



GRAPHISCHE ERSTELLUNG UND VERARBEITUNG VON KÜNSTLICHEN NEURONALEN NETZEN

Fakultät für Informatik und Mathematik
Goethe Universität Frankfurt

Abschlussarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

vorgelegt von

Pascal Fischer
geboren am 12.06.1995 in Salzgitter

Abgabetermin 29.10.2019
Betreuer: Prof. Dr.-Ing. D. Kroemker

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: _____ Unterschrift: _____

Abstract

In dieser Arbeit wird eine Einführung in die künstliche Intelligenz, speziell in maschinelles Lernen, sowie ein kurzer Einblick in die Bildverarbeitung anhand von bestimmten Methoden gegeben. Im weiteren Verlauf werden neben einer aktuellen Software zur Erstellung und Verarbeitung von künstlichen neuronalen Netzen, zwei Open-Source Bibliotheken zur Verwendung von Verfahren des maschinellen Lernens vorgestellt. Zuletzt wird eine im Rahmen dieser Arbeit entstandene Software präsentiert, mit welcher der Versuch gewagt wurde, basierend auf einer der beiden präsentierten Open-Source Bibliotheken, eine einfachere, benutzerfreundlichere , leicht zu erlernende Software zu entwickeln, welche die Grafische Erstellung und Verarbeitung von künstlichen Neuronalen Netzen ermöglicht.

Inhaltsverzeichnis

Eidesstattliche Erklärung	2
Abstract	3
1 Einleitung	8
2 Grundlagen	10
2.1 Bildverarbeitung	10
2.1.1 Digitale Bilder	10
2.1.2 Faltung	11
2.1.3 Pooling	13
2.2 Maschinelles Lernen	14
2.2.1 Lernmethoden	14
2.3 Künstliche neuronale Netze	17
2.3.1 Einschichtiges Perceptron	17
2.3.2 Mehrschichtiges Perceptron	19
2.3.3 Convolutional Neural Networks	21
2.3.4 Rekurrente neuronale Netze	25
3 Stand der Technik	30
3.1 MemBrain neuronale Netze Editor und Simulator	30
3.1.1 Was ist MemBrain ?	30
3.1.2 Technische Details	30
3.2 TensorFlow	34
3.2.1 Was ist TensorFlow?	34
3.2.2 Grundlegende Funktionsweise	35
3.2.3 Keras	35
3.3 Pytorch	35
3.3.1 Was ist Pytorch ?	35
3.3.2 Grundlegen Funktionsweise	35
4 Methoden	37
4.1 Die Software	37
4.1.1 Aufbau	37
4.1.2 Die Bausteine	38
4.1.3 Technische Details	39
4.2 ein künstliches Neuronales Netz erstellen	40
4.2.1 MNIST-Datensatz	40

4.2.2	Das Netz erstellen	41
5	Ergebnis und Ausblick	52
6	Diskussion	53
6.1	Zusammenfassende Bewertung	53
6.2	Ausblick	53

Abbildungsverzeichnis

2.1	digitale Bilderzeugung	10
2.2	Abtastung	11
2.3	Faltung horizontale Ableitung	12
2.4	2D-Faltung	12
2.5	Max-,Average-Pooling	14
2.6	ReinforcementLearning	16
2.7	BiologischesNeuron	17
2.8	Einschichtiges Perceptron	18
2.9	MehrschichtigesPerceptron	19
2.10	Convolutional Neural Network	22
2.11	Filteraktivierung	23
2.12	Filteraktivierung2	24
2.13	Rekurrentes neuronales Netz	25
2.14	Rekurrentes neuronales Netz Training	26
2.15	LSTM-Zelle	27
3.1	Membrain	31
3.2	Membrain FeedForward NN	32
3.3	Tensoren	36
4.1	Networkx1	37
4.2	Veraenderbare Bausteine	38
4.3	Unveraenderbare Bausteine	39
4.4	MNIST-Datensatz	41
4.5	NeuronalesNetzErstellen1	41
4.6	NeuronalesNetzErstellen2	42
4.7	NeuronalesNetzErstellen3	43
4.8	NeuronalesNetzErstellen4	43
4.9	NeuronalesNetzErstellen5	44
4.10	NeuronalesNetzErstellen6	44
4.11	NeuronalesNetzErstellen7	45
4.12	NeuronalesNetzErstellen8	45
4.13	NeuronalesNetzErstellen9	46
4.14	NeuronalesNetzErstellen10	46
4.15	NeuronalesNetzErstellen11	47
4.16	NeuronalesNetzErstellen12	47
4.17	NeuronalesNetzErstellen13	48

4.18 NeuronalesNetzErstellen14	49
4.19 NeuronalesNetzErstellen15	49
4.20 NeuronalesNetzErstellen16	49
4.21 NeuronalesNetzErstellen17	50
4.22 NeuronalesNetzErstellen18	50
4.23 NeuronalesNetzErstellen19	51
4.24 NeuronalesNetzErstellen20	51
4.25 NeuronalesNetzErstellen21	51

1 Einleitung

Selbstfahrende Autos, Drohnen die Post ausliefern und Staubsaugerroboter sind schon lange kein Science-Fiction mehr. Sogar das in naher Zukunft Roboter entwickelt werden, welche dem Menschen nahezu die gesamte Arbeit abnehmen können, ist nur noch eine Frage der Zeit. Das Einsatzgebiet der *künstlichen Intelligenz*, hauptsächlich des *maschinellen Lernen*, ist nahezu grenzenlos und hat unseren technischen Fortschritt maßgeblich vorangetrieben. Da die künstliche Intelligenz noch in ihren Fußstapfen steht und die weitere Entwicklung unaufhaltsam sein wird, ist es kein Geheimnis, dass sie auch in Zukunft den technischen Fortschritt weiter vorantreiben wird und damit eine große Veränderung für unsere Welt mit sich bringen wird.

Die mächtigsten und heutzutage am meisten verwendeten Verfahren des *maschinellen Lernens* sind sämtliche Arten von *künstlichen neuronalen Netzen*, welche für einen bestimmten Problemtyp erstellt werden und anschließend durch geschicktes Trainieren lernen ein spezielleres Problem zu lösen. Die Entwicklung von künstlichen neuronalen Netzen, war bis vor einigen Jahren noch sehr kompliziert, da alles selbst programmiert werden musste. Inzwischen gibt es Open Source Bibliotheken wie *Tensorflow* von Google und *PyTorch* von Facebook, welche die Entwicklung etwas erleichtern. TensorFlow und PyTorch sind ohne Frage sehr mächtige Open-Source Bibliotheken. Das die Erstellung von künstlichen neuronalen Netzen allerdings immer noch zu kompliziert und Aufwändig ist, zeigt ein Blick in die Arbeit von vielen kleineren Firmen, welche Lösungen mit künstlichen neuronalen Netzen entwickeln. In der Regel werden die neuronalen Netze dort nicht selbst erstellt, sondern fertige meist sogar vortrainierte Netze aus dem Internet oder anderen Quellen verwendet und versucht diese auf das Problem anzupassen. Wer selbst einmal diese Herangehensweise getestet hat, der weiß wie mühselig das aufsetzen dieser künstlichen Neuronalen Netze ist. Die Anpassung der Trainings- bzw. Testdaten für das Training und die Evaluierung der künstlichen neuronalen Netze, was auch einen Großteil der Entwicklung aus macht, wird zudem weder von TensorFlow noch von PyTorch unterstützt. Zugegebenermaßen ist die Automatisierung der Datenverarbeitung auch ein komplexer und schwieriger Vorgang.

Als im Jahre 1992 *OpenGL* (Open Graphics Library, deutsch offene Grafikbibliothek) veröffentlicht wurde, war damit auch das Fundament für die Entwicklung der Grafischen Programmierung gelegt. Allerdings konnte z.B. im Bereich der Videospiele erst durch sogenannte *Game Engines* ein großer Durchbruch erreicht werden. Game Engines wie beispielsweise *Unity* oder *Unreal Engine* bieten eine Grafische Oberfläche welche die Funktionen von OpenGL einfach und intuitiv bedienbar und mit wenig bis gar keinen Programmieraufwand umsetzbar machen. Ein neues System, welches die Funktionen von TensorFlow und PyTorch zur Erzeugung von künstlichen neuronalen Netzen, sowie Funktionen zur Vor- und Nachverarbeitung der Daten welches eine leichte und intuitive Bedienung am besten über eine grafische

Benutzeroberfläche bietet, muss her. Mit so einer Software könnte die Entwicklung von Verfahren mittels künstlicher neuronaler Netze deutlich vereinfacht werden, so wie es die Game Engines mit der Entwicklung von Videospielen getan haben. In dieser Arbeit wird versucht eine Art Baukasten für künstliche Neuronale Netze, welcher basierend auf PyTorch, in Form einer grafischen Benutzeroberfläche, die Möglichkeit bietet ein Neuronales Netz aus einzelnen Bausteinen einfach und übersichtlich grafisch zusammen zu setzen und anschließend ein echtes Neuronales Netz daraus erzeugen kann. Zudem wird der Versuch gewagt ein Verfahren zu entwickeln welches die vor und Nachverarbeitung der Daten automatisiert, sodass die Erstellung und Verarbeitung von künstlichen Neuronalen Netzen ohne Programmcode möglich ist.

2 Grundlagen

2.1 Bildverarbeitung

Die Bildverarbeitung ist ein Teilgebiet der Informatik, welcher sich mit der Verarbeitung von Bildsignalen, wie Fotos, oder Einzelbildern aus Videos, beschäftigt. Die Ergebnisse von Bildverarbeitung können neben Bildern auch Merkmalsextraktionen oder Erkennungen in einem Bild sein. So kann neben klassischen Anwendungen wie Bildverarbeitungssoftwares (z.B Photoshop oder Gimp), welche ausschließlich für die Nachverarbeitung von Bildern in Form von Helligkeitsveränderungen, Farbveränderungen, Schärfungen, Bildzusammenschnitten, Bereinigung von Teilen des Bildes und vielem mehr, auch die Erkennung und Segmentierung von Objekten, Abschätzungen von räumlichen Entfernungen etc. Anwendungen sein.

2.1.1 Digitale Bilder

Digitale Bilder werden in der Regel aus Lichtaufnahmen von Kameras, welche ein kontinuierliches Lichtsignal auf eine Bildebene projizieren, erzeugt. Das Digitale Bild entsteht anschließend durch 2 aufeinander folgende Prozesse: Die *Abtastung* (Sampling) gefolgt von der *Quantisierung*, wie in Abb.2.1 dargestellt. Bei der Abtastung wird das Bild diskretisiert, was bedeutet, dass das kontinuierliche Lichtsignal auf eine endliche Anzahl an Abtastpunkten, sogenannten Pixeln, beschränkt wird. Anschließend werden den Pixeln bei der Quantisierung einem geeigneten Wert aus einem passenden Wertebereich zu geordnet. In der Regel handelt es sich bei einem Pixel für ein Schwarz-Weiß-Bild um einen Wert aus dem Intervall [0,255], was einem Byte (8Bit) entspricht. 0 Bezeichnet hier den Wert Schwarz und 255 den Wert Weiß die Werte dazwischen sind Grauwerte. Das Digitale Schwarz-Weiß Bild liegt also in

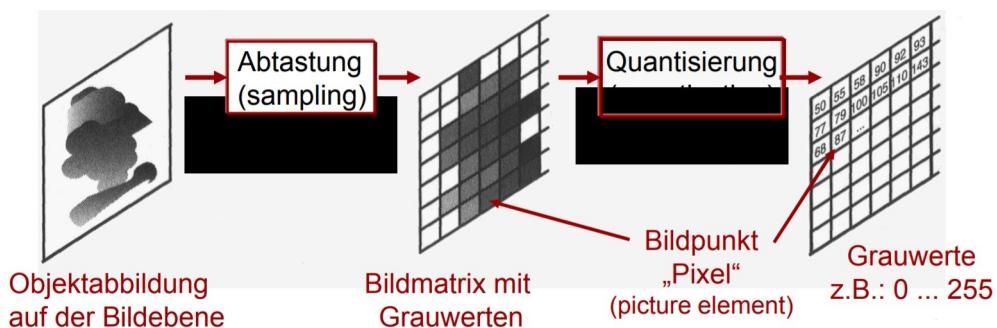


Abbildung 2.1: Erzeugung eines digitalen Grauwertbildes von der Bildebene mittels Abtastung und Quantisierung. [1]



Abbildung 2.2: Verdeutlichung der Qualitätsverbesserung eines Bildes durch Erhöhung der Anzahl an Abtastpunkten. [1]

Form einer 2 Dimensionalen Matrix vor, wobei der Wert (x,y) den Pixel aus Zeile x und Spalte y bezeichnet. Bei Farbbildern verhält es sich ähnlich, hierbei gilt in der Regel nur der Unterschied, dass es sich um eine 3 Dimensionale Matrix handelt wie Abb. 2.2 verdeutlicht. Die Matrix ist hierbei normalerweise eine X-Y-3 Matrix. Hier bezeichnet X wieder die Anzahl der Zeilen und Y die Anzahl der Spalten und die 3 die Anzahl der Kanäle. Es besteht also jedes Bild aus der additiven Farbmischung dreier Bildkanäle dem Rot-, Grün,- und Blau Kanal., Die Abtastwerte kommen ebenfalls aus dem Intervall [0,255], mit dem Unterschied, dass dieses mal 0 Schwarz bezeichnet und 255 rot, grün oder blau. Aus diesen 3 Farben lassen sich ja bekanntlich alle Farbtöne additiv mischen. Abb.2.2 verdeutlicht den Effekt der Abtastung noch etwas genauer. Es ist klar erkennbar, dass das Bild schärfer wird je mehr Abtastpunkte verwendet werden, allerdings steigt mit der Anzahl an Abtastpunkten der Speicherverbrauch des Bildes linear an. Zur Verdeutlichung wie viel Abtastpunkte denn ein qualitativ gutes Bild erzeugen: eine einfache Spiegelreflexkamera mit 24 Megapixeln z.B. nimmt ein Bild mit 24.000.000 Pixeln auf.

2.1.2 Faltung

In der Signaltheorie und speziell in der Bildverarbeitung ist die *Faltung* (engl. Convolution) neben der Fouriertransformation die mit Abstand wichtigste Operation. Mittels diskreter Faltung und Verwendung der richtigen Filterkernels können unter anderem Kanten eines Bildes in sämtlichen Richtungen gefunden werden, Bilder geschärft oder geglättet werden. Durch komplexere Anwendungen der Faltung können in Bildern durch künstliche Neuronale Netze ganze Objekte klassifiziert bzw. segmentiert werden. Die Faltung beschreibt umgangssprachlich den Vorgang Filter über Bilder zu ziehen. Im Sinne der Signalverarbeitung beschreibt die Faltung den Mathematischen Operator $*$ der aus zwei Signalen $S1$ und $S2$ ein drittes Signal $S1 * S2$ erzeugt. Im diskreten Fall bedeutet das für die Bildverarbeitung, dass 2 digitale Bilder $B1$ und $B2$ miteinander gefaltet werden und ein neues Bild $B1 * B2$ erzeugen.

Abb. 2.3 demonstriert den Effekt der Faltung eines Bildes mit einem Filterkernel für hori-

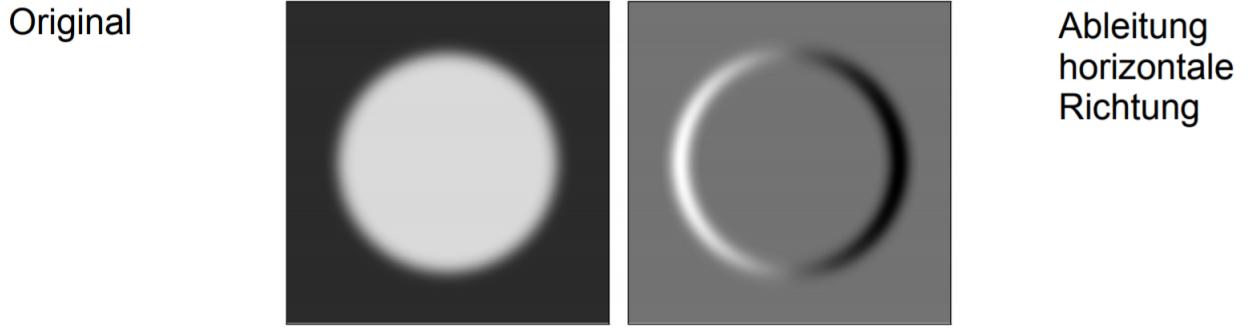


Abbildung 2.3: Anwendung eines horizontalen Ableitungskernels $\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$ zur Kanten-detection von einem weißen Kreis auf schwarzem Hintergrund. [2]

zontale Kanten. Es ist gut zu sehen, dass der Filter die höchste Aktivierung in horizontaler Richtung am Rande des Kreises beim Übergang von schwarz zu weiß hat. Die niedrigste Aktivierung findet im Gegenzug beim Übergang von weiß zu schwarz statt. Um die Faltung

0	1	1	1	0	0	0	0
0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0
0	0	0	0	1	1	0	0
0	0	1	1	0	0	0	0
0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0

*

1	0	1
0	1	0
1	0	1

=

1	4	3	4	1
1	2	4	3	3
1	2	3	4	1
1	3	3	1	1
3	3	1	1	0

I **K** **$I * K$**

Abbildung 2.4: [3]

intuitiv zu verstehen, reicht es sich die Faltung wie in Abb. 2.4 etwas vereinfacht als 2 zweidimensionale Matrizen vorzustellen, wobei I das zu filternde Bild und K der Filterkernel ist. Der horizontal gespiegelte Filterkernel wird nun oben links auf das Bild gelegt und anschließend alle sich überlappendenden Einträge miteinander multipliziert und addiert. Daraus entsteht dann der Eintrag im Ausgabebild. Anschließend wird der Filterkernel iterativ in horizontale Richtung zum Rande des Bildes bewegt. In jedem Schritt wird dabei wieder der passende Wert der Ausgabematrix berechnet. Ist der Rand des Bildes erreicht, wird der Kernel wieder an die Anfangsposition gelegt und einen Schritt in die vertikale Richtung bewegt. Dieses Verfahren wird iterativ in vertikale, wie auch in horizontale Richtung solange weiter geführt, bis das Bild vollständig abgedeckt wird. Die Größe der Ausgabematrix ist dabei Abhängig von der Bild- und Filterkernel-Matrix, sowie auch von der Schrittweite (engl. SS-

tride") abhängig. Die Schrittweite gibt an, um wie viele Einträge der Kernel horizontal, wie auch vertikal bewegt werden soll. Für den Fall, dass der Filterkernel an den Rändern das Bild überlappt gibt es das sogenannte "Padding". Dabei werden den überlappenden Stellen je nach Art unterschiedliche Werte zugeordnet. Beim Zero Padding beispielsweise werden die Ränder einfach mit Nullen aufgefüllt. Durch etwas überlegen und nachrechnen wird schnell klar, das in Abb. 2.4 kein Padding benutzt wurde und die Schrittweite 1 beträgt.

Mathematisch betrachtet sieht die Berechnung für die Faltung wie folgt aus. Seien I und K zweidimensionale Matrizen, wobei $I[i, j]$ und $K[p, q]$ den Wert der Matrizen an Stelle i, j und p, q bezeichnet. mit $i, j \in \{1, \dots, \text{Dim}(I)\}$, $p, q \in \{1, \dots, \text{Dim}(K)\}$.

für:

$$O = I * K, \quad W = \text{Dim}(K) \quad (2.1)$$

gilt dann:

$$O[i, j] = \sum_{p=-W/2}^{W/2} \sum_{q=-W/2}^{W/2} I[i - p, j - q] K[W/2 + p, W/2 + q] \quad (2.2)$$

für die Berechnung eines einzelnen Ausgabewertes. Zur Vereinfachung gilt offensichtlich Stride=1, sowie keine Berücksichtigung des Paddings. Die Formel müsste also ein klein wenig modifiziert werden, um Berechnungen am Rand durchführen zu können. Im Fall des Zero-Paddings könnte dies wie folgt aussehen:

$$O[i, j] = \sum_{p=-W/2}^{W/2} \sum_{q=-W/2}^{W/2} I_{\text{zero}}[i - p, j - q] K[W/2 + p, W/2 + q] \quad (2.3)$$

mit

$$I_{\text{zero}}[i, j] = \begin{cases} I[i, j] & 1 \leq i, j \leq \text{Dim}(I) \\ 0 & \text{sonst} \end{cases} \quad (2.4)$$

2.1.3 Pooling

Eine weitere Methode für die Verarbeitung von digitalen Bildern ist das *Pooling*. Dabei wird das Bild mit Hilfe eines Poolingkernels verkleinert, was einen Vorteil in Bezug auf Minimierung des Speicherplatzes, sowie der Berechnungszeit für nachfolgende Verarbeitungsoperationen bringt. Ein Nachteil ist der dabei entstehende Informationsverlust. Möchte man zum Beispiel den genauen Standort von Pixeln in einem durch Pooling verkleinertem Bild lokalisieren, ist dies wahrscheinlich nicht mehr exakt möglich. Für viele Anwendungen, wie der Erkennung von Objekten, reicht es aber das Bild auf die für die Lösung der Aufgabe wichtigen Informationen herunter zu brechen. In tiefen neuronalen Netzen allerdings kann der Verlust von unwichtigen Informationen sogar ein Vorteil sein, aber dazu im folgenden Kapitel mehr.

Am häufigsten verwendet werden das Max- bzw. Durchschnitts-Pooling. Intuitiv wird der Kernel auch beim Pooling auf das Bild gelegt und in horizontale bzw. vertikale Richtung über das Bild geschoben. Beim Max-Pooling wird der höchste Wert des vom Kernel überlappenden Bereichs übernommen und beim Durchschnitts-Pooling ist der Ausgabewert jeder Iteration der Durchschnitt aller vom Kernel getroffenen Werte, wie in Abb.2.5 dargestellt.

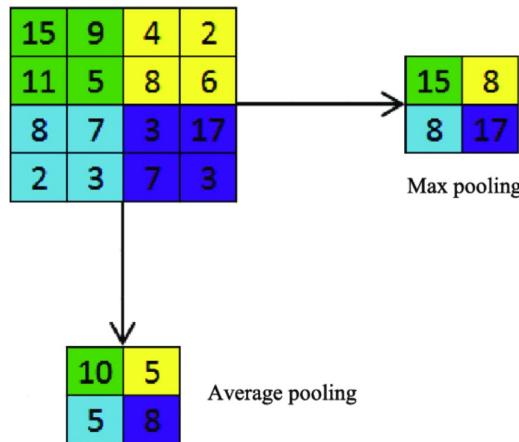


Abbildung 2.5: Anwendung eines 2x2 Max- und Average-Pooling Kernels auf eine zweidimensionale 4x4 Matrix. Figure2 in [4]

2.2 Maschinelles Lernen

Das *maschinelle Lernen* ist ein Oberbegriff für Verfahren, die durch geschicktes Lernen von Datensätzen bzw. Beispielen, Muster und Gesetzmäßigkeiten in diesen erkennen und über unbekannte Daten bestimmte Vorhersagen treffen können. Das geschieht im Trainingsverlauf der Verfahren über Anpassung bestimmter Gewichte innerhalb des Systems. Dabei entwickeln diese Verfahren eigenständig Algorithmen über statistische Modelle. Das bedeutet, dass die Beispiele nicht einfach auswendig gelernt werden, sondern bestimmte Eigenschaften und Merkmale innerhalb der Daten erkannt werden, welche maßgeblich entscheidend für die Vorhersage sind. Soll Beispielsweise über ein Bild vorhergesagt werden, ob sich auf diesem ein Mensch befindet, dann lernt das System, was einen Menschen optisch kenntlich macht. Wenn im Volksmunde von *künstlicher Intelligenz* die Rede ist, handelt es sich in den meisten Fällen um Verfahren des maschinellen Lernens. Der heutige technische Erfolg von Sprach-, Text- und Bilderkennung, Übersetzern, selbstfahrenden Autos, Marktanalysen und sogar sozialen Netzwerken wie Facebook und Instagram, ist zu einem großen Teil der Verdienst des maschinellen Lernens. Die Anwendungsmöglichkeiten sind nahezu unbegrenzt und universell auf fast jeden Bereich anwendbar, was zur Folge hat das maschinelles Lernen fest in unserem Alltag integriert ist, ohne das viele Menschen überhaupt etwas davon wissen. E-Mail Spam Ordner, Bildschirmsperrung für moderne Smartphones mit Fingerabdruck- und Gesichtserkennung, Computer Gegner in Videospielen von Schachgegnern bis hin zu Soldaten in Egoshootern, sowie die Entscheidungen, welche Werbung auf Onlineshops wie Amazon welchem Kunden angezeigt wird und welche Profile in der Timeline auf Sozialen Netzwerken vorgeschlagen werden, sind nur einige Beispiele von vielen.

2.2.1 Lernmethoden

Es existieren 3 Lernmethoden, für das Trainieren von Verfahren des maschinellen Lernens, welche sich in der Art der Überwachung beim Lernen unterscheiden[5]. Das *überwachte Lern-*

nen (engl. supervised learning), das *unüberwachte Lernen* (engl. 'unsupervised learning') und das *bestärkende Lernen* (engl. 'reinforcement learning'). Grundlage für diese Arbeit ist das überwachte Lernen.

Überwachtes Lernen

Beim überwachten Lernen liegen Die Daten mit den gewünschten Lösungen, den sogenannten *Labels* vor. Das System wird sowohl mit den Daten als auch deren Labels trainiert. Eine Grundlegende Aufgabe ist die Klassifikation, wobei passend zu den Daten die Labels bestimmte Klassen bzw. Kategorien sind. Ein Beispiel hierfür sind Spamfilter die mit einem Datensatz an E-Mails und der für jede E-Mail passenden Kategorie *Ham* oder *Spam* darauf trainiert werden, neue E-Mails zu klassifizieren.

Eine weitere Aufgabe neben der Klassifizierung ist die vorhersage von numerischen Werten, wie etwa den Preis einer Immobilie vorherzusagen. Die Daten liegen im Datensatz zusammengesetzt aus mehreren Merkmalen, den sogenannten Attributen und ihren Werten, wie Ort, Wohnfläche, Anzahl der Räume etc. vor und die Labels sind dabei die Preise der Immobilien.

Einige der wichtigsten Verfahren des überwachten Lernens sind k-nächste-Nachbarn, Lineare Regression, Logistische Regression, Support Vector Machines, Entscheidungsbäume, Random Forests und künstliche neuronale Netze. In dieser Arbeit wurden ausschließlich künstliche neuronale Netze verwendet.

Unüberwachtes Lernen

Beim unüberwachten Lernen, liegen die Daten ohne Labels vor. Das System versucht also nur mit den Daten an sich zu Lernen und kann somit keinen direkten Zusammenhang zwischen Label und Daten erkennen, sondern erkennt Strukturen nur innerhalb der Daten. Die *Dimensionsreduktion* ist eine solches Verfahren. Das Ziel dabei ist es den Datensatz der Vereinfachung halber zu reduzieren und unwichtige Informationen zu verwerfen. Korrelierende Attribute wie beispielsweise in einem Auto *PS* und *Höchstgeschwindigkeit* lassen sich zu einem verschmelzen und das System kann beide zu einem einzigen Attribut verbinden.

Ein weiterer Aufgabenbereich ist das Erkennen von Anomalien. Beispiele sind das Bemerken und Entfernen von Ausreißern in einem Datensatz zur Verbesserung von diesem, sowie das Vorbeugen von Produktionsfehlern, oder die Feststellung von ungewöhnlichen Transaktionen auf Kreditkarten.

Bestärkendes Lernen

Die Methode des bestärkenden Lernens ist völlig anders. Das Lernsystem, der sogenannte *Agent* bekommt keine Daten in Form wie beim un/überwachten Lernen, sondern beobachtet seine Umgebung und dessen aktuelle Gegebenheiten, wählt anhand dieser Aktionen und führt diese aus. beeinflusst wird das Lernen des Agenten in Form von Belohnung und Bestrafung. Wählt der Agent richtige bzw. vorteilhafte Aktionen, so wird er belohnt. Macht er einen Fehler, dann wird er bestraft. Ziel des Agenten ist es eine Strategie oder *Policy*

zu finden, welche ihm die meisten Belohnungen bringt. Eine Policy gibt vor unter welchen Gegebenheiten der Agent eine bestimmte Aktion wählt. In der Robotik verwenden viele Ro-

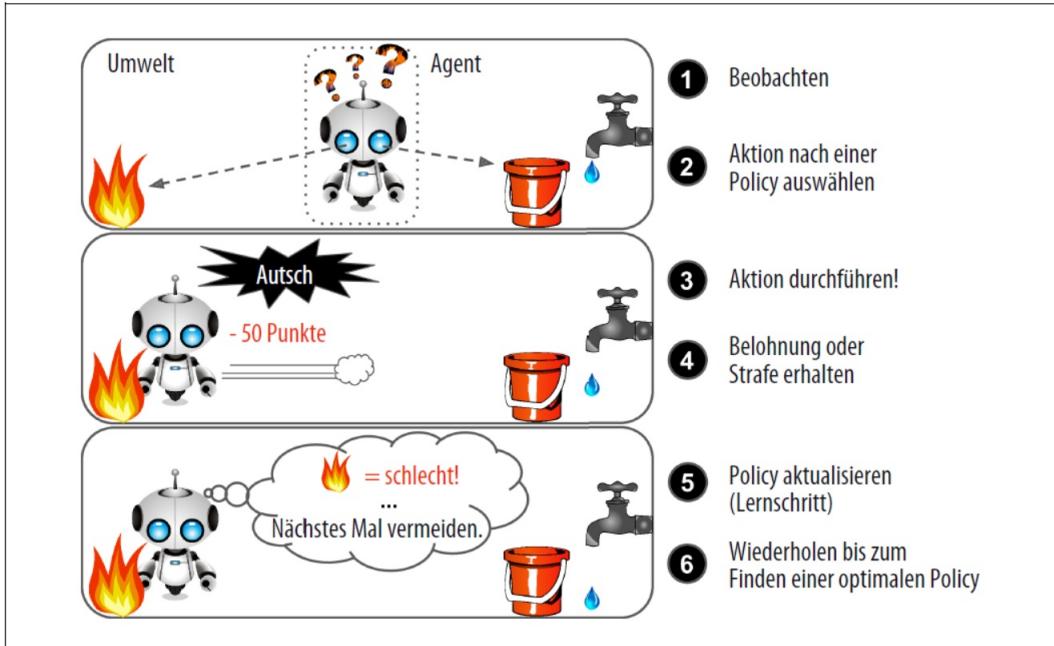


Abbildung 2.6: Ablauf des Lernzyklus für das laufen Lernen eines Roboters mittels bestärkendem Lernen. Seite 28 in [5]

boter bestärkendes Lernen, um motorische Fähigkeiten wie das Laufen zu erlernen. Abb. demonstriert den Ablauf eines Lernzyklus des überwachten Lernens. Im ersten Schritt beobachtet der Roboter mit seinen Sensoren die Umwelt und nimmt diese mit ihren aktuellen Gegebenheiten war. Anschließend wählt er über seine bisherige gewählte Policy eine Aktion aus, wie beim Laufen lernen ein Bein zu heben, oder den Oberkörper zum ausbalancieren in eine Richtung zu Beugen. Führt der Agent die gewählte Aktion aus und stabilisiert sich bzw. bewegt sich weiter nach vorne, dann bekommt er eine Belohnung. Wird er instabil oder fällt sogar um, dann wird er Bestraft. In diesem Fall aktualisiert er seine Policy und fängt wieder von vorne an bis eine Optimale Policy gefunden wurde, die ihm das Laufen ermöglicht. Computergegner in Videospielen von Schach bis hin zu Egoshootern nutzen in der Regel ebenfalls bestärkendes Lernen um eine gute Spielstrategie zu finden.

Das Programm *AlphaGo* [6] von Googles Team *DeepMind* verdeutlicht die Stärke des bestärkenden Lernens. Der Lernprozess von AlphaGo[6] bestand aus Millionen simulierten Partien, um eine effiziente Policy für das Brettspiel Go zu erlernen. Das Programm schlug anschließend den Go Weltmeister *Ke Jie*, wobei es nur die bis dahin erlernte Policy verwendete und der Lernprozess beim Spiel gegen Ke Jie abgeschaltet wurde.

2.3 Künstliche neuronale Netze

2.3.1 Einschichtiges Perceptron

Biologisches Neuron

Der Aufbau von künstlichen Neuronen basiert auf dem Konzept der biologischen Neuronen, welche die Grundlage für sämtliche Wahrnehmung und Auslöser für jegliche Art von Verhalten der Lebewesen sind. Abb.2.7 zeigt den Aufbau eines biologischen Neuron. Es gibt zwar

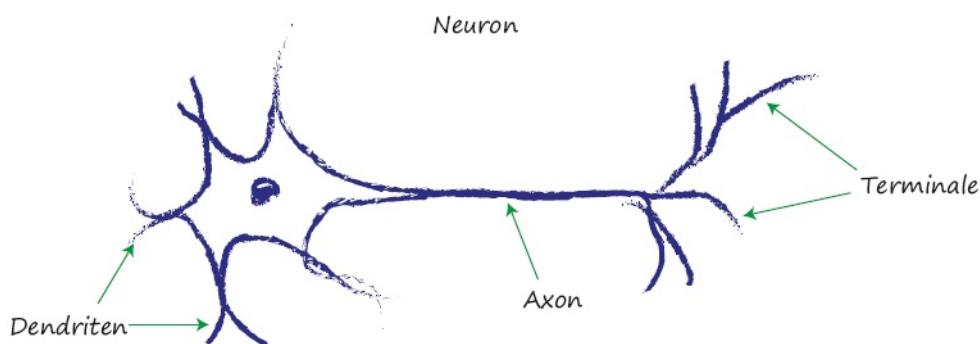


Abbildung 2.7: Modell eines biologischen Neuron. Abbildung 1-27 in [7]

unterschiedliche Formen von Neuronen, allerdings übertragen alle Informationen in Form eines elektrischen Signals von den Dendriten über die Axone zu den Terminalen. Anschließend wird das Signal über die Terminalen an weitere Neuronen übergeben. Dadurch können beispielsweise Reize über die Sinne aufgenommen und von dort weiter in das Gehirn transportiert werden, welches ebenfalls aus einem Netz aus Neuronen besteht. Durch Beobachtung konnte festgestellt werden, dass ein Neuron nicht jedes von den Dendriten empfangene Signal weiter leitet, sondern nur solche für die eine bestimmte Größe erreicht wurde. Das könnte man mit einem Schwellwert vergleichen, der erreicht werden muss bis das Neuron eine Ausgabe macht.

Künstliches Neuron

1958 stellte Frank Rosenblatt[8] erstmalig das *Perceptron* vor, welches ein vereinfachtes künstliches neuronales Netz ist. In der Grundversion besteht es aus einem einzigen künstlichen Neuron mit veränderlichen Gewichtungen und einem Schwellenwert für die Ausgabe wie in Abb.2.8 dargestellt.

Das *einschichtige Perceptron* besteht lediglich aus einer Eingabeschicht mit einer endlichen Anzahl n an Eingabewerten (x_n) und einer Ausgabeschicht mit ebenfalls endlicher Anzahl m an Ausgabewerten (o_m). Jeder Eingabewert i der Eingabeschicht wird mit jedem Knoten der Ausgabeschicht j durch eine Gewichtung w_{ij} verbunden, wobei jeder Ausgabeknoten einen Schwellwert s_j besitzt. Es gibt unterschiedliche Varianten mit leichten Abwandlungen für die Berechnung der Ausgabewerte und die Lernregeln des Perceptrons. Im folgenden werden

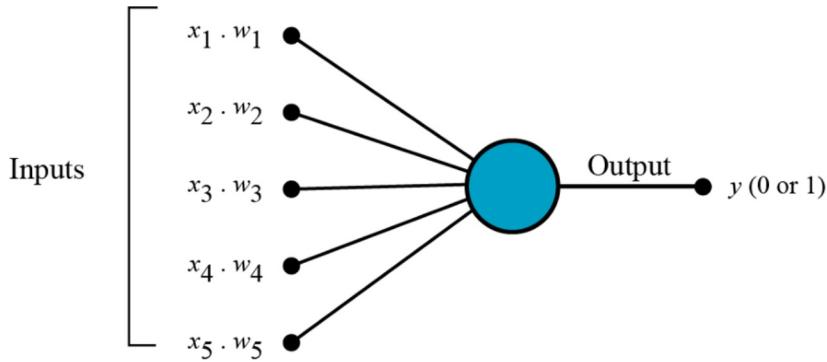


Abbildung 2.8: Modell eines künstlichen Neurons mit einer Eingabeschicht von 5 Eingabewerten und einem Ausgabewert. [9]

zur Vereinfachung binäre Ein- und Ausgabewerte angenommen. Für die Berechnung der j-ten Ausgabe, werden die Eingabewerte mit den passenden Gewichtungen multipliziert und anschließend aufsummiert. Danach wird der Schwellwert von der Summe subtrahiert und eine Aktivierungsfunktion angewendet, welche anschließend aus der Differenz die Ausgabe generiert. Mathematisch wird die Berechnung für die j-te Ausgabe o_j wie folgt beschrieben:

$$o_j = H\left(\sum_{i=1}^n x_i w_{ij} - s_j\right) \quad (2.5)$$

wobei H die Heaviside-Funktion bezeichnet, welche ihrem Eingabewert $X \in \mathbb{R}$, abhängig von seinem Vorzeichen, den Wert 0 oder 1 zuweist:

$$H(X) = \begin{cases} 0 & X < 0 \\ 1 & X \geq 0 \end{cases} \quad (2.6)$$

Das Neuron gibt also in Ausgabe j den Wert 1 aus, falls der Schwellwert über die Eingabe erreicht wurde ansonsten 0.

Wie bereits angekündigt gibt es mehrere Versionen der Lernregel des Perceptrons die im folgenden vorgestellte Lernregel terminiert nur dann, wenn der Trainingsdatensatz linear separierbar ist. Intuitiv kann die mathematische Definition der Lernregel durch folgende Überlegungen nachvollzogen werden:

1. Wenn die Soll-Ausgabe gleich der Ist-Ausgabe ist, dann werden die Gewichte nicht geändert.
2. Wenn die Soll-Ausgabe des Neurons 1, die tatsächlichen Ausgabe aber 0 ist, dann werden die Gewichte dekrementiert.
3. Wenn die Soll-Ausgabe des Neurons 0, die tatsächlichen Ausgabe aber 1 ist, dann werden die Gewichte inkrementiert.

mathematisch ist die Lernregel folgendermaßen definiert:

$$w_{iJ}^{neu} = w_{ij}^{alt} + \Delta w_{ij}, \quad (2.7)$$

mit:

$$\Delta w_{ij} = \alpha(t_j - o_j)x_i. \quad (2.8)$$

Dabei ist

- t_j die Soll-Ausgabe des Neurons j
- $\alpha > 0$ die Lernrate

Das mit dem angegebenen Lernverfahren alle Lösungen erlernt werden können, welche ein Perceptron repräsentieren kann, konnte Robert Rosenblatt[8] durch das Konvergenztheorem beweisen.

2.3.2 Mehrschichtiges Perceptron

Es existieren viele unterschiedliche Architekturen für künstliche Neuronale Netze. Die bekannteste Architektur stellt des mehrschichtige Perceptron da, welches eine Feed-Forward Architektur mit vollständig verbundenen Schichten darstellt. Das bedeutet, dass jeder Knoten einer Schicht mit jedem Knoten der nächsten Schicht verbunden ist. Der Aufbau eines

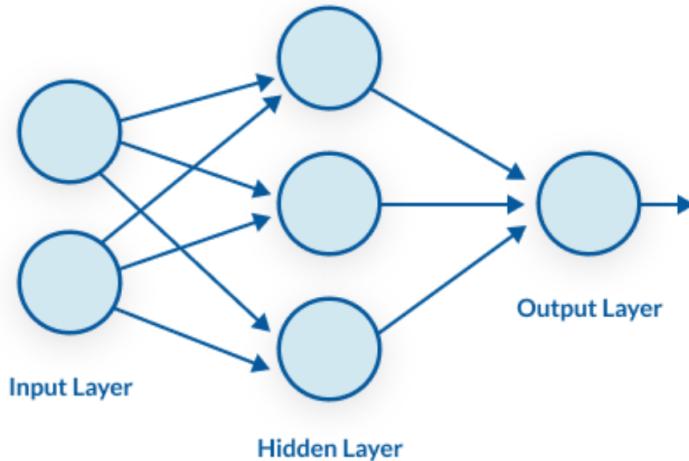


Abbildung 2.9: Darstellung eines Mehrschichtigen Perceptron mit 2 Eingabe-Neuronen, 3 Versteckten Neuronen, und einem Ausgabe-Neuron. [9]

mehrschichtigen Perceptron ist in Abb.2.9 dargestellt. Es erweitert das einschichtige Perceptron durch sogenannte versteckte Schichten (engl. 'hidden layer'). Das Netz besitzt weiterhin eine Eingabe- bzw. Ausgabeschicht mit endlich vielen Neuronen, sowie beliebig viele versteckte Schichten mit endlich vielen Neuronen. Jedes Neuron der Versteckten- bzw. Ausgabeschicht besitzt hierbei wieder einen Schwellwert und eine Aktivierungsfunktion, welche

aus der Summe der Eingabewerte multipliziert mit seinem Gewicht, den Wert des Neurons durch die Differenz der Summe mit dem Schwellwert berechnet. Zudem hat jedes Neuron einschließlich der Eingabeschicht eine Gewichtung für jede Verbindung mit einem Neuron der darauffolgenden Schicht.

Forward Pass

Im sogenannten *Forward Pass* wird die Ausgabe des Neuronalen Netz erzeugt. Die Ausgabe der ersten versteckten Schicht wird genau so berechnet, wie die Ausgabe des einschichtigen Perceptrons. Anschließend wird iterativ weiter Verfahren, indem die Ausgabe der versteckten Schicht zur Eingabe der nächsten Schicht wird, bis die Ausgabeschicht erreicht wird. Die Eingabe wird also, wie bei einem biologischen neuronalen Netz durch das Neuronen Netz geschickt dort von jedem Neuron weiter verarbeitet und anschließend vom Ausgabeneuron eine Ausgabe generiert. Mathematisch kann das ganze über folgende Rekursionsgleichung definiert werden:

$$n_j^{(k)} = \phi \left(\left(\sum_{i=1}^n n_i^{(k-1)} w_{ij}^{(k-1)} - s_j^{(k)} \right) \right) \quad (2.9)$$

mit Rekursionsanfang:

$$n_j^{(1)} = e_j \quad (2.10)$$

wobei gilt:

- ϕ ist die Aktivierungsfunktion.
- $n_j^{(k)}$ ist der Wert des j-ten Neurons in der k-ten Schicht.
- w_{ij}^k ist das Gewicht, der Verbindung des i-ten Neurons aus der k-ten Schicht mit dem j-ten Neuron der (k+1)-ten Schicht
- $s_j^{(k)}$ ist der Schwellwert des j-ten Neurons der k-ten Schicht ist.
- e_j der j-te Wert der Eingabeschicht ist.

Backward Pass

Das Lernen erfolgt auch im mehrschichtigen Perceptron durch anpassen der Gewichte. Allerdings wird hierbei der entstandene Fehler Rückwärts durch das Netz geschickt und dabei die Gewichte angepasst, dieses Verfahren wird *Backpropagation* genannt. Es gilt ebenfalls, dass es hierfür mehrere Arten der Durchführung gibt, der Prozess aber nahezu derselbe ist. Der Fehler eines Ausgabeknotens j für die vorhersage des n-ten Datenpunkts aus den Trainingsdaten, wird mit

$$e_j(n) = t_j(n) - y_j(n) \quad (2.11)$$

dargestellt, wobei t die Soll-Ausgabe und y die Ist-Ausgabe des neuronalen Netz bezeichnet. Die Gewichte werden durch die Minimierung des quadratischen Fehlers der gesamten

Ausgabe ε :

$$\varepsilon(n) = \frac{1}{2} \sum_j e_j(n) \quad (2.12)$$

angepasst. Bei der Verwendung des Gradienten Abstiegs, ist die Anpassung der Gewichte durch

$$\Delta w_{ji}(n) = -\alpha \frac{\partial \varepsilon(n)}{\partial v_j(n)} y_i(n) \quad (2.13)$$

definiert. Wobei y_i die Ausgabe des vorherigen Neurons und α die Lernrate bezeichnet. Die zu berechnende Ableitung hängt vom sich verändernden induzierten Feld v_j ab. Die Ableitung kann zu

$$-\frac{\partial \varepsilon(n)}{\partial v_j(n)} = e_j(n) \phi'(v_j(n)) \quad (2.14)$$

vereinfacht werden, wobei ϕ' die Ableitung der verwendeten Aktivierungsfunktion ist. Für verdeckte Neuronen ist die Analysis ein bisschen komplizierter es kann allerdings gezeigt werden, das für die Ableitung gilt:

$$-\frac{\partial \varepsilon(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \varepsilon(n)}{\partial v_k(n)} w_{kj}(n) \quad (2.15)$$

was abhängig von der Gewichtsänderung des k-ten Neurons ist, welches die Ausgabeebene darstellt. Um die versteckten Gewichte zu verändern müssen also die Gewichte in der Ausgabeschicht geändert werden. Dadurch ist klar, dass der Backpropagation Algorithmus eine rückwärts Durchführung der Aktivierungsfunktion darstellt.

2.3.3 Convolutional Neural Networks

Seit der Einführung der Convolutional Neural Networks, auf deutsch etwa neuronale Faltungsnetzwerke, von Yann LeCun[10] in den frühen 90er Jahren, bringen neuronale Netze sehr gute Ergebnisse in den Bereichen der Bild und Audioverarbeitung. Aufgaben wie die Sprachanalyse, Objekterkennung und der Semantischen Segmentierung haben ihren Durchbruch durch Convolutional Neural Networks erreicht. Die biologische Inspiration von Convolutional Neural Networks ist der visuelle Cortex im Gehirn. Neuronen im primären visuellen Cortex, sogenannte *simple cells* reagieren auf ein kleines Rezeptives Feld der Retina und analysieren somit quasi kleine Abschnitte des Sehfeldes. Die Informationen werden an weitere Neuronen des visuellen Kortex weiter gegeben und Schichtweise zusammen gefügt, wodurch am Ende eine optische sowie semantische Deutung des Bildes entsteht. Das kann man sich etwa so vorstellen, das einmal das Bild aus vielen Pixeln entsteht, aber unser Verstand dem Bild weitere Bedeutungen zuschreibt, wie die Zuweisung einzelner Pixel zu bestimmten Objekten deren Entfernung und vielem mehr.

Der grundlegende Aufbau eines Convolutional Neural Network ist in Abb.2.10 dargestellt. Bei dieser Art von neuronalen Netzen, handelt es sich nicht ausschließlich um vollständig verbundene Schichten. In der Regel gibt es eine oder mehrere Faltungsschichten wobei die Verknüpfungen der Neuronen so angeordnet werden, das eine oder mehrere Faltungen auf

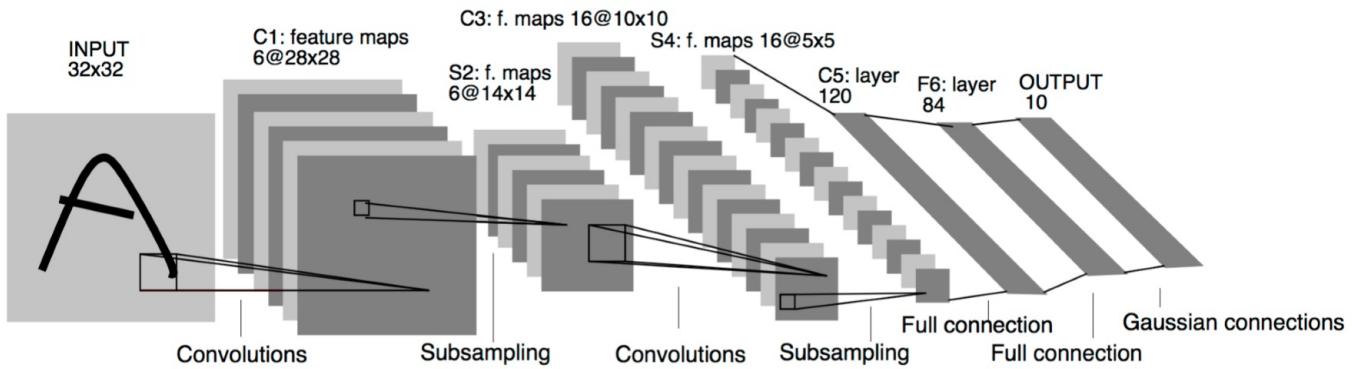


Abbildung 2.10: Darstellung eines Convolutional Neural Network mit 2 Faltungs- und 2 Poolingschichten, für die Erkennung von Buchstaben. Figure 1 in [11]

die Eingabe angewendet werden. Anschließend folgt eine Pooling Schicht, meist Max Pooling, in der redundante Informationen verworfen werden und Rechenzeit gespart wird, da in der Regel nur die stärksten Filteraktivierungen von Interesse sind. Der Anwendung von Faltungen gefolgt von Pooling kann dabei beliebig oft wiederholt werden. Bei sehr vielen Faltungs- und Poolingschichten spricht man von *Deep Convolutional Neural Networks*. Auf die letzte Poolingschicht folgen in der Regel in vollständig verbundene Schichten, wie beim mehrlagigen Perceptron, wobei die letzte Schicht die Ausgabe generiert.

Die Faltungsschicht

Normalerweise liegt die Eingabe des Convolutional Neural Networks in Form von einer zwei- bzw. dreidimensionalen Matrix vor (Grauwert bzw. Farbbild bei Bildverarbeitungsaufgaben). Die Neuronen in der Faltungsschicht (engl. 'convolution layer') sind dementsprechend angeordnet. Über diskrete Faltung wird die Aktivität jedes Neurons berechnet. Instinktiv wird dabei wie schon im Kapitel der Faltung beschrieben, ein Faltunskern über die Eingabe bewegt, wobei unterschiedliche Merkmalskarten (engl. 'featuremaps') aus der Eingabe erstellt werden. Zu beachten ist hierbei, dass jedes Neuron nur auf Reize eines kleinem Feldes in der Eingabe reagiert, was dem biologischen Verhalten des rezeptiven Feldes entspricht. Die Gewichte für alle Neuronen sind hierbei identisch, das Netz lernt also nicht die Gewichte anzupassen, sondern die Werte der Filterkernel. Das bedeutet, dass das Netz im Training selbstständig passende Filter lernt, welche die für die Vorhersage notwendigen Eigenschaften aus der Eingabe extrahieren. Analog zum visuellen Cortex steigt in tieferen Faltungsschichten die Größe der rezeptiven Felder, als auch die Komplexität der erkannten Merkmale (engl. 'features'). In den früheren Schichten bei z.B. der Erkennung von Gesichtern, werden einfache Strukturen wie Kanten, Ecken oder Kreise durch die Filter erkannt und in den späteren tieferen Schichten komplexere zusammengesetzte Strukturen wie Augen, Mund, Nase etc. erkannt. Abb. 2.11 zeigt die Aktivierung eines Filters zur Erkennung von Kreisen im oberen Teil, sowie die Aktivierung eines Filters für Rechtecke im unteren Kreis. Die hellen Bereiche der Merkmalskarten stellen die starken Aktivierungen der Filter dar, also jene Bereiche in

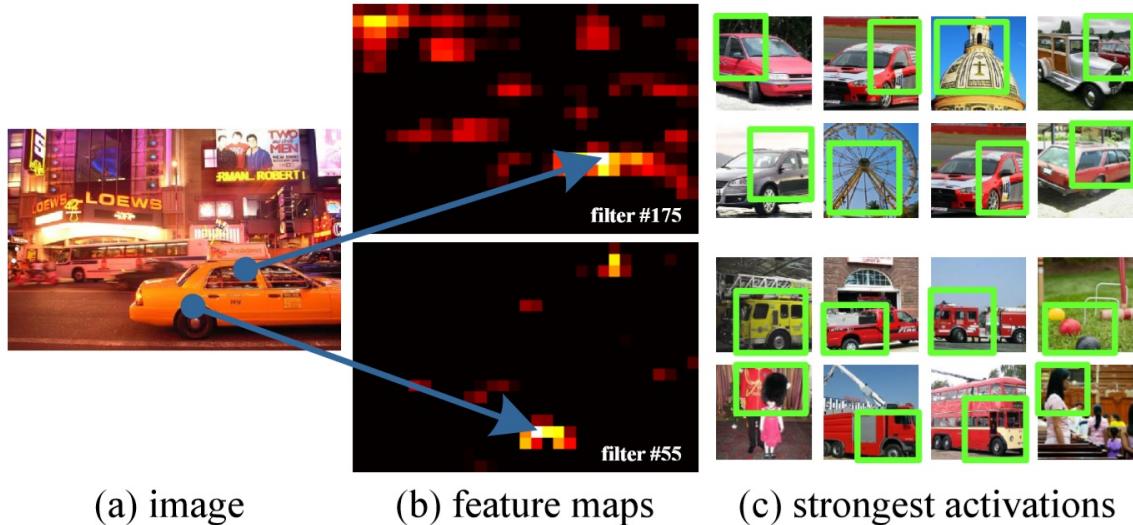


Abbildung 2.11: Darstellung von Filteraktivierungen zweier Filter für die Erkennung von Kreisen und Rechtecken in Bildern. Figure 2 in [12]

denen sich zu hoher Wahrscheinlichkeit ein Kreis oder ein Rechteck befindet. Hierbei wird der große Nutzen der Faltungsschichten für die Merkmalsextraktion aus den Daten deutlich.

Poolingschichten

In den Poolingschichten (engl. 'Poolinglayer'), werden die Aktivierungen der Neuronen gefiltert. Die wichtigen Aktivierungen werden mittels Pooling, wie im Kapitel zu Pooling übernommen, und die unwichtigen verworfen. Das biologische Vorbild des Poolings ist die *laterale Hemmung* im visuellen Cortex. Bei der Erkennung von Objekten in Bildern zum Beispiel ist nur die relative Position von erkannten Strukturen untereinander im Bild von nötiger Relevanz für eine richtige Vorhersage. Wie bereits erläutert gibt es verschiedene Arten des Poolings, jedoch zeigt sich in den meisten Fällen das Max-Pooling die beste Wahl ist. Trotz der großen Datenreduktion, 75 % Verlust bei einem 2x2 Poolingkernels, verringert sich in der Regel die Performance des Netzwerks nicht durch das Pooling. Im Gegenteil bietet es sogar einige signifikante Vorteile:

- Verringelter Speicherbedarf
- Erhöhte Berechnungsgeschwindigkeit
- Die Möglichkeit der Erstellung tieferer Netze durch weniger Rechenkosten
- Automatisches Wachstum der Größe der rezeptiven Felder in den Faltungsschichten, ohne dabei größere Faltungskernels verwenden zu müssen.
- Vorbeugende Maßnahme gegen Überanpassung (engl. 'overfitting') durch zu langen Trainingsprozess.

Vollständig verbundene Schichten

Im Anschluss an die sich wiederholenden Faltungs- und Poolingschichten schließt das Netz mit vollständig verbundenen Schichten ab. In diesen werden die gefundenen Merkmale weiter verarbeitet und aus ihrer Deutung eine Ausgabe generiert. Bei der Erkennung von Objekten zum Beispiel, wobei ein Bild in 5 verschiedene Klassen fallen kann, wird die Ausgabeschicht der vollständig verbundenen Schicht in den meisten Fällen eine Softmax-Schicht sein, welche aus 5 Neuronen besteht, die alle eine Ausgabe machen. Die Ausgabe eines jeden Neurons entspricht hierbei der Wahrscheinlichkeit für eine Klasse, die tatsächlich Ausgabe ist dann sozusagen ein Vektor der an der Stelle für die Klasse mit der höchsten Wahrscheinlichkeit eine 1 hat und allen anderen eine 0. Dieses sehr einfache Verfahren, genannt *One-Hot-Encoding*, hat den Vorteil, dass das Netz keine impliziten Annahmen über Zusammenhänge zwischen den Klassen machen kann. Abb. 2.12 verdeutlicht nochmal, den Nutzen

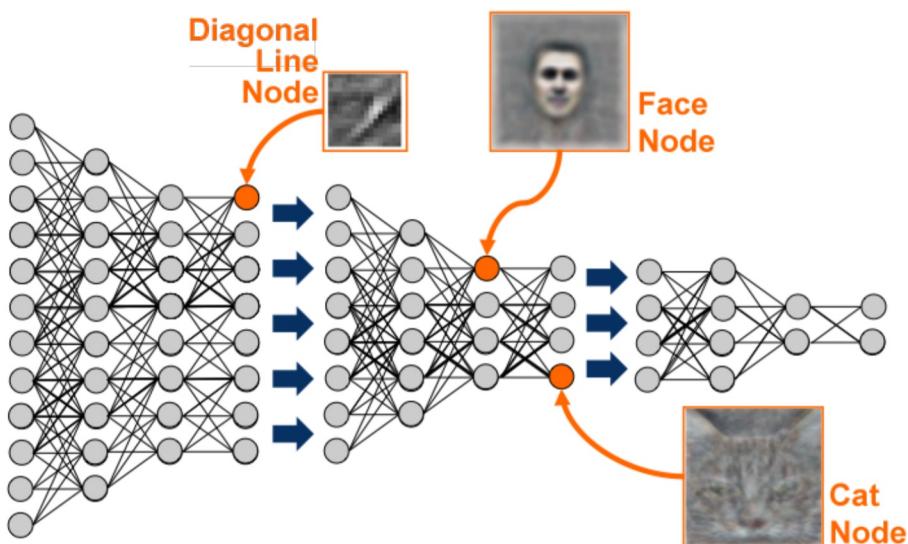


Abbildung 2.12: Darstellung eines Convolutional Neural Networks zur Objekterkennung in Bildern. Figure 2 in [12]

einzelner Neuronen innerhalb des Convolutional Neural Networks. In diesem Beispiel handelt es sich um ein Convolutional Neural Network zur Objekterkennung in Bildern. Das Neuron oben links in einer anfänglichen Faltungsschicht wird stark aktiviert, wenn sich im für das Neuron festgelegten rezeptiven Feld der Daten eine Diagonale Kante befindet. Die beiden gekennzeichneten Neuronen in den späteren Schichten werden dann durch komplexere Strukturen, in einem größeren, für sie festgelegtem rezeptivem Feld, nämlich einer Katze oder einem Menschlichen Gesicht aktiviert. Convolutional Neural Networks können genau wie das mehrschichtige Perceptron über Backpropagation Algorithmen trainiert werden, zu bemerken ist hier allerdings, dass das Trainieren und Lernen von künstlichen Neuronalen Netzen sich vollständig von ihren biologischen Vorbildern unterscheidet. Wie genau ein biologisches neuronales Netz lernt ist, allerdings unklar.

2.3.4 Rekurrente neuronale Netze

Rekurrente neuronale Netze, oder rückgekoppelte neuronale Netz (engl. 'recurrent neural networks') sind nicht wie die gewöhnlichen hier bereits besprochenen *feed forward networks*, welche nur Verbindungen von Neuronen zu anderen Neuronen aus tieferen Schichten haben, sondern auch Verbindungen zu sich selbst, oder zu Neuronen der gleichen bzw. vorangegangenen Schichten. Im Gehirn sind diese Verknüpfungsweisen die bevorzugten, vor allem im Neocortex. Bei Aufgaben welche als Eingabe eine Sequenz erwarten wie z.B. Handschrifterkennung, Sprachverarbeitung, und maschinelles Übersetzung von Sprachen, sind Rekurrente neuronale Netze die erste Wahl. Dabei werden die Sequenzen Stück für Stück durch das Netz geschickt. Durch die Rückkopplung der Ausgaben von Neuronen aus den vorherigen Schritten mit dem aktuellen Teil der Eingabe zusammen wieder in das Netz geschickt werden können Zusammenhänge zwischen den Teilen der Eingabe berücksichtigt werden. Um den Vorteil dieser Art der Verarbeitung zu demonstrieren, reicht es die Übersetzung von Wörtern und Sätzen von einer Sprache in eine andere Sprache zu betrachten, dabei können einzelne Wörter, sowie Wörter innerhalb eines Satzes im Zusammenhang mit anderen Wörtern ganz unterschiedliche Bedeutungen haben. Rekurrente neuronale Netzwerkarchitekturen wie (*bidirectional*)*long short term memorys* auf deutsch etwa (bidirektionales) Lang- Kurzzeitgedächtnis oder Abwandlungen davon, sind momentan die vorherrschende Variante. Diese können nicht nur den Zusammenhang zwischen 2 Schritten der Eingabesequenz, sondern von vielen aufeinanderfolgenden Teilen bis hin zur gesamten Eingabe berücksichtigen.

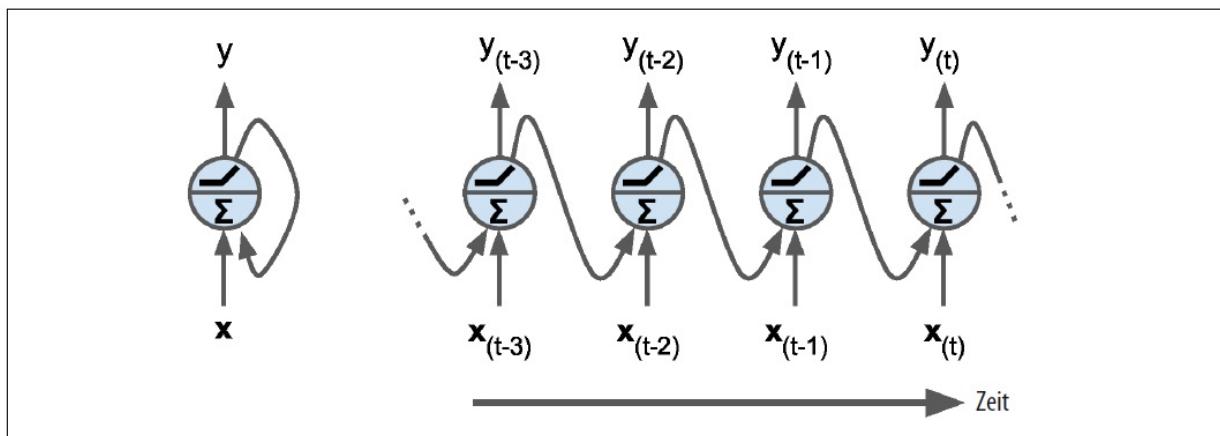


Abbildung 2.13: Links: Ein rekurrentes Neuron, Rechts: Das rekurrente Neuron entlang der Zeitachse aufgerollt. Abbildung 14-1 in [5]

Rekurrentes Neuron

Ein einzelnes Neuron, welches eine Eingabe erhält, eine Ausgabe produziert, diese aus gibt und zusätzlich an sich selbst schickt, wie in Abb. 2.13 (links), ist das einfachste mögliche rekurrente Neuronale Netz. Zu jeder Eingabe x_t des rekurrenten Neurons zum Zeitpunkt t , erhält dieses seine eigene Ausgabe aus dem vorherigen Schritt y_{t-1} . In Abb. 2.13 (rechts) ist

das winzige Netz der Zeitachse entlang aufgerollt dargestellt. Das rekurrente Neuron enthält zwei veränderbare Gewichte w_x für die Eingabe x_t und w_y für die Ausgabe des vorangegangenen Schrittes y_{t-1} . Daraus folgt für die Ausgabe y_t eines einzelnen Datenpunktes für ein solches rekurrentes Neuron mathematisch die Formel

$$y_t = \phi\left(x_t * w_x + w_y * y_{t-1} - s\right). \quad (2.16)$$

Zu beachten ist hierbei, dass es sich eigentlich um eine Rekursive Funktion handelt, da die Ausgabe zum Zeitpunkt t von der Eingabe x_t und der Ausgabe y_{t-1} abhängt, wobei y_{t-1} wieder von der Eingabe x_{t-1} und der Ausgabe y_{t-2} abhängt und so weiter. Dadurch wird die Funktion y_t eine Funktion aus allen Eingaben $x_0, x_1..x_t$, wobei y_{-1} zur Berechnung von y_0 auf 0 gesetzt wird. Man sagt auch, dass das rekurrente Neuron ein 'Gedächtnis' hat, wobei man den Teil eines neuronalen Netzes dessen Zustand über mehrere zeitliche Schritte erhalten bleibt, als Gedächtniszelle (oder einfach als Zelle) bezeichnet. Ein einzelnes rekurrentes Neuron ist dabei eine sehr einfache Zelle, allerdings gibt es auch deutlich komplexere Architekturen.

Training

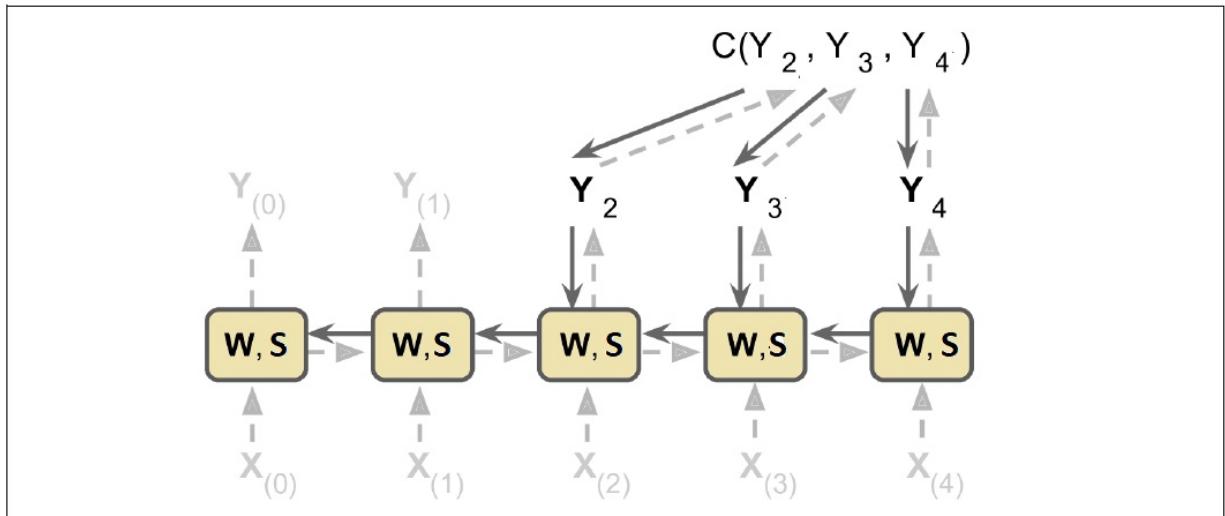


Abbildung 2.14: Backpropagation through time in einem rekurrenten Neuron über mehrere Zeitschritte. Abbildung 14-5 in [5]

Das Trainieren eines rekurrenten neuronalen Netzes ist offensichtlich nicht so einfach möglich, wie bei den bisher bekannten Netzen. Wenn das Netz jedoch entlang der Zeitachse wie in Abb.2.14 aufgerollt wird, dann ist es möglich, den gewohnten Backpropagation Algorithmus anzuwenden, diese Strategie nennt man *Backpropagation through time*. Die Bewertung der Ausgabesequenz, erfolgt hierbei über eine Kostenfunktion $C(y_{t_{min}}, y_{t_{min+1}}, \dots, y_{t_{max}})$, welche den Wert des Fehlers aus der Ausgabe extrahiert, wobei t_{min} den ersten und t_{max} den letzten

Zeitschritt der Ausgabe bezeichnen (ignorierte Ausgaben zählen dabei nicht). Die Gradienten der Kostenfunktion werden wie gewohnt rückwärts durch das aufgerollte Netz geschickt und schließlich die Gewichte über die Gradienten aktualisiert. Dabei werden die Gradienten der gesamten, von der Kostenfunktion verwendeten Ausgaben, benutzt. In Abb.2.14 also, wird die Kostenfunktion nur aus den Ausgaben y_2 , y_3 und y_4 berechnet, daher fließen die Gradienten nur durch diese 3 Ausgaben. Der Backpropagation Algorithmus ist so richtig definiert, da in jedem Zeitschritt der Schwellwert S , sowie dieselben Gewichte w_x und w_y verwendet werden.

LSTM-Zellen

Die Architektur der *Long Short Term-Memory*, auf deutsch Lang-Kurzzeitgedächtnis wurde im Jahr 1997 erstmals von Sepp Hochreiter und Jürgen Schmidhuber [13] vorgeschlagen. Im Laufe der Zeit wurde sie von unterschiedlichen Wissenschaftlern wie Alex Graves und Hasim Sak [14] oder Wojciech Zaremba [15] weiter entwickelt. Im Vergleich zur einfachen Zelle kann die LSTM-Zelle, bei deutlich besseren Ergebnissen, im wesentlichen genau so einfach verwendet werden. Der Trainingsprozess konvergiert schneller, wobei weitreichende Abhängigkeiten innerhalb der Daten erkannt werden. In der folgenden Erklärung sind die

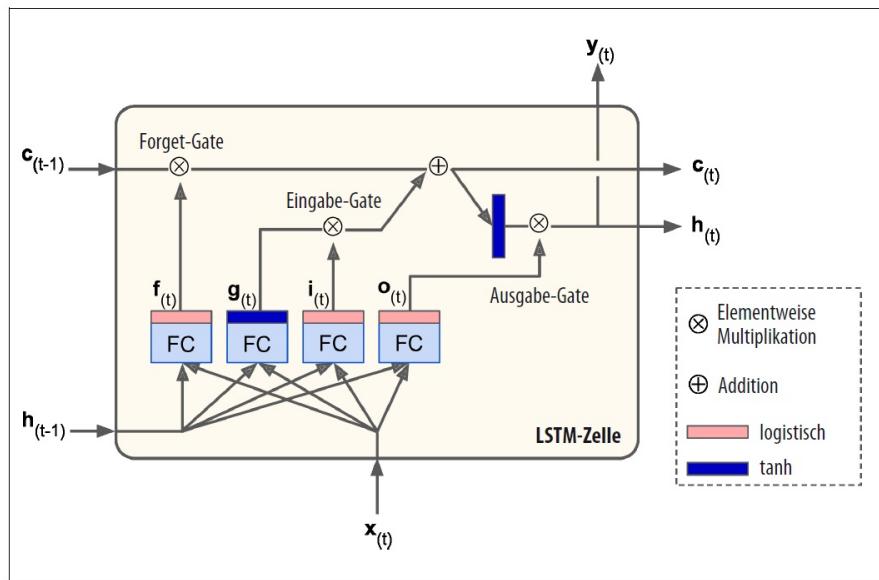


Abbildung 2.15: Aufbau einer LSTM-Zelle. Abb. 14-13 in [5]

Gewichtete Matrizen statt Vektoren und die Eingabedaten Vektoren statt einzelne Punkte, das hat den Grund, da hierbei sogenannte *batches* verwendet werden. Ein batch ist eine Zusammensetzung aus mehreren Daten. Die Daten werden also beim Training nicht einzeln in das Netz eingegeben sondern in Blöcken von Daten, welche als Vektoren dargestellt werden. Abb. 2.15 zeigt den gewöhnlichen Aufbau einer LSTM-Zelle. Der Zustand einer LSTM-Zelle ist im Gegensatz zu einer normalen Zelle, in zwei Vektoren aufgeteilt, wobei man sich $h_{(t)}$ als Kurzzeitgedächtnis und $c_{(t)}$ als Langzeitgedächtnis. Im Grundlegenden Gedanken soll

das Netz lernen, was es vergessen oder im Langzeitgedächtnis speichern soll und wie das Gedächtnis interpretiert werden kann. In jedem Zeitschritt vergisst der Langzeitzustand $c_{(t-1)}$ im *Forget Gate* einige Erinnerungen bevor ihm neue Erinnerungen, durch Addition einer vom Input-Gate ausgewählten Erinnerungen, hinzugefügt werden. Das Ergebnis $c_{(t)}$ wird anschließend direkt ausgegeben. Der Langzeitzustand wird zusätzlich nach der Addition kopiert und durchläuft die Aktivierungsfunktion

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (\text{Tangens hyperbolicus}). \quad (2.17)$$

Aus dem vom Output Gate gefilterten Ergebnis ergibt sich dabei der Zustand des Kurzzeitgedächtnisses $h(t)$ (der der Ausgabe der Zelle bei diesem Schritt $y(t)$ entspricht). Im unteren Teil der LSTM-Zelle woher die neuen Erinnerungen stammen, werden der aktuelle Eingabevektor $x_{(t)}$ und der Zustand des Kurzzeitgedächtnisses $h(t-1)$ in vier einzelne vollständig verbundene Schichten übergeben, welche unterschiedlichen Zwecken dienen. Die zweite Schicht gibt $g_{(t)}$ aus. Sie hat die gewöhnliche Aufgabe, wie in einer einfachen Zelle, die aktuellen Eingaben $x_{(t)}$ und den vorherigen Zustand des Kurzzeitgedächtnisses $h_{(t-1)}$ zu verarbeiten. Die Ausgabe dieser Schicht wird später teilweise im Langzeitgedächtnis gespeichert. Die 3 weiteren Schichten verwenden die logistische Aktivierungsfunktion

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad (2.18)$$

welche auch 'Sigmoidfunktion' genannt wird und als Ausgabe entweder 0 oder 1 ausgibt. Die Ausgaben werden in Elementweisen Multiplikationen verwendet, was die Schichten zu *Gate Controllern* macht, welche bei Ausgabe 0 das Gate schließen und bei Ausgabe 1 das Gate öffnen. Das von $f_{(t)}$ kontrollierte *Forget Gate* steuert, was aus dem Langzeitgedächtnis gelöscht wird. Das von $i_{(t)}$ kontrollierte *Input Gate* steuert, welche Informationen von $g_{(t)}$ im Langzeitgedächtnis gespeichert werden. Das von $o_{(t)}$ kontrollierte *Output Gate* steuert, welche Teile des Langzeitgedächtnis ausgelesen und ausgegeben werden.

Der Zustand des Kurz- und Langzeitgedächtnis, sowie die Ausgabe eines Zeitschritts für einen einzelnen Zeitschritt werden wie folgt berechnet:

$$i_{(t)} = \sigma(W_{xi}^T \cdot x_{(t)} + W_{hi}^T \cdot h_{(t-1)} + s_i) \quad (2.19)$$

$$f_{(t)} = \sigma(W_{xf}^T \cdot x_{(t)} + W_{hf}^T \cdot h_{(t-1)} + s_f) \quad (2.20)$$

$$o_{(t)} = \sigma(W_{xo}^T \cdot x_{(t)} + W_{ho}^T \cdot h_{(t-1)} + s_o) \quad (2.21)$$

$$g_{(t)} = \tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot h_{(t-1)} + s_g) \quad (2.22)$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \quad (2.23)$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)}) \quad (2.24)$$

Dabei gilt:

- W_{xi} , W_{xf} , W_{xo} und W_{xg} sind die Gewichtsmatrizen des Eingabevektors der Verbindungen $x_{(t)}$ zu den 4 Schichten sind.

- W_{hi} , W_{hf} , W_{ho} und W_{hg} sind die Gewichtsmatrizen des Kurzzeitgedächtnisses $h_{(t-1)}$ der Verbindungen zu den vier Schichten.
- b_i , b_j , b_o und b_g sind die Schwellwerte der vier Schichten

3 Stand der Technik

3.1 MemBrain neuronale Netze Editor und Simulator

3.1.1 Was ist MemBrain ?

MemBrain ist ein leistungsstarker graphischer Neuronale Netze Editor und Simulator für Microsoft Windows, welcher die Entwicklung neuronaler Netze von beliebiger Größe und Architektur unterstützt. Es ist momentan die einzige Software, welche die grafische Erstellung von neuronalen Netzen ermöglicht.

MemBrain wird an Universitäten zu Forschungszwecken und wird zunehmend in der industriellen Fertigungs- und Steuerungstechnik verwendet. Die Verwendung von MemBrain für den privaten oder nicht kommerziellen Zweck ist kostenlos und kann auf der offiziellen Internetseite <https://www.membrain-nn.de/> heruntergeladen werden. Für die kommerzielle Verwendung kann eine Lizenz erworben werden.

3.1.2 Technische Details

Modellbildung

Mit Membrain können im Prinzip vom einfachen, zeitinvarianten Feed-Forward-Netz bis hin zu Netzen mit diskret feuernenden Neuronen, sowie beliebigen Rückkopplungen und Laufzeiten alle Arten von neuronalen Netzen simuliert werden. Allerdings gibt es nur die Möglichkeit ein neuronales Netz aus einzelnen Neuronen zu erstellen. Es ist also nicht möglich ganze Schichten oder komplexere Schichten wie Pooling, Faltung, LSTM-Zellen etc. direkt direkt zu erzeugen. In MemBrain kann echtes Laufzeitverhalten dadurch abgebildet werden, das die Verbindungen zwischen den Neuronen (sogenannte links) beliebige logische Längen aufweisen können. In Abb.3.2 ist ein Feed-Forward-Netz im Membrain Editor dargestellt. Die Schichten werden in Membrain aus einzelnen Neuronen erstellt. Ein Eingabeneuron hat am nicht belegten Eingang einen Pfeil in Richtung des Neurons, ein Ausgabeneuron einen Pfeil in Richtung vom Neuron weg.

Lernalgorithmen

Membrain verfügt momentan über folgende Lernalgorithmen für überwachtes Lernen:

- Backpropagation mit/ohne Unterstützung für Rückkopplungslinks
- Backpropagation mit Momentum mit/ohne Unterstützung für Rückkopplungslinks

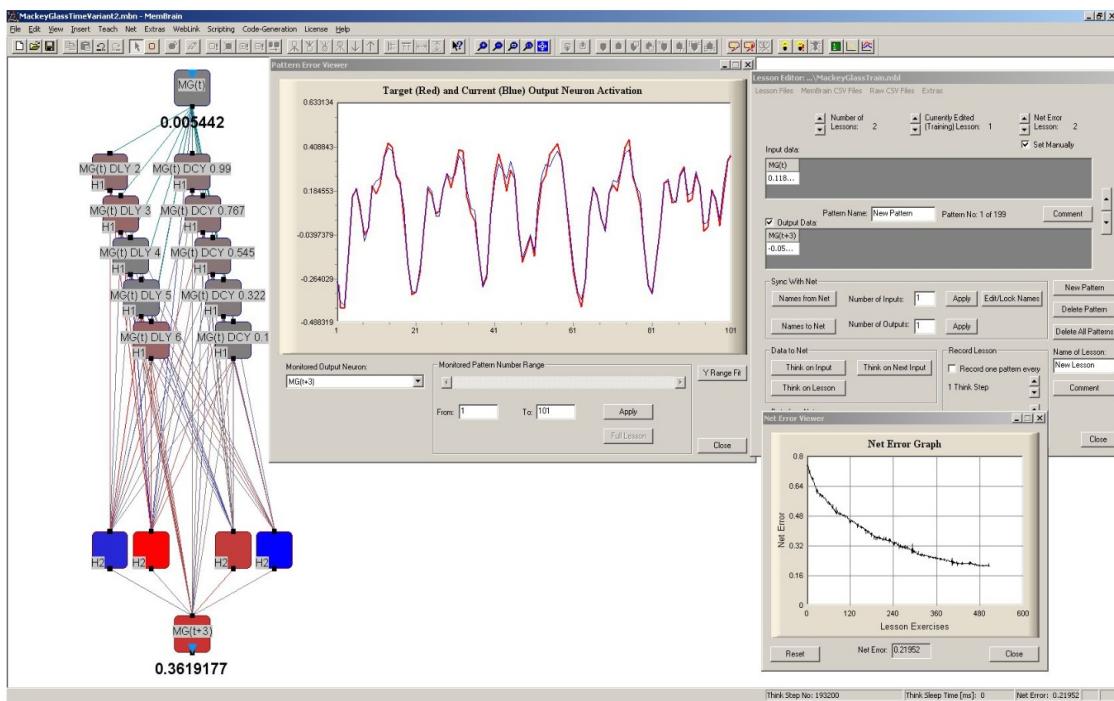


Abbildung 3.1: Screenshot von einer Membrain Anwendung. <https://www.membrain-nn.de>

- RPROP (Resilient Backpropagation) mit Unterstützung für Rückkopplungslinks
- Levenberg-Marquardt ohne Unterstützung für Rückkopplungslinks
- Cascade Correlation mit Unterstützung für Rückkopplungslinks, basierend auf Backpropagation mit Momentum oder basierend auf RPROP
- Trial and Error mit Unterstützung für Rückkopplungslinks

Für unüberwachtes Lernen ist der 'Winner Takes it All' Algorithmus für SOMs ('Self Organizing Maps') der einzige zur Verfügung stehende Lernalgorithmus für unüberwachtes Lernen.

DLL - Dynamic Link Library

Membrain verfügt über eine DLL (Dynamic Link Library) für C/C++, in der die meisten Funktionen auch verfügbar sind. Mit der MemBrain DLL können neuronale Netze, die mit MemBrain erstellt und trainiert wurden, unabhängig von der verwendeten Programmiersprache, in eigene Applikationen eingebunden werden. Zudem bietet die DLL Funktionen um neue Netze zu erstellen oder bereits existierende Netze zu editieren. Die DLL bietet folgende Funktionen:

- Zeitgleiches Laden von beliebiger Anzahl neuronaler Netzen.
- Wertzuweisung an die Eingangsneuronen.

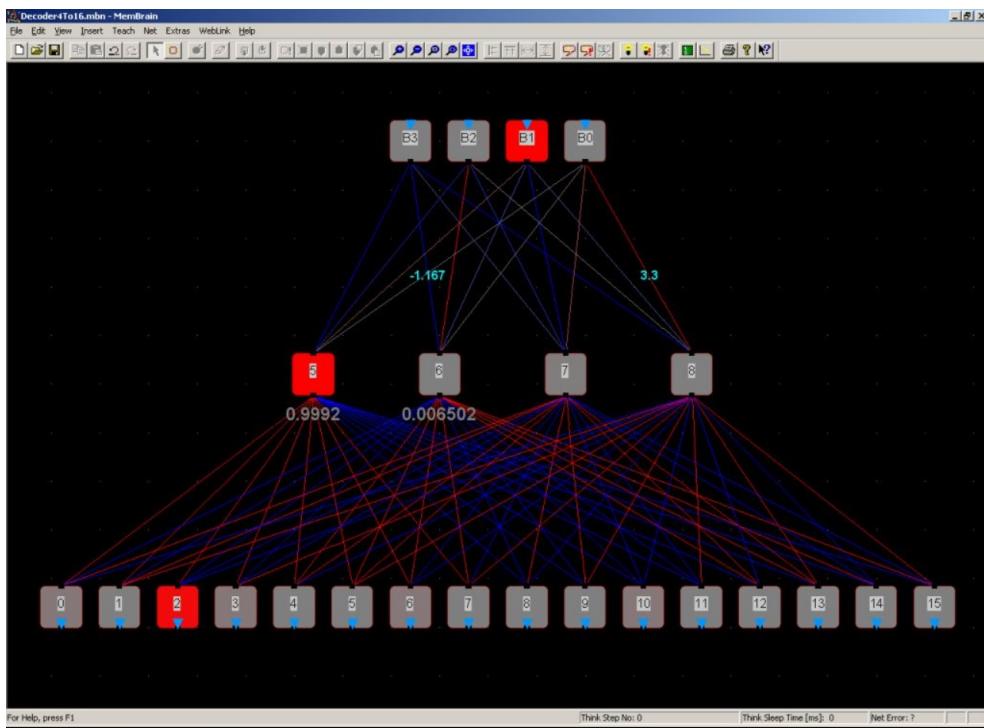


Abbildung 3.2: Screenshot von einer Membrain Anwendung in der ein Feed-Forward-Netz mit aus 3 Schichten mit 4 Eingabeneuronen 4 versteckten Neuronen und 12 Ausgabeneuronen. <https://www.membrain-nn.de>

- Simulation von beliebig vielen Schritten.
- Zurücklesen der Ergebnisse von den Ausgangsneuronen.
- Abspeichern der Netze in Dateien.
- Erstellen, Laden, Importieren, Exportieren von Datensätzen.
- Trainieren, Erstellen, Editieren der Netze.

Dadurch können Neuronale Netze, die mit MemBrain erstellt wurden durch die DLL in Produktivsysteme zu überführen, ohne MemBrain selbst zu verwenden und ohne auf das TCP/IP Interface oder die Skript-Sprache von MemBrain zurückzugreifen. Die MemBrain DLL kann direkt von Java aus, durch eine für Java-Programmierer zur Verfügung stehende JNI Wrapper-DLL und eine zugehörige Java Klasse angesprochen werden. C# Anwender können die DLL durch eine objektorientierte C# Wrapper Bibliothek nutzen. Für Python, welches momentan hauptsächlich für Maschinelles Lernen, besonders für Anwendungen neuronaler Netze verwendet wird, gibt es allerdings keine Möglichkeit die DLL zu benutzen.

Automatische Codegenerierung

Ein Code-Generator, welcher aus einem aktuell geladenen neuronalen Netz C-Code generiert, der mit jedem beliebigen C-Compiler übersetzt und so in eigene Software eingebunden werden kann, steht in MemBrain zur Verfügung.

Die folgenden Möglichkeiten zum Einsatz des erzeugten neuronalen Netzes sind durch den erzeugten Code gegeben:

- Initialisieren und Zurücksetzen des neuronalen Netzes
- Wertzuweisung an die Eingangsneuronen des neuronalen Netzes
- Ausführen von beliebig vielen Simulationsschritten
- Zurücklesen der Ergebnisse von den Ausgangsneuronen des Netzes

Scriptsprache und Debugger

MemBrain beinhaltet eine eigene Skript-Sprache, die es ermöglicht, Operationen wie das Laden, Trainieren, Validieren und Speichern von neuronalen Netzen zu automatisieren. Die Skriptsprache basiert auf der Scripting-Engine 'AngelScript', die eine C++/Java ähnliche Sprache mit starker Typisierung zur Verfügung stellt. MemBrain Skripte können durch unterschiedliche Arten wie z.B. Message-Boxen, Dialoge zum Öffnen und Speichern von Dateien, numerische und Text-Eingabefelder, Ausgaben im Trace-Fenster mit dem Benutzer interagieren. MemBrains Skriptsprache enthält Funktionen zum Erzeugen und Editieren von neuronalen Netzen auch die Entwicklung eigener evolutionärer Algorithmen ist möglich. Das Ausführen der MemBrain Skripte ist über verschiedene Wege möglich:

- Über das Menü <Scripting> in MemBrain
- Als Parameter zu MemBrain.exe von der Kommandozeile aus
- Durch ein spezielles Datei-Interface zur Laufzeit von MemBrain
- Über den externen grafischen Quellcode Debugger von MemBrain

Membrain beinhaltet zur Skriptsprache zusätzlich einen passenden Debugger, welcher folgende Leistungen bietet:

- Syntax highlighting
- Text Editor mit Blockeinrückung, Copy/Paste, Suchfunktion, Lesezeichen
- Compilieren, Bauen, Starten und Pausieren von Scripten aus dem Debugger heraus
- Direktes Navigieren über Compiler-Meldungen zum Source-Code via Mausklick
- Debug Ausgabefenster

- Variablenüberwachung
- Grafische Breakpoints
- Step Into, Step Over, Step Out
- Anhängen des Debuggers an laufendes Script
- Der Debugger auf einem anderen Rechner innerhalb des LANs laufen oder auf der selben Maschine wie MemBrain
- Starten des Debuggers von MemBrain aus und Starten von MemBrain vom Debugger aus.

Einlesen von Dateien

MemBrain bietet die Möglichkeit, Pixeldaten aus Bilddateien als Graustufenwerte einzulesen. Vor dem Einlesen und Umwandeln in Graustufen werden die Bilder optional neu gesampelt, um eine Anpassung hinsichtlich Auflösung und Format zu erzielen. Grauwertbilder sind allerdings die einzigen Dateitypen neben Zahlen für die es eine Möglichkeit gibt eingelesen zu werden. Textformate oder Audiodateien werden nicht unterstützt und müssen zur Verwendung wie sämtliche andere Daten, außer Grauwertbildern, vorverarbeitet und als Zahlen dargestellt werden.

3.2 TensorFlow

3.2.1 Was ist TensorFlow?

TensorFlow ist ein vom GoogleBrain[16] Team entwickeltes Framework zur datenstromorientierten Programmierung und fällt in die Kategorie der Open-Source Programmbibliotheken für maschinelles Lernen und künstliche Intelligenz. Die Software zeichnet sich durch ihre Leistungsfähigkeit und gute Skalierbarkeit aus. Mit Hilfe von gerichteten, zykelfreien Graphen können beliebige neuronale Netze dargestellt werden. Andere Programmbibliotheken aus dem Bereich des maschinellen Lernens werden oft mit vordefinierten Modellen geliefert. Im Gegensatz dazu ist Tensorflow in der Lage, eigene Modelle zu entwickeln und zu bearbeiten. Es ist also kein Wunderd, das die meisten heutigen Anwendungen, die auf künstlichen neuronalen Netzen basieren, mit TensorFlow entwickelt wurden. Auch in der Forschung wird TensorFlow reichlich verwendet. Tensorflow erfordert keine Übersetzung des Quellcodes in andere Programmiersprachen und ist auf unterschiedlichen Plattformen wie Smartphones, Embedded Devices, Einzelrechnern, Servern und verteilten Systemen lauffähig. Tensorflow unterstützt die Programmiersprachen Python C, C++, Go, Java, JavaScript und Swift. Von Drittanbietern gibt es weitere Bibliotheken für die Sprachen C#, Haskell, Julia, R, Scala, Rust, OCaml, und Crystal.

3.2.2 Grundlegende Funktionsweise

Die Hauptfunktionsweise von TensorFlow basiert auf dem Prinzip eines gerichteten Graphens im Sinne der Informatik. In Tensorflow werden mathematische Probleme in einen gerichteten Graphen abstrahiert. Der Graph besteht dabei aus sogenannten Knoten, welche mit gerichteten Kanten untereinander verbunden werden. Die Knoten repräsentieren in TensorFlow mathematische Operationen und Daten. Durch die Verbindung der Knoten mit den Kanten wird ein Graph erzeugt, welcher das neuronale Netz darstellt. Tensorflow ist ein Low-Level-Framework, was zwar ermöglicht jedes Detail selbst zu bestimmen, allerdings den Schwierigkeitsgrad beim Erlernen erhöht und den Quellcode unübersichtlich und kompliziert erscheinen lässt.

3.2.3 Keras

Da es für die meisten Fälle nicht nötig ist jedes Detail selbst zu bestimmen, sondern lediglich ein Netz aus bestimmten Schichten mit einer gewissen Anzahl an Neuronen und festen Eigenschaften ausreichend ist, wird hauptsächlich kein reines Tensorflow verwendet. Stattdessen wird das von Francois Chollet entwickelte Keras, welches eine Open-Source Deep-Learning-Bibliothek ist, verwendet. Keras ist geschrieben in Python und seit TensorFlow 1.4 Teil der TensorFlow Core-API. Es ist ein High-Level Framework, welches leicht erlernt werden kann und übersichtlichen gutlesbaren Code liefert. Mit Keras kann mit wenigen Zeilen Quellcode und nur ein paar Befehlen tiefe neuronale Netze erzeugt werden.

3.3 Pytorch

3.3.1 Was ist Pytorch ?

Pytorch ist ein vom Facebook Forschungsteam für künstliche Intelligenz [17] entwickelte Open-Source-Programmbibliothek. Zum einen lassen sich neuronale Netze auf Basis eines bandbasierten Autograd-Systems mit der Programmbibliothek erstellen und zum anderen lassen sich mit GPUs beschleunigte Tensor-Analysen erstellen. Dabei können oft genutzte Python-Bibliotheken wie NumPy, SciPy und Cython genutzt werden. Flexibilität und eine hohe Geschwindigkeit zeichnen Pytorch aus. Zum Austausch von Modellen mit anderen Programmbibliotheken wird ONNX unterstützt. Zudem bietet Pytorch die Möglichkeit Python Standard-Python-Debugger zu verwenden ohne das dabei Abstriche bei der Performance gemacht werden müssen. Pytorch ist im Gegensatz zu Tensorflow noch sehr neu, findet jedoch zunehmend immer mehr Anwendung in der Deep-Learning-Forschung und ist bei Entwicklern im Bereich des Natural-Language-Processing sehr beliebt. Zurzeit werden die Programmiersprachen Python und C++ unterstützt.

3.3.2 Grundlegen Funktionsweise

In PyTorch ist *torch.Tensor* die elementare Datenstruktur zur Verarbeitung und Repräsentation von Daten. Der mathematische Begriff eines 'Tensors' beschreibt die Generalisierung

von Matrizen und Vektoren, siehe Abb.3.3. In Form von mehrdimensionalen Arrays werden



Abbildung 3.3: Verdeutlichung des Mathematischen Begriffs Tensor. [18]

Tensoren in PyTorch implementiert, wobei ein Vektor nichts anderes als ein eindimensionaler Tensor (Rang 1) ist. Die Elemente der Vektoren sind Zahlen eines bestimmten Datentyps beispielsweise `torch.int32` oder `torch.float64`. In PyTorch ist ein zweidimensionaler Tensor (Rang 2) also eine Matrix und ein nulldimensionaler Tensor (Rang 0) ein Skalar. In höheren Dimensionen gibt es keine speziellen Namen mehr, man daher nur noch von Tensoren. Das Interface für PyTorch Tensoren ist stark an das Design von multidimensionalen Arrays in der Programmiersprache *Numpy* angelehnt. Genau wie NumPy werden in PyTorch Methoden, zum verändern von Tensoren und anwenden der linearen Algebra auf Tensoren, bereitgestellt. Eine performante Ausführung von Tensoroperationen auf der CPU (vor allem mit Intel-Prozessoren), wird durch die Anwendung optimierter Bibliotheken wie BLAS, LAPACK und MKL ermöglicht. Zusätzlich wird auch die Ausführung der Operationen auf NVIDIA-Grafikkarten mit Hilfe des CUDA-Toolkits und der CuDNN-Bibliothek ermöglicht. PyTorch ist im Gegensatz zu TensorFlow ein High-Level-Framework, was leichter zu erlernen ist und übersichtlicheren Quellcode generiert. Das besondere an PyTorch ist allerdings die Möglichkeit flexible Architekturen erstellen zu können, da PyTorch einen sogenannten *dynamic computation graph* verwendet. Welcher im Gegensatz zum bei TensorFlow verwendeten *static computation graph* nicht immer gleich sein muss, sondern die Anzahl an Neuronen zur Laufzeit ändern kann. Wenn in TensorFlow beispielsweise ein Convolutional Neural Network für Bildverarbeitung entwickelt wird, muss die Bildauflösung der Bilddaten immer identisch sein. Durch die richtige Verwendung in PyTorch können die Bilddaten aber auch unterschiedliche auflösen sein.

4 Methoden

4.1 Die Software

4.1.1 Aufbau

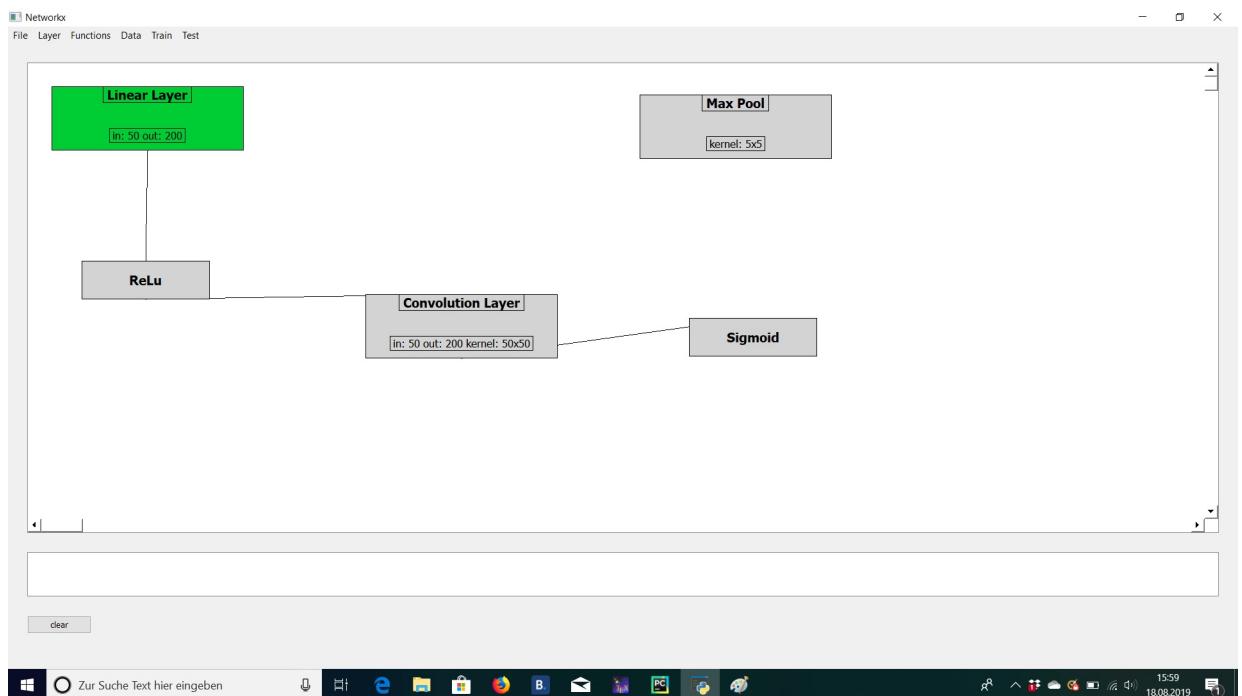


Abbildung 4.1: Screenshot einer offenen Anwendung zur Verdeutlichung des Aufbaus der Software

Abb. 4.1 zeigt eine offene Anwendung der Software. Der Grundlegende Bestandteil der Benutzeroberfläche besteht aus dem sogenannten *Worldwidget*, welches den weißen horizontal bzw. vertikal scrollbaren Bereich darstellt, und der *Konsole*, welche den weißen Bereich direkt darunter darstellt. Zudem gibt es eine Toolbar direkt am Oberen Rand der Software, welche in bestimmten Kategorien Funktionen zum Benutzen der Software bereitstellt. Dem Worldwidget können, durch die Untermenüs *Layer* und *Functions*, Bausteine für die Generierung von künstlichen neuronalen Netzen hinzugefügt werden. Im Worldwidget können die Bausteine, wie in Abb. 4.1 zu einem Graphen verknüpft werden und Anhand ihrer möglichen Einstellungen verändert werden. Die Eingabeschicht ist dabei in Grün gekennzeichnet und

kann jeder Zeit verändert werden. Die Software erstellt im Trainings- bzw. Testvorgangs automatisch aus dem grafisch erstellten Graphen ein künstliches Neuronales Netz und verarbeitet die Gewichte für den Benutzer unsichtbar im Hintergrund. Die Konsole hat die Aufgabe dem Benutzer Informationen über falsche Bedienung, den Trainingsstatus, Testergebnisse etc. zu übermitteln. Falls die Konsolengröße nicht fähig ist, alle bisherigen Ausgaben anzuzeigen, wird diese, genau wie das Worldwidget, scrollbar. Mit dem Clear Button auf linken Seite unterhalb der Konsole kann der Inhalt der Konsole gelöscht werden. Neben den Untermenüs *Layer* und *Functions*, enthält die Software noch 4 weitere Untermenüs. Das Untermenü *File*, welches Funktionen für den Import und Export von neuronalen Netzen bereitstellt. Das Untermenü *Data*, welches das Laden und Verarbeiten von Trainingsdaten ermöglicht. Das Untermenü *Train* welches Funktionen zum Training bereitstellt (Zurzeit nur eine Art des Trainings mit unterschiedlichen Einstellungen möglich) und das Untermenü *Test* Welches Funktionen zum Evaluieren und Testen bereitstellt. Die Größe des *Worldwidgets* und der *Konsole* sind, durch die Maus am Grauen Rand dazwischen, veränderbar.

4.1.2 Die Bausteine

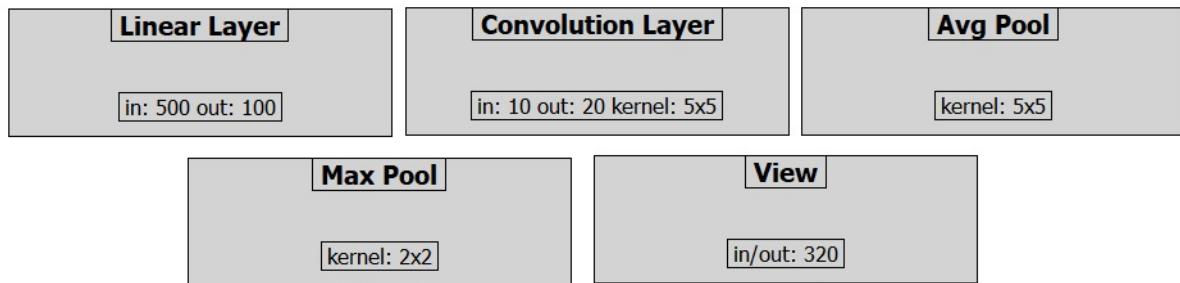


Abbildung 4.2: Veränderbare Bausteine zum erstellen neuronaler Netze in Networkx.

In Abb. 4.2 sind die aktuell vorhandenen veränderlichen Bausteine, für die Erzeugung von künstlichen Neuronalen Netzen, abgebildet. Der Baustein mit der Aufschrift *Linear Layer* ist eine Schicht vollständig verbundener Neuronen, wobei *in* die Anzahl der Eingabeneuronen und *out* die Anzahl der Ausgabeneuronen angibt. Die Anzahl der Eingabe und Ausgabeneuronen ist hierbei frei wählbar. Es handelt sich also auf dem Bild um eine Erwartete Eingabe von 100 Neuronen bzw. Werten welche vollständig mit einer Schicht aus 100 Neuronen verbunden sind.

Der nächste Baustein das *Convolution Layer* erzeugt eine zweidimensionale Faltungsschicht. In diesem Fall beschreibt *in* die Anzahl der Kanäle. Startet beispielsweise ein neuronales Netz mit einem Convolution Layer und verwendet als Eingabedaten ein Farbbild, dann muss *in* auf 3 gesetzt werden, da ein Farbbild 3 Kanäle besitzt. Genau wie *in* beschreibt *out* auch die Anzahl der Kanäle und nicht die Anzahl der Neuronen. Hierbei wird also durch die Anzahl der Kanäle entschieden, wie viele Faltungskerne in dieser Schicht verwendet werden. Der Parameter *kernel* gibt an, welche Größe die verwendeten Faltungskerne haben. Der Vorteil des *dynamic computation graph* im gegensatz zum *static computation graph* wird

im Faltungslayer gut deutlich. Die Anzahl der Neuronen wird abhängig von der Größe der Kanäle automatisch erzeugt. In dem Beispiel in Abb. 4.2 werden 10 Eingabekanäle auf 20 Ausgabekanäle abgebildet, wobei jeweils ein 5x5 Kernel verwendet wird.

Die Bausteine *Avg Pool* und *Max Pool* sind Poolingschichten. Bei *Avg Pool* handelt es sich um das Durchschnittspooling und bei *Max Pool* um das Max-Pooling. Der Parameter *kernel* gibt die Größe des verwendeten Poolingkernels an. In Abb. 4.2 handelt es sich um ein Durchschnittspooling mit 5x5 Kernel sowie ein Maxpooling mit 2x2 Kernel. Der Baustein mit dem Namen *View* ist ein besonderer Baustein, dieser verbindet die Neuronen aller Kanäle mit einer Schicht, welche die gleiche Anzahl an Neuronen in nur einen Kanal enthält. Der Parameter *in/out* Bezeichnet hier die Summe aller Neuronen der Eingabekanäle sowie die Anzahl der Neuronen des Ausgabekanals.

Die aktuell vorhandenen unveränderbaren Bausteine sind in Abb. 4.3 dargestellt. Der Bau-



Abbildung 4.3: Unveränderbare Bausteine zum erstellen neuronaler Netze in Networkx.

stein mit der Bezeichnung *ReLU* bezeichnet die Anwendung der Aktivierungsfunktion *rectified linear unit* (ReLU) auf die vorhergehende Schicht. ReLU ist als Positivteils seines Arguments definiert:

$$ReLU(x) = \max(0, x) \quad (4.1)$$

Der *Sigmoid* Baustein wendet die bereits bekannte Sigmoid Aktivierungsfunktion, welche auch *logistische Funktion* genannt wird, auf die vorherige Schicht an.

Der letzte Baustein ist der sogenannte *Dropout*, welcher zufällig einige Gewichte beim Training ausschaltet und für den nächsten Berechnungsschritt nicht berücksichtigt. Dropout reduziert die Gefahr der Überanpassung und hat sich in der Praxis als sehr effektiv erwiesen.

4.1.3 Technische Details

Die Software bietet zum aktuellen Zeitpunkt die folgenden Bausteine und Funktionen zum erstellen von künstlichen neuronalen Netzen:

- Faltungs-, Dropoutschichten sowie vollständig verbundene Schichten
- Max- und Durchschnittspooling
- Die Aktivierungsfunktionen rectified linear unit (ReLU) und logistische Funktion (Sigmoidfunktion)
- Eine Funktion um die Neuronen aus mehreren feature maps in eine featuremap zur weiterverarbeitung zu überführen
- Erstellen von Verbindungen zwischen den Schichten mit automatischer Erstellung der Verbindungen der richtigen Neuronen zwischen den Schichten, abhängig vom gewählten Typ der Schicht.

- Automatische Anwendung von beliebig vielen hintereinander ausgeführten Aktivierungsfunktionen auf den Ausgabeneuronen einer Schicht, abhängig von der Wahl der Aktivierungsfunktionen und der Art der Schicht.
- Unbegrenzte Fläche zum Hinzufügen von Bausteinen neuronaler Netze im Worldwidet.
- Automatische Erstellung eines künstlichen neuronalen Netzes aus den verbundenen Schichten innerhalb des Worldwidgets

Neben den Bausteinen und Funktionen für die Erstellung künstlicher neuronaler Netze bietet die Software die Funktionen:

- Laden und Speichern von künstlichen neuronalen Netzen und ihren aktuellen Gewichten.
- Trainieren von künstlichen neuronalen Netzen
- 2 verschiedenen Arten für das Einlesen und die automatische Vorverarbeitung von Bilddaten und deren Labels für das Training eines neuronalen Netzes zur Objekterkennung in Bildern.
- Trainieren von künstlichen neuronalen Netzen zur Klassifikation von Bildern.
- Test und Evaluierung von künstlichen neuronalen Netzen zur Klassifikation von Bildern mit einzelnen Bildern.
- Test und Evaluierung von Datensätzen beim Training sowie außerhalb des Trainings.
- Exportieren von trainierten künstlichen neuronalen Netzen in ein ausführbares Pythonskript.
- Ausgabe von Anweisungen zur Bedienung der Software, des aktuellen Trainingsstatus, der Testergebnisse etc. in der Konsole.

4.2 ein künstliches Neuronales Netz erstellen

4.2.1 MNIST-Datensatz

Im folgenden wird ein künstliches Neuronales Netz mit Networkx zur Erkennung von Handgeschriebenen Ziffern (0-9) erstellt. Es handelt sich hierbei also offensichtlich um ein Convolutional-Neural-Network zur Bildklassifizierung. Für das Training und die Evaluierung wird der MNIST-Datensatz[19], welcher auf <http://yann.lecun.com/exdb/mnist/> zum kostenlosen Download verfügbar ist, verwendet. Der MNIST-Datensatz[19] (Modified National Institute of Standards and Technology) ist ein großer Datensatz handgeschriebener Ziffern, der üblicherweise zum Trainieren verschiedener Bildverarbeitungssysteme verwendet wird. Der Datensatz für auch häufig für Schulungen und Tests im Bereich des maschinellen Lernens verwendet.



Abbildung 4.4: Einige Ziffern aus dem MNIST-Datensatz. Quelle: Abbildung 3-1 in [5]

Der Trainingdatensatz beinhaltet 60.000 und der Testdatensatz 10.000 Bilder. Die Bilder sind normalisierte Grauwertbilder mit einer fixen Größe von 28x28 Pixeln. In Abb.4.4 sind einige Ziffern aus dem Datensatz dargestellt, wobei jede Spalte aufsteigend die Zahlen 0-9 beinhaltet.

4.2.2 Das Netz erstellen

Als erstes wird ein Convolution Layer erstellt. Um ein solches dem Worldwidget hinzuzufügen muss, wie in Abb 4.5, das Untermenü *Layer* ausgewählt werden und anschließend auf den Reiter *Convolution Layer* gedrückt werden. Nun sollte das Convolution Layer am

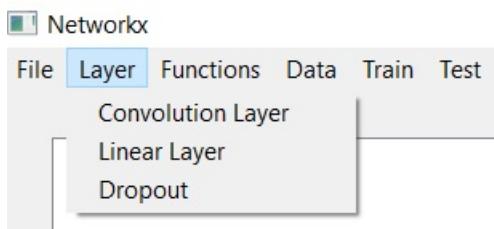


Abbildung 4.5

oberen linken Rand des Worldwidget erscheinen, wie in 2.2. Der Baustein kann so wie jeder andere auch durch gedrückt halten der Maus, auf die Fläche des Bausteins, mit der Maus verschoben werden. Um die Parameter zu verändern muss die Maus ebenfalls auf die Fläche des Bausteins gelegt werden und die rechte Maustaste gedrückt werden. Es erscheint ein Dropdownmenü, wie in Abb.4.6, welches die Ausführung der Funktionen Edit, Draw, Delete, Delete links und First Layer ermöglicht. Beim Drücken auf *Draw* kann mit der Maus, durch einen Linksklick auf ein einen anderen Baustein, eine Verbindung mit diesem erstellt

werden. Durch einen Doppelklick mit der Linken Maustaste auf einen Baustein kann ebenfalls eine Verbindung von diesem gezogen werden. Die Verbindung wird hierbei vom Ausgang des Bausteins aus welchem eine Verbindung gezogen wird, zum Eingang des Bausteins zu dem die Verbindung anschließend mit Linksklick abgeschlossen wird, erstellt. Durch drücken der linken Maustaste auf denselben Baustein kann das Erstellen einer Verbindung abgebrochen werden. Jeder Baustein kann nur eine Verbindung am Ausgang und eine am Eingang besitzen. Eine Ausnahme ist dabei das *First Layer*, welches keine Verbindung am Eingang haben kann. Mit der Funktion *Delete* wird der Baustein gelöscht. Durch *Delete Links* werden die Verbindungen des Bausteins sowohl am Eingang als auch am Ausgang gelöscht. Durch die das Auswählen von *First Layer* wird der aktuelle Baustein zum First Layer, färbt sich Grün und wird beim Erzeugen des künstlichen Neuronalen Netzes von der Software als Eingabeschicht des Netzes gedeutet. Um die Parameter eines veränderbaren Bausteins zu bearbeiten,

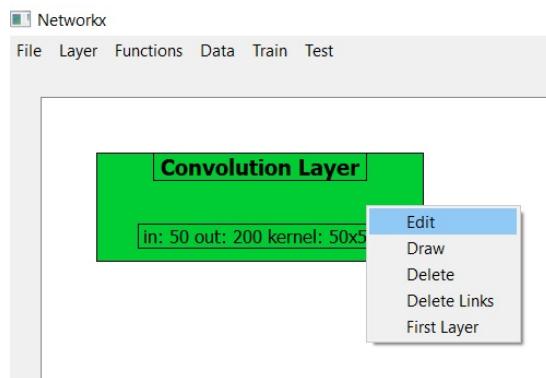


Abbildung 4.6

muss auf *Edit* gedrückt werden. Zum Verändern der Werte öffnet sich ein Popup-Fenster, wie in Abb.4.7 dargestellt, in dem die Werte gewählt werden können. Mit dem Button *Ok* werden die Werte anschließend im Baustein übernommen, durch den *Cancel* Button wird die Änderung verworfen. In diesem Fall werden der Parameter *in* auf 1 gesetzt, da es sich bei den MNIST-Daten um Grauwertbilder handelt, welche bekanntlich nur einen Kanal besitzen. Um einen Max-Pooling Baustein hinzuzufügen muss im Untermenü *Functions* der Eintrag *Max-Pool*, wie in Abb.4.8, ausgewählt werden. Der Parameter *kernel* wird genau so wie beim Convolution Layer auch geändert. In diesem Fall wird für die gewollte Architektur ein 2x2 Kernel gebraucht. Anschließend müssen das Convolutional Layer und das Max-Pooling so miteinander verbunden werden dass die Verbindung vom Ausgang des Convolution Layers zum Eingang gesetzt wird. Das geschieht wie zuvor beschrieben und sollte nach verschieben des Max-Poolings unter den Convolution Layer ungefähr so aussehen wie in Abb.4.9.

Das Netz

Das endgültige hier verwendete künstliche neuronale Netz ist in Abb.4.10 zu sehen. Das Netz kann genau wie vorher beschrieben erstellt werden.

Die Bilder werden, wie in Convolutional Neural Networks gewohnt, zuerst von einer Faltungsschicht verarbeitet. Der Parameter *in* wird hierbei auf 1 gesetzt, da der MNIST-Datensatz wie

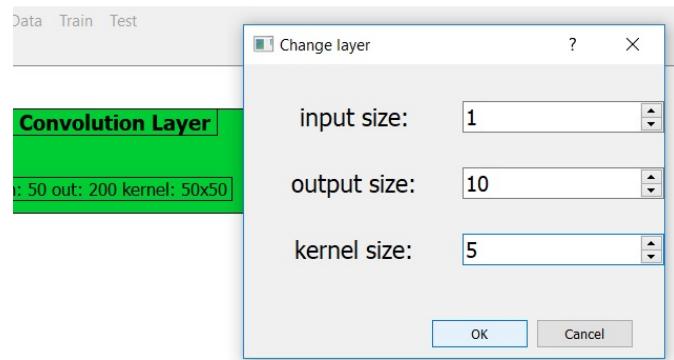


Abbildung 4.7

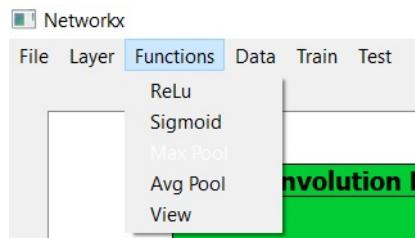


Abbildung 4.8

bereits erwähnt aus Graustufenbildern besteht (1 Kanal). In der 1. Faltungsschicht werden 10 verschiedene 5×5 Filterkerne auf die Ausgabe angewendet und es entstehen 10 Kanäle, weil der Parameter *out* den Wert 10 erhalten hat. Auf die Merkmalskarten der einzelnen Kanäle wird nun in der ersten Max-Poolingschicht ein 2×2 Filterkern angewendet, welcher nur die Karten auf ein Viertel der Ursprunggröße reduziert und nur die stärksten Aktivierungen übernimmt. Der nächste Baustein definiert dabei die Aktivierungsfunktion der aktuellen Ausgabeneuronen, hierbei wird offensichtlich *ReLU* verwendet. Im nächsten Schritt wird nun ein Dropout verwendet, welcher in jedem Trainingsschritt zufällig irgendwelche Gewichte unterdrückt um Übertraining zu verhindern. Das Verfahren wiederholt sich danach einmal mit der Ausnahme, dass in der 2. Faltungsschicht nur 2 Filterkerne der Größe 5×5 angewendet werden und der Dropout direkt in der 2. Faltungsschicht statt findet anstatt auf der 2. Poolingschicht. Der nachfolgende *View* Baustein ermöglicht nun die Verbindung der einzelnen Neuronen der 20 Merkmalskarten mit der Eingabe des vollständig verbundenen Schicht im *Linear Layer*. Um zu verstehen wieso im *View* Baustein der Parameter *in/out* auf 320 gesetzt wurde ist es am Besten sich erst einmal die Größe eines einzelnen Bildes anzuschauen. Die Bilder liegen im MNIST-Datensatz in 28×28 Pixeln, also 784 Pixeln vor. Die Bilder werden durch die Faltungskerne der 1. Faltungsschicht auf eine Größe von 24×24 , also 576 Neuronen, verkleinert. Da die erste Faltungsschicht aber 10 Filterkerne verwendet besteht die Ausgabe aus 10 Merkmalskarten mit 576 Neuronen, was insgesamt 5760 Neuronen sind. In der ersten Max-Poolingschicht werden die Merkmalskarten dann auf 12×12 , also auf 144 Neuronen und insgesamt 1440 Neuronen verkleinert. Die zweite Faltungsschicht reduziert die

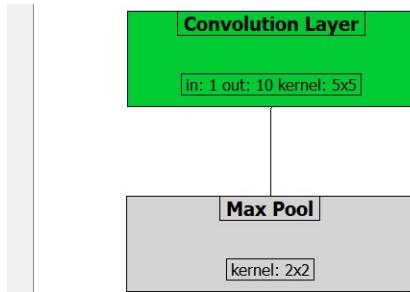


Abbildung 4.9

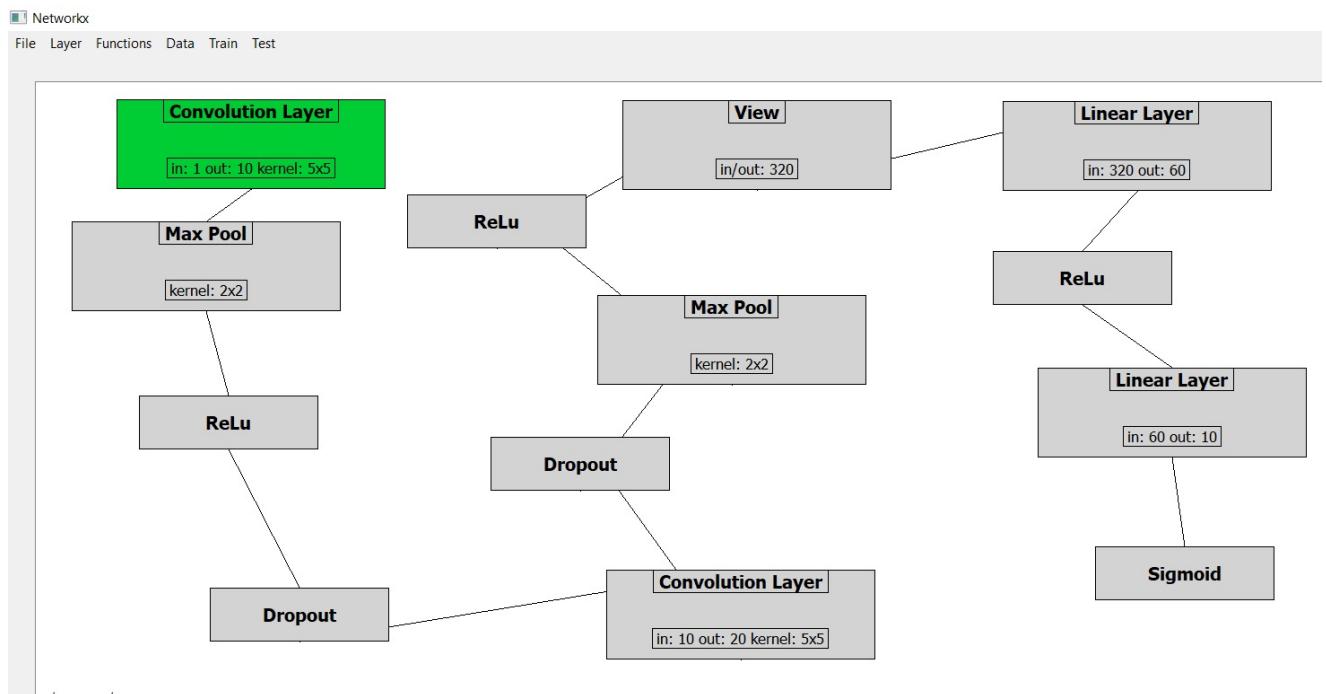


Abbildung 4.10

Anzahl Neuronen einer Merkmalskarte dann auf 8x8, also 64 Neuronen. Da die Ausgabe in der zweiten Faltungsschicht aus 20 Merkmalskarten besteht, beinhaltet sie insgesamt 1280 Neuronen. Die zweite Max-Poolingschicht verkleinert die Merkmalskarten dann genau wie die erste um ein Viertel. Eine Merkmalskarte enthält also 4x4 bzw. 16 Neuronen und die gesamte Ausgabe 320 Neuronen, genau wie in dem *View* Baustein angegeben. Bis zu diesem Schritt soll das Netz beim Training die richtigen Filter lernen, welche die Eigenschaften der 10 verschiedenen Ziffern gut genug abbilden können, um sie von einander zu unterscheiden. In den nächsten 2 vollständig verbundenen Schichten sollen die Aktivierungen der Filter ausgewertet werden und abschließend eine Entscheidung getroffen werden. Im ersten Linear Layer wird eine Schicht von 60 Neuronen dafür verwendet. Das zweite Linear Layer welches die Ausgabeschicht des gesamten Netzes darstellt enthält 10 Neuronen, eine Ausgabe für jede Ziffer, wobei die mit dem höchsten Wert schließlich als Vorhersage des Netzes gewertet.

Daten einlesen

Um das Netz für das Training mit Daten zu versorgen müssen diese eingelesen und vorverarbeitet werden. Abb. 4.11 zeigt, wie Daten eingelesen werden können. Das Untermenü *Data* enthält im aktuellen Stand 2 verschiedene Arten für die Verarbeitung von Bilddaten. Der

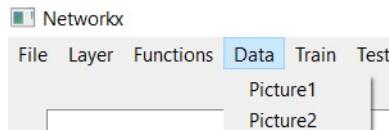


Abbildung 4.11

Grundlegende Unterschied zwischen *Picture1* und *Picture2* ist die Art wie die Daten auf der Festplatte vorliegen. Wenn eines der beiden ausgewählt wird, öffnet sich wie in Abb. 4.12 ein File-Dialog, welcher dazu auffordert, den Ordner mit den Trainingsdaten auszuwählen. Im Fall von *Picture1* müssen die Bilder für das Training wie in Abb. 4.13 vorliegen. Hierbei

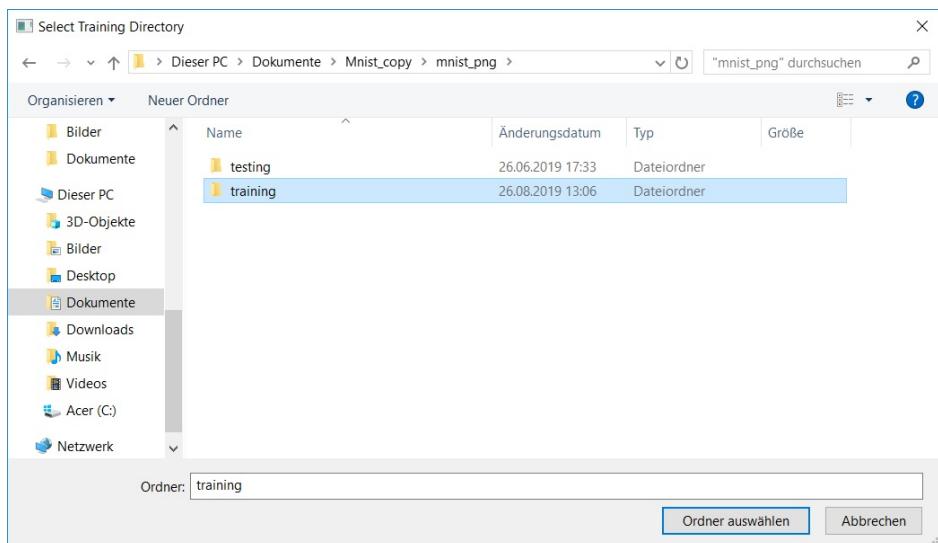


Abbildung 4.12

sind die Bilder anhand ihrer Labels in eigene Unterordner sortiert. In diesem Fall alle Nullen im Ordner *Zero*, alle Einsen im Ordner *One* und so weiter. Das neuronale Netz übernimmt später beim Trainieren für die Vorhersage genau die Namen der einzelnen Ordner als Labels. Bei *Picture2* sieht das ganze etwas anders aus. Hier sind alle Trainingsdaten in einem Ordner enthalten, wie in Abb. 4.14 dargestellt. Die Labels der einzelnen Bilder sind dabei im Namen des Bildes enthalten, welcher bei *Picture1* irrelevant ist. Nach dem auswählen der Trainings- und Testdaten, öffnet sich ein weiteres Popup-Fenster in dem dann die einzelnen Kategorien für die Labels eingetragen werden können. Im Fall von Abb. 4.14 handelt es sich um den sogenannten *Dogs vs. Cats* Datensatz, welcher auf <https://www.kaggle.com/c/dogs-vs-cats> kostenlos zum Download zur Verfügung steht. Dieser besteht aus 25.000 Bildern im

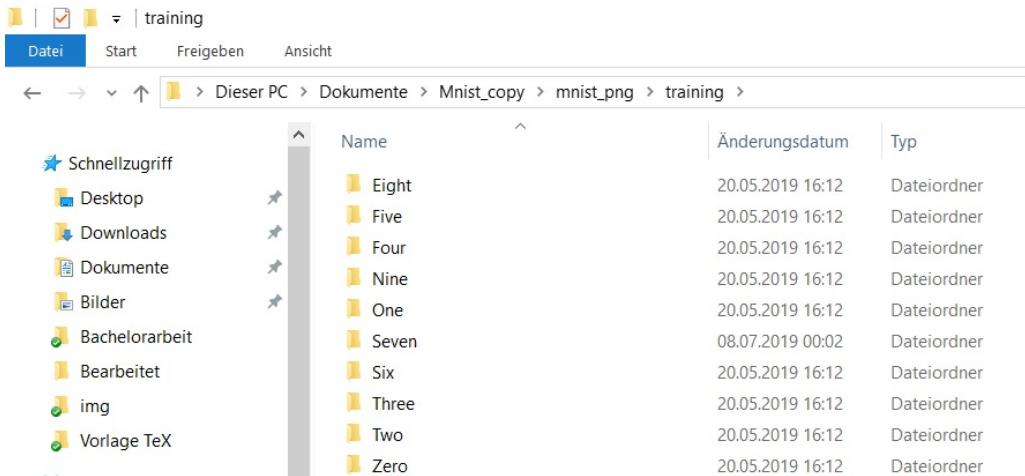


Abbildung 4.13

Trainingsdatensatz und aus 12.500 Bildern im Testdatensatz, welche zur Hälfte Bilder von Hunden und zur Hälfte Bilder von Katzen in unterschiedlichen Perspektiven und Größen enthält. Die Titel der Bilder enthalten hierbei immer den Namen der Klasse und die Zahl des Bildes, es müssten im folgenden um diesen Datensatz zu verarbeiten die Klassen *cat* und *dog* angegeben werden. Die Software weiß beim trainieren dann jedem Bild automatisch die passende Klasse als Label zu, sofern der Name einer Klasse im Titel enthalten ist. Nach

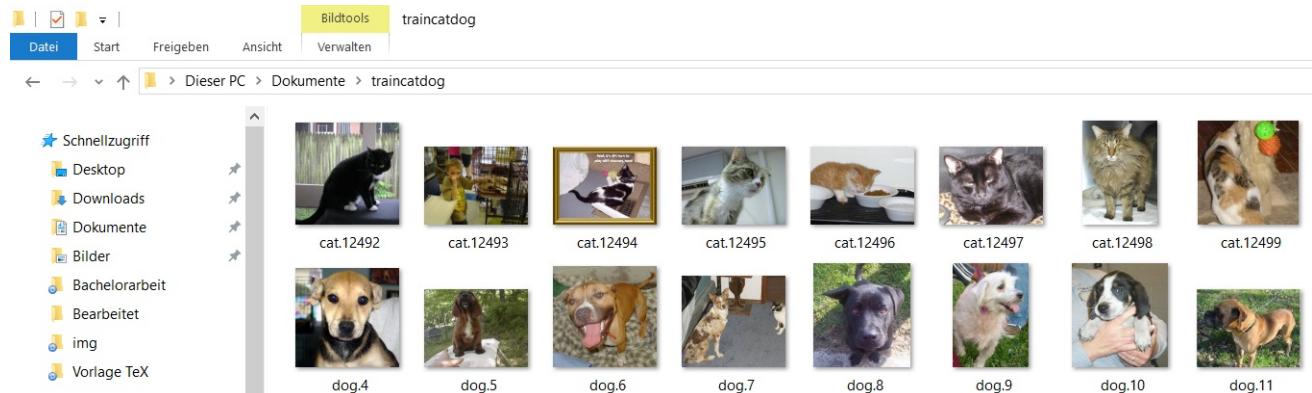


Abbildung 4.14

dem auswählen des Trainingsdatensatzes, öffnet sich wieder ein File-Dialog, welcher nun den Testdatensatz fordert. Für diesen gilt dasselbe Format, wie für den Trainingsdatensatz. Als letztes öffnet sich dann das bereits erwähnte Popup-Fenster, welches in Abb.4.15 dargestellt ist. Der Parameter *Resize* gibt an, in welcher Größe die Bilder ins Netz geschickt werden sollen. Der Zahlenwert gibt dabei die Pixelgröße der Bilder in X und Y Richtung an. Der Rand der Bilder über diesen Bereich um das Zentrum herum werden dann durch einen sogenannten *Crop* abgeschnitten und nicht weiter verwendet. Für das hier verwendete Netz wird 28 gewählt, da die Bilder schon alle dasselbe Format haben. Es ändert sich für

den MNIST-Datensatz in diesem Fall also nichts. Bei Datensätzen wie dem *Dogs vs Cats* Datensatz und einer ebenso statischen Architektur, wie der hier verwendeten, ist allerdings ein *Resize* notwendig, da die Bilder in unterschiedlichen Formen und Größen vorliegen. Der zweite Parameter die *batch size* gibt an wie viele Trainingsdaten in einem *Batch* pro Trainingsschritt durch das Netz geschickt werden sollen. Die letzten beiden Parameter *normalize mean* und *normlize std*, sind optional. Durch die Angabe beider Parameter werden die Bilder normalisiert, das ist sinnvoll, falls die Bilder stark über- oder unterbelichtet sind. Das hat den Effekt, dass der hellste Punkt nahezu weiß und der dunkelste Punkt schwarz abgebildet wird. Der Parameter *normalize mean* gibt hierbei im stochastischen Sinne den Erwartungswert des gesamten Datensatzes und *normalize std* die Standardabweichung eines einzelnen Bildpunktes an. Bei Grauwertbildern muss jeweils nur ein Wert angegeben werden. Bei Farbbildern allerdings muss für jeden Kanal (RGB in genau dieser Reihenfolge) ein eigener Wert angegeben ermittelt werden und durch Komma getrennt eingegeben werden. Da der MNIST-Datensatz bereits normalisiert ist, würde es auch nichts bringen erneut zu normalisieren. Die Eingabefelder bleiben also leer! Für den dogs vs cats Datensatz wäre aber auch eine Normalisierung vorteilhaft. Durch drücken auf den *OK* Button werden schließlich die Daten

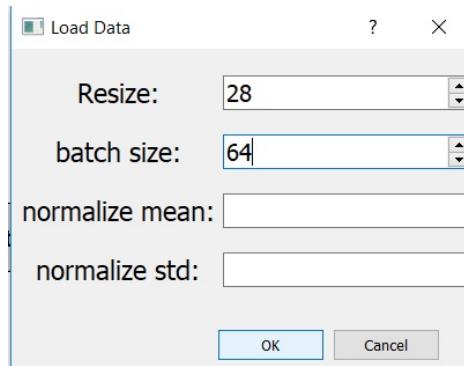


Abbildung 4.15

geladen und verarbeitet. Der aktuelle Status der Verarbeitung wird dabei in der Konsole, wie in Abb.4.16, ausgegeben. Die Ausgabe in der Konsole findet für jeden geladenen Batch statt, wobei die Batchnummer des aktuell verarbeiteten Batches, sowie der Prozentsatz des geladenen Datensatzes angegeben wird.

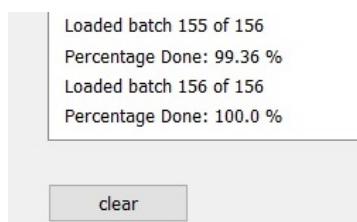


Abbildung 4.16

Trainieren

Sobald die Daten geladen und fertig verarbeitet sind, kann das Training, durch drücken auf den Button *Train* in der Toolbar, gestartet werden. Es öffnet sich wieder ein Popup-Fenster, wie in Abb. 4.17, in dem die Eigenschaften des Trainingsvorgangs festgelegt werden können. Der erste Parameter *optimizer* legt fest welches Trainingsverfahren benutzt werden soll.

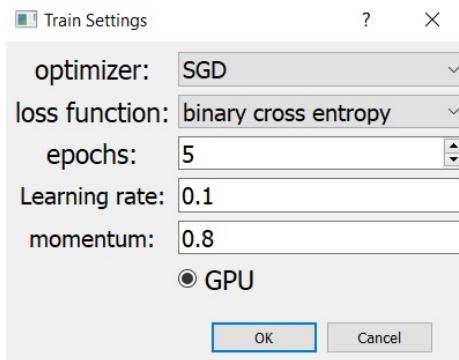


Abbildung 4.17

Zurzeit gibt es den schon bekannten *stochastischen Gradientenabstieg* (SGD) und den sogenannten *Adam*, welcher in PyTorch zur Verfügung steht. Für den MNIST-Datensatz wurde hier SGD gewählt, was allerdings keinen besonderen Grund hat. Der zweite Parameter *loss function*, gibt die verwendete Fehlerfunktion an. Auch hier gibt es bereits 2 Optionen *binary cross entropy* und *negativ log likelihood*, welche ebenfalls in PyTorch enthalten sind und auf die im Rahmen dieser Bachelorarbeit nicht weiter eingegangen wird. Das Hier *binary cross entropy* gewählt wurde hat ebenso keinen besonderen Grund. Die Anzahl der Epochen, also die Anzahl an Trainingsdurchläufen aller Trainingsdaten, wird über den Parameter *epochs* bestimmt. 5 Durchläufe sollten für den MNIST-Datensatz bei seiner Größe reichen um eine Genauigkeit von über 90 % zu erreichen in anderen Datensätzen wie dem Dog vs Cats Datensatz, würden Aufgrund der hohen Komplexität deutlich mehr Durchläufe benötigt werden. Die *learning rate* bestimmt hierbei die bereits bekannte Lernrate, welche den Trainingsfortschritt festlegt. Das Momentum ist eine zusätzliche Option von Pytorch für den stochastischen Gradientenabstieg, da dieses eine Version mit Momentum verwendet. Worauf allerdings im weiteren Verlauf auch nicht mehr eingegangen wird, da dies den Rahmen der Bachelorarbeit sprengen würde. Zum Schluss kann noch ein Häkchen neben *GPU* gesetzt werden, das bewirkt, dass die Trainingsdaten und das künstliche neuronale Netz auf die GPU geladen werden und das Training dort ausgeführt wird. Diese Funktion erfordert allerdings eine CUDA fähige Grafikkarte. Nach drücken des *OK* Buttons, startet schließlich ein weiter File-Dialog, welcher nach einem Zielordner verlangt, in dem nach jeder Epoche das Netz, sowie die Gewichte in separaten Dateien gespeichert werden. Danach startet dann der Traingsprozess. Der Trainingsfortschritt wird dabei für jede Epoche, wie in Abb.4.18, in der Konsole ausgegeben. Es ist zu beachten, dass das Training bei Epoche 0 gestartet wird. Zu jedem Trainingsschritt wird der aktuelle Fehler ausgegeben. Am Ende jeder zu trainierenden Epoche stoppt das Training kurz um das Testdatensatz zu evaluieren. Falls

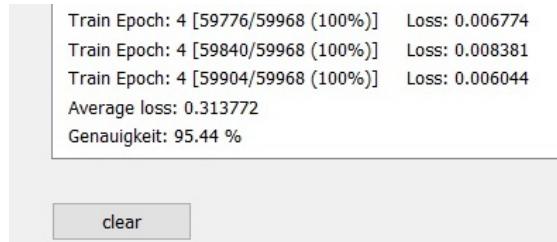


Abbildung 4.18

GPU gewählt wurde, wird auch dieses zuerst auf die Grafikkarte geladen. Die Genauigkeit der Vorhersage des neuronalen Netzes auf dem Testdatensatz wird dann, neben dem durchschnittlichen Fehler, ebenfalls in der Konsole ausgegeben. Das neuronale Netz zur Erkennung handschriftlicher Ziffern erreicht, wie in Abb. 4.18 zu sehen, nach Abschluss des Trainings eine Genauigkeit von 95,44%, was ohne Optimierung ein wirklich gutes Ergebnis ist.

Evaluierung/Test

Um das trainierte Netz nochmal mit anderen Daten zu testen, existiert ein Untermenü *Test* in der Toolbar, welches bisher die Funktionen *Testset*' und *Picture* zur Verfügung hat, was in Abb. 4.19 zu sehen ist. Durch drücken auf Testset wird ein geladenes Testset durch das



Abbildung 4.19

Netz geschickt und evaluiert. Das Ergebnis wird anschließend in der Konsole ausgegeben. Um die Vorhersage für ein einzelnes Bild zu machen muss *Picture* gewählt werden. Hierbei

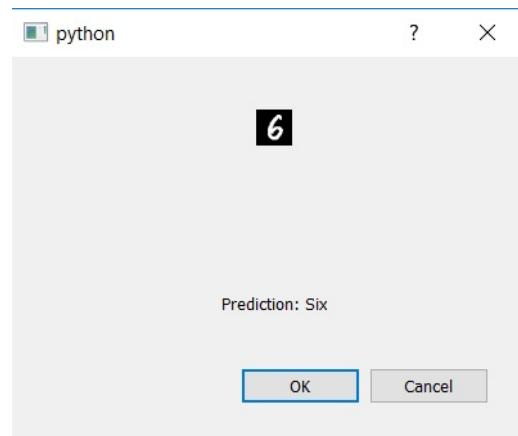


Abbildung 4.20

öffnet sich ein ähnliches Popup-Fenster, wie bereits für das Einlesen und Vorverarbeiten des Trainings- und Testdatenatzes in Abb.4.15, in dem das Bild anschließend vorverarbeitet werden kann. Zudem öffnet sich wieder ein Filedialog welcher das Bild öffnen kann. Hierbei wird kein Label benötigt, da das Netz hier einfach nur eine Vorhersage trifft. Das Bild und die anschließende Vorhersage wird dann in einem neuen Popup-Fenster, wie in Abb.4.20, angezeigt. Offenstichtlich wurde ein Bild welches eine Handgeschriebene 6 darstellt gewählt und zusammen mit der Vorhersage *Six* des trainierten Netzes im Popup-Fenster angezeigt.

Speichern, Laden, Exportieren

Beim Training wird das künstliche neuronale Netz und seine Gewichte, wie bereits erwähnt, nach jeder Epoche gespeichert. Abb.4.22 zeigt die gespeicherten Dateien aus den ersten 3 Epochen. Die Gewichte werden dabei in einer .pt Datei gespeichert und das Grafische Netz in

 weights_epoch_0.pt	26.06.2019 22:16	PT-Datei	118 KB
 weights_epoch_0_Network.nx	26.06.2019 22:16	NX-Datei	1 KB
 weights_epoch_1.pt	26.06.2019 22:17	PT-Datei	118 KB
 weights_epoch_1_Network.nx	26.06.2019 22:17	NX-Datei	1 KB
 weights_epoch_2.pt	26.06.2019 22:19	PT-Datei	118 KB
 weights_epoch_2_Network.nx	26.06.2019 22:19	NX-Datei	1 KB

Abbildung 4.21

einer .nx Datei. Beim späteren Laden ist darauf zu achten, das beide Dateien sich innerhalb des selbe Ordners befinden und die Namen nicht geändert wurden. Zum Laden des Netzes mit seinen Gewichten muss nur die .nx Datei geladen werden. Um ein Netz in der Software zu öffnen muss im Untermenü *File*, wie in Abb. 4.21 dargestellt, die Funktion *Load Network* verwendet werden. Es öffnet sich ein File-Dialog, indem die .nx Datei ausgewählt werden

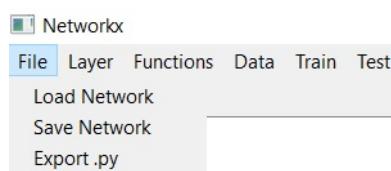


Abbildung 4.22

muss. Anschließend ist das Netz im Worldwidget sichtbar und kann, wie bisher beschrieben, verwendet werden. Um ein Netz ohne seine Gewichte zu speichern, steht die Funktion *Save Network* zur Verfügung. Hierbei öffnet sich wieder ein File-Dialog, in dem der Speicherort, sowie der Name der .nx Datei gewählt werden kann. Das Netz kann ohne Gewichte, genau wie das Netz mit Gewichten durch Load Network, geladen werden. Mit *Export.py* kann ein bereits trainiertes Netz als Python Skript exportiert werden. Allerdings müssen dafür neben dem Netz und seinen Gewichten auch Daten geladen werden, da das Skript empfangene Daten direkt umwandelt um sie dann durch das neuronale Netz zu schicken. Nachdem auf

Export.py gedrückt wurde, öffnet sich wieder ein File-Dialog in dem der Name der .py Datei und ihr Speicherort gewählt werden muss. Das Pythonskript sowie eine .pt Datei, welche die Gewichte des neuronalen Netz speichert, werden anschließend im gewählten Ordner, wie in Abb.4.23 gespeichert. Das Pythonskript kann vom Prinzip her leicht in anderen Quellcode integriert und benutzt werden, da nicht der gesamte Code verstanden werden muss. Zeile

 Thesis_MNIST	28.08.2019 12:39	JetBrains PyCharm ...	3 KB
 Thesis_MNIST_weights.pt	28.08.2019 12:39	PT-Datei	118 KB

Abbildung 4.23

integriert und benutzt werden, da nicht der gesamte Code verstanden werden muss. Zeile

```

8   path_weights = 'C:/Users/Pasca/Dropbox/Bachelorarbeit/Thesis_MNIST_weights.pt'
9   path_picture = 'Your Path here'
10  resize = 28
11
12  categories = ['Eight', 'Five', 'Four', 'Nine', 'One', 'Seven', 'Six', 'Three', 'Two', 'Zero']
13  transform = transforms.Compose([transforms.Resize(resize),
14                                transforms.CenterCrop(resize),
15                                transforms.ToTensor()])

```

Abbildung 4.24

8-15 des Quellcodes sind in Abb. 4.24 zu sehen. In Zeile 8 wird der Dateipfad der .pt Datei mit den Gewichten des neuronalen Netz angegeben und in der Variable *path_weights* als String gespeichert. Sollte die Datei verschoben werden, muss natürlich auch der Dateipfad an dieser Stelle geändert werden. Um Ein Bild von der Festplatte einzulesen muss in Zeile 9 der String *Your Path here* gegen den Dateipfad des Bildes ersetzt werden. In Zeile 12 wird in der Variable *categories* die einzelnen Labels in einer Liste gespeichert. Die Namen der einzelnen Labels können zwar geändert werden, allerdings muss die Reihenfolge bestehen bleiben. Die restlichen Zeilen sind hierbei für die Benutzung eher unbedeutend und beziehen sich nur auf die Datenverarbeitung. Die Vorhersage des neuronalen Netzes wird in Zeile 60 konstruiert, welche in Abb.4.25 dargestellt ist, wobei die vorhergesagte Kategorie über den Index der Liste *categories* ausgegeben wird, weshalb die Reihenfolge nicht verändert werden darf. die Variable *prediction*, speichert und generiert dabei den Index der Kategorie,

```

59      prediction = out.data.max(1, keepdim=True)[1].item()
60      print(categories[prediction])

```

Abbildung 4.25

welche das Netz vorhergesagt hat.

5 Ergebnis und Ausblick

Das Ziel einen Baukasten basierend auf PyTorch in Form einer grafischen Benutzeroberfläche zu entwickeln, mit dem aus einzelnen Bausteinen ein neuronales Netz erzeugt werden kann, ist wie im Kapitel zuvor beschrieben erreicht worden. Die Software wurde so entwickelt, das mit wenig Aufwand neue Bausteine aus dem breiten Pool von PyTorch aufgenommen werden können. Beim Entwickeln der Software wurde klar, dass die hauptsächliche Schwierigkeit nicht darin liegt ein neuronales Netz grafisch einfach zu erstellen, sondern die Art der Datenverarbeitung für jede Art von Problem angepasst werden muss und somit einen sehr großen Aufwand für die Entwicklung darstellt. Die Software bietet zum aktuellen Stand nur die Möglichkeit Daten in Form von Bildern für die Klassifizierung von Objekten zu verarbeiten. Allerdings wurde die Software so programmiert, dass einfach neue Funktionen zur Verarbeitung von Daten hinzugefügt werden können. Der Aufwand Funktionen für die Verarbeitung für sämtliche Arten von Daten und ihrer Problemklasse zu erstellen ist für eine Person aufgrund des hohen Aufwands nur sehr schwer umsetzbar. Mit mehreren Leuten und etwas Zeit sollte das aber zu schaffen sein.

Der Code für die Erzeugung des neuronalen Netzes funktioniert zwar universal für jedes neuronale Netz, der Voraussetzung entsprechend ein gültiges neuronales Netz zu sein, jedoch müsste der Code für beispielsweise sogenannte Encoder-Decoder Architekturen, wie sie typisch für z.B. die maschinelle Übersetzung sind, nochmal überarbeitet werden. Zudem würde die hier erstellte Software aufgrund der wenigen Freiheit im Bezug auf die Verarbeitung von Daten zwar das Erstellen und Trainieren von neuronalen Netzen extrem erleichtern, jedoch können dann auch nur Problemtypen gelöst werden, für welche eine Funktion zur Verarbeitung der Daten zur Verfügung steht. Das bedeutet, dass die Software für die Forschung eher ungeeignet wäre. Die Zielgruppe richtet sich also klar an Entwickler, welche mit bekannten Verfahren des maschinellen Lernens, speziell mit neuronalen Netzen, Lösungen entwickeln. Man könnte die Software eventuell um die Möglichkeit erweitern selbst die Daten zu verarbeiten oder eine Objektorientierte Skriptsprache hinzufügen, durch welche mehr Freiraum gegeben werden könnte. Allerdings würde dies den Schwierigkeitsgrad zum bedienen und erlernen der Software deutlich erhöhen. Das hier gedachte Konzept würde dies allerdings ausschließen und die Software würde im Grunde so bleiben wie sie ist, mit dem Unterschied das neben Bildern auch Ton, Text und sämtliche anderen vorstellbaren Datenformen einlesbar sind. Dazu soll der Baukasten mit allen in PyTorch zur Verfügung stehenden Funktionen und Layern ergänzt werden. Die Software soll dann für jede zum bisherigen Stand der Technik möglichen Art der Nutzung von neuronalen Netzen ein neuronales Netz erzeugen können, sowie eine Funktion zum einlesen und Verarbeiten der Daten für dieses Problem anbieten, sodass das Netz erstellt, trainiert und evaluiert werden kann und anschließend leicht in bestehende Systeme zur Nutzung integriert werden kann.

Literaturverzeichnis

- [1] KONEN, Zielke: Vorlesung: Bildverarbeitung und Algorithmen. In: *SS06* (2006)
- [2] MARTIN HERRMANN, Philipp G.: point of interest. In: *SS09 Prosem Votraege* (2009)
- [3] PETARV: 2D Convolution. In: *github* (2016)
- [4] RAWAT, Waseem ; WANG, Zenghui: Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. In: *Neural Computation* 29 (2017), 06, S. 1–98. http://dx.doi.org/10.1162/NECO_a_00990. –
- [5] GÉRON, Aurélien: *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme*. O'Reilly, 2018
- [6] GIBNEY, Elizabeth: Google AI algorithm masters ancient game of Go. In: *Nature News* 529 (2016), Nr. 7587, S. 445
- [7] RASHID, Tariq: *Neuronale Netze selbst programmieren: ein verständlicher Einstieg mit Python*. O'Reilly, 2017
- [8] ROSENBLATT, Frank: The perceptron: a probabilistic model for information storage and organization in the brain. In: *Psychological review* 65 (1958), Nr. 6, S. 386
- [9] SHAMAR, Sagar: What the Hell is Perceptron? The Fundamentals of Neural Networks. In: *towardsdatascience* (2017). <https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53>
- [10] LECUN, Yann ; BOSER, Bernhard ; DENKER, John S. ; HENDERSON, Donnie ; HOWARD, Richard E. ; HUBBARD, Wayne ; JACKEL, Lawrence D.: Backpropagation applied to handwritten zip code recognition. In: *Neural computation* 1 (1989), Nr. 4, S. 541–551
- [11] KOUSHIK, Jayanth: Understanding convolutional neural networks. In: *arXiv preprint arXiv:1605.09081* (2016)
- [12] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: Spatial pyramid pooling in deep convolutional networks for visual recognition. In: *IEEE transactions on pattern analysis and machine intelligence* 37 (2015), Nr. 9, S. 1904–1916
- [13] HOCHREITER, Sepp ; SCHMIDHUBER, Jürgen: Long short-term memory. In: *Neural computation* 9 (1997), Nr. 8, S. 1735–1780

- [14] SAK, Hasim ; SENIOR, Andrew ; BEAUFAYS, Françoise: Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. In: *arXiv preprint arXiv:1402.1128* (2014)
- [15] ZAREMBA, Wojciech ; SUTSKEVER, Ilya ; VINYALS, Oriol: Recurrent neural network regularization. In: *arXiv preprint arXiv:1409.2329* (2014)
- [16] ABADI, Martín ; BARHAM, Paul ; CHEN, Jianmin ; CHEN, Zhifeng ; DAVIS, Andy ; DEAN, Jeffrey ; DEVIN, Matthieu ; GHEMAWAT, Sanjay ; IRVING, Geoffrey ; ISARD, Michael u. a.: Tensorflow: A system for large-scale machine learning. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, S. 265–283
- [17] PASZKE, Adam ; GROSS, Sam ; CHINTALA, Soumith ; CHANAN, Gregory: Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. In: *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration* 6 (2017)
- [18] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep learning*. MIT press, 2016
- [19] LECUN, Yann: The MNIST database of handwritten digits. In: <http://yann.lecun.com/exdb/mnist/> (1998)