Künstliche Intelligenz

Künstliche neuronale Netze und deren praktische Anwendung

Besondere Lernleistung von Daniel Simon Stoll

Fachlehrer: Herr Wehmeyer

1.6.2017

Gliederung

Vorwort

Die Menge an Daten, die erstellt, vervielfältigt und konsumiert werden, wird 2020 bei etwa 40 Zettabytes liegen. Ein Zettabyte sind 10^{21} bytes. Und diese Datenanzahl wird in 20 Jahren als gering angesehen werden, denn schätzungsweise verdoppelt sich der weltweite Datenbestand alle 2 Jahre. Grund dafür sind die bereits zahlreich vorhandenen Datenquellen: Nutzer in sozialen Netzwerken legen ständig Informationen nach, Bilder von Überwachungssystemen, Betriebsdaten aus Unternehmensinformationssystemen, GPS-Daten und Daten über Einkaufsverhaltensweisen von Benutzern des amerikanischen Online-Versandhändlers Amazon.com sind nur wenige Beispiele der gewaltigen Datensätze, welche in digitaler Form in physischen Speichern auf der Welt gelagert werden.

Die Ansammlung solcher außerordentlich großen und detaillierten Informationen über jeden Menschen dieser Welt ist durchaus umstritten. Wo die einen eine Möglichkeit zur Gefahrenabwehr und Aufklärung schwerer Straftaten oder eine Verbesserung der Produktqualität von Industrieunternehmen aufgrund einer vorher nicht vorhandenen Nähe zum Kunden sehen, kritisieren die anderen eine Erosion des Rechtsstaates und einen Abbau von Bürgerrechten. Ohne eine Position in dieser Diskussion zu vertreten, erweitere ich die Dimensionalität der Streitthematik um einen Punkt, der Fragen wie Sicherheit oder Freiheitsrechte in den Schatten stellen könnte. Denn seine Ausmaße und seine Beeinflussung unseres alltäglichen Lebens wären deutlich stärker als der internationale Terrorismus oder ein langsam mächtiger werdender totalitärer Überwachungsstaat. Die Mengen an Daten, die digital erfasst werden, könnten zu einem Schlüssel werden. Ein Schlüssel, der uns die Tür zu einer der größten ungeklärten Frage der Menschheit öffnet: Die **menschliche Intelligenz**.

Doch wie wird Intelligenz definiert und warum bilden riesige Datensätze ein Fundament, um sie zu erforschen und zu verstehen? Mathematische und logische Ansätze und Konzepte zur Darstellung eines künstlichen intelligenten Systems existieren bereits seit Mitte des 20. Jahrhunderts. Allen Newell (1927-1992) und Herbert A. Simon (1916-2001) entwickelten in den 1960er Jahren den General Problem Solver, ein Programm, das mit einfachen Methoden beliebige Probleme lösen sollte. Das Projekt war jedoch nicht erfolgreich und wurde nach 10-jähriger Entwicklung eingestellt. Viele Informatiker und Neurobiologen begründen die Erfolglosigkeit des Ansatzes von Newell und Simon mit der fehlenden Beachtung der Unterteilung der menschlichen Intelligenz in vier Bereiche nach Wolfgang Wahlsten: kognitive Intelligenz, sensomotorische Intelligenz, emotionale Intelligenz und soziale Intelligenz. Ein maschinelles System, welches über die gleiche Intelligenz verfügt, wie der Mensch, muss demnach Lösungen für alle der vier Teilprobleme der künstlichen Intelligenz haben.

Bei der kognitiven Intelligenz ist die Maschine dem Menschen schon lange überlegen. Diese Form der Intelligenz bezeichnet das Aufnehmen und Erlernen von Wissen, das Kombinieren aus diesem Wissen und das Schlussfolgern aus diesem Wissen. Als Beispiel kann hierbei das Schachspielen betrachtet werden und bekanntlich existieren schon lange Verfahren, welche in der Lage sind, die besten menschlichen Schachspieler zu schlagen. In allen anderen Bereichen, wie beispielsweise dem Kombinieren von Sinneseindrücken, das Hineinfühlen in andere Menschen oder das soziale Verständnis einer Gruppe, ist jedoch noch der Mensch der Maschine weit überlegen. Doch natürlich arbeiten Neurowissenschaftler und Informatiker an der künstlichen Nachbildung jener Intelligenzbereiche, bei denen Maschinen derzeit noch unterlegen sind. Besonders im Fokus der derzeitigen Entwicklung im Bereich der künstlichen Intelligenz steht die sensomotorische Intelligenzkomponente des Menschen. Hier wurden in den letzten Jahren großartige Erfolge mit

Netzen aus künstlichen Neuronen gemacht. Diese Netze sind dem natürlichen neuronalen Netz des Menschen nachempfunden und beweisen eindrucksvoll, dass die Lösung für eine künstliche Intelligenz kein mathematisch kompliziertes Modell sein muss, sondern dass einfachere Modelle, welche jedoch eine höhere biologische Plausibilität haben, erfolgreicher sind.

Nun kommen die vorher erwähnten Datensätze ins Spiel. Denn künstliche neuronale Netze sind maschinelle Lernalgorithmen. Maschinelles Lernen heißt, Computer so zu programmieren, dass ein bestimmtes Leistungskriterium anhand von Beispieldaten oder Erfahrungswerten aus der Vergangenheit optimiert wird. Als die Theorie des künstlichen neuronalen Netzes in 1950er Jahren entwickelt wurde, mangelte es einerseits an Optimierungs- und Lernregeln, andererseits waren aber auch zu wenig Datensätze vorhanden, um kommerziell anwendbaren Lernalgorithmen ausreichend Lernmaterial zur Verfügung zu stellen. Mit dem Aufstieg des Internets und der Massenspeicherung von Daten, welche die Benutzerinteraktion mit großen Websites wie Google oder Facebook sammelte, entstanden riesige Datenmengen, auch genannt *Big Data*. Mit der ständigen Weiterentwicklung und Optimierung von neuronalen Netzen und der nun möglichen Anwendung auf riesiges Lernmaterial erlebt die Lernmethode des KNN (= künstliches neuronales Netz) einen Boom. Maschinelle künstliche Mustererkennungssysteme mit übermenschlicher Leistung werden sowohl von Internet-Giganten wie Instagram, Facebook oder Google angewendet, als auch von Automobilherstellern wie Tesla, Mercedes-Benz oder BMW. Gesichtserkennungssysteme, autonome Fahrsysteme, Aktienkursvorhersagen, Klassifizierungsalgorithmen, Echt-Zeit-Übersetzungen und Data Mining sind nur wenige Anwendungsbereiche, in welchen KNNs Durchbrüche erzielt haben. Überall dort, wo schier unermessliche Daten nur mit Hilfe von Rechnern analysiert und Wissen extrahiert werden kann, werden neuronale Netze angewendet. Wir können nur erahnen, was für Anwendungen mit Hilfe des maschinellen Lernens künftig realisierbar sein werden.

Ich habe eine große persönliche Begeisterung für das Arbeitsfeld des maschinellen Lernens. Schade ist es daher, dass ich in dieser besonderen Lernleistung nur Teilgebiete der derzeitigen Entwicklung in der künstlichen Intelligenz behandeln kann. Die Wahl zur Behandlung von KNN und nicht anderer maschineller Lernalgorithmen, wie die *k-means-Clusteranalyse* oder *Reinforcement Learning*, fiel relativ schnell, da neuronale Netze im Vergleich ein noch aufregenderes Arbeitsfeld bieten, als die anderen Teilbereiche. Die theoretische Behandlung wurde zwar nicht langweilig, doch nach mehreren Wochen Beschäftigung mit der Thematik wuchs in mir der Tatendrang, das gelernte Wissen praktisch anzuwenden. Zunächst entwickelte ich ein Gesichtserkennungssystem, dessen Erfolg jedoch aufgrund des unzureichenden Datenerzeugungswillens meiner Familie mangelhaft war. Deshalb suchte ich nach einem Projekt, dessen Trainingsdaten ich selber erstellen konnte und stieß auf das „Self Driving Toy Car“-Projekt von Ryan Zotti, welches er auf der PyData DC 2016 vorstellte. Da ich Vorkenntnisse bei der Benutzung des Einplatinencomputers Raspberry Pi und der Programmierung der Programmbibliothek Tensorflow, welche von Google Brain für maschinelles Lernen im Umfeld von von Sprache und Bildverarbeitungsaufgaben geschaffen wurde, hatte, habe ich auf dieser Grundlage ein selbstfahrendes Spielzeugauto konstruiert.

Ich habe das Projekt daher selbst umgesetzt und werde meine Ergebnisse im Rahmen dieser besonderen Lernleistung im zweiten Teil präsentieren. Zunächst folgt allerdings eine theoretische Einführung rund um das Thema der neuronale Netze.

Ich wünsche Ihnen viel Freude beim Lesen meiner Arbeit!

1. Theoretische Einführung – Neuronale Netze

1.1 Neuronen und Perzeptronen

Das Modell eines künstlichen neuronalen Netzes ist dem Neuronennetz eines menschlichen Gehirns nachempfunden. Jedes einzelne Neuron kann hierbei als Verarbeitungseinheit betrachtet werden, wobei verschiedene Neuronen parallel operieren. Das menschliche Gehirn besteht aus einer sehr großen Anzahl an Neuronen, schätzungsweise 10^{11}. Bei genauerer Betrachtung einer solchen Nervenzelle kann seine Funktionsweise analysiert werden.

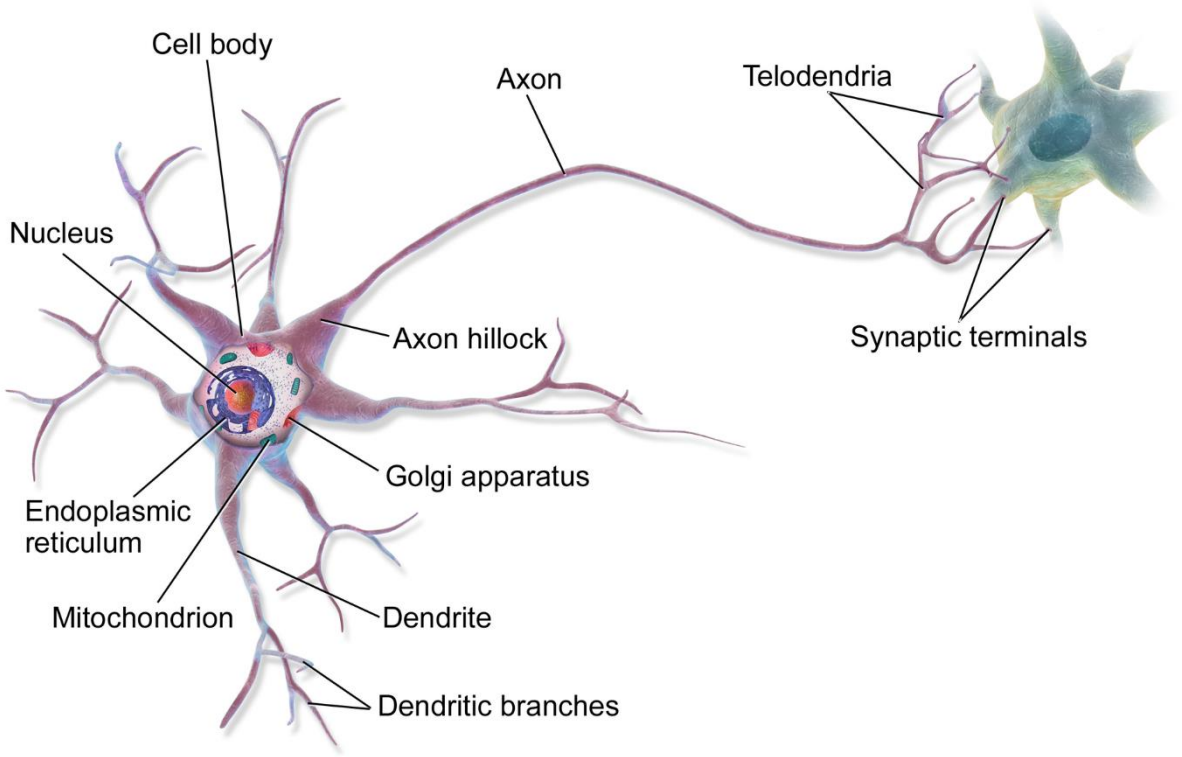

Abbildung 1.1.1: Darstellung einer menschlichen Nervenzelle. Klar zu erkennen sind die Synapsen, welche das Neuron mit weiteren Neuronen vernetzt

Was das Gehirn von derzeitigen Rechenprozessen in Computern unterscheidet ist, dass die einzelnen Nervenzellen deutlich langsamer arbeiten als moderne Prozessoren. Trotzdem ist die gesamte Rechenleistung des Gehirns deutlich größer als die eines jeden Supercomputers. Grund dafür ist die große innere Konnektivität der Neuronen. Nervenzellen im Gehirn sind über sogenannte Synapsen mit etwa 10^4 anderen Neuronen verbunden, die alle parallel arbeiten. Und es besteht ein weiterer gravierender Unterschied zwischen Computer und Gehirn. In einem Computer ist der Prozessor aktiv und die separate Speichereinheit passiv. Beim Gehirn hingegen wird angenommen, dass sowohl die Verarbeitung als auch die Speicherung gemeinsam auf das Netz verteilt werden. Dabei übernimmt die Nervenzelle die Verarbeitung und die Speicherung findet in den Synapsen zwischen den einzelnen Zellen statt.

Bei einer künstlichen Erzeugung eines neuronalen Netzes werden die einzelnen Nervenzellen und ihre Verbindungen untereinander elektronisch simuliert. Dieses künstliche Neuron wird als *Perzeptron* bezeichnet und ist das grundlegende Verarbeitungselement des Netzes. Es nimmt

Eingaben entgegen, die entweder der Umgebung entstammen oder die Ausgabe anderer Perzeptronen sind. Das Perzeptron ist in der Lage, jede Eingabe durch die eingehenden Verbindungen zu verarbeiten und jegliche ausgehende Verbindung mit dem berechnet Wert anzuregen. Die Berechnung der Ausgabe ist dabei denkbar einfach. Jede eingehende Verbindung besitzt ein Verbindungsgewicht oder auch synaptisches Gewicht. Somit kann jeder Eingabe x_i mit $i = 1, \ldots . d$ ein Gewicht w_i zugeordnet werden. Je größer der Absolutbetrag des Gewichtes ist, desto größer ist der Einfluss einer Unit auf eine andere Unit. Ein positives Gewicht zwischen zwei Perzeptronen drückt aus, dass die Ausgabe des ersten künstlichen Neurons einen exzitatorischen Einfluss auf das zweite Perzeptron hat. Ein negatives Gewicht meint, dass das Neuron einen inhibitorischen Einfluss auf folgende Neuronen hat. Ein Gewicht von Null tötet die Synapse. Somit wird kein Einfluss mehr ausgeübt.

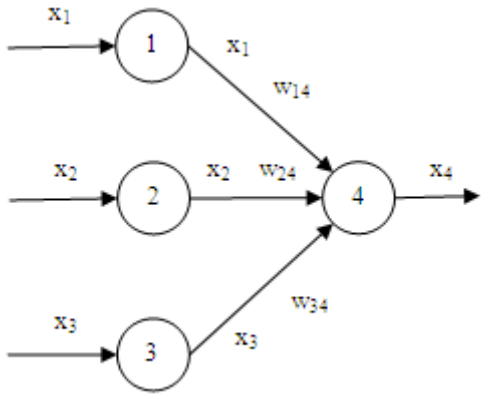

Abbildung 1.1.2: Ein Perzeptron mit Input Units und einer Output Unit

Die Ausgabe ist im einfachsten Falle eine gewichtete Summe der Eingaben:

$$y = \sum_{i=1}^{d} w_i x_i + b$$

Die Konstante b beschreibt hierbei die allgemeine Neigung des Perzeptrons. Ist es positiv, tendiert das Neuron zu einer positiven Ausgabe, ist es negativ, tendiert es zu einer negativen Ausgabe. Die oben genannte Rechnung kann auch als Skalarprodukt ausgedrückt werden:

$$y = \vec{w} \cdot \vec{x} + b$$

Wobei $\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \ldots \\ w_d \end{pmatrix}$ und $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_d \end{pmatrix}$

Wird das Perzeptron in Abbildung 1.2 betrachtet, so fällt schnell auf, dass hiermit keine großartigen Berechnung durchgeführt werden können vollkommen unabhängig davon, wie die Gewicht w_i und die Neigung b gewählt wird. Vielmehr handelt es sich hierbei um eine Darstellung lineare Berechnung einer Ausgabe verschiedener Inputs, welche je nach synaptischen Gewicht unterschiedlich starke Auswirkungen auf den Output haben. Wäre nur eine Input Unit vorhanden, so würde sich der Output durch $y = w_1 * x_1 + b$ berechnen lassen. Hierbei würde es sich demnach um eine adaptive neuronale Darstellung einer linearen Funktion mit dem Ordinatenabstand b und der Änderungsrate

w_1 handeln. Mit mehr als einer Eingabe lässt sich die Ausgabe jedoch nicht mehr zweidimensional darstellen. Es handelt sich dann um eine (Hyper-) Ebene, welche von i Parametern abhängig ist.

1.2 Aktivierungsfunktionen

In den meisten Anwendungsfällen werden Perzeptronen verwendet um Klassifikationen durchzuführen. Um eine Entscheidung, um welche Klasse es sich handelt, treffen zu können, muss der Ausgaberaum in zwei (oder mehr) Teilräume geteilt werden. Eine solche lineare Diskriminanzfunktion ist leicht in ein einlagiges Perzeptron zu implementieren. Jedoch muss die Berechnung der Unit um eine weitere Komponente erweitert werden: Eine Aktivierungsfunktion $\sigma(y)$.

Legen wir fest, dass unsere Aktivierungsfunktion eine binäre Funktion ist, also entweder den Wert 0 oder 1 ausgeben kann, so kann beispielsweise definiert werden:

$$\sigma(y)\begin{cases}1 \text{ falls } y > 0\\ 0 \text{ anderfalls}\end{cases}$$

Somit können wir zwischen zwei Klassen K_1 und K_2 unterscheiden. Beispielweise ist

$$K_1, wenn\ \sigma(y) = 1$$

$$K_2, wenn\ \sigma(y) = 0$$

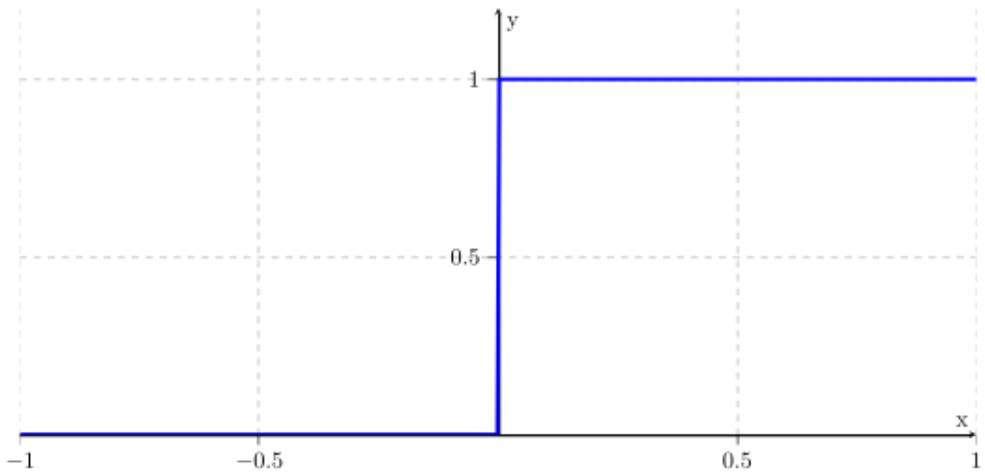

Abbildung 1.2.1: Binäre Aktivierungsfunktion

Nun gibt es folgende Problematik: Wie bereits im Vorwort erwähnt sind die Gewichte w_i nicht definiert und müssen in der Trainingsphase des Perzeptrons ermittelt werden. Das Adaptieren der

einzelnen Verbindungsgewichte geschieht durch eine Fehlerfunktion, welche optimiert wird. Da der Rechenaufwand zu groß wäre, sämtliche Funktionswerte der Fehlerfunktion zu berechnen, wird das Gradientenabstiegsverfahren angewendet. Eine binäre Funktion hat jedoch die Eigenschaft, keine Änderungsratenfunktion zu haben und daher kann auch kein Gradient berechnet werden. Folglich muss eine Aktivierungsfunktion verwendet werden, welche der Funktionalität einer booleschen Funktion ähnelt, jedoch auch eine Änderungsrate besitzt. Derzeit wird hierfür die Sigmoid-Funktion benutzt:

$$sigmoid(y) = \frac{1}{1 + e^{-a\sum_{i=1}^{d} w_i x_i + b}}$$

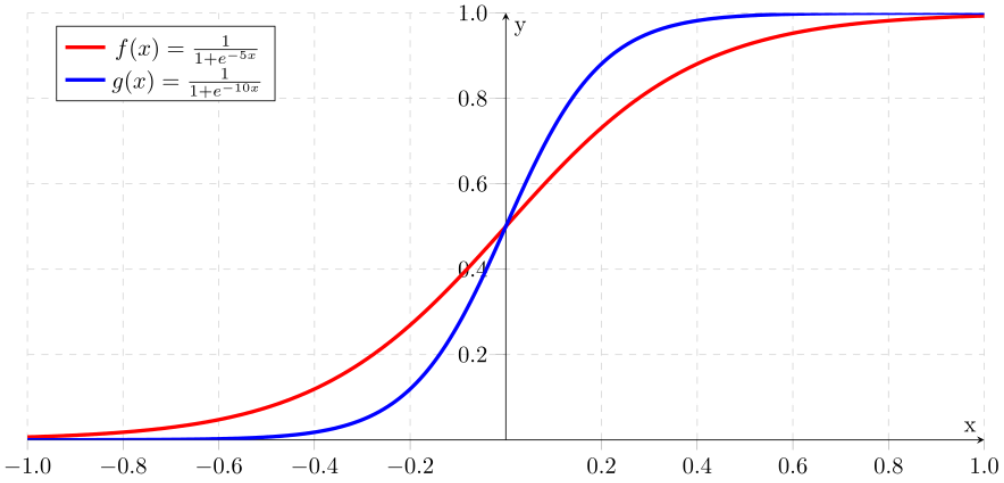

Abbildung 1.2.2: Sigmoid Aktivierungsfunktion mit a = 5 und a=10

1.3 Mehrlagige Perzeptronen

Ein Perzeptron, welches eine einzelne Schicht von Gewichten besitzt, kann nur lineare Funktionen approximieren und ist daher nicht fähig, sich komplexeres Wissen anzueignen. Sogenannte Feedforward-Netze mit Zwischenschichten oder verborgenen Schichten zwischen den Eingabe- und Ausgabeschichten verfügen über die Eigenschaft, nicht-lineare Diskriminanzfunktionen darzustellen und sind daher bei Klassifikationsproblemen deutlich erfolgreicher als einschichtige Perzeptronen.

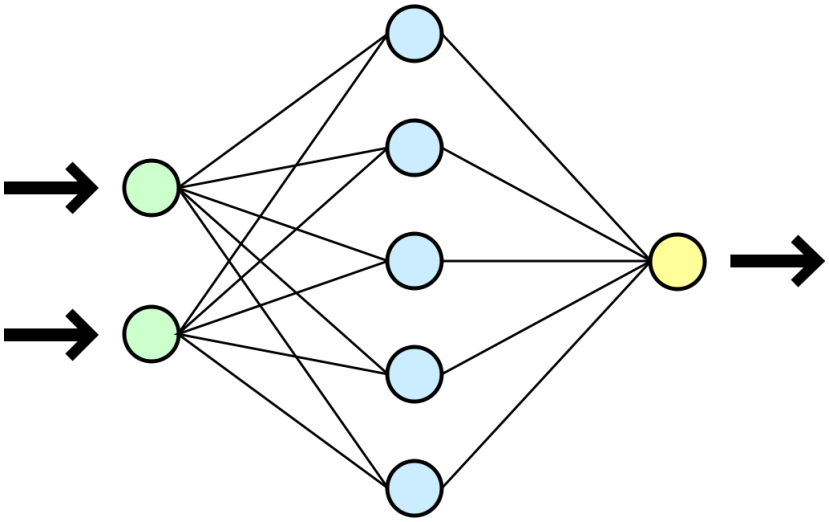

Abbildung 1.3.1: Deep Neuronal Network mit einer Hidden-Layer

Solche Netzwerke werden als *mehrlagiges Perzeptron (MLP)* bezeichnet oder auch *Deep neural network (DNN)*. In einem MLP werden die einzelnen Neuronen in 3 Gruppen unterteilt: Die Input-Units empfangen Reize von der Außenwelt. Sie bilden somit zwangsmäßig immer die erste Schicht eines DNN. Die Hidden-Units befinden sich in Schichten zwischen der Eingabe-Schicht und dem letztendlichen Output. Sie beinhalten eine interne Repräsentation der analysierten Daten und sind in der Lage Wissen darzustellen. Die Output-Units präsentieren das berechnete Ergebnis der Außenwelt. Die Ausgabe $y_{i,l}$ eines i-ten Neurons in der l-ten Schicht berechnet sich aus den kumulierten Ausgaben aller Neuronen aus der vorherigen Schicht:

$$y_{i,l} = \sigma\left(\sum_{i=1}^{d} w_i y_{i,l-1} + b\right) = \frac{1}{1 + e^{-\sum_{i=1}^{d} w_i y_{i,l-1} + b}}$$

Mehrschichtige Perzeptronen sind derzeit die effizientesten und erfolgreichsten Mustererkennungs- und Klassifikationsalgorithmen im Bereich des maschinellen Lernens und erzielen regelmäßig auf internationalen Wettbewerben neue Rekordergebnisse. Bei der vorherig erwähnten Problematik der Zweiklassendiskriminanz bedarf es lediglich einer Ausgabeeinheit, welche durch eine Sigmoid-Funktion aktiviert wird. Wenn jedoch $K > 2$ Klassen existieren, welche im Lernverfahren voneinander unterschieden werden sollen, reicht eine einzelne Output-Unit nicht mehr aus. Eine Lösung des Mehrfachklassenproblems ist die Simulation von $i = K$ Neuronen in der Ausgabeschicht. Jede Klasse wird hierbei einer einzelnen Ausgabeeinheit zugeordnet und die synaptischen Gewichte entsprechend der Datenstruktur angepasst. Das mehrlagige Perzeptron ist nun im Stande, anhand der ihm übergebenen Eingangsreize eine Aussage über die Klasse der übermittelten Daten zu treffen. Normalerweise wird diejenige Klasse als Ausgabeklasse ausgewählt, deren zugehörige Output-Unit am stärksten erregt wurde. Um die höchste Erregung feststellen zu können, müssen jedoch alle Ausgabereize miteinander verglichen werden und bei einer drohenden Gleichbewertung von mehreren Klassen der Empfänger des Klassifizierungssystems vor der möglichen Falschbewertung gewarnt werden. Eine Möglichkeit der Darstellung des Ergebnisses der Berechnungen im neuronalen Netzwerk ist daher die *Softmax Regression*. Hierbei wird eine Wahrscheinlichkeit für jede Klasse in einem Array angeben, welche sich zu einer Summe von 1 addieren. Daher wird in der letzten Output-Schicht die Softmax-Aktivierungsfunktion genutzt, welche die einzelnen Wahrscheinlichkeitswerte wie folgt ausrechnet:

$$P(y = k \mid \vec{x}) = \frac{e^{-\sum_{i=1}^{d} w_{i,k} y_{i,l-1} + b_k}}{\sum_{k=1}^{K} e^{-\sum_{i=1}^{d} w_{i,k} y_{i,l-1} + b_k}}$$

$$Mit\ k \in [1; K]$$

Die Frage, welchen Nutzen eine Sigmoid-Aktivierungsfunktion bei der verborgenen Einheiten hat, ist sicherlich berechtigt. Doch wenn die Ausgabe der verborgenen Einheiten linear wäre, so würden die verborgenen Schichten ihren Zweck nicht erfüllen. Die lineare Kombination von linearen Kombinationen ergibt eine weitere lineare Kombination. Die Sigmoid-Funktion ist die kontinuierliche, differenzierbare Version der binären Aktivierungsfunktion. Wir benötigen Differenzierbarkeit, weil die Lerngleichung, die wir betrachten werden, gradientenbasiert ist. Eine andere sigmoidale nichtlineare Basisfunktion, die verwendet werden kann, ist die hyperbolische Tangentenfunktion, $\tanh(x)$, welche zwischen -1 und +1 liegt, statt zwischen 0 und +1. In der Praxis besteht kein Unterschied darin, ob die Sigmoid-Funktion oder die tanh-Funktion genutzt wird. Jedoch hat sich

herausgestellt, dass eine Optimierung über das Gradientenabstiegsverfahren teilweise bei einer tanh-Aktivierungsfunktion effizienten und leistungssparender ist, als die einer Sigmoid-Funktion.

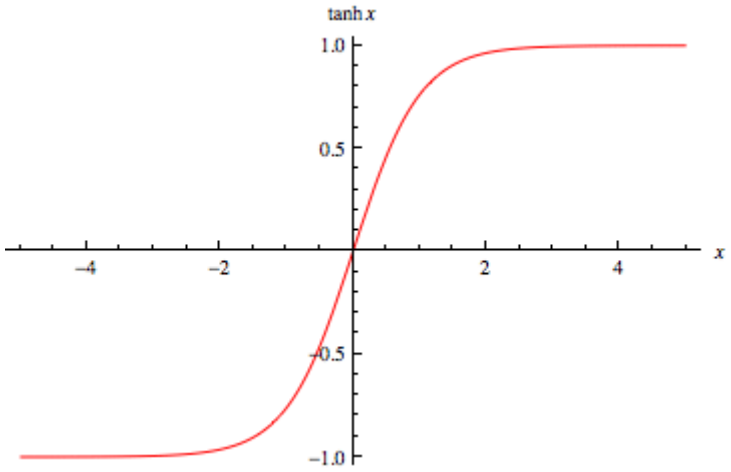

Abbildung 1.3.2: tanh(x)-Aktivierungsfunktion

1.4 Training eines MLP

Während der Trainingsphase eines mehrlagigen Perzeptrons wird üblicherweise der zu lernende Datensatz in eine Trainingsstichprobe und eine Teststichprobe geteilt. Der erste Datenstapel wird nun zu der Anpassung der Verbindungsgewichte verwendet, während der zweite Datenstapel den Erfolg und die Präzision des Netzes ermittelt. Bei Datenmangel kann auch der Trainingsdatensatz gleichzeitig zur Leistungsüberprüfung eingesetzt werden, doch es ist anzuzweifeln, ob die ermittelte Fehlerrate mit der Fehlerrate in einem neuen, vorher nicht gelerntem Umfeld übereinstimmt.

Die Aktualisierung von Verbindungsgewichten wird mit dem *Backpropagation*-Verfahren oder auch *Backpropagation of Error* bzw. auch Fehlerrückführung durchgeführt. Der Backpropagation-Algorithmus wurde bereits 1960 durch die Software-Entwickler Henry J. Kelley und Arthur E. Bryson hergeleitet. Er basiert auf der Annahme, dass mit jeder Instanz die richtige Ausgabe mitgeliefert wird, da er als Grundlage der Gewichtsanpassung den Fehler zwischen dieser Ausgabe und der tatsächlichen Ausgabe des MLP berechnet. Er ist damit ein *überwachtes Lernverfahren*.

Wird angenommen, dass die richtige Ausgabe in Form eines Labels für jede einzelne Instanz gegeben ist und dieses Label jeder einzelnen Output-Unit einen Ausgabewert zuordnen, beispielsweise in Form eines binären Arrays bei der Klassifikation oder einer rationale Zahl für das einzig vorhandene Ausgabeneuron bei der Regression, so lässt sich mithilfe folgender Fehlerfunktion der Fehler für die gesamte Stichprobe berechnen:

$$F(\boldsymbol{W} \mid \boldsymbol{X}, \boldsymbol{r^t}) = \frac{1}{2}\sum_t (r^t - y^t)^2$$

Mit $\boldsymbol{W}$ als Gewichtsmatrix, welche jegliche Verbindungsgewichte des MLP enthält, $\boldsymbol{X}$ als Stichprobenmatrix, welche die Inputreize enthält und r^t als Label, sowie y^t als ermittelter Ausgabewert der Instanz t.

Das Ziel der Trainingsphase ist es nun diese Fehlerfunktion zu minimieren. Da die Funktion jedoch eine Hyperebene mit teilweise mehreren hundert Parametern und Dimensionen hat, ist es selbst mit

hohem rechnerischem Aufwand nicht möglich, jegliche Funktionswerte zu berechnen und dann den Minimalwert zu bestimmen. Die Lösung des Problems ist das Gradientenabstiegverfahren, bei welchem lediglich der Wert des Gradienten bei momentanen gegeben Gewichten berechnet werden muss und daraufhin eine Tendenz festgestellt werden kann, in welcher Richtung der Hyperebene ein lokales oder globales Minimum liegen muss.

Die Veränderung der Gewichte geschieht nun von hinten nach vorne. Zunächst wird die erste verborgene Schicht vor der Ausgabeschicht als einzige Eingabeschicht betrachtet und die Gewichte optimiert. Danach wird die vorherige verborgene Schicht als Output-Schicht betrachtet und die Gewichte zur nächst tieferen Schicht aktualisiert. Deshalb wird das Verfahren Fehlerrückführung genannt. Die Gewichtsänderung für die Hidden-Unit ist dabei so zu bestimmen, als würde es sich lediglich um ein einlagiges Perzeptron mit der verborgenen Schicht als Input-Unit und der Output-Schicht handeln. $\vec{z}$ wird hierbei als Eingangsreize der Hidden-Unit definiert, welche sie aus der Input-Unit erhält, und $\vec{v}$ als Gewichtsvektor, welche die Verbindungsgewichte zur Output-Schicht enthält. Durch das Gradientenabstiegsverfahren ergibt sich folgende Lernregel:

$$F(\boldsymbol{W} \mid \boldsymbol{Z}, \boldsymbol{r^t}) = \frac{1}{2}\sum_t \left(r^t - (\vec{v}^t \vec{z}^t)\right)^2$$

$$\Delta v = -\varepsilon \frac{\partial F}{\partial v} = \varepsilon \sum_t (r^t - y^t)\vec{z}^t$$

Wobei Δv die Gewichtsänderung ist, die durch den negativen Gradienten von der Fehlerfunktion $F(\boldsymbol{W}, \boldsymbol{Z}, \boldsymbol{r^t})$, welche zu einem Minimum leitet, und den Lernparameter ε bestimmt. Der Vorgang kann sich wie folgt vorgestellt werden: Es wird ein Ball in einer hügligen Ebene platziert. Durch den Hang fängt der Ball mit der Geschwindigkeit ε in Richtung $-\frac{\partial F}{\partial v}$ zu rollen. Ist der Ball im Tal angekommen, bleibt er stehen, denn er hat ein lokales Minimum erreicht.

Die Gewichte der ersten Schicht können wir nach demselben Prinzip anpassen. Jedoch ergibt sich eine Problematik. Wenn die verborgene Schicht als Ausgabeeinheit betrachtet wird und daher wieder lediglich ein einlagiges Perzeptron betrachtet wird, sind zwar die Eingangsreize klar, jedoch die gewünschte Reizung der verborgenen Neuronen kann nur indirekt bestimmt werden.

Um diese Problematik zu lösen, nutzen wir für die Gewichte $w_{i,l}$ der ersten Schicht die Kettenregel, um den Gradienten zu berechnen:

$$\frac{\partial F}{\partial w_{i,l}} = \frac{\partial F}{\partial y_i}\frac{\partial y_i}{\partial z_l}\frac{\partial z_l}{\partial w_{i,l}}$$

Nun gilt, wie bereits vorher geklärt:

$$\Delta w_{i,l} = -\varepsilon \frac{\partial F}{\partial w_{i,l}}$$

Nun kommt die Kettenregel ins Spiel:

$$\Delta w_{i,l} = -\varepsilon \sum_t \frac{\partial F^t}{\partial y^t}\frac{\partial y^t}{\partial z_l^t}\frac{\partial z_l^t}{\partial w_{i,l}}$$

$$= -\varepsilon \sum_t -(r^t - y^t)\, v_l\, z_l^t (1 - z_l^t) x_i^t$$

$$= \varepsilon \sum_t (r^t - y^t)\, v_l\, z_l^t (1 - z_l^t) x_i^t$$

Hierbei lässt sich die Fehlerrückführung deutlich erkennen. Das Produkt der ersten beiden Terme $(r^t - y^t)v_l$ berechnet den Fehler für die Ausgabe der Input-Schicht für die verborgene Schicht. Der dritte Term $z_l^t(1 - z_l^t)$ beschreibt die Ableitung der Sigmoid-Funktion und x_i^t ist die Ableitung der gewichteten Summe hinsichtlich des Gewichts $w_{i,l}$.

Um das Verfahren zu optimieren, ist auch die Wahl der anfänglichen Werte für die synaptischen Gewichte wichtig. Die Initialisierung dieser Variablen ist jedoch höchst umstritten. Viele Professoren plädieren für kleine zufällige Werte, beispielsweise im Intervall [-0,01;0,01], um die Sigmoid-Funktion nicht zu sättigen. Andere behaupten, größere zufälligere Werte würden die Lerngeschwindigkeit von MLP-Systemen deutlich erhöhen.

Ebenfalls ist die Wahl des Wertes von dem Lernparameter ε von großer Bedeutung. Ein zu hoher Wert bewirkt eine starke Veränderung, wobei das Minimum verfehlt werden kann, während eine zu kleine Lernrate das Einlernen unnötig verlangsamt.

Ein weit verbreiteter Optimierungsalgorithmus in der Deep Learning-Community ist der *Adam-Optimizer*, welcher auf dem Backpropagation-Algorithmus basiert, ihn jedoch um zwei Funktionen erweitert:

1. Während des Lernprozesses kann es zu einer Oszillation des Netzes d.h. alternierende Verbindungsgewichte kommen. Um dies zu vermeiden, ändert der Algorithmus den Lernparameter ε in Abhängigkeit von den bereits durchgeführten Lernschritten, um eine Stagnation des Lernerfolgs zu verhindern.
2. Die Verwendung eines variablen Trägheitsterms (*Momentum*) $\propto$ ergänzt dem derzeitigen Gradienten eine Beachtung der letzten Änderung. Ist das *Momentum* $\propto$ gleich 0, so ist die Änderung nur vom derzeitigen Gradienten abhängig, ist $\propto$ gleich 1, so wird lediglich der Gradient der letzten Änderung betrachtet. Mathematisch ausgedrückt ergibt sich der Optimierungsgradient:

$$\Delta w_{i,l}(T) = (1 - \propto)\, \varepsilon \sum_t (r^t - y^t)\, v_l\, z_l^t (1 - z_l^t) x_i^t + \propto \Delta w_{i,l}(T-1)$$

Der Umfang des Trainingsdatensatzes mit der Länge t wird als *Batch* bezeichnet. Mit der vorherig gegebenen Lerngleichung berechnen wir die Änderung über alle Muster der Stichprobe, sammeln diese und bringen sie schließlich einmal ein. Einen kompletten Durchlauf über alle Muster in der Trainingsmenge bezeichnet man als *Epoch*.

Eine weitere Lernmethode ist das *online-Lernen*. Hierbei werden die Gewichte nach jedem einzelnen Muster aktualisiert. Um diese Methode zu perfektionieren, werden bei jedem Epoch zufällig Instanzen aus der Stichprobe ausgewählt. Dieser Ansatz wird daher auch stochastischer Gradientenabstieg genannt. Der Lernfaktor ε sollte hierbei jedoch deutlich verringert werden, um die Auswirkung einzelner Muster auf die Gewichtsanpassung zu verringern.

Das folgende Diagramm zeigt beispielhaft den Trainingserfolg eines mehrlagigen Perzeptrons. Klar zu erkennen ist, dass der Wert der Fehlerfunktion sukzessive nach mehreren Tausend *Epochs* abnimmt.

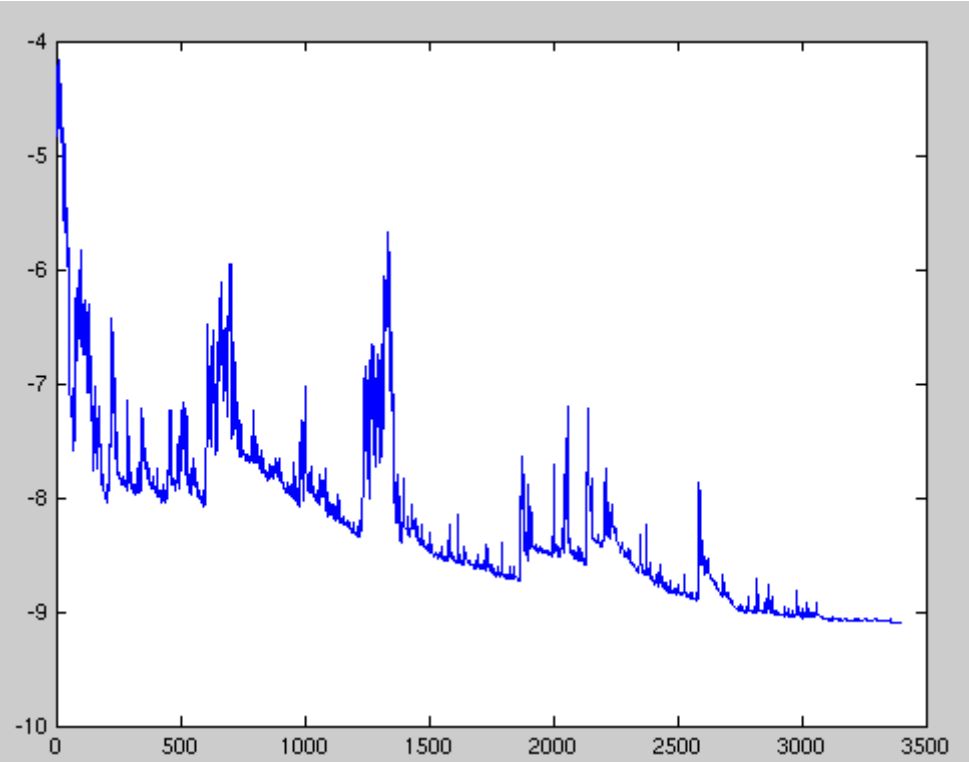

Abbildung 1.4.1: Fehlerquote eines mehrlagigen Perzeptrons in der Trainingsphase

1.5 Convolutional Neural Network

Künstliche neuronale Netze sind der Versuch, menschliche Nervenverbindungen im Gehirn zu simulieren und somit gleiche Ergebnisse zu erreichen. Auch wenn der Ansatz des mehrlagigen Perzeptrons momentan in der emotionalen und sozialen Intelligenz erfolgslos ist, konnten in letzter Zeit neuronale Netze den Menschen in seiner sensomotorischen Intelligenz übertreffen. Ein maschinelles Lernsystem erreichte bereits auf eine der am häufigsten genutzten Bilddatenbanken, MNIST, eine Fehlerquote von 0,23%. Ein Mensch wäre nicht in der Lage, einen annäherungsweise gleichen Klassifikationserfolg aufzuweisen. Der Lernalgorithmus, welcher verwendet wurde, ist eine Sonderform des mehrlagigen Perzeptrons. Die Struktur des Netzes wird *Convolutional Neural Network (CNN)* (= faltendes neuronales Netz) genannt und hat eine hohe biologische Plausibilität.

Das Konzept der Convolutional Neural Networks ist von der Funktionsweise des menschlichen Gehirns inspiriert. Das Gehirn reagiert im primären visuellen Cortex lediglich auf kleine Abschnitte der inneren Augenhaut. Es ist dadurch im Stande, Ecken und Kanten durch Nervenreizung zu identifizieren und in späteren Nervenverbindungen diese Informationen zu Objekten zu kombinieren.

CNNs versuchen diesen biologischen Prozess mithilfe von 3 speziellen Neuronen-Anordnungen zu kopieren.

1.5.1 Convolutional Layer

Normalerweise werden Convolutional Neural Networks in der Bilderkennung eingesetzt. Hierbei wird meistens das zu klassifizierende Bild in ein Graubild umgewandelt. Es ist dann eine zweidimensionale Matrix, welche jedem einzelnen Pixel einen Grauwert zuordnet. Liegt das Bild jedoch im RGB-Format

vor, wird die Input-Matrix dreidimensional, da sie jedem Pixel 3 Werte zuordnet. Die Pixelwerte des Bildes dienen als Eingabereize des Netzwerks. Die Besonderheit einer *Convolutional Layer* besteht darin, dass nicht sämtliche künstliche Neuronen mit den Input-Units verbunden sind. Es werden vorher mehrere Faltungsmatrizen definiert, welche lediglich lokale Konnektivitäten zu den Input-Neuronen aufweisen und durch bestimmte synaptische Gewichten nach bestimmten Mustern innerhalb des Bildes suchen. Wurde beispielweise eine 4x4 Faltungsmatrix definiert, so wird jedem möglichen 4x4-Pixelfeld in dem Bild ein Neuron in der Convolutional Layer zugeordnet. Über die festgelegten Verbindungsgewichte, welche das faltende Neuron mit den jeweils 16 Input-Neuronen verbindet wird nun die Reizung des Neurons für ein bestimmtes Muster in einem bestimmten Bereich des Bildes berechnet und über die Rectified Linear Unit-Funktion (kurz ReLU) aktiviert.

Die ReLU-Funktion ist definiert als $f(x) = \max(0, x)$. Da bei dem Backpropagation-Algorithmus jedoch ein Gradient verlangt wird, muss die differenzierbare Approximation $f(x) = \ln(1 + e^x)$ benutzt werden.

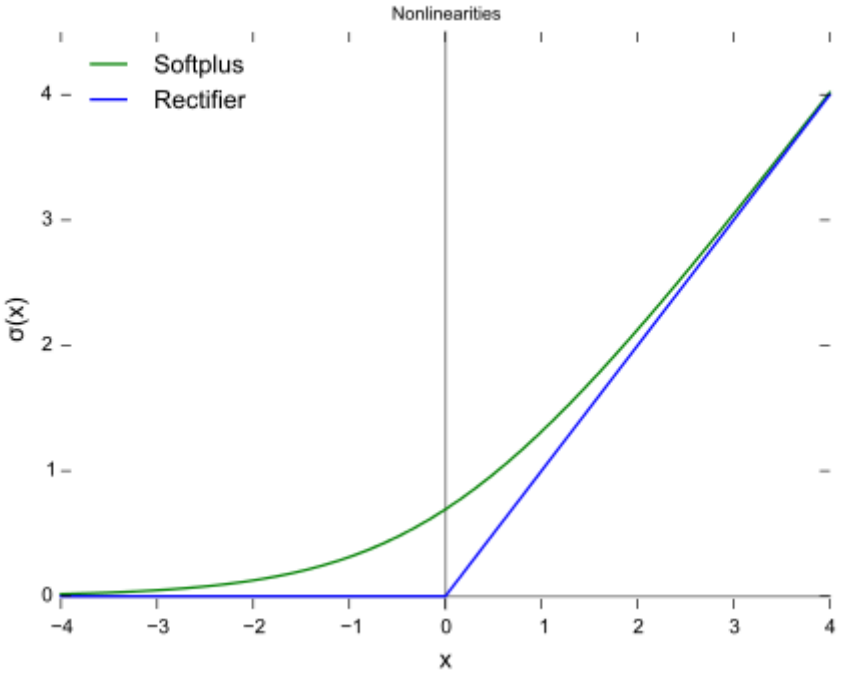

Abbildung 1.5.1.1: Die Aktivierungsfunktion ReLU. Als Softmax wird hierbei die Annäherungsfunktion $f(x) = \ln(1 + e^x)$ bezeichnet.

Es wird die ReLU-Aktivierungsfunktion genutzt, da eine erfolgslose Suche nach einem Muster nicht zu einer hemmenden Wirkung des Neurons führen soll, sondern es lediglich nicht aktivieren soll.

Aufgrund der Funktionsweise des faltendenden Neurons, dass es nur auf eine lokale Umgebung der vorherige Layer reagiert, folgt es dem biologischen Vorbild des rezeptiven Feldes. Der Befürchtung, dass aufgrund der höheren Anzahl an Synapsen ein deutlich höherer Rechenaufwand bei dem Training des Netzwerkes zu erwarten ist, kann entgegnet werden, dass alle faltenden Neuronen, welche dieselbe Faltungsmatrix benutzen, automatisch auch dieselben Gewichte benutzen. Diese Gewichtsteilung wird als *shared weights* bezeichnet und führt automatisch dazu, dass Translationsvarianz eine inhärente Eigenschaft von CNNs ist.

1.5.2 Pooling Layer

Die funktionale Aufgabe der Pooling Layer ist es, unnötige Informationen, welche in der Convolutional Layer gesammelt wurde, zu verwerfen und somit Rechenleistung zu sparen. Zudem erhöhen sie die Abstraktionsebene des generierten Wissens.

Zwar gibt es verschiedene Arten des Poolings, als effizientester Algorithmus hat sich jedoch das Max-Pooling erwiesen. Dabei wird aus einem $n \times m$ –Quadrat aus Neuronen der Convolutional-Neuronen nur der aktivste Wert beibehalten und dem zuständigen Pooling-Neuron als Aktivitätslevel mitgegeben. Auch wenn es hierbei zu einer massivsten Datenreduktion kommt, wird meistens die Performance des Netzwerkes nicht verringert, da die verworfen Daten oft nicht von hoher Bedeutung für die Wissensgenerierung sind.

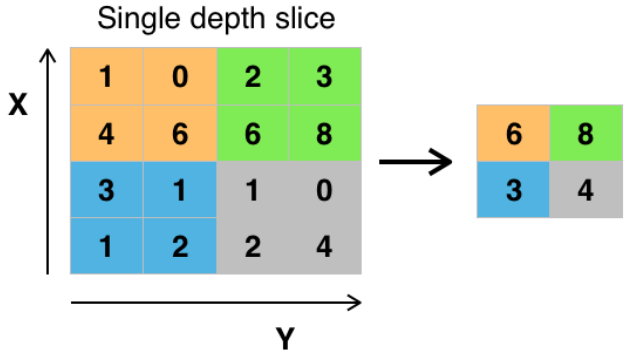

Abbildung 1.5.2.1: Max Pooling mit einem 2x2 Filter und Schrittgröße 2. Die Schrittgröße gibt an wie viele Pixel der Filter pro Operation verschoben wird

1.5.3 Fully Connected Layer

Die Fully Connected Layers bilden üblicherweise die letzten Schichten in der Netzwerkarchitektur eines CNNs. Die Struktur einer Fully Connected Layer entspricht der eines mehrlagigen Perzeptrons und dient dazu, vorher gewonnene Erkenntnisse über erkannte Muster und Objekte in Wissen zu verarbeiten und schließlich bei Klassifzierungs- und Diskriminanzproblemen eine Entscheidung zu fällen. Hier werden auch wieder die vorher gelernten Aktivierungsfunktionen, wie die Sigmoid-Funktion oder die tanh-Funktion eingesetzt. Lediglich in der Output-Layer wird, wie bereits vorher erläutert, die Softmax-Funktion genutzt.

Interessant bei dem Training von CNNs ist, dass drei Faktoren die Performance und die Genauigkeit des Netzes deutlich erhöhen:

1. Pooling - Unnötige Erkenntnisse werden verworfen und können daher keinen Einfluss auf die spätere Wissengenerierung mehr haben.
2. ReLU - Jegliche negative Reizung wird annulliert. Hierbei werden unnötige Rechenoperationen vermieden.
3. Dropout - Durch das Einfügen von einzelnen Dropout-Schichten, welche pro Trainingsschritt zufällig ausgewählte Neuronen entfernen und wahlweise danach wieder einfügen, wird das Netzwerk präziser.

2. Praktische Anwendung – Selbstfahrendes Auto

2.1 Aufbau des Projektes

Bevor ich Ihnen erläutere, wie das selbstfahrende Auto strukturiert und programmiert ist, sollten die Anforderungen, welche ich an die letztendliche Leistung des Autos gestellt habe, definiert werden:

- Der autonome Fahralgorithmus des Autos soll die Autonomiestufe 5 erfüllen. Diese wird von der Bundesanstalt für Straßenwesen wie folgt definiert: „Kein Fahrer erforderlich. Außer dem Festlegen des Ziels und dem Starten des Systems ist kein menschliches Eingreifen erforderlich. “
- Das Fahrzeug soll unabhängig von seiner Fahrumgebung fähig sein, zu lernen sich in neuen Verkehrssituationen zurecht zu finden.
- In späteren Entwicklungsstufen soll das Fahrzeug eigenständig Verkehrszeichen erkennen, deuten und reagieren können.

Das Fundament des Projektes bildet der ferngesteuerte Mercedes-Benz E-Klasse mit Polizeiauto-Lackierung von BUSDUGA GmbH. Prinzipiell kann jedoch jedes andere RC-gesteuerte Spielzeugauto, welches auf dem Markt verfügbar ist, benutzt werden. Wichtig ist dabei nur, dass die interne Elektronik des Wagens entfernt und die Stromkontrolle der beiden Elektromotoren verlötet werden kann. Das genannte Spielzeugauto verfügt über insgesamt 2 Gleichstrommotoren zur Steuerung des Fahrzeuges. Ein Motor ist mit der hinteren Achse verbunden und sorgt lediglich für die Beschleunigung bzw. die Bremsung des Autos. Der andere Motor ist mit der vorderen Achse verbunden und kann diese seitlich verschieben, sodass dem Fahrzeug eine Links-Rechts-Steuerung ermöglicht wird.

Herzstück des selbstfahrenden Autos wird ein Raspberry Pi 3, Modell B, sein. Dieser ist ein winziger Linux-basierter Einplatinencomputer mit einer ARM-CPU, der bei diesem Projekt zur Steuerung der beiden Elektromotoren und zur Simulation eines neuronalen Netzwerks, welches Bildmaterial von einer Frontkamera des Autos auswertet und Bewegungsvorgaben liefert, genutzt wird. Auch wenn der RPi3-B der zurzeit leistungsfähigste Raspberry Pi mit einer 64-Bit-CPU und Taktfrequenz von 1,2 GHz ist und somit deutlich schneller ist als seine Vorgängermodelle, kann er aufgrund seiner geringen Herstellungskosten und seiner Kompaktheit nicht mit modernen Rechnern mithalten. Daher spielen Effizienz und Performance der Programmierung des Mini-PCs eine große Rolle.

Für mein Projekt nutzte ich das Linux-basierte Betriebssystem *Raspbian* für den Raspberry Pi. Raspbian ist die populärste Linux-Distribution für den Raspberry Pi. Das Wortgebilde *Raspbian* setzt sich aus „Raspberry Pi“ und „Debian“ zusammen. Es erweitert eine der ältesten, einflussreichsten und am weitesten verbreiteten GNU/Linux-Distribution *Debian* um einige Funktionalitäten zur Steuerung von speziellen Anschlüssen des RPi3-B, den GPIO-Pins (General Purpose Input/Output). Diverse Raspberry-Pi-Zusatzpakete stehen ausschließlich für Raspbian zur Verfügung bzw. müssen beim Einsatz anderer Distributionen extra kompiliert werden. Für die meisten Raspberry-Pi-Nutzer gibt es daher keinen plausiblen Grund, eine andere Distribution zu verwenden.

Natürlich reicht es nicht aus, elektronische Erweiterungen an den Raspberry Pi anzuschließen. Es bedarf zusätzlich Code um diese Komponenten zu steuern. Zudem muss das Convolutional Neural Network, welches ich benutzen werde, um dem Computer Bewegungsvorschläge mitzuteilen, mit durch ein Skript simuliert werden. In der Raspberry Pi Welt wird eigentlich nur die Programmiersprache *Python* verwendet. Zudem ist die Machine Learning Library *Tensorflow*, welche

ich bei dem Projekt für jegliche maschinellen Lernalgorithmen benutze, zwar in C++ geschrieben, genutzt wird jedoch fast ausschließlich die benutzerfreundliche Python-Schnittstelle der Bibliothek. Folglich sind jegliche Programme in Python geschrieben.

Als einziger Sensor, welcher dem neuronalen Netz Informationen über sein Bewegungsumfeld mitteilt, dient das Raspberry Pi Kamera Modul NoIR V2, das ich am vorderen Ende des Autos befestigt habe. Die Ausgangsreize des CNNs werde dementsprechend insgesamt $128 \times 64 = 8.192$ Pixel sein, da die Kamera mit einer Auflösung von 128 Pixel in der Breite und 64 Pixel in der Höhe Fotos im Graubild-Format aufnimmt. Anhand dieser Bilder wird das Netzwerk in der Lage sein, Bewegungsvorschläge abzugeben, welche dann in ein Steuerung-Signal für die beiden Motoren umgewandelt wird.

Die Projekt-Struktur habe ich in dieser Skizze zusammengefasst:

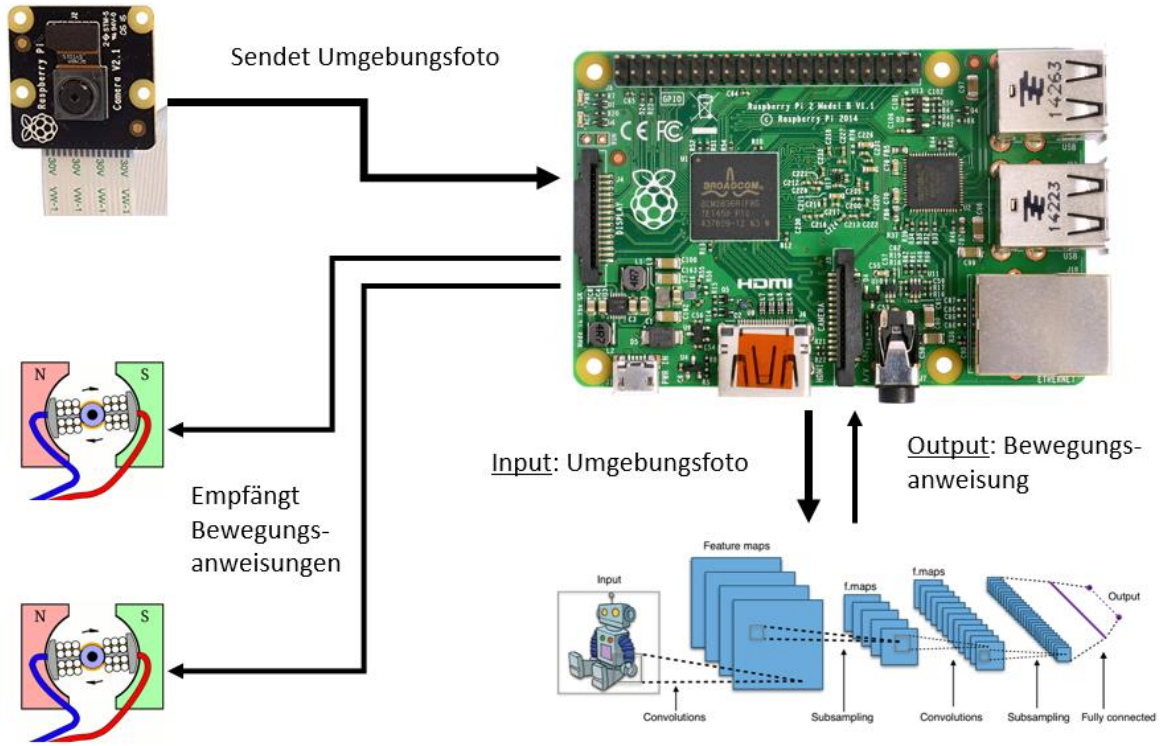

Abbildung 2.1.1: Struktur des Projektes

2.2 Steuerung der Elektromotoren

Gleichstrommotoren sind häufig verwendete Motoren im Modellbau. Durch die Gleichspannungsversorgung kann auf die gefährliche Wechselspannung verzichtet werden, sodass sich die Steuerung deutlich vereinfacht.

Zur Bedienung der beiden Gleichstrommotoren nutze ich den Motortreiber L298. Dieser ist ein dualer Treiberbaustein mit integrierter H-Brücke. Eine H-Brücke ist eine Bezeichnung für eine Schaltung, die es unter anderem ermöglicht, Motoren sehr einfach umzupolen, um die Drehrichtung zu ändern. Somit können wir einerseits den vorderen Elektromotor umpolen, um in Kurven nach links oder rechts zu fahren, sowie den hinteren Elektromotor umpolen, um auch rückwärtsfahren zu

können. Angenehm an dem L298 Bauteil ist, dass dieser gleichzeitig zwei Motoren steuern kann. Somit kristallisiert sich dieses Bauteil als ideal für das Projekt heraus.

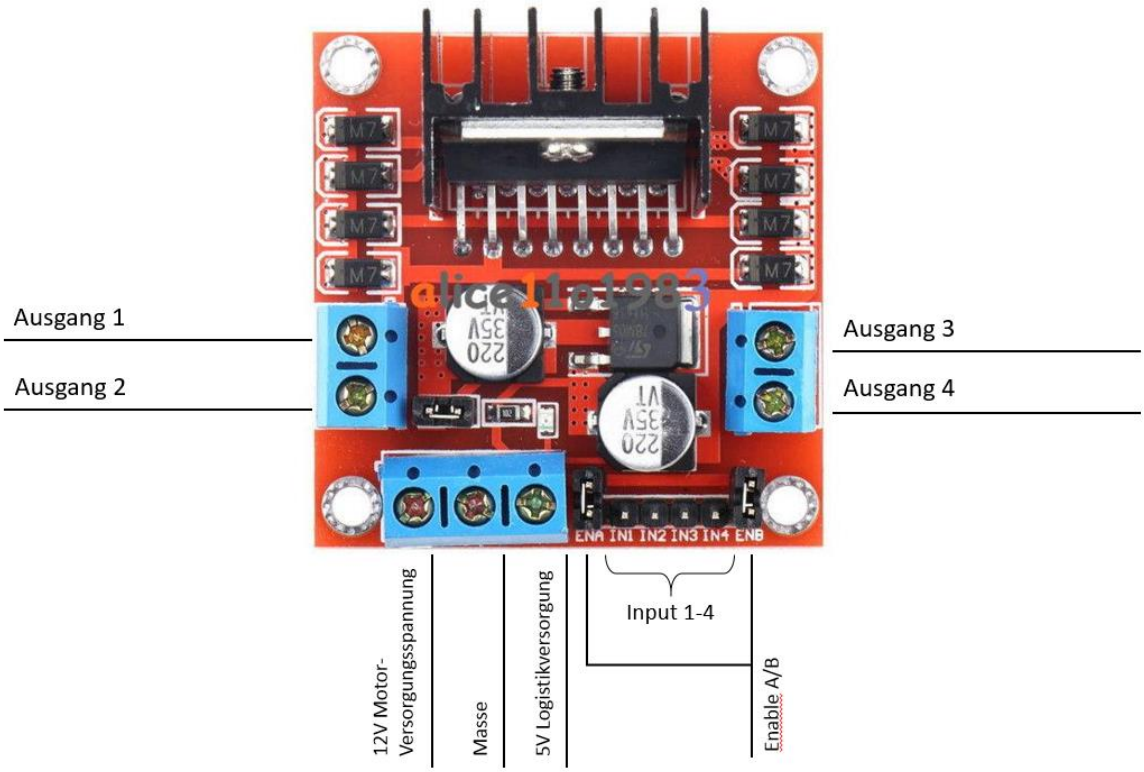

Abbildung 2.2.1: Anschlüsse des Motortreibers L298

Trotz der vielen verfügbaren Pins ist das System leicht zu verstehen:

- Die **Ausgänge:** Hier werden die Versorgungsanschlüsse der Motoren A (1/2) und B (3/4) angeschlossen
- **12V Motor-Versorgungsspannung:** Dieser Pin muss mit der gemeinsamen Versorgungsspannung der Motoren bedient werden. In diesem Projekt benötigten beide Motoren jeweils eine Gleichspannung von 6V. Daher habe ich hier eine externe Batterie mit einer 12V Spannung angeschlossen
- **Masse:** Gemeinsame Masseanbindung. Ich habe hierbei die Masse der externen Stromversorgung genutzt
- **5V Logistikversorgung:** Logistikversorgung des Bauteils. Hier wird eine Spannung von 5V angeschlossen, um den internen Logikteil des L298 zu versorgen.
- **Input 1-4:** Die Input-Pins dienen zur Ansteuerung des Bausteins. Dabei sind die Input Pins 1/2 und **Enable-Pin A** Motor A zugehörig. Folglich gehören die Input Pins 3/4 und **Enable-Pin B** zu Motor B. Jeder Input-Pin kann mit einem High-Pegel von 2,3V aktiviert werden, gleiches gilt für die Enable-Pins. Die Auswirkungen einer Aktivierung der einzelnen Pins sind in dieser Tabelle beispielhaft für Motor A dargestellt:

Input 1	Input 2	Enable A	Funktion
0/1	0/1	0	Motor läuft aus
0	0	1	Sofortiger Stopp
1	0	1	Vorwärts
0	1	1	Rückwärts
1	1	1	Sofortiger Stopp

Die eigentliche Besonderheit des Raspberry Pi ist weder seine winzige Größe noch sein Preis. Die riesige Faszination des Raspberry Pi geht vielmehr von den 26 GPIO (General Purpose Input/Output) aus, die zur Messung und Steuerung elektronischer Geräte verwendet werden können.

Die Steckerleiste, auf welcher sich die Pins befinden, wird J8-Header genannt und enthält 2×13 Kontakte, deren Rasterabstand 2,54mm beträgt. Auf dieser Steckerleiste befinden sich neben den allgemein verwendbaren Kontakten auch zwei Versorgungsspannungen (3,3V und 5V), sowie die Masse (0V).

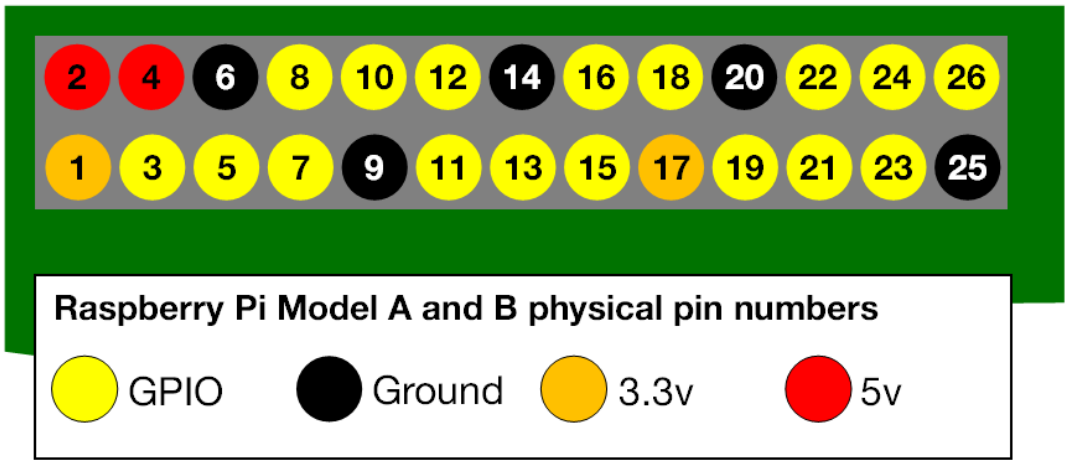

Abbildung 2.2.2: Pin-Belegung des J8-Headers des RPi-3B

Die GPIO-Pins können bei der Interaktion mit Peripheriegeräten zweierlei wirken: Entweder sie dienen als Input und übertragen Signale über eine Spannungsänderung der Verbindung, oder sie fungieren als Informationsempfänger und werten Signale aus, welche sie in Form von Spannungsänderungen über ihre Verbindung erhalten.

Einige Pins erfüllen alternative Funktionen. Zum Beispiel kann Pin 7 vom 1-Wire-Kerneltreiber verwendet oder als Taktgeber eingesetzt werden. Pins 11, 12 und 13 können zum Anschluss von SDI-Komponenten genutzt werden. Da diese Funktionalitäten in dem Projekt nicht genutzt werden, ist eine weitere Erklärung unnötig.

Um nun die Eingänge am L298-Modul zu steuern, habe ich die GPIO Pins 11,13,15 und 16 mit den Input Pins verbunden, sowie die GPIO Ausgänge 7 und 8 mit den beiden Enable-Pins. Zusätzlich versorgt ein 5V-Pin des Raspberry Pi die interne Logik des Moduls.

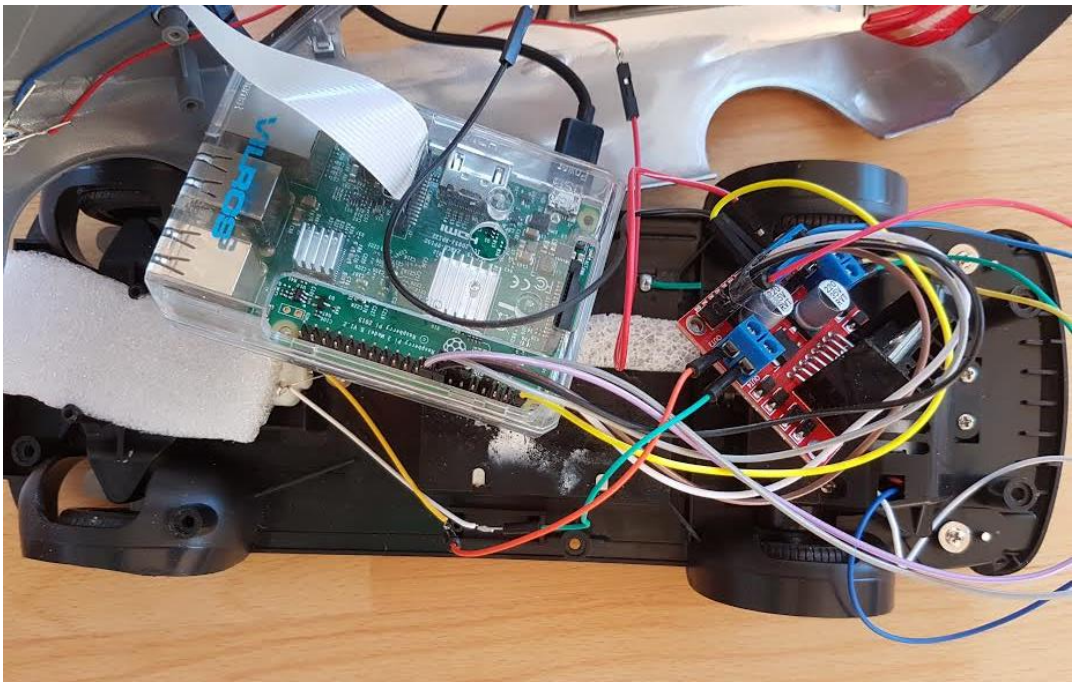

Abbildung 2.2.3: Vollständige Verkabelung des RPi-3B mit dem L298-Modul

Nun bedarf es noch eines Python-Skripts zur Steuerung der L298 H-Brücke. Jedoch wurde bisher noch keine Lösung für die Regulierung der Motorleistung gefunden. Momentan ist es dem Auto nur möglich, entweder vollständig vorwärts oder rückwärts bzw. rechts oder links zu fahren. Um diese Problematik zu lösen, nutze ich bei der Programmierung die Pulsweitenmodulation (= PWM). Hierbei wird die Spannung schnell gepulst. Zwar bleibt der Spannungspegel in den High-Phasen immer gleich, jedoch sorgt das PWM-Signal dafür, dass das ständige An- und Ausschalten des Motors zu einer Drosselung der Leistung führt. Somit ist eine Regulierung des Motors möglich. Im Folgenden finden Sie das Skript, welches zur Steuerung des Motors im Projekt immer wieder genutzt wird:

```
import RPi.GPIO as GPIO


class Motor:

    def __init__(self, pinForward, pinBackward, pinControlStraight,pinLeft,
pinRight, pinControlSteering):
        self.pinForward = pinForward
        self.pinBackward = pinBackward
        self.pinControlStraight = pinControlStraight
        self.pinLeft = pinLeft
        self.pinRight = pinRight
        self.pinControlSteering = pinControlSteering
        GPIO.setup(self.pinForward, GPIO.OUT)
        GPIO.setup(self.pinBackward, GPIO.OUT)
        GPIO.setup(self.pinControlStraight, GPIO.OUT)

        GPIO.setup(self.pinLeft, GPIO.OUT)
        GPIO.setup(self.pinRight, GPIO.OUT)
        GPIO.setup(self.pinControlSteering, GPIO.OUT)

        self.pwm_forward = GPIO.PWM(self.pinForward, 100)
        self.pwm_backward = GPIO.PWM(self.pinBackward, 100)
```

```
        self.pwm_forward.start(0)
        self.pwm_backward.start(0)

        self.pwm_left = GPIO.PWM(self.pinLeft, 100)
        self.pwm_right = GPIO.PWM(self.pinRight, 100)
        self.pwm_left.start(0)
        self.pwm_right.start(0)

        GPIO.output(self.pinControlStraight,GPIO.HIGH)
        GPIO.output(self.pinControlSteering,GPIO.HIGH)

    def forward(self, speed):
        self.pwm_right.ChangeDutyCycle(0)
        self.pwm_left.ChangeDutyCycle(0)
        self.pwm_backward.ChangeDutyCycle(0)
        self.pwm_forward.ChangeDutyCycle(speed)

    def forward_left(self, speed_forward, speed_left):
        self.pwm_backward.ChangeDutyCycle(0)
        self.pwm_forward.ChangeDutyCycle(speed_forward)
        self.pwm_right.ChangeDutyCycle(0)
        self.pwm_left.ChangeDutyCycle(speed_left)

    def forward_right(self, speed_forward, speed_right):
        self.pwm_backward.ChangeDutyCycle(0)
        self.pwm_forward.ChangeDutyCycle(speed_forward)
        self.pwm_left.ChangeDutyCycle(0)
        self.pwm_right.ChangeDutyCycle(speed_right)

    def backward(self, speed):
        self.pwm_forward.ChangeDutyCycle(0)
        self.pwm_backward.ChangeDutyCycle(speed)

    def left(self, speed):
        self.pwm_right.ChangeDutyCycle(0)
        self.pwm_left.ChangeDutyCycle(speed)

    def right(self, speed):
        self.pwm_left.ChangeDutyCycle(0)
        self.pwm_right.ChangeDutyCycle(speed)

    def stop(self):
        self.pwm_forward.ChangeDutyCycle(0)
        self.pwm_backward.ChangeDutyCycle(0)
        self.pwm_left.ChangeDutyCycle(0)
        self.pwm_right.ChangeDutyCycle(0)
```

2.3 Generierung von Trainingsdaten

Zwar hat der Raspberry Pi nun über die Motor-Klasse die Möglichkeit das Fahrzeug zu steuern, er verfügt jedoch noch nicht über einen Algorithmus oder eine künstliche Intelligenz, um entscheiden zu können, wie er die Spur hält oder eine Kurve mit den gegebenen Möglichkeiten bewältigt. In dem ersten Teil dieser besonderen Lernleistung sind Konzepte vorgestellt worden, welche über die Fähigkeit verfügen, dem Spielzeugauto eine Art Intelligenz zu geben.

Um die Umgebungsfotos des Autos verarbeiten und mit vorher generiertem Wissen ein Vorschlag für die Steuerung des Fahrzeuges zu machen, simuliert der RPi-3B ein Convolutional Neural Network. Die

Netzwerkarchitektur dieses neuronalen Netzes wird in späteren Kapiteln behandelt. Zunächst muss ein Datensatz erzeugt werden, aus welchem die künstliche Intelligenz Wissen generieren kann. Je nachdem wie erfolgreich der Erzeuger der Daten sich im Straßenverkehr verhält, desto besser ist die KI, denn schließlich lernt das Netzwerk vom Verhalten des Erzeugers in bestimmten Situation.

Die Generierung des Datensatzes geschieht nach einem einfachen Muster. Ein menschlicher Pilot steuert das Fahrzeug durch eine Teststrecke, welche ich mit weißen Klebeband in meinem Zimmer gebaut habe. Die Kamera nimmt währenddessen kontinuierlich Fotos auf und ordnet jedem aufgenommenen Foto die Bewegung, die von dem menschlichen Fahrer während der Aufnahme getätigt worden ist, in Form eines Klassenlabels zu.

Da ich selber der Erzeuger war, aber seit Jahren kein ferngesteuertes Fahrzeug mehr gesteuert hatte, musste ich zunächst mein eigenes neuronales Netz wieder trainieren. Nach mehreren gescheiterten Anläufen gelang es mir schließlich, mehrere fast fehlerfreien Fahrten durchzuführen.

Der Python-Code zur Erzeugung der Trainingsdaten ist folgender:

```
import sys, tty, termios
import RPi.GPIO as GPIO
from motor import Motor
from drive import drive
import picamera
import os

GPIO.setmode(GPIO.BOARD)
motor = Motor(11,13,7,15,16,18)
camera = picamera.PiCamera()
camera.resolution = (128, 64)
script_path = os.path.abspath(__file__)
script_dir = os.path.split(script_path)[0]
rel_path1 = 'TrainingData/mydataset.txt'
abs_file_path1 = os.path.join(script_dir, rel_path1)
f = open(abs_file_path1,'a')
index = 0

def getKey():
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    try:
        tty.setraw(sys.stdin.fileno())
        ch = sys.stdin.read(1)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    return ch

try:
    while(True):
        y_label = ''
        e = getKey()
        if e == 'x':
            GPIO.cleanup()
            exit()
        elif e == 'a':
            y_label = '0'
            motor.forward_left(30,100)
        elif e == 'q':
            y_label = '1'
            motor.forward_left(30,35)
```

```
        elif e == 'e':
            y_label = '3'
            motor.forward_right(30,35)
        elif e == 'd':
            y_label = '4'
            motor.forward_right(30,100)
        else:
            y_label = '2'
            motor.forward(30)

        index = str(index)
        rel_path2 =
'/home/daniel/Schreibtisch/VirtualBoxUbuntu/TrainingData/img{}.jpeg
{}\n'.format(index,y_label)
        f.write(rel_path2)
        rel_path3 = 'TrainingData/img{}.jpeg'.format(index)
        abs_file_path3 = os.path.join(script_dir, rel_path3)
        camera.capture(abs_file_path3,format='jpeg')
        index = int(index)
        index += 1
finally:
    f.close()
    camera.close()
    GPIO.cleanup()
```

2.4 Verarbeitung der Daten

Bevor ein neuronales Netz mit den gesammelten Daten und ihren Labels gefüttert werden kann, müssen die Daten präpariert werden. D.h. die Bilder müssen in eine Reihung von 2D-Matrizen umgewandelt werden und die Labels in eine Labelmatrix. Da derzeit jedoch noch jeder Pixel in den Bildern einen RGB-Wert hat können die Bilder nur als 3D-Matrizen dargestellt werden. Die Dimensionalität der Bilder kann glücklicherweise verringert werden, indem sie während der Datenverarbeitung in das Grauformat umgewandelt werden. Somit wird jedem Pixel nur noch ein Wert zugeordnet und es können 2D-Matrizen erstellt werden. Die Datenverarbeitung geschieht über folgendes Python-Skript:

```
from tflearn.data_utils import build_hdf5_dataset
import h5py

dataset_file = '/Desktop/Self-Driving-Car/TrainingData/my_dataset.txt'
build_hdf5_image_dataset(dataset_file, image_shape=(128,64), mode='file',
ouput_path='/Desktop/Self-Driving-
Car/PreparedData/dataset.h5',grayscale=True)
```

Die Trainingsdaten befinden sich nun in einem .h5-Datenformat. HDF5 steht für „Hierarchical Data Format“ und wurde von dem *National Center for Supercomputing Applications* entwickelt. Optimierte Strukturen und Algorithmen erlauben das effiziente Speichern von mehrdimensionalen Datensätzen, ideal für dieses Projekt um die Labelmatrix und die Reihung von 2D-Bildmatrizen für die spätere Fütterung zu speichern.

2.5 Simulation des neuronalen Netzes

Wir sind im Herzstück des selbstfahrenden Autos angekommen. Hier soll ein Convolutional Neural Network über jegliche Bewegungen des Autos anhand von aufgenommen Umgebungsfotos entscheiden.

Für die Simulation des neuronalen Netzes nutze ich *Tensorflow*, eine plattformunabhängige Open-Source-Programmbibliothek für künstliche Intelligenz bzw. maschinelles Lernen im Umfeld von Sprache und Bildverarbeitungsaufgaben. Tensorflow ist momentan Entwicklungsstandart und bildet das infrastrukturelle Fundament der meisten Deep Learning-Projekte. Google selbst benutzt die Bibliothek bei kommerziellen Produkten wie die Google-Spracherkennung, Gmail und Google Fotos.

Um die Übersichtlichkeit des Codes zu erhöhen, nutze ich zudem die Deep learning library *tflearn*, welche auf der Grundlage von Tensorflow programmiert worden ist und auch die gleichen Funktionalitäten bereitstellt, jedoch auf einer höheren Abstraktionsebene.

Wie bereits vorher erwähnt, nutze ich ein CNN für die Erzeugung eines intelligenten Bildverarbeitungsnetzwerks. Das Netz besteht aus insgesamt 9 Schichten:

1. Convolutional Layer (Aktivierungsfunktion: ReLU)
2. Pooling Layer
3. Convolutional Layer (Aktivierungsfunktion: ReLU)
4. Pooling Layer

Diese Schichtenanordnung dient der Erkennung von Mustern und Objekten in dem Grauformat-Bild. Unnötige Daten werden hierbei durch die Pooling-Schichten verworfen. Zudem ist die doppelte Anwendung des max-poolings ein bewährter Schutz gegen das sogenannte *overfitting*. Dieses Phänomen lässt sich am besten durch eine menschliche Analogie erklären. Teilweise lernen Schüler Vokabeln auswendig und können diese in Tests erfolgreich wiedergeben. Eine Anwendung in späteren Kontexten ist manchen Schülern jedoch nicht möglich. Eine Begründung dafür ist, dass die synaptischen Gewichte überangepasst wurden und das neuronale Netz sich nur auf die Trainingsdaten in einem speziellen Anwendungskontext spezialisiert hat. Eine Generalisierung des Wissens hat jedoch nicht stattgefunden und somit ist das Netz in neuen Umgebungen erfolgslos.

Auf die 4 Schichten folgen 3 Fully-connected-Schichten, jedoch unterbrochen durch *drop outs*:

5. Fully Connected Layer (Aktivierungsfunktion: tanh)
6. drop out
7. Fully Connected Layer (Aktivierungsfunktion: tanh)
8. drop out
9. Fully Connected Layer (Aktivierungsfunktion: softmax)

Die Funktionalität und Wirkung von *drop outs* wurden bereits im ersten Teil erläutert. Die Schichten 5 und 7 dienen der Generierung von Wissen aus den vorher identifizierten Objekten und Kanten. Dieses Wissen wird in der letzten Schicht verwendet, um eine Entscheidung über die auszuführende Bewegung zu treffen.

Für das Training des Netzes habe ich den *Adam-Optimizer* gewählt. Die benutzte Fehlerfunktion ist *Categorical Crossentropy*, welche im Grundsatz der bereits erwähnten Fehlerfunktion mit einigen Verbesserungen entspricht.

Nun folgt der Code für die Simulation. Durch die Methode *model.fit()*werden die vorher generierten und präparierten Trainingsdaten dem Netzwerk für die Trainingsphase zur Verfügung gestellt. Die Methode *model.save()* speichert die Verbindungsgewichte des Netzwerks in einer Gewichtsmatrix zur späteren Wiederverwendung.

```
from __future__ import division, print_function, absolute_import

import h5py
import tflearn
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.normalization import local_response_normalization
from tflearn.layers.estimator import regression

h5f =
h5py.File('/home/daniel/Schreibtisch/VirtualBoxUbuntu/PreparedData/dataset.
h5','r')
X = h5f['X']
Y = h5f['Y']
X.reshape([-1, 64, 128, 1])

network = input_data(shape=[None, 64, 128, 1], name='input')
network = conv_2d(network, 32, 3, activation='relu', regularizer='L2')
network = max_pool_2d(network, 2)
network = local_response_normalization(network)
network = conv_2d(network, 64, 3, activation='relu', regularizer='L2')
network = max_pool_2d(network, 2)
network = local_response_normalization(network)
network = fully_connected(network, 128, activation='tanh')
network = dropout(network, 0.8)
network = fully_connected(network, 256, activation='tanh')
network = dropout(network, 0.8)
network = fully_connected(network, 10, activation='softmax')
network = regression(network, optimizer='adam', learning_rate=0.01,
loss='categorical_crossentropy', name ='target')


model = tflearn.DNN(network, tensorboard_verbose=0)
model.fit({'input': X}, {'target': Y}, n_epoch=20,
run_id='convnet_self_driving_car')

model.save('cnnsdc.model')
```

Das Training des Models verlief größtenteils positiv. Die Werte der Fehlerfunktion fielen nach den ersten *Epochs* rapide ab. Zwar trat in den letzten Durchläufen das Phänomen des *overfittings* auf, doch die Auswirkung auf die Performance war derart minimal, dass ich auf eine Optimierung des Netzes verzichtete:

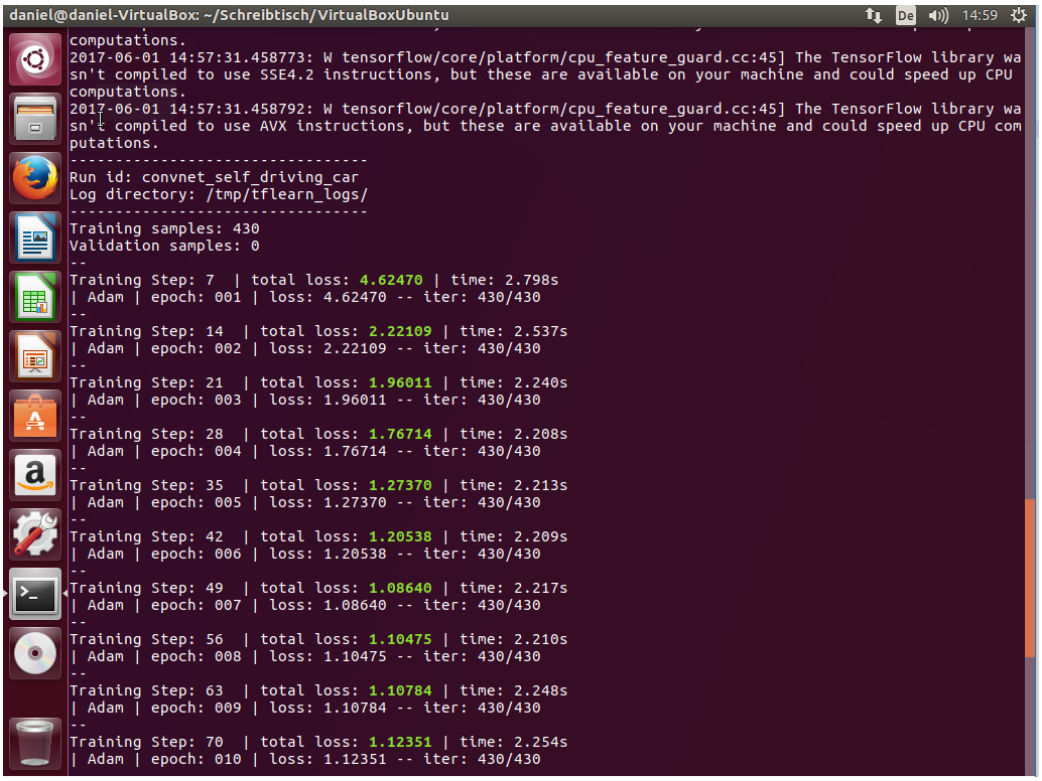

Abbildung 2.5.1: Log-Daten der Trainingsphase des CNNs

2.6 Vollautomatisierung des Fahrzeuges

Grundsätzlich sind nun alle benötigten Komponenten vorhanden:

- Durch die Motor-Klasse des motor.py Skripts lässt sich das Fahrzeug manövrieren.
- Die PiCamera liefert das Bildmaterial, anhand dessen der maschinelle Lernalgorithmus eine Bewegung vorgibt
- Das trainierte CNN ist fähig richtige Bewegungsentscheidung für jedes der ihm gelieferten Umgebungsfotos zu treffen

Ein letztes Python-Programm vereint nun diese 3 Komponenten und lässt somit das Spielzeugauto nun vollautonom Fahren:

```
from __future__ import division, print_function, absolute_import

import tflearn
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.normalization import local_response_normalization
import picamera
import RPi.GPIO as GPIO
from motor import Motor
from PIL import Image
from numpy import array
import os
import operator

GPIO.setmode(GPIO.BOARD)
```

```
motor = Motor(11,13,7,15,16,18)
camera = picamera.PiCamera()
camera.resolution = (128, 64)
script_path = os.path.abspath(__file__)
script_dir = os.path.split(script_path)[0]
rel_path='currentimg.jpeg'
abs_file_path = os.path.join(script_dir, rel_path)

network = input_data(shape=[None, 64, 128], name='input')
network = conv_2d(network, 32, 3, activation='relu', regularizer='L2')
network = max_pool_2d(network, 2)
network = local_response_normalization(network)
network = conv_2d(network, 64, 3, activation='relu', regularizer='L2')
network = max_pool_2d(network, 2)
network = local_response_normalization(network)
network = fully_connected(network, 128, activation='tanh')
network = dropout(network, 0.8)
network = fully_connected(network, 256, activation='tanh')
network = dropout(network, 0.8)
network = fully_connected(network, 5, activation='softmax')

model.load('cnnsdc.model')
try:
    while True:
        camera.capture(abs_file_path, format='jpeg')
        image = np.zeros((128,64))
        img = Image.open(abs_file_path).convert('LA')
        image = array(img).reshape(128,64)

        prediction = model.predict_label(image)
        result = max(prediction.iteritems(), key=operator.itemgetter(1))[0]
        if result=='0':
            motor.forward_left(30,100)
        elif result =='1':
            motor.forward_left(30,35)
        elif result =='2':
            motor.forward(30)
        elif result =='3':
            motor.forward_right(30,35)
        elif result =='4':
            motor.forward_right(30,100)
finally:
    f.close()
    camera.close()
    GPIO.cleanup()
```

Es traten allerdings unglücklicherweise bei den ersten autonomen Testfahrten einige Probleme auf, welche den Erfolg des selbstfahrenden Mercedes-Benz sichtbar senkten:

1. Das neuronale Netz machte als Entscheidungsfinder natürlicherweise in manchen Situationen Fehler. Aufgrund der Tatsache, dass es sich hierbei um eine künstliche Intelligenz handelt, welche der menschlichen Intelligenz nachempfunden ist, ist das System, wie der Mensch, teilweise fehlerhaft in der Einschätzung der Situation. Selbstverständlich sollte dieser Fehlerursprung zügig minimiert werden, da ein Einsatz dieser Technologie im Straßenverkehr auf der Annahme beruht, dass das System besser und intelligenter als der Mensch in vielen Umständen handelt. Sollte es im Gegensatz dazu mehr Unfälle und Todesfälle erzeugen, so wäre die derzeitige künstliche Intelligenz nicht zukunftsfähig.

2. Die Rechenleistung des Raspberry Pi ist für eine schnelle Einschätzung der Verkehrsverhältnisse unzureichend. Bei derzeitiger Performance erreicht der Raspberry Pi eine Rate von 3 Entscheidungen pro Sekunde. In kurvigen Streckenabschnitte reicht diese Anzahl allerdings nicht aus, um das Fahrzeug erfolgreich zu steuern. Eine Lösung dieser Problematik wäre die Auslagerung der internen Berechnungen durch das neuronale Netz. Ein leistungsstärkerer Computer könnte hierbei über einen Webserver und das heimische WLAN mit dem RPi-3B kommunizieren und jedem aufgenommen Bild eine Verhaltensanweisung zuordnen, welche wiederrum dem Mini-Computer im Fahrzeug übermittelt wird.
3. Bei jedem Fehler 1.Art oder 2.Art kam das Testfahrzeug von der markierten Strecke ab. In dem nicht-markierten Bewegungsumfeld fand das Fahrzeug fatalerweise nicht wieder in die Teststrecke zurück. Erst durch einen menschlichen Eingriff konnte das Auto seine autonome Fahrt fortsetzen.

Auch die Verkehrszeichenerkennung war mit dem genutzten CNN leider nicht möglich. In einer späteren Entwicklungsstufe werde ich dem maschinellen Lernalgorithmus einige weitere Schichten hinzufügen, welche der gezielten Objekterkennung von Verkehrszeichen dient und bei der erfolgreichen Identifikation von bestimmten Schildern über spezielle Synapsen Einfluss auf die abschließende Bewegungsentscheidung nehmen kann.

Schlussendlich lässt sich jedoch feststellen, dass das Spielzeugauto zwei der drei Anforderung bereits erfüllt. Das Fahrzeug ist fähig sich in neuen Verkehrssituationen zurecht zu finden, sofern diese sich innerhalb des markierten Teststreckengeländes befinden. Auch wenn bei Fehlern des Systems menschliche Eingriffe nötig sind, so ist anzumerken, dass die Hauptfehlerquelle die geringe Rechenleistung des Raspberry Pi ist. Diese kann mit der genannten Lösungsalternative in der weiteren Entwicklung größtenteils annulliert werden.

Die Arbeit an diesem Projekt hat mich dermaßen begeistert, dass ich vorhabe die Arbeit an dem Auto fortzuführen. Bereits bestehende Algorithmen werde ich optimieren und weitere Lernsysteme für die Erhöhung des Fahrerfolges ergänzen.

3. Fazit / Rückblick

Die meisten von uns nutzen Autos tagtäglich. Manche telefonieren, während sie fahren. Einige schreiben WhatsApp-Nachrichten oder SMS. Das traurige Ergebnis: Mehr als eine Million Menschen sterben jedes Jahr durch Verkehrsunfälle. Entwicklung in der Autonomie von Fahrzeugen in der Automobilbranche machen Hoffnungen auf eine drastische Senkung der gegenwärtigen zahlreichen täglichen Verkehrstoten. Durch moderne autonome Autos wird unser Fahren nicht nur wesentlich sicherer, sie befreien uns auch davon, während unseres täglichen Arbeitsweges auf die Straße achten zu müssen.

Die vorgestellten Theorien und die praktische Anwendung sind der Ansatz für den Traum eines großflächigen intelligenten Verkehrswesens, in welchem jedes öffentliche Verkehrsmittel, Auto und Motorrad selbständig durch die Straßen fährt. Minimierung der Unfälle, Vermeidung von unnötigen Staus durch perfektioniertes Verhalten der einzelnen Verkehrsteilnehmer und somit eine Verringerung der Fahrzeit sind nur einige wenige Vorteile, die das autonome Fahren in Aussicht stellt.

Doch die angewendeten maschinellen Lernalgorithmen sind nicht nur auf selbstfahrenden Autos eingeschränkt anwendbar, sondern universell einsetzfähig. Die Anwendungsbereiche dieser großartigen Technologie der künstlichen Intelligenz sind zahlreich und daher nur beispielhaft aufzählbar. In gefährlichen Umgebungen, wie dem Kernkraftwerken Tschernobyl und Three Mile Island werden Robotor zur Unterstützung von Menschen eingesetzt. Sollten diese mithilfe von neuronalen Netzen in ihrem Handeln und ihrer Bewegung dem Menschen ebenbürtig sein, wäre eine Gefährdung von Menschenleben unnötig und die Robotor könnten sämtliche gefährliche Aufgaben übernehmen. Dieselben intelligenten Robotor könnten Erkundungen von fernen Planeten oder den Bau von Weltraumstationen durchführen. Bei gleicher oder höherer menschlicher Intelligenz würde ihnen jeglicher Schwachstelle des humanen Organismus fehlen. Künstliche Intelligenz könnte bahnbrechende Ergebnisse in der medizinischen Forschung erzielen oder jedem Aktienhändler an der Börse aufgrund speziell trainierten mehrlagigen Perzeptronen überlegen sein. Moderne Verschlüsselungssysteme basieren auf neuronalen Netzes, sofortige semantisch-korrekte elektronische Übersetzer werden bald auf dem Markt kommen und Wissen über jede einzelne Person dieser Welt aus Googles gigantischen Datenbergen wird schon lange in vielschichtigen elektronischen Neuronenverbindungen generiert.

Wie lange wird es dauern bis die emotionale und soziale Intelligenz des Menschen künstlich erzeugbar ist? Feststeht, dass Wissenschaftler der vollständigen Decodierung der Königsklasse des Menschen, seiner Intelligenz, immer näher kommen und ihre Ergebnisse das Leben aller Menschen drastisch verändert wird.

Das Fundament der gesamten Technologie bildet die Erforschung des Gehirns und die Ableitung von neuronalen Strukturen des Menschen. Hieraus resultiert die künstliche Erzeugung vergleichbarer Netzwerke, wie beispielsweise das mehrlagige Perzeptron (MLP).

Wie diese Erkenntnisse praktisch angewendet werden können, habe ich im zweiten Teil dieser Arbeit in Form eines selbstfahrenden Spielzeugautos gezeigt. Die Durchführung und Dokumentation des Projektes hat mir selber unglaublich viel Spaß gebracht und ich hoffe Ihnen hat das Lesen dieser besonderen Lernleistung ebenfalls gefallen.

Daniel Stoll

4. Abbildungsverzeichnis

5. Quellenangabe

Literatur:

Rusell, Stuart & Norvig, Peter; Pearson Deutschland GmbH (2012): *Künstliche Intelligenz – Ein moderner Ansatz*

Ertel, Wolfgang; Springer Vieweg (2016): *Grundkurs Künstliche Intelligenz – Eine praxisorientierte Einführung (Computational Intelligence)*

Eberl, Ulrich, Carl Hanser Verlag GmbH (2016): *Smarte Maschinen – Wie Künstliche Intelligenz unser Leben verändert*

Rashid, Tariq; M.P. Media-Print Informationstechnologie GmbH (2016): *Neuronale Netze selbst programmieren- Ein verständlicher Einstig mit Python*

Alpaydin, Ethem; Oldenbourg Wissenschaftsverlag GmbH (2008) : *Maschinelles Lernen*

Raschka, Sebastian; mitp (2017): *Machine Learning mit Python – Das Praxis-Handbuch für Data Science, Predictive Analytics und Deep Learning*

Weigend, Michael; mitp (2016): *Raspberry Pi programmieren mit Python*

Kofler, Michael & Kühnast, Charly & Scherbeck, Christoph; Rheinwerk Computing (2016): *Raspberry Pi – Das umfassende Handbuch*

Internet:

Künstliche Intelligenz - *https://de.wikipedia.org/wiki/Convolutional_Neural_Network* (Zugriff: 1.6.2017)

Perzeptron - *https://de.wikipedia.org/wiki/Perzeptron* (Zugriff: 1.6.2017)

Künstliches neuronales Netz - *https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz* (Zugriff: 1.6.2017)

Convolutional Neural Network - *https://de.wikipedia.org/wiki/Convolutional_Neural_Network* (Zugriff: 1.6.2017)

Neuronale Netze: Eine Einführung (Rey, Günter Daniel & Beck, Fabian) - *http://www.neuronalesnetz.de/* (Zugriff: 1.6.2017)

Tensorflow: An open-source software library for Machine Intelligence - *https://www.tensorflow.org/* (Zugriff: 1.6.2017)

TFLearn: Deep learning library featuring a higher-level API for TensorFlow - *http://tflearn.org/* (Zugriff: 1.6.2017)