling layer with a 2x2 filter and stride of 2
# Input Tensor Shape: [batch_size, 14, 14, 64]
# Output Tensor Shape: [batch_size, 7, 7, 64]
pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

# Flatten tensor into a batch of vectors
# Input Tensor Shape: [batch_size, 7, 7, 64]
# Output Tensor Shape: [batch_size, 7 * 7 * 64]
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

# Dense Layer
# Densely connected layer with 1024 neurons
# Input Tensor Shape: [batch_size, 7 * 7 * 64]
# Output Tensor Shape: [batch_size, 1024]
dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)

# Add dropout operation; 0.6 probability that element will be kept
dropout = tf.layers.dropout(
    inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

# Logits layer
# Input Tensor Shape: [batch_size, 1024]
# Output Tensor Shape: [batch_size, 10]
logits = tf.layers.dense(inputs=dropout, units=10)

predictions = {
    # Generate predictions (for PREDICT and EVAL mode)
    "classes": tf.argmax(input=logits, axis=1),
    # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
```

```
      # 'logging_hook'.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):
  # Load training and eval data
  mnist = tf.contrib.learn.datasets.load_dataset("mnist")
  train_data = mnist.train.images  # Returns np.array
  train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
  eval_data = mnist.test.images  # Returns np.array
  eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

  # Create the Estimator
  mnist_classifier = tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")

  # Set up logging for predictions
  # Log the values in the "Softmax" tensor with label "probabilities"
  tensors_to_log = {"probabilities": "softmax_tensor"}
  logging_hook = tf.train.LoggingTensorHook(
      tensors=tensors_to_log, every_n_iter=500)

  begin_time = time.clock()

  # Train the model
  train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": train_data},
      y=train_labels,
      batch_size=100,
```

```
      num_epochs=None,
      shuffle=True)
  mnist_classifier.train(
      input_fn=train_input_fn,
      steps=20000,
      hooks=[logging_hook])

  end_time = time.clock()

  # Evaluate the model and print results
  eval_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      num_epochs=1,
      shuffle=False)
  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
  print(eval_results)
  print("Training took ", end_time - begin_time," seconds")


if __name__ == "__main__":
  tf.app.run()
```

7.3 Code CNN MNIST

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)
```

```
def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  # Reshape X to 4-D tensor: [batch_size, width, height, channels]
  # MNIST images are 28x28 pixels, and have one color channel
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Convolutional Layer #1
  # Computes 32 features using a 5x5 filter with ReLU activation.
  # Padding is added to preserve width and height.
  # Input Tensor Shape: [batch_size, 28, 28, 1]
  # Output Tensor Shape: [batch_size, 28, 28, 32]
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #1
  # First max pooling layer with a 2x2 filter and stride of 2
  # Input Tensor Shape: [batch_size, 28, 28, 32]
  # Output Tensor Shape: [batch_size, 14, 14, 32]
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

  # Convolutional Layer #2
  # Computes 64 features using a 5x5 filter.
  # Padding is added to preserve width and height.
  # Input Tensor Shape: [batch_size, 14, 14, 32]
  # Output Tensor Shape: [batch_size, 14, 14, 64]
  conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #2
  # Second max pooling layer with a 2x2 filter and stride of 2
  # Input Tensor Shape: [batch_size, 14, 14, 64]
  # Output Tensor Shape: [batch_size, 7, 7, 64]
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

  # Flatten tensor into a batch of vectors
  # Input Tensor Shape: [batch_size, 7, 7, 64]
  # Output Tensor Shape: [batch_size, 7 * 7 * 64]
  pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

  # Dense Layer
```

```
  # Densely connected layer with 1024 neurons
  # Input Tensor Shape: [batch_size, 7 * 7 * 64]
  # Output Tensor Shape: [batch_size, 1024]
  dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)

  # Add dropout operation; 0.6 probability that element will be kept
  dropout = tf.layers.dropout(
      inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

  # Logits layer
  # Input Tensor Shape: [batch_size, 1024]
  # Output Tensor Shape: [batch_size, 10]
  logits = tf.layers.dense(inputs=dropout, units=10)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
      # `logging_hook`.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):
  # Load training and eval data
  mnist = tf.contrib.learn.datasets.load_dataset("mnist")
  train_data = mnist.train.images  # Returns np.array
  train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
  eval_data = mnist.test.images  # Returns np.array
  eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)
```

```
  # Create the Estimator
  mnist_classifier = tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")

  # Set up logging for predictions
  # Log the values in the "Softmax" tensor with label "probabilities"
  tensors_to_log = {"probabilities": "softmax_tensor"}
  logging_hook = tf.train.LoggingTensorHook(
      tensors=tensors_to_log, every_n_iter=500)

  begin_time = time.clock()

  # Train the model
  train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": train_data},
      y=train_labels,
      batch_size=100,
      num_epochs=None,
      shuffle=True)
  mnist_classifier.train(
      input_fn=train_input_fn,
      steps=20000,
      hooks=[logging_hook])

  end_time = time.clock()

  # Evaluate the model and print results
  eval_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      num_epochs=1,
      shuffle=False)
  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
  print(eval_results)
  print("Training took ", end_time - begin_time," seconds")


if __name__ == "__main__":
  tf.app.run()
```

7.4 Code MLP Speech

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
```

```
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf
from mydata import DataSetGenerator

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

def mlp_model_fn(features, labels, mode):
  """Model function for MLP."""
  # Input Layer
  # Reshape X to flat tensor: [batch_size, size]
  input_layer = tf.reshape(features["x"], [-1, 16000])

  # Hidden Layer #1
  # 16384 Neurons
  hidd1 = tf.layers.dense(
      inputs=input_layer,
      units=4096,
      activation=tf.nn.relu)

  # Hidden Layer #2
  # 4096 Neurons
  hidd2 = tf.layers.dense(
      inputs=hidd1,
      units=4096,
      activation=tf.nn.relu)

  # Hidden Layer #3
  # 1024 Neurons
  hidd3 = tf.layers.dense(
      inputs=hidd2,
      units=1024,
      activation=tf.nn.relu)

  # Logits layer
  # Input Tensor Shape: [batch_size, 1024]
  # Output Tensor Shape: [batch_size, 10]
```

```
  logits = tf.layers.dense(inputs=hidd3, units=2)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add 'softmax_tensor' to the graph. It is used for PREDICT and by the 'logging_hook'.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.05)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):

  # Load training and eval data
  dg = DataSetGenerator(".\Data")
  #train_data, train_labels = dg.get_mini_batches(10000,(128,128), allchannel=False)
  #eval_data, eval_labels = dg.get_mini_batches(1000,(128,128), allchannel=False)
  print("Reading test and training files...")
  train = dg.get_mini_batches(4000)
  eval = dg.get_mini_batches(250)
  for data, labels in train:
      train_data, train_labels = data,labels
  for data, labels in eval:
      eval_data, eval_labels = data,labels
  print("Got ",len(train_data),"/",len(train_labels)," files for training and ",
              len(eval_data),"/",len(eval_labels)," for evaluation")
  print("Done. Beginning training...")

  # Create the Estimator
  mnist_classifier = tf.estimator.Estimator(
      model_fn=mlp_model_fn, model_dir="/tmp/mnist_mlp_model")
```

```
  # Set up logging for predictions
  # Log the values in the "Softmax" tensor with label "probabilities"
  tensors_to_log = {"probabilities": "softmax_tensor"}
  logging_hook = tf.train.LoggingTensorHook(
      tensors=tensors_to_log, every_n_iter=100)

  begin_time = time.clock()

  # Train the model
  train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": train_data},
      y=train_labels,
      batch_size=10,
      num_epochs=None,
      shuffle=True)
  mnist_classifier.train(
      input_fn=train_input_fn,
      steps=5000,
      hooks=[logging_hook])

  end_time = time.clock()

  # Evaluate the model and print results
  eval_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      batch_size=25,
      num_epochs=10,
      shuffle=False)
  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
  print(eval_results)
  print("Training took ", end_time - begin_time," seconds")


if __name__ == "__main__":
  tf.app.run()
```

7.5 Code CNN Speech

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
```

```
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf
from mydata import DataSetGenerator

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

def cnn_model_fn(features, labels, mode):
    """Model function for CNN."""
    # Input Layer
    input_layer = tf.reshape(features["x"], [-1, 16000, 1])

    # Convolutional Layer #1
    conv1 = tf.layers.conv1d(
            inputs=input_layer,
            filters=32,
            kernel_size=[5],
            padding="same",
            activation=tf.nn.relu)
    conv2 = tf.layers.conv1d(
            inputs=conv1,
            filters=64,
            kernel_size=[10],
            padding="same",
            activation=tf.nn.relu)

    # Pooling Layer #1
    pool1 = tf.layers.max_pooling1d(inputs=conv2, pool_size=[2], strides=2)

    # Convolutional Layer #2
    conv3 = tf.layers.conv1d(
            inputs=pool1,
            filters=32,
            kernel_size=[10],
            padding="same",
            activation=tf.nn.relu)
    conv4 = tf.layers.conv1d(
            inputs=conv3,
            filters=128,
            kernel_size=[20],
```

```
        padding="same",
        activation=tf.nn.relu)

# Pooling Layer #2
pool2 = tf.layers.max_pooling1d(inputs=conv4, pool_size=[2], strides=2)

conv5 = tf.layers.conv1d(
        inputs=pool2,
        filters=64,
        kernel_size=[20],
        padding="same",
        activation=tf.nn.relu)
conv6 = tf.layers.conv1d(
        inputs=conv5,
        filters=32,
        kernel_size=[40],
        padding="same",
        activation=tf.nn.relu)
pool3 = tf.layers.max_pooling1d(inputs=conv6, pool_size=[2], strides=2)

# Flatten tensor into a batch of vectors (got value from error message lol)
pool3_flat = tf.reshape(pool3, [-1, 2000 * 32])

# Dense Layer
# Densely connected layer with 1024 neurons
dense = tf.layers.dense(inputs=pool3_flat, units=1024, activation=tf.nn.relu)

# Add dropout operation; 0.4 probability that a neuron is blocked to prevent overfitting
dropout = tf.layers.dropout(
        inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

# Logits layer
logits = tf.layers.dense(inputs=dropout, units=2)

predictions = {
        # Generate predictions (for PREDICT and EVAL mode)
        "classes": tf.argmax(input=logits, axis=1),
        # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
        # `logging_hook`.
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
```

```
        train_op = optimizer.minimize(
                loss=loss,
                global_step=tf.train.get_global_step())
        return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

    # Add evaluation metrics (for EVAL mode)
    eval_metric_ops = {
            "accuracy": tf.metrics.accuracy(
                    labels=labels, predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(
            mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):

    # Load training and eval data
    dg = DataSetGenerator(".\Data")
    #train_data, train_labels = dg.get_mini_batches(10000,(128,128), allchannel=False)
    #eval_data, eval_labels = dg.get_mini_batches(1000,(128,128), allchannel=False)
    print("Reading test and training files...")
    train = dg.get_mini_batches(40)
    eval = dg.get_mini_batches(20)
    for data, labels in train:
        train_data, train_labels = data,labels
    for data, labels in eval:
        eval_data, eval_labels = data,labels
    print("Got ",len(train_data),"/",len(train_labels)," files for training and ",
                len(eval_data),"/",len(eval_labels)," for evaluation")
    #cv2.imshow('image',train_data[0])
    #cv2.waitKey(0)
    #cv2.destroyAllWindows()
    print("Done. Beginning training...")

    # Create the Estimator
    classifier = tf.estimator.Estimator(
            model_fn=cnn_model_fn, model_dir="/tmp/convnet_model")

    # Set up logging for predictions
    # Log the values in the "Softmax" tensor with label "probabilities"
    tensors_to_log = {"probabilities": "softmax_tensor"}
    logging_hook = tf.train.LoggingTensorHook(
            tensors=tensors_to_log, every_n_iter=100)

    begin_time = time.clock()

    # Train the model
    train_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": train_data},
            y=train_labels,
            batch_size=20,
```

```
            num_epochs=None,
            shuffle=True)
    classifier.train(
            input_fn=train_input_fn,
            steps=100,
            hooks=[logging_hook])

    end_time = time.clock()

    # Evaluate the model and print results
    eval_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": eval_data},
            y=eval_labels,
            num_epochs=1,
            shuffle=False)
    eval_results = classifier.evaluate(input_fn=eval_input_fn)
    print(eval_results)
    print("Training took ", end_time - begin_time," seconds")


if __name__ == "__main__":
    tf.app.run()
```

7.6 Code Datenschnittstelle Speech

```
# Based on https://thecodacus.com/prepare-data-set-train-tensorflow-model/


import wave # to load the wave files
import struct # to decode the wave files
import numpy as np # to do matrix manipulations
from os.path import isfile, join # to manipulate file paths
from os import listdir # get list of all the files in a directory
from random import shuffle # shuffle the data (file paths)

import time

def parse_wave_python(filename):
    with wave.open(filename, 'rb') as wave_file:
        # would be 16000 for Google's speech dataset
        length = wave_file.getnframes()
        samples = np.zeros(length,dtype=int)
        for i in range(length):
            samples[i] = struct.unpack('<h', wave_file.readframes(1))[0]
    return samples

class DataSetGenerator:
```

```
    def __init__(self, data_dir):
        self.data_dir = data_dir
        self.data_labels = self.get_data_labels() # 1D-List
        self.data_info = self.get_data_paths() # 2D-List, first dimension is
                                               #the corresponding label's index

    def get_data_labels(self):
        data_labels = []
        for filename in listdir(self.data_dir):
            if not isfile(join(self.data_dir, filename)): # is folder
                data_labels.append(filename)
        return data_labels

    def get_data_paths(self):
        data_paths = []
        for label in self.data_labels:
            wav_lists=[]
            path = join(self.data_dir, label)
            for filename in listdir(path):
                tokens = filename.split('.')
                if tokens[-1] == 'wav':
                    wave_path=join(path, filename)
                    wav_lists.append(wave_path)
            shuffle(wav_lists)
            data_paths.append(wav_lists)
        return data_paths

    def get_mini_batches(self, batch_size=100):
        files = []
        labels = []
        empty=False
        counter=0
        each_batch_size=int(batch_size/len(self.data_labels))
        while True:
            for i in range(len(self.data_labels)):
                label = i
                if len(self.data_info[i]) < counter+1:
                    empty=True
                    continue
                empty=False

                wav = parse_wave_python(self.data_info[i][counter])

                floatwav = wav*(1./32767); # convert 16bit integer to float between -1 and 1
                files.append(floatwav)
                labels.append(label)
            counter+=1

            if empty:
                break
```

```
            # if the iterator is multiple of batch size return the mini batch
            if (counter)%each_batch_size == 0:
                yield np.array(files,dtype=np.float32), np.array(labels,dtype=np.int32)
                del files
                del labels
                files=[]
                labels=[]
                break
```

7.7 Code CNN Speech-Spectrograms

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf
from mydata import DataSetGenerator
import cv2

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

def cnn_model_fn(features, labels, mode):
    """Model function for CNN."""
    # Input Layer
    # Reshape X to 4-D tensor: [batch_size, width, height, channels]
    # MNIST images are 28x28 pixels, and have one color channel
    input_layer = tf.reshape(features["x"], [-1, 128, 128, 1])

    # Convolutional Layer #1
    # Computes 32 features using a 5x5 filter with ReLU activation.
    # Padding is added to preserve width and height.
    # Input Tensor Shape: [batch_size, 128, 128, 1]
```

```
    # Output Tensor Shape: [batch_size, 128, 128, 32]
    conv1 = tf.layers.conv2d(
            inputs=input_layer,
            filters=32,
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu)
    conv2 = tf.layers.conv2d(
            inputs=conv1,
            filters=64,
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu)

    # Pooling Layer #1
    # First max pooling layer with a 2x2 filter and stride of 2
    # Input Tensor Shape: [batch_size, 128, 128, 32]
    # Output Tensor Shape: [batch_size, 64, 64, 32]
    pool1 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

    # Convolutional Layer #2
    # Computes 64 features using a 5x5 filter.
    # Padding is added to preserve width and height.
    # Input Tensor Shape: [batch_size, 64, 64, 32]
    # Output Tensor Shape: [batch_size, 64, 64, 64]
    conv3 = tf.layers.conv2d(
            inputs=pool1,
            filters=32,
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu)
    conv4 = tf.layers.conv2d(
            inputs=conv3,
            filters=128,
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu)

    # Pooling Layer #2
    # Second max pooling layer with a 2x2 filter and stride of 2
    # Input Tensor Shape: [batch_size, 64, 64, 64]
    # Output Tensor Shape: [batch_size, 32, 32, 64]
    pool2 = tf.layers.max_pooling2d(inputs=conv4, pool_size=[2, 2], strides=2)

    conv5 = tf.layers.conv2d(
            inputs=pool2,
            filters=64,
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu)
```

```
conv6 = tf.layers.conv2d(
        inputs=conv5,
        filters=32,
        kernel_size=[5, 5],
        padding="same",
        activation=tf.nn.relu)
pool3 = tf.layers.max_pooling2d(inputs=conv6, pool_size=[2, 2], strides=2)

# Flatten tensor into a batch of vectors
# Input Tensor Shape: [batch_size, 32, 32, 32]
# Output Tensor Shape: [batch_size, 32 * 32 * 32]
pool3_flat = tf.reshape(pool3, [-1, 16 * 16 * 32])

# Dense Layer
# Densely connected layer with 1024 neurons
# Input Tensor Shape: [batch_size, 32 * 32 * 64]
# Output Tensor Shape: [batch_size, 1024]
dense = tf.layers.dense(inputs=pool3_flat, units=1024, activation=tf.nn.relu)

# Add dropout operation; 0.4 probability that a neuron is blocked to prevent overfitting
dropout = tf.layers.dropout(
        inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

# Logits layer
# Input Tensor Shape: [batch_size, 1024]
# Output Tensor Shape: [batch_size, 2]
logits = tf.layers.dense(inputs=dropout, units=2)

predictions = {
        # Generate predictions (for PREDICT and EVAL mode)
        "classes": tf.argmax(input=logits, axis=1),
        # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
        # `logging_hook`.
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
    train_op = optimizer.minimize(
            loss=loss,
            global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
```

```
    eval_metric_ops = {
            "accuracy": tf.metrics.accuracy(
                    labels=labels, predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(
            mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):

    # Load training and eval data
    dg = DataSetGenerator(".\Data")
    #train_data, train_labels = dg.get_mini_batches(10000,(128,128), allchannel=False)
    #eval_data, eval_labels = dg.get_mini_batches(1000,(128,128), allchannel=False)
    print("Reading images...")
    train = dg.get_mini_batches(4000,(128,128), allchannel=False)
    eval = dg.get_mini_batches(250,(128,128), allchannel=False)
    for data, labels in train:
        train_data, train_labels = data,labels
    for data, labels in eval:
        eval_data, eval_labels = data,labels
    print("Got ",len(train_data),"/",len(train_labels)," images for training and ",
                len(eval_data),"/",len(eval_labels)," for evaluation")
    # cv2.imshow('image',train_data[0])
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()
    print("Done. Beginning training...")

    # Create the Estimator
    classifier = tf.estimator.Estimator(
            model_fn=cnn_model_fn, model_dir="/tmp/convnet_model")

    # Set up logging for predictions
    # Log the values in the "Softmax" tensor with label "probabilities"
    tensors_to_log = {"probabilities": "softmax_tensor"}
    logging_hook = tf.train.LoggingTensorHook(
            tensors=tensors_to_log, every_n_iter=100)

    begin_time = time.clock()

    # Train the model
    train_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": train_data},
            y=train_labels,
            batch_size=10,
            num_epochs=None,
            shuffle=True)
    classifier.train(
            input_fn=train_input_fn,
            steps=2000,
            hooks=[logging_hook])
```

```
    end_time = time.clock()

    # Evaluate the model and print results
    eval_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": eval_data},
            y=eval_labels,
            batch_size=25,
            num_epochs=10,
            shuffle=False)
    eval_results = classifier.evaluate(input_fn=eval_input_fn)
    print(eval_results)
    print("Training took ", end_time - begin_time," seconds")


if __name__ == "__main__":
    tf.app.run()
```

7.8 Code Datenschnittstelle Speech-Spectrograms

```
# Based on https://thecodacus.com/prepare-data-set-train-tensorflow-model/


import cv2 # to load the images
import numpy as np # to do matrix manipulations
from os.path import isfile, join # to manipulate file paths
from os import listdir # get list of all the files in a directory
from random import shuffle # shuffle the data (file paths)

import time

class DataSetGenerator:
    def __init__(self, data_dir):
        self.data_dir = data_dir
        self.data_labels = self.get_data_labels() # 1D-List
        self.data_info = self.get_data_paths() # 2D-List, first dimension is
                                               #the corresponding label's index

    def get_data_labels(self):
        data_labels = []
        for filename in listdir(self.data_dir):
            if not isfile(join(self.data_dir, filename)): # is folder
                data_labels.append(filename)
        return data_labels

    def get_data_paths(self):
        data_paths = []
```

```
        for label in self.data_labels:
            img_lists=[]
            path = join(self.data_dir, label)
            for filename in listdir(path):
                tokens = filename.split('.')
                if tokens[-1] == 'png':
                    image_path=join(path, filename)
                    img_lists.append(image_path)
            shuffle(img_lists)
            data_paths.append(img_lists)
        return data_paths

    def get_mini_batches(self, batch_size=100, image_size=(200, 200), allchannel=True):
        images = []
        labels = []
        empty=False
        counter=0
        each_batch_size=int(batch_size/len(self.data_labels))
        while True:
            for i in range(len(self.data_labels)):
                label = i
                if len(self.data_info[i]) < counter+1:
                    empty=True
                    continue
                empty=False

                img = cv2.imread(self.data_info[i][counter])

                img = self.resizeAndPad(img, image_size)
                if not allchannel:
                    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                    img = np.reshape(img, (img.shape[0], img.shape[1], 1))
                floatimg = img*(1./255);
                images.append(floatimg) # the image
                labels.append(label) # 1 if [yes, no] and is no, 0 if is yes
            counter+=1

            if empty:
                break
            # if the iterator is multiple of batch size return the mini batch
            if (counter)%each_batch_size == 0:
                yield np.array(images,dtype=np.float32), np.array(labels,dtype=np.int32)
                del images
                del labels
                images=[]
                labels=[]
                break

    def resizeAndPad(self, img, size): # brings the image to specified size
        h, w = img.shape[:2]
```

```
    sh, sw = size
    # interpolation method
    if h > sh or w > sw:  # shrinking image
        interp = cv2.INTER_AREA
    else: # stretching image
        interp = cv2.INTER_CUBIC

    # aspect ratio of image
    aspect = w/h

    # padding
    if aspect > 1: # horizontal image
        new_shape = list(img.shape)
        new_shape[0] = w
        new_shape[1] = w
        new_shape = tuple(new_shape)
        new_img=np.zeros(new_shape, dtype=np.uint8)
        h_offset=int((w-h)/2)
        new_img[h_offset:h_offset+h, :, :] = img.copy()

    elif aspect < 1: # vertical image
        new_shape = list(img.shape)
        new_shape[0] = h
        new_shape[1] = h
        new_shape = tuple(new_shape)
        new_img = np.zeros(new_shape,dtype=np.uint8)
        w_offset = int((h-w) / 2)
        new_img[:, w_offset:w_offset + w, :] = img.copy()
    else:
        new_img = img.copy()
    # scale and pad
    scaled_img = cv2.resize(new_img, size, interpolation=interp)
    return scaled_img
```

7.9 Code CNN Lernraten-Test-Automation

```
import shutil
import subprocess


subprocess.call("cnn_mnist_timer-0.0005.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.001.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")
```

```
subprocess.call("cnn_mnist_timer-0.005.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.01.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.05.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.1.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.2.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.45.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-0.5.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")

subprocess.call("cnn_mnist_timer-1.0.py", shell=True)
shutil.rmtree("/tmp/mnist_convnet_model")
```

7.10 Codebeispiel CNN Ergebnisausgabe in Datei

```
f = open("output_cnn-0.001.txt","w")
sys.stdout = f
print("learning_rate_=_0.001")
print(eval_results)
print("Training_took_", end_time - begin_time,"_seconds")
sys.stdout = sys.__stdout__
```

7.11 Code Datenschnittstelle Aktienkurse

```
import numpy as np # to do matrix manipulations and load csv
from random import shuffle # shuffle the data (file paths)

SAMPLE_COUNT = 100
```

```
class DataSetGenerator:
    def __init__(self, csv):
        self.csv = np.loadtxt(open(csv, "rb"), delimiter=",", skiprows=1, usecols=(1))

    def get_mini_batches(self, batch_size=100):
        data = []
        labels = []
        empty=False
        c=0 # we get SAMPLE_COUNT data points +1 as label
        pos = np.arange(SAMPLE_COUNT+1,len(self.csv))
        shuffle(pos)
        while True:
            datapoint = []
            label = self.csv[pos[c]]

            if len(pos) < c+1:
                empty=True
                continue
            empty=False

            for i in range(1,SAMPLE_COUNT+1):
                datapoint.append(self.csv[pos[c]-i])

            data.append(datapoint)
            labels.append(label)
            c+=1

            if empty:
                break
            # if the iterator is multiple of batch size return the mini batch
            if (c)%batch_size == 0:
                yield np.array(data,dtype=np.float32), np.array(labels,dtype=np.float32)
                del data
                del labels
                data=[]
                labels=[]
                break
```

7.12 Code MLP Aktienkurse

```
#    Based on Convolutional Neural Network Estimator for MNIST, built with tf.layers.
#    which has Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
```

```
#     http://www.apache.org/licenses/LICENSE-2.0

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf
from mydata import DataSetGenerator

import time

tf.logging.set_verbosity(tf.logging.WARN)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

def mlp_model_fn(features, labels, mode):
  """Model function for MLP."""
  # Input Layer
  # Reshape X to flat tensor: [batch_size, size]
  input_layer = tf.reshape(features["x"], [-1, 100])
  labels = tf.reshape(labels, [-1, 1])

  # Hidden Layer #1
  # 16384 Neurons
  hidd1 = tf.layers.dense(
      inputs=input_layer,
      units=500,
      activation=tf.nn.relu)

  # Hidden Layer #2
  # 4096 Neurons
  hidd2 = tf.layers.dense(
      inputs=hidd1,
      units=200,
      activation=tf.nn.relu)

  # Hidden Layer #3
  # 1024 Neurons
  hidd3 = tf.layers.dense(
      inputs=hidd2,
      units=50,
      activation=tf.nn.relu)

  # Logits layer
  # Input Tensor Shape: [batch_size, 1024]
  # Output Tensor Shape: [batch_size, 10]
  logits = tf.layers.dense(inputs=hidd3, units=1)
```

```
    if mode == tf.estimator.ModeKeys.PREDICT:
      return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

    # Calculate Loss (for both TRAIN and EVAL modes)
    loss = tf.losses.mean_squared_error(labels=labels, predictions=logits)

    # Configure the Training Op (for TRAIN mode)
    if mode == tf.estimator.ModeKeys.TRAIN:
      optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.0000001)
      train_op = optimizer.minimize(
          loss=loss,
          global_step=tf.train.get_global_step())
      return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)


    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss)


def main(unused_argv):

    # Load training and eval data
    dg = DataSetGenerator("OIL.csv")
    #train_data, train_labels = dg.get_mini_batches(10000,(128,128), allchannel=False)
    #eval_data, eval_labels = dg.get_mini_batches(1000,(128,128), allchannel=False)
    print("Reading test and training files...")
    train = dg.get_mini_batches(2000)
    eval = dg.get_mini_batches(500)
    for data, labels in train:
        train_data, train_labels = data,labels
    for data, labels in eval:
        eval_data, eval_labels = data,labels
    print("Got ",len(train_data),"/",len(train_labels)," data samples for training and ",
                len(eval_data),"/",len(eval_labels)," for evaluation")
    print("Done. Beginning training...")

    # Create the Estimator
    mnist_classifier = tf.estimator.Estimator(
        model_fn=mlp_model_fn, model_dir="/tmp/mnist_mlp_model")

    begin_time = time.clock()

    # Train the model
    train_input_fn = tf.estimator.inputs.numpy_input_fn(
        x={"x": train_data},
        y=train_labels,
        batch_size=1,
        num_epochs=None,
        shuffle=True)
    mnist_classifier.train(
```

```
      input_fn=train_input_fn,
      steps=20000)

  end_time = time.clock()

  # Evaluate the model and print results
  eval_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      #batch_size=25,
      num_epochs=1,
      shuffle=False)
  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
  print(eval_results)
  print("Training took ", end_time - begin_time," seconds")


if __name__ == "__main__":
  tf.app.run()
```

7.13 Tabelle Lernraten

Metadaten			
Szenario	Modell	Aufbau	Lernrate
MNIST	CNN	Pooling: 2x2, Stride 2 ConvFilter: 32*5x5, 64*5x5 Hidden Neurons: 1024 ReLU GradientDescentOptimizer Dropout: 0.4 Schritte: 2000	0,0005 0,001 0,005 0,01 0,05 0,1 0,2 0,45 0,5 1
	MLP	Hidden Neurons: 16384, 4096, 1024 ReLU GradientDescentOptimizer Dropout: LastLayer: 0.4 Schritte: 2000	0,0005 0,001 0,005 0,01 0,05 0,1 0,2 0,45 0,5 1 2

	Genauigkeit			
Lernrate	1	2	3	Durchschnitt
0,0005	95,10%	95,18%	95,22%	95,17%
0,001	97,03%	97,13%	96,88%	97,01%
0,005	98,72%	98,71%	98,57%	98,67%
0,01	99,07%	99,08%	98,93%	99,03%
0,05	99,15%	99,16%	99,13%	99,15%
0,1	99,32%	99,30%	99,22%	99,28%
0,2	99,36%	99,29%	99,25%	99,30%
0,45	99,27%	10,28%	99,17%	69,57%
0,5	11,35%	10,28%	11,35%	10,99%
1	10,10%	10,10%	11,35%	10,52%
0,0005	11,35%	11,35%	12,38%	11,69%
0,001	11,69%	17,27%	16,75%	15,24%
0,005	68,59%	68,67%	69,38%	68,88%
0,01	84,42%	84,21%	85,20%	84,61%
0,05	91,80%	91,71%	90,57%	91,36%
0,1	92,26%	91,81%	92,95%	92,34%
0,2	93,69%	93,38%	93,77%	93,61%
0,45	95,20%	95,20%	95,10%	95,17%
0,5	95,22%	95,45%	95,17%	95,28%
1	94,24%	94,83%	95,31%	94,79%
2	10,28%	10,32%	10,09%	10,23%

7.14 Tabelle MNIST

Szenario	Metadaten				Ergebnisse				
	Modell	Trainingsdurchläufe	Konstruktion	Sonstiges	Lauf	CPU	GPU	Zeit (s)	Genauigkeit
MNIST	CNN	20000	CONV POOL CONV POOL FC	Pooling: 2x2, Stride 2 ConvFilter: 5x5: 32, 64 Hidden Neurons: 1024 ReLU GradientDescentOptimizer Dropout: 0.4 Lerning Rate: 0.001	1	36,00%	60,00%	371,884	97,01%
					2	37,00%	61,00%	369,07	97,03%
					3	35,00%	60,00%	369,57	97,03%
					Durchschnitt	36,00%	60,33%	370,175	97,02%
	MLP	20000	3 HL	Hidden Neurons: 16384,4096,1024 Sigmoid GradientDescentOptimizer Learning Rate: 0.005	1	39,00%	87,00%	1408,597	86,95%
					2	39,10%	89,00%	1397,483	80,57%
					3	38,30%	89,67%	1399,49	80,86%
					Durchschnitt	38,80%	88,56%	1401,857	82,79%
	MLP	20000	3 HL	Hidden Neurons: 16384,4096,1024 ReLU GradientDescentOptimizer Learning Rate: 0.005	1	39,00%	88,88%	1390,649	97,20%
					2	38,60%	87,40%	1393,034	97,26%
					3	38,30%	91,00%	1390,751	97,27%
					Durchschnitt	38,63%	89,09%	1391,478	97,24%

7.15 Tabelle Speech

Szenario	Metadaten				Ergebnisse		
	Modell	Trainingsdurchläufe	Konstruktion	Sonstiges	Lauf	Zeit (s)	Genauigkeit
Speech	CNN (spectrograms)	2000	CONV CONV POOL CONV CONV POOL CONV CONV POOL FC	Pooling: 2x2, Stride 2 ConvFilter: 5x5: 32,64,32,128,64,32 Hidden Neurons: 1024 ReLU GradientDescentOptimizer Dropout: 0.4 Lerning Rate: 0.01	1	360,1	99,00%
					2	359	99,00%
					3	358,7	99,20%
					Durchschnitt	359,267	99,07%
	CNN (raw wave)	2000	CONV CONV POOL CONV CONV POOL CONV CONV POOL FC	Pooling: 2, Stride 2 ConvFilter: 5:32,10:64,32, 20:128,64,40:32 Hidden Neurons: 1024 ReLU GradientDescentOptimizer Dropout: 0.4 Lerning Rate: 0.01	1	-	50,00%
					2	-	50,00%
					3	-	65,00%
					Durchschnitt	-	55,00%
	MLP	5000	3 HL	Hidden Neurons: 4096,4096,1024 ReLU GradientDescentOptimizer Learning Rate: 0.05	1	306,29	99,60%
					2	306,34	98,00%
					3	303,98	98,80%
					Durchschnitt	305,537	98,80%

Kapitel 8

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die Facharbeit selbständig und ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe. Außerdem habe ich alle fremden Aussagen als solche markiert.

Lars Erber
Dortmund, den 04.03.2018