Machine Learning und tiefe neuronale Netze mit TensorFlow

David Baumgartner

BACHELORARBEIT

Nr. XXXXXXXXXX-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Januar 2017

Diese Arbeit entstand im Rahmen des Gegenstands

. . . . . .

im

Wintersemester 2016/17

Betreuer:

Stephan Dreiseitl, FH-Prof. PD DI Dr.

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 14. Januar 2017

David Baumgartner

Inhaltsverzeichnis

Erklärung **iii**

Kurzfassung **vi**

Abstract **vii**

1 Einleitung **1**
1.1 Motivation 1
1.2 Problemstellung 1
1.3 Zielsetzung 2

2 Begriffe im Maschinellen Lernen **3**
2.1 Data Science 3
2.2 Machine Intelligence 4
2.3 Machine Learning 4
2.4 Neuronale Netzwerke 4
2.4.1 Neuron 4
2.5 Ebenen/Layer 6
2.6 Informationen Merken und Wiedererkennung 6
2.7 Konvergieren im Maschinellen Lernen 7
2.8 Backpropagation 7
2.9 Trainieren 8
2.10 Allgemeine Probleme 8
2.11 Domänenklassen 9
2.11.1 Clustering 9
2.11.2 Regression 9
2.11.3 Klassifikation 9
2.11.4 Predict 10
2.11.5 Robotics 10
2.11.6 Computer Vision 10
2.12 Neuronale Netzwerktypen 10
2.12.1 FeedForward 11
2.12.2 Self-Organizing Map 11

2.12.3 Hopfield Neuronal Network 11
2.12.4 Boltzmann Machine 12
2.12.5 Deep FeedForward 12
2.12.6 NEAT 12
2.12.7 Convolutional Neural Network 12
2.12.8 Recurrent Network 13
2.13 Domänen und Typen Matrix 13
2.14 Optimierung 14
2.15 Trainingsgeschwindigkeitssteigerung 15

3 TensorFlow **17**
3.0.1 Graphs/Dataflowgraph 18
3.0.2 Datenfluss Programmierung 19
3.0.3 Operation 19
3.0.4 Sessions 19
3.0.5 Tensor 20
3.0.6 Hyperparameter 20
3.1 Bibliotheksinhalt 20
3.1.1 Datentypen 20
3.1.2 Operationen 20
3.1.3 TensorBoard 31

4 Facial Keypoints Detection **37**
4.1 Ausgangssituation 37
4.2 Vorbereitung 38
4.2.1 Daten vorbereiten und normalisieren 39
4.2.2 Evaluations- und Errorfunktion 40
4.3 Neuronale Ebenen vorbereiten 41
4.4 Neuronale Ebenen verknüpfen 42
4.5 Trainieren 43
4.6 Validierungsergebnisse 44
4.7 Ergebnis 45
4.8 Graphenvisualisierung 49
4.9 Verbesserungen 49

5 Zusammenfassung & Ausblick **51**
5.1 Zusammenfassung 51
5.2 Ausblick 52

Quellenverzeichnis **53**
Literatur 53

Kurzfassung

NN sind seit Jahren vorhanden Zeitlich bedingt nicht umsetzbar da technische Voraussetzung nicht erfüllt werden immer mehr eingesetzt Unterstützung im Alltagsleben

Ziel der Arbeit eine Einführung in die Welt der NNN mit Hilfe eines Frameworks

Abstract

Kapitel 1

Einleitung

1.1 Motivation

Kaum ein Gebiet existiert so lange wie maschinelles Lernen und erlebte in den letzten 20 Jahren einen Aufschwung in den Punkten Relevanz, Weiterentwicklung und Alltagstauglichkeit wie kein anderes Gebiet der Informationstechnologie. Ein Umstand dafür ist unter anderem die technische Voraussetzung große Systeme zu entwickeln und diese auch der Menschheit zugänglich zu machen. Im Speziellen wurde hierbei der Fokus auf neuronale Netzwerke gelegt, welche noch weiter erforscht werden, aber auch schon im Einsatz sind.

Die großen Datenmengen, die in den letzten Jahren zu verarbeiten und zu analysieren sind, stellen ein Problem dar. So befindet sich kein Mensch in der Lage, effizient Millionen an Datensätzen zu analysieren und Zusammenhänge darin zu finden und dies in adäquater Zeit zu tun.

Im Zuge dieser Arbeit sollen die Grundlagen im Bereich des maschinellen Lernens, im Speziellen mit neuronalen Netzwerken näher gebracht werden. Dabei soll gezeigt werden, wie weit sich diese in der Praxis einsetzen lassen.

1.2 Problemstellung

Die grundlegende Problemstellung lässt sich mit folgender Frage beschreiben:

Wie weit ist TensorFlow als Bibliothek im Gebiet des maschinellen Lernens in praktischen Fällen einsatzfähig?

In dieser Frage stecken mehrere nichttriviale Punkte.
Ein Punkt stellt das Gebiet des maschinellen Lernens generell dar, sowie die

praktische Umsetzung von Problemstellungen. Im Speziellen bieten neuronale Netzwerke neue Möglichkeiten, Probleme in der Informationstechnologie zu lösen, sowie Vorgänge in der Natur besser zu verstehen.

Zum anderen die Frage, was ist TensorFlow, wofür steht und wofür kann dies eingesetzt werden? Diese Bibliothek bietet eine gute Unterstützung sich dem Gebiet des maschinellen Lernens zu nähern, aber auch die Möglichkeit, dies in praktischen Fällen einzusetzen.

1.3 Zielsetzung

Die Arbeit soll das Thema maschinelles Lernen - neuronale Netzwerke so beleuchten, dass es möglich ist, diese mit den Grundlagen zu verstehen und zu erlernen. Des Weiteren wird die Bibliothek TensorFlow näher gebracht, welche eine gute Unterstützung bietet, eine Problemstellung zu lösen und diese Lösung direkt in einer Applikation einzusetzen.

Die Betrachtung der benötigten Punkte sowie Gebiete sollte auf breiter Front erfolgen, am Ende sollte jeder Leser ein Verständnis dafür haben. Außerdem sollte er in der Lage sein können, den Umfang einer solchen Aufgabenstellung festzustellen. Als Konsequenz aus dieser Betrachtungsweise können allerdings nicht alle Punkte in ihrer ganzen Tiefe erfasst werden. Hier wäre es erforderlich, noch weitere Lektüre einzubeziehen und diese zu studieren.

Am Ende der Arbeit wird eine mögliche Lösung für ein praktisches Beispiel entwickelt und näher erklärt. Dieses wird als praktische Repräsentation verwendet, um den Umfang der Bibliothek noch besser zu verstehen.

Der Anspruch, dass sich mit dieser Technik der Datenanalyse jegliche Problemstellung gelöst werden kann, ist ausdrücklich kein Ziel dieser Arbeit. Des Weiteren wird teilweise nicht tiefer auf Themen eingegangen, da dies den Rahmen dieser Arbeit übersteigen würde. Diese Arbeit sollte aber als möglicher Startpunkt, für einen Einstieg in die Welt des maschinellen Lernens - neuronale Netzwerke, dienen.

Kapitel 2

Begriffe im Maschinellen Lernen

Diese Erklärung der Begriffe und Elemente verfolgt zwei Ziele. Zum einen stellt dies die Grundlage des gesamten Themas dar und soll für Interessierte, die nicht so vertraut sind, eine Einführung in die Thematik bieten. Und zum anderen werden viele der Begriffe erläutert, die in dieser Arbeit noch häufig zum Einsatz kommen werden (u.A. Neuron, Aktivierungsfunktion, ...).

2.1 Data Science

Data Science wird generell als die Extraktion von Wissen aus Daten bezeichnet. Dabei werden die Fachbereiche Statistik und Mathematik, Informatik und Machine Learning sowie einige weitere zu diesem Begriff zusammengefasst. Das Gebiet für sich wird auch als Berufstätigkeit bezeichnet, wobei meist spezialisierte Formen für die Berufsbezeichnung verwendet werden.

Damit Wissen aus Daten überhaupt extrahiert werden kann, muss ein ganzer Prozess durchlaufen werden. Dieser beginnt mit dem Zusammentragen von Rohdaten aus der Realität, welche zu diesem Zeitpunkt noch keinen Zusammenhang offenbaren. Im zweiten Prozessschritt werden diese Daten meist umgebaut und neu sortiert, wobei dieser Schritt nicht immer erforderlich ist. Auf diese zurecht gelegten Daten besteht nun die Möglichkeit, Modelle, Algorithmen sowie weitere Extraktionen durchzuführen. Die erneut extrahierten Daten werden in weiterer Folge als Ausgangsdaten verwendet. Auf diese Daten ausgeführte Modelle und Algorithmen liefern Ergebnisse, die visuell dargestellt für eine größere Gruppe von Personen geeignet sind. Aus diesem gelernten Wissen besteht zusätzlich die Möglichkeit, dieses zum Generieren von neuen Daten zu verwenden und neue Modelle zu entwickeln, die zum Beispiel Vorgänge in der Natur noch akkurater widerspiegeln.

2.2 Machine Intelligence

Machine Intelligence ist ein Begriff, der noch nicht eindeutig definiert worden ist, aber schon Verwendung findet. Einige namhafte Unternehmen wie Google Inc. und Microsoft Corporation bieten jeweils unterschiedliche Definitionen oder Beschreibungen. Die Definitionen dieser Firmen weichen nur unwesentlich voneinander ab. Dieser Begriff wird als Überbegriff für das gesamte Gebiet von Machine Learning, Künstlicher Intelligenz, Konversationsintelligenz und allen Bereichen, die in näherer Beziehung dazu stehen, verwendet.

2.3 Machine Learning

Machine Learning definiert eine große Anzahl an Theorien und Umsetzungen von nicht explizit programmierten Abläufen. Diese wurden aus Studien in den Bereichen der Mustererkennung und der rechnerischen Lerntheorie mit Künstlicher Intelligenz teilweise entwickelt. Dieses Gebiet umfasste im Jahr 2016 aber sehr viel mehr. So existieren zusätzliche Ansätze aus dem Bereich der Biologie, wie zum Beispiel Neuronale Netzwerke, die dem Gehirn nachempfunden sind und genetische Algorithmen, die der Weiterentwicklung eines Lebewesens ähneln. Ein ganz anderer Zugang wurde in der Sowjetunion verfolgt, mit sogenannten 'Support Vektor Machines', bei welchem man einen rein mathematischen Ansatz anstrebte [11].

2.4 Neuronale Netzwerke

Die Theorie und die ersten Grundlagen wurden im Jahre 1943 von Warren McCulloch und Walter Pitts geschaffen, die ein Modell entwickelten, jedoch nicht die technischen Möglichkeiten hatten dieses umzusetzen. Dieses führte zur 'Threshold Logik', welche bestimmt, ab wann und wie stark ausgeprägt etwas weitergegeben wird [12]. Durch die Entwicklung des 'Backpropagation'-Algorithmus ist es möglich, Netzwerke mit mehr als drei Ebenen zu trainieren. Neuronale Netzwerke bestehen aus Neuronen, die miteinander verbunden sind und gemeinsam ein Netzwerk ergeben [6].

2.4.1 Neuron

Ein Neuron wurde einer Nervenzelle in einem Gehirn mit den folgenden Bestandteilen nachempfunden:

Informationseingangsstrom ist der Dateneingang, wobei ein Neuron ein bis theoretisch beliebig viele solcher Eingänge haben kann. Dies hängt von der jeweiligen Architektur des Netzwerks ab.

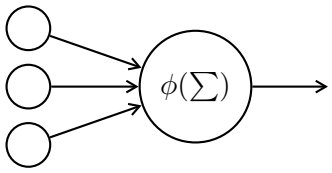


Abbildung 2.1: Neuron mit Eingang, Kernfunktion, Aktivierungsfunktion

Informationsgewichtung bezeichnet die Gewichtung mit der der Eingangsstrom gewertet wird. So wird ein Informationseingangsstrom mehr oder weniger berücksichtigt. Diese Gewichtung wird durch den Backpropagation-Algorithmus angepasst und nachjustiert.

Kernfunktion bewirkt das Verarbeiten der gewichteten Informationseingänge. Im einfachsten Fall werden alle Werte aufsummiert. Es wäre aber möglich, jegliche Berechnung hier einfließen zu lassen, welche mehrere Werte verwendet und daraus einen neuen Wert berechnet.

Aktivierungsfunktion berechnet den Ausgang eines Neurons. Dabei wird eine weitere Funktion auf das im Kern berechnete Ergebnis ausgeführt, das dazu führt, dass ein Ergebnis noch stärker ausgeprägt weitergegeben wird oder minimiert wird, beziehungsweise in einen Wertebereich eingepasst wird. Diese Aktivierungsfunktion ist meist die Sigmoid-Funktion oder eine lineare Funktion, welcher in der Abbildung 2.2 zu erkennen sind.

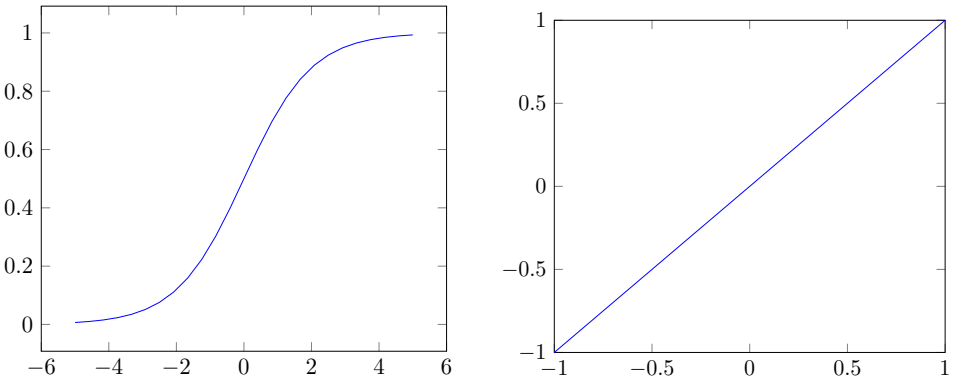


Abbildung 2.2: Basisaktivierungsfunktionen: (l) eine Sigmoid-Funktion, (r) eine lineare Funktion

Die einfachste Repräsentation eines Neurons lässt sich mathematisch folgendermaßen darstellen. Im Kern wird eine Summenberechnung durchgeführt. Dabei werden die Eingangswerte und deren Gewichtung miteinander mul-

tipliziert, sowie diese Ergebnisse aufsummiert. Der griechische Buchstabe ϕ (phi) steht für die Aktivierungsfunktion des Neurons und stellt damit die Ausgabe des Neurons dar.

$$f(x, w) := \phi(\sum_i w_i * x_i) \tag{2.1}$$

Bias Neuron definiert einen Spezialfall eines Neurons, welches keine Dateneingänge, somit auch keine Gewichtung hat und keine Berechnung im Kern durchführt. Dieses liefert nur einen konstanten Wert, wie zum Beispiel eine 1. Durch die konstante Auslieferung wird auch die Aktivierungsfunktion überflüssig. Das Bias Neuron stellt somit einen stetigen Wert für das Netzwerk dar, beziehungsweise für die darauffolgende Ebene.

2.5 Ebenen/Layer

Ebenen sind Zusammenschlüsse von Neuronen, welche sich auf derselben Stufe befinden. Diese Neuronen sind aber nicht miteinander verbunden, sondern bekommen Daten aus der Ebene davor und geben diese an die darauffolgende Ebene weiter. Dieser Typ wird **Hiddenlayer** bezeichnet. Jedes Netzwerk benötigt zusätzlich zwei weitere Ausprägungen an Ebenen. Diese sind:

Inputlayer stellt den Übergang zwischen der Welt außerhalb des Neuronalen Netzwerks und dem Netzwerk dar. Diese Ebene nimmt die Daten ohne Gewichtung auf und gibt sie an die darauffolgende Ebene weiter.

Outputlayer befindet sich am Ende eines Netzwerkes. Dieser Layer hat die Aufgabe, die Daten nach außen, oder an das darauffolgende Netzwerk weiterzugeben. Hierbei werden die Informationen meist nur mehr für die Ausgabe aufbereitet. In manchen Netzwerken existieren keine Outputlayer in diesem Sinne, sondern ein Layer, der als Hiddenlayer und Outputlayer fungiert. Dies ist der Fall, wenn nur zwei Layer sich im Netzwerk befinden und einer davon vom Inputlayer eingenommen wird.

2.6 Informationen Merken und Wiedererkennung

Durch das Anpassen der Gewichtungen bei jedem Dateneingangsstrom mit Hilfe des Backpropagation-Algorithmus ist es möglich, Zustände zu speichern und diese auch zu merken. Sollte ein ähnlicher Dateneingang stattfinden, wo zuvor schon einer vorhanden war, dann sollte dieser ähnlich behandelt werden. Dieser kann möglicherweise zu derselben Kategorie gehören, wie der zuvor schon bekannt gemachte und gelernte Dateneingang.

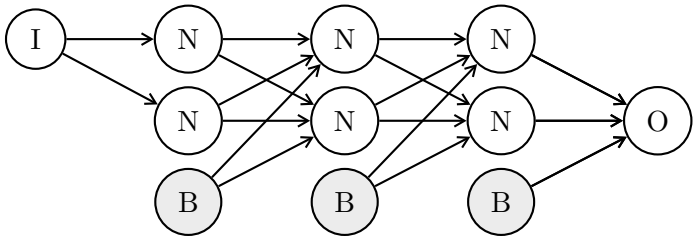


Abbildung 2.3: Einfaches Neuronales FeedFordward Netzwerk

2.7 Konvergieren im Maschinellen Lernen

Konvergieren im Maschinellen Lernen bezeichnet das Minimieren der Fehlerquote gegen 0. Die Fehlerquote wird dabei als Error bezeichnet und ist ausschlaggebend für den Lernprozess. Ein Error von 0 würde bedeuten, dass das Netzwerk keinen Fehler machen würde. Für die Feststellung des Fehlers gibt es diverse Funktionen, wie zum Beispiel die 'Mean-Square-Error'-Methode.

2.8 Backpropagation

Bis zum Jahre 1986 gab es keine automatisierte Möglichkeit, die Gewichtungen in einem Netzwerk automatisch anzupassen. In diesem Jahre entwickelten Rumelhart, Hinton & Williams eine mögliche Lösung, welche sehr ähnlich zu anderen Ansätzen von früher war [7]. Die zentrale Idee in ihrer Lösung liegt darin, die Abweichung des produzierten Ergebnisses zum wirklich erwarteten Ergebnis zu bestimmen. Aufgrund dieses Fehlers lassen sich im Anschluss die Gewichtungen im Netzwerk vom Ende zum Anfang nachjustieren. Diese Technik ermöglichte damit Netzwerke mit verschachtelten Schichten zu konstruieren und auch zu trainieren.

Lernrate skaliert den Lernprozesses, mit der Auswirkung, ob schneller oder langsamer gelernt wird. Eine Lernrate unter 0 würde die Lerngeschwindigkeit stark verlangsamen und ist somit nicht sinnvoll. Ein Wert über 1 würde eine hohe Lerngeschwindigkeit zur Folge haben. Eine zu hohe Rate würde nicht zum Konvergieren führen, sondern zum Springen.

Momentum stellt wie die Lernrate eine Skalierung des Lernprozesses dar. Dabei werden mit dem definierten Faktor die früheren Gewichtsupdates berücksichtigt. Dies führt dazu, dass lokale Tiefpunkte überwunden werden können und das System doch zum globalen Tiefpunkt konvergiert.

2.9 Trainieren

Überwachtes Trainieren definiert, dass die Daten, welche zur Verfügung stehen aus zwei Teilen bestehen. Erstens aus den Daten selbst, aus welchen gelernt und verstanden werden soll. Zweitens aus den Ergebnissen, zu welchen das Netzwerk kommen sollte, welche meistens als Lable bezeichnet werden. In dieser Situation liefert das Netzwerk ein Ergebnis, welches mit dem erwarteten Wert verglichen werden kann. Dieser Unterschied wird zum Feststellen des Fehlers verwendet, welcher besagt, wie inkorrekt das Ergebnis ist. Des Weiteren wird dieser Fehlerwert für die Backpropagation benötigt.

Unüberwachtes Trainieren kommt dann zu tragen, wenn nur Daten zum Trainieren zur Verfügung stehen. Der erwartete Ausgang ist unbekannt. Diese Strategie wird in Fällen von Clustering (2.11.1) verwendet. Dabei sollen nicht bekannte Gruppen von zusammengehörenden Daten identifiziert werden. Self-Organizing Maps (2.12.2) entdecken zusammengehörende Muster und geben diese in einer Grafik zur weiteren Interpretation weiter. Diese Technik ist insbesondere interessant, da einem Netzwerk nie erklärt worden ist, warum etwas so ist und steht damit in Relation zu einem natürlichen Lernprozess, wie bei einem Menschen.

2.10 Allgemeine Probleme

Ein Grundsatz von neuronalen Netzwerken ist, dass sie nicht jede Frage dieser Welt beantworten können, sondern nur dies durchführen, wofür sie konstruiert wurden. Das Entwickeln eines neuen Netzwerks ist eine sehr schwierige und eine lang andauernde Aufgabe. Dabei können Fehler auftreten, wie Overfitting, aber auch vom Entwickler verursacht sein können.

Diese Arbeit wird auf die bekanntesten Probleme eingehen und auch Lösungen oder mögliche Lösungsansätze beinhalten.

Overfitting bezeichnet ein Problem, welches nicht nur maschinelles Lernen betrifft, sondern auch Menschen und andere Lebewesen. Ein Student lernt zum Beispiel für eine Prüfung und ist im Besitz einer Klausur aus einem Vorjahr. Nach öfterem Durchspielen der Fragen und sich selbst testen, befindet er sich in der Lage diese Klausur mit einer sehr hohen Wahrscheinlichkeit zu bestehen. Dabei hat sich die Klausur in seinem Gehirn eingeprägt, aber nicht das Stoffgebiet zu welchem er eine Klausur schreiben muss. Das Problem wird als Overfitting bezeichnet und beschreibt, dass etwas gemerkt wurde, aber nicht gelernt worden ist und somit eine Abwandlung von Informationen nicht wiedererkannt wird.

Daten , die zum Trainieren von Netzwerken verwendet werden, können selbst ein Probleme darstellen. Eine zu geringe Menge an Daten stellt den Entwickler vor das Problem, dass er für diese Daten ein akkurates Netzwerk entwickeln kann, dieses aber im weiteren Verlauf nicht die gewünschten Resultate liefern wird. Zusätzlich kann es sein, dass die zur Verfügung gestellten Daten selbst nicht vollständig sind und somit wieder nur ein Subset verwendet werden kann.

2.11 Domänenklassen

Neuronale Netzwerke können sehr vielseitig eingesetzt werden. Grundsätzlich lässt sich jedes Problem, welches als Funktion repräsentiert werden kann, durch ein Neuronales Netzwerk approximieren.

In dieser Arbeit werden sieben Hauptdomänen erklärt und beschrieben, welche von Heaton [6] definiert wurden.

2.11.1 Clustering

Das Clustering Problem bezeichnet das Einordnen von Daten in Klassen oder Gruppierungen. Diese Gruppierungen können von einem Netzwerk selbst definiert werden oder manuell festgelegt werden. Im Falle einer Self-Organizing-Map werden die Gruppierungen selbst durch das System festgelegt.

2.11.2 Regression

Regression beschreibt den Fall, in welchem Rohdaten generiert werden und diese so verwendet werden. Dies kann im Sinne einer Klassifikation verstanden werden mit dem Unterschied, dass die Anzahl der Klassen unendlich ist, wie die der Reelle Zahlen von $[0 - 1]$. Ein Anwendungsfall ist das Finden einer zugrundeliegenden Funktion, bei der nur Resultate dieser Funktion vorliegen. So gibt es Abläufe in der Natur, welche schwer in einer Funktion festgehalten werden kann. In diesem Fall lässt sich ein Netzwerk mit den Aktionen und Reaktionen trainieren und erhält eine Approximation der Vorgänge [3].

2.11.3 Klassifikation

Bei der Klassifikation werden die Ergebnisse einer Regression, welche meist als Gleitkommazahl vorliegen, auf eine bestimmte Anzahl an Klassen übertragen. Der Unterschied liegt im Ergebnis, welches produziert wird. Hier werden Daten dem Netzwerk übergeben und dieses muss vorhersagen, zu welcher Klasse sie gehören. Dies wird in einer überwachten Umgebung durchgeführt.

Die Klassen für die Vorhersage sind vorab schon bekannt und können mit den Daten aus dem Outputlayer des Netzwerks verglichen werden und infolge Justierungen durchgeführt werden [6].

Ergebnisse einer Klassifikation sagen aus, mit welchem prozentualen Anteil etwas auf den gegebenen Input zutrifft. Das Gesamtergebnis ergibt immer 100 Prozent. Das Ergebnis bei einer Regression wird dabei nicht in Prozent angegeben, sondern stellt einen konkreten Wert dar.

2.11.4 Predict

Predict-Problemstellungen kommen im Kontext von Business, beziehungsweise von E-Business zur Anwendung. Hier muss anhand von meist zeitgesteuerten Ereignissen eine Vorhersage getroffen werden. Zum Beispiel an der Börse ändern sich täglich die Kurse relativ rasch, sodass es für Menschen praktisch nicht mehr möglich ist, diese zu verfolgen. Im Falle der Börse sind Aktienkurse mit zeitlichem Verlauf aus der Vergangenheit verfügbar. Diese können als Trainingsdaten für ein Netzwerk verwendet werden, um den nächsten Tag möglicherweise vorherzusagen.

2.11.5 Robotics

Auch bekannt unter dem Namen Robot-Learning. Dabei lernen Roboter eigenständig neue Techniken oder passen sich automatisch ihrer Umgebung an. Eines der Kernprobleme dabei ist, dass in Echtzeit etwas zur selben Zeit gelernt werden muss, aber auch Aktionen eingeleitet werden müssen, wie das Steuern von Motoren, um zum Beispiel nicht umzufallen.

2.11.6 Computer Vision

Computer Vision zielt darauf ab, einem Computer das Sehen und Verstehen von Bildern zu ermöglichen. Diese Technik findet im Jahr 2016 schon häufig Einsatz. So werden automatisiert Bilder analysiert, beschrieben sowie auch in Gruppen nach diversen Kategorien eingeordnet, wie zum Beispiel nach Gesichtsausdrücken. Solche Dienste werden auch kommerziell eingesetzt und angeboten. In autonom gesteuerten Fahrzeugen findet diese Technologie bereits Verwendung, um Objekte zu erkennen und zu verstehen. So muss zum Beispiel ein Verkehrszeichen von einem Passanten unterschieden werden können.

2.12 Neuronale Netzwerktypen

In den letzten Jahren haben sich diverse gut funktionierende Neuronale Netzwerktypen gebildet, beziehungsweise sind entwickelt und erforscht wor-

den. Diese Netzwerktypen definieren Richtlinien oder Ansätze zu möglichen Netzwerken, welche aber nicht komplett übernommen werden müssen, sondern einen kreativen Spielraum ermöglichen.

2.12.1 FeedForward

FeedForward Netzwerke (FFN) waren bis vor einigen Jahren noch der Stand der Forschung. Auf ihnen basieren einige andere Typen von Netzwerken, die bekanntesten werden in dieser Arbeit noch behandelt. Ein FFN basiert auf den Grundlagen eines Neurons, sowie ihrem Ausbau zu Ebenen mit mehreren Neuronen. So ein Netzwerk besitzt einen Inputlayer, einen Hiddenlayer sowie einen Outputlayer. Sobald das Netzwerk eine große Anzahl an Hiddenlayer aufweist, wird es als Deep FeedForward Netzwerk (2.12.5) bezeichnet. Das FFN weist dabei eine Charakteristik auf, in der der Datenfluss eindeutig definiert ist. Der Datenfluss beginnt beim Inputlayer und endet beim Outputlayer, ohne dass ein Datenrückfluss zum Beispiel vom Hiddenlayer in den vorhergehenden Hiddenlayer vorhanden ist. Dies würde einer Rekursion oder einem Kurzzeitgedächtnis entsprechen. Die einzelnen Ebenen müssen dabei aber nicht voll verbunden sein, die Vernetzung kann selbst bestimmt werden.

2.12.2 Self-Organizing Map

Self-Organizing Map (SOM) findet vor allem im Bereich der Classification Verwendung und wurde von Kohonen (1988) erfunden. Es ist nicht erforderlich einer SOM die Information zu geben, in wie viele Gruppen oder Klassen die Daten unterteilt werden sollen. Dadurch gehört sie zu den Systemen, welche unsupervised trainiert werden. Außerdem besitzen sie die Möglichkeit, neue Daten weiter zu klassifizieren und dies über die Trainingsphase hinaus. Kohonen entwarf die SOM mit zwei Ebenen, einem Inputlayer und einem Outputlayer ohne Hiddenlayer. Der Inputlayer propagiert Muster an den Outputlayer, wo der Dateneingang gewichtet wird. Im Outputlayer gewinnt das Neuron, welches den geringsten Abstand zu den Eingangsdaten hat. Dies geschieht durch das Berechnen der euklidischen Distanz. Diese Art von Netzwerk kommt ohne Bias Neuron aus und es kommen ausschließlich Lineare Aktivierungsfunktionen zur Verwendung.

2.12.3 Hopfield Neuronal Network

Ein Hopfield Neuronal Network (HNN) [4] ist ein einfaches Netzwerk, welches aus einem Layer besteht. In diesem Layer sind alle Neuronen mit jedem anderen Neuron verbunden. Dieses Muster wurde von Hopfield (1982) erfunden. Im Gegensatz zu anderen Netzwerken können Hopfield Netzwerke in einer Matrix abgebildet werden, in welcher die Gewichtung zu den

einzelnen Neuronen abgebildet werden. Das Problem bei diesem Typ ist, dass jedes Neuron auf dem Status des anderen aufbaut. Dies stellt ein Problem für die Reihenfolge der Berechnung dar, was zu einem nicht stabilen Zustand führt. Durch das Hinzugeben einer Energiefunktion kann festgestellt werden, in welchem Zustand sich das Netzwerk befindet. Hiermit kann ein Haltepunkt definiert werden, ab welchem keine Trainingsiteration mehr durchlaufen werden soll.

2.12.4 Boltzmann Machine

Im Jahre 1985 stellten Hinton & Sejnowski [8] das erste Mal eine Boltzmann Maschine vor. Es stellt ein Zwei-Ebenensystem dar, mit einem Inputlayer und einem Outputlayer, wo jeder Knoten mit jedem verbunden ist, außer mit sich selbst. Das voll vernetzte System unterscheidet eine Boltzmann Maschine von einer eingeschränkten Boltzmann Maschine (RBM), welche eine Grundlage für tiefes Lernen und tiefe Neuronale Netzwerke darstellt. In einer RBM sind alle sichtbaren Neuronen mit allen Neuronen im Outputlayer verbunden. Die Verbindungen zwischen den Neuronen im selben Layer entfallen. Der alte uneingeschränkte Typ der Boltzmann Maschinen eignet sich gut für Optimierungsprobleme sowie für Mustererkennungen.

2.12.5 Deep FeedForward

Deep FeedForward Netzwerke unterscheiden sich von den normalen FeedForward Netzwerken in dem, dass sie mehrere Hiddenlayer beinhalten anstatt nur einem.

2.12.6 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) Netzwerke sind relativ jung, wobei NEAT für einen Algorithmus steht, der Neuronale Netzwerke entwickelt. Er wurde von Stanley und Miikkulainen (2002) entwickelt. Dieser Typ verwendet genetische Algorithmen, um die Struktur und die Gewichtungen im Netzwerk zu optimieren. Die Input- und Outputlayer sind identisch mit einem FeedForward Netzwerk. Dafür fehlt diesem Typ eine innere Struktur. Die Verbindungen sind lose, nicht klar definiert und können während dem Trainieren entfernt werden, aber auch wieder hinzugefügt werden.

2.12.7 Convolutional Neural Network

Convolutional Neural Network werden selbst nicht als komplettes eigenes Netzwerk verwendet, sondern in FeedForward Netzwerken. Im Speziellen, wenn es um Bilderkennung geht. Dabei werden zwei Ebenen nicht voll vernetzt sondern nur teilweise und somit Gewichtungen eingespart. Außerdem

können die Gewichtungen geteilt werden, sodass immer in dieselbe Richtung verlaufende Verbindungen dieselbe Gewichtung aufweisen. Dies ermöglicht es, komplexe Strukturen zu speichern und trotzdem die Speicherauslastung niedrig zu halten und die Effektivität aufrecht zu halten.

2.12.8 Recurrent Network

Recurrent Network sind Netzwerke, die nicht nur einen Kontrollfluss haben, sondern auch Rekursionen beinhalten. Diese Rekursionen können jedes andere Neuron ansprechen, ausgenommen der Neuronen im Inputlayer. Ein Problem durch Rekursionen sind Endlosschleifen, welche behandelt werden müssen. So können Kontext Neuronen verwendet werden, aber auch eine definierte Anzahl an Iterationen durchlaufen werden, damit die Rekursion nicht mehr fortgeführt wird. Eine weitere Option ist so lange zu warten, bis sich die Ausgabe des Neurons stabilisiert hat und sich nicht mehr ändert. Das Kontext Neuron nimmt dabei die Stelle eines kurzen Speichers ein, wo ein Zustand für die nächste Iteration zwischengespeichert wird. Die Informationen, die in einem solchen Neuron gespeichert werden, werden bei diesem Speichervorgang nicht gewichtet, sondern erst, wenn diese Informationen an das Netzwerk zurückgegeben werden. Diese Rekursionen werden vor allem in Fällen verwendet, wenn es um zeitliche Abläufe und Änderungen geht, wie zum Beispiel bei der Temperatur für den nächsten Tag, wo man Daten etlicher vorangegangener Jahre zur Verfügung hat.

Elman Network wurde im Jahre 1990 vorgestellt, sie verwenden Rekursionen mit Kontext Neuronen. Dabei existieren zwei Hiddenlayer mit einem Layer für normale Neuronen und einem mit Kontext Neuronen. Die Kontext Neuronen sind dabei voll mit dem Hiddenlayer verbunden und dieser gibt die Informationen ungewichtet an die Kontext Neuronen weiter. In diesem System existieren so viele Kontext Neuronen wie Neuronen im Hiddenlayer, sodass jedes Neuron dort ein Kontext Neuron mit dem neuen Status befüllt.

Jordan Network wurde 1993 der Öffentlichkeit präsentiert, sie sind den Elman Netzwerken sehr ähnlich. Es werden wieder Kontext Neuronen für das Zwischenspeichern verwendet, nur wird dieser Zustand durch den Outputlayer definiert. So wird der Ausgang gespeichert und in der nächsten Iteration wieder verwendet. Das Kontext Neuron ist dabei nur mit dem Outputlayer wieder verbunden und nicht mit einem Hiddenlayer.

2.13 Domänen und Typen Matrix

Wie in der Abbildung 2.4 erkennbar ist, existiert kein Netzwerkgrundtyp, der für alle Problemdomänen geeignet ist. Dies führt zu der Schlussfolgerung,

	Clust	Regis	Classif	Predict	Robot	Vision	Optim
Self-Organizing Map	✓✓✓				✓	✓	
Feedforward		✓✓✓	✓✓✓	✓✓	✓✓	✓✓	
Hopfield			✓			✓	✓
Boltzmann Machine			✓				✓✓
Deep Belief Network			✓✓✓		✓✓	✓✓	
Deep Feedforward		✓✓✓	✓✓✓	✓✓	✓✓✓	✓✓	
NEAT		✓✓	✓✓		✓✓		
CPPN					✓✓✓	✓✓	
HyperNEAT		✓✓	✓✓		✓✓✓	✓✓	
Convolutional Network		✓	✓✓✓		✓✓✓	✓✓✓	
Elman Network		✓✓	✓✓	✓✓✓			
Jordan Network		✓✓	✓✓	✓✓	✓✓		
Recurrent Network		✓✓	✓✓	✓✓✓	✓✓	✓	

Abbildung 2.4: Domänen zu Typen Matrix [6]

dass je nach Aufgabe und Ziel ein entsprechendes Grundgerüst gewählt werden muss. Auf Basis dieses Grundgerüsts können uneingeschränkt weitere Eigenheiten aus anderen Netzwerken eingebaut werden. In der Praxis findet man selten ein Netzwerk von nur einem Typ für eine größer Problemstellung. Das Problem wird herabgebrochen, auf mehrere kleinere Problemstellungen, welche hintereinander und teilweise parallel gelöst werden. Dabei übernimmt jedes Teilnetzwerk eine kleine Aufgabe des Gesamten und zwar die eine, für die es entwickelt wurde. Aktuelle Netzwerke wie das Inception v3 Netzwerk von Google Research benötigt zwei Wochen mit acht Grafikkarten zum Trainieren. Ab diesem Zeitpunkt ist es im Stande, akkurate Resultate zu liefern. Dieses Netzwerk ist sehr komplex und besteht nicht nur aus 3 Ebenen, was zur Schlussfolgerung führt, dass je tiefer das Netzwerk ist, desto aufwändiger ist es zu trainieren.

2.14 Optimierung

Optimierungen beeinflussen das Lernverhalten und das Speicherverhalten eines Netzwerkes.

Lernrate skaliert die Lerngeschwindigkeit, wie im Punkt Backpropagation beschrieben.

Momentum gehört auch zur Optimierung im Algorithmus zur Backpropagation. Dieser bestimmt wie stark frühere Gewichtsaktualisierungen berücksichtigt werden sollen.

DropOut gehört zur Kategorie der Regulatoren. Hinton [10] beschreibt DropOuts als eine effektive Art, um Overfitting zu vermeiden. DropOut kann als System integriert werden, aber auch als eigener Layer in einem Netzwerk. In einem DropOutlayer werden immer Neuronen deaktiviert, inklusive ihrer Verbindungen zum nächsten Layer. Dies hat zur Folge, dass nur ein geringerer Teil an Informationen aus dem vorhergehenden Layer in den nächsten übergeht. Durch diesen Prozess des künstlichen Geringhaltens von Informationen während des Trainings führt dazu, dass das Netzwerk trotz dieser Einschränkung versucht, ein gutes Ergebnis zu erzielen. Während der Test- und Produktivphase werden diese DropOutlayer aber meist deaktiviert, da das volle Potenzial des Netzwerks verwendet werden soll.

L1 und L2 Regularisierung sind ebenfalls Techniken, die zur Verhinderung von Overfitting beitragen. Im Gegensatz zu der DropOut Strategie sind diese zwei Regularisierungstechniken Teil der Backpropagation oder werden als Funktion eingesetzt. Beide Techniken arbeiten mit Strafen, welche verteilt werden und so die Gewichtungen vor dem Ausarten hindern. Durch das Bestrafen der Gewichtungen im Netzwerk werden diese Werte gering gehalten. Wenn ein Gewicht Richtung 0 geht, führt dies unweigerlich zu einem indirekten Ausschluss aus dem Netzwerk. Das Netzwerk wird spärlicher und leichtgewichtiger, was aber wiederum das Resultat hat, dass ein Rauschen in den Daten ignoriert wird.

$$E := \frac{\lambda}{n} \sum |w| \tag{2.2}$$

Die Funktion 2.2 bildet die Berechnung der Strafe in der Backpropagation ab. Das λ definiert wie stark die Regularisierung den Error-Wert des Netzwerkes beeinflussen soll. Ein Wert von 0 führt dazu, dass die Regularisierung keinen Einfluss besitzt. Im einem normalen Fall ist dieser Wert kleiner als 0.1(10%). Der Divisor n wird durch die Anzahl an Elementen im Trainingssatz und der Anzahl an Neuronen im Outputlayer bestimmt. Zum Beispiel bei 100 Elementen im Trainingssatz und 3 Neuronen im Outputlayer würde der Divisor den Wert 300 einnehmen. Dies ist erforderlich, da diese Funktion bei jeder Evaluierung der Trainingsdaten berechnet wird.

2.15 Trainingsgeschwindigkeitssteigerung

GPU - GPGPU ist die Bezeichnung für die Verwendung des Grafikprozessors über seine ursprüngliche Auslegung darüber hinaus. In der aktuellen Zeit ist es nicht mehr möglich, eine Geschwindigkeitssteigerung zu erreichen, indem die Taktrate des Prozessors erhöht wird. Deshalb wird mehr parallelisiert, da die Recheneinheiten kleiner werden und so mehrere auf derselben Fläche Platz finden. Eine Grafikkarte besitzt die Eigenschaft, gleichförmige

Operationen in einem Schritt auf sehr viele Objekte gleichzeitig auszuführen. So werden viele Pixel auf einmal eingefärbt oder eine Multiplikation großer Matrizen. Der Geschwindigkeitsvorteil kommt dabei durch den hohen Grad an Parallelität, da die Grafikkarte hauptsächlich für solche Operationen ausgelegt worden ist.

Batch Learning führt dazu, dass immer ein Paket an Daten in das System eingeführt wird. Dieses Paket wird je nach Implementierung parallel verarbeitet oder sequenziell. Der Unterschied zu 'Online Learning' ist nun, dass nicht nach jedem Datensatz die Gewichtungen und das System nachjustiert wird, sondern dass zuerst das Paket verarbeitet und das System einmal angepasst wird. Dabei werden die Gradienten zusammengerechnet und einmal auf den Graphen adaptiert [6].

Kapitel 3

TensorFlow

TensorFlow repräsentiert eine Bibliothek für Machine Intelligence und entstand in der Google Brain Abteilung. Das Projekt wird als Open Source Projekt weiterentwickelt, wobei das Projekt von Google weiterhin gepflegt wird. Das Offenlegen des Projekts führt dazu, dass auch Personen außerhalb von Google die Möglichkeit bekommen die Bibliothek zu verwenden, sowie auch etwas einflechten können.

Das Hauptkonzept hinter TensorFlow sind sogenannte Tensoren, welche einen Graphen durchlaufen. Der Graph selbst stellt damit einen Datenflussgraphen dar, welcher Knoten beinhaltet. Diese Knoten bilden numerische Operationen ab. Der Informationsaustausch zwischen den Knoten erfolgt mit multidimensionalen Arrays, den Tensoren. TensorFlow bietet wie andere Bibliotheken die Möglichkeit, die Berechnungen auf eine Grafikkarte auszulagern. Zusätzlich sind weitere Routinen eingebaut, damit das Trainieren über mehrere Grafikkarten, sowie auf weitere Computer verteilt werden kann.

TensorFlow steht für mehrere Programmiersprachen zur Verfügung, welche offiziell unterstützt werden, wobei noch weitere durch die Open Source Gemeinschaft unterstützt werden. Den Hauptbereich stellt die Python API dar, welche die vollständigste Implementierung enthält. Der Kern von TensorFlow ist mit C++ und Python implementiert und wurde sehr stark optimiert, um eine sehr gute Performanz zu erzielen. Die Python API wird im Umfeld von TensorFlow dazu verwendet, einen Graphen zu erstellen, zu trainieren und zu testen. Durch die Verwendung von Python besteht die Möglichkeit, sehr schnell Änderungen am Graphen für die Ergebnisdarstellung durchführen zu können, ohne die ganze Applikationsstrukturen übersetzen zu müssen. Dieser Graph wird nach seiner Trainingsphase exportiert und beinhaltet alle Knoten sowie die dazugehörigen Gewichtungen. Die C++ API sowie die Java API und GO API zielen auf eine sehr effiziente Ausführung ab. Durch die Verwendung des trainierten Graphen kann dieser auch

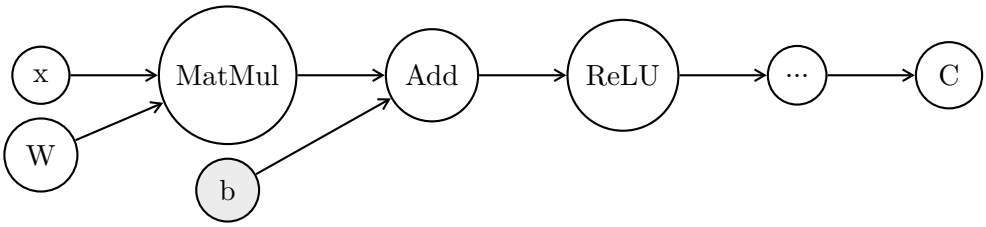


Abbildung 3.1: Der resultierenden Teilgraph aus dem Codefragment 3.1 nach dem Beispiel in [1]

auf mobilen Plattformen eingesetzt werden.

3.0.1 Graphs/Dataflowgraph

```
1 import tensorflow as tf
2
3 b = tf.Variable(tf.zeros([100]))
4   # 100-d Vektor, initialisiert mit 0
5 W = tf.Variable(tf.random_uniform([784,100],-1,1))
6   # 784x100 Matrix w/rnd vals
7 x = tf.placeholder(name="x")
8   # Platzhalter für Eingangsdaten
9 relu = tf.nn.relu(tf.matmul(W, x) + b)
10  # Relu(Wx+b) Aktivierungsfunktion mit impliziter Addition
11 C = [...]
12  # Kostenfunktion und noch weitere Knoten
13 s = tf.Session()
14 for step in xrange(0, 10):
15   input = ...construct 100-D input array ...
16     # Erstellen eines 100-d Vektor mit den Eingangsdaten
17   result = s.run(C, feed_dict={x: input})
18     # Graphen mit den Eingangsdaten ausführen
19   print step, result
20     # Ausgabe des Berechneten Resultats
```

Listing 3.1: TensorFlow Codefragment zur Definition eines Teils des Graphen

Ein TensorFlow Graph kann wie im Codefragment 3.1 ersichtlich, beschrieben werden und wie in Abbildung 3.1 grafisch dargestellt werden. Dieser wurde zum Beispiel mit der Python API erstellt. Die Knoten und Verbindungen ergeben einen Datenfluss, dieser beinhaltet alle erforderlichen Komponenten, auch für das Persistieren und Aktualisieren der Daten. Dies sind Erweiterungen für den Hauptgraphen und beinhalten auch Logik für Schleifenverwaltungen. Ein Knoten in einem Graphen besitzt 0 bis n Ein- und Ausgänge und eine Kernfunktion. Zum Datenhauptfluss mit den Tensoren gibt es zusätzlich spezielle Verbindungen, welche „control dependencies“genannt werden. Anhand dieser Verbindungen werden keine Daten im

Sinne der Tensoren übertragen, sondern werden herangezogen um Abhängigkeiten zu definieren, wie zum Beispiel eine Ausführung in einem anderen Knoten vor einem anderen zu definieren. Ein Quellknoten muss die Ausführung abgeschlossen haben, bevor der darauf Wartende mit der Ausführung beginnen kann [1].

3.0.2 Datenfluss Programmierung

In traditionellen Programmierparadigmen, wie dem Sequenziellen, Prozeduralen oder Imperativen, werden eine Reihe von Operationen definiert, welche in einer speziellen Reihenfolge abgearbeitet werden sollen, wie von Neumann beschrieben. TensorFlow wird nicht im traditionellen verwendet, sondern im Sinne eines Datenflusses. Hierbei werden die Bewegungen von Daten in einem Modell beschrieben. In diesem Modell existieren Operationen mit explizit definierten Ein- und Ausgängen, wobei die Aktion in der Operation für das Modell verborgen bleibt wie eine „black box“. Eine Operation wird ausgeführt sobald alle Eingänge vorhanden und valide sind. Diese Art der Programmierung bietet von sich aus die Möglichkeit der parallelen Ausführung, sowie ein einfaches Verteilen der Berechnungen auf ein dezentrales System.

3.0.3 Operation

Die Operation stellt in jedem Knoten den Kern dar, wie zum Beispiel eine Matrix Multiplikation oder eine Addition. In TensorFlow selbst gibt es einen Unterschied zwischen Operation und Kernel. Operationen besitzen Attribute, welche spätestens zum Zeitpunkt der Grapherstellung bekannt sein müssen. Ein solches Attribut wäre zum Beispiel, um einer Operation polymorphismus zu ermöglichen. Der Kernel hingegen ist die Implementierung der Operation selbst. Dieser kann auf verschiedenen Geräten ausgeführt werden, wie zum Beispiel auf der CPU oder GPU. Die Operationen und die dazugehörigen Kernels werden über einen Registrierungsmechanismus zur Verfügung gestellt. Diese Sammlung an Operationen kann auch erweitert werden [1].

3.0.4 Sessions

Die Session repräsentiert die Laufzeit eines Graphen. Dieser Session wird ein Graphen übergeben, welcher zuvor initialisiert werden muss. Ohne die Initialisierung der Knoten und Verbindungen würde die weitere Ausführung nichts produzieren, da alle Werte 0 sind. Sie stellt eine weitere Funktion zur Verfügung genannt *Run*. Der Run-Funktion wird eine Liste an Endknoten übergeben, welche berechnet werden sollen und die zu dem initialisierten Graphen gehören. Die Platzhalter Tensoren werden mit Daten verknüpft und an den Graphen gereicht. In den meisten Fällen wird ein Graph einmal

erstellt und mehrfach ausgeführt [1].

3.0.5 Tensor

In TensorFlow ist ein Tensor ein typisiertes multidimensionales Array. Die verwendbaren Typen reichen von Datentypen mit und ohne Vorzeichen, sowie bis hin zu Doubles und Zeichenketten [1].

3.0.6 Hyperparameter

Hyperparameter werden im Umfeld von maschinellem Lernen verwendet, um Variationen an Kombinationen zu testen. Dabei werden verschiedenste Parameter getestet, wie unterschiedliche Aktivierungsfunktionen oder Optimierungsalgorithmen, aber auch die Anzahl an Ebenen und Breiten dieser. Im Gesamten führt dies meist zu sehr vielen Permutationen, welche ausgetestet werden müssen und somit voll trainiert werden. Da die Zeit, die dafür benötigt werden würde, nicht in sinnvoller Relation dazu steht, werden solche Brute-Force Tests nur mehr selten durchgeführt. Für diesen Fall existieren eigene Techniken, welche sich nur um das Optimieren der Hyperparameter kümmern [3].

3.1 Bibliotheksinhalt

3.1.1 Datentypen

TensorFlow besitzt eine große Anzahl an Datentypen, welche verwendet werden können. Diese reichen von Grunddatentypen wie 'Boolean' und 'String' bis hin zu verschiedenen Integer Datentypen. Sie stehen in verschiedenen Wertebereichen zur Verfügung. Es gibt Gleitkommazahlen mit unterschiedlichen Genauigkeiten. 16-bit steht für eine halbe Genauigkeit und 64-bit Genauigkeit entspricht einer doppelten Genauigkeit. Der Grund für diese verschiedenen Anzahlen an Datentypen ist, dass diese zur Optimierung verwendet werden können. Ein trainiertes Netzwerk, welches nie in den Wertebereich von 64-bit signierte Integers gekommen ist, wird diesen möglicherweise nie benötigen. In einem solchen Fall können die Wertebereiche reduziert werden, zum Beispiel auf 32-bit signierte Integer, somit können die Berechnungen hochperformanter ausgeführt werden [13].

3.1.2 Operationen

Konstanten und Zufallswerte

Konstanten liegen in TensorFlow vordefiniert zur Verwendung. Diese stellen initialisierte Tensoren für den ersten Trainingsdurchlauf zur Verfügung.

- *tf.zeros* erstellt einen Tensor mit den angegebenen Matrizendimensionen, bestehend aus 0 und von einem definierten Datentypen.
- *tf.zeros_like* gibt einen Tensor zurück, welcher dieselben Dimensionen wiedergegeben besitzt. Alle Werte in diesem Tensor sind auf 0 gesetzt. In diesem Schritt kann der Datentyp mit angepasst werden, wenn die Dimensionen übernommen werden sollen.
- *tf.ones* agiert genau wie der Tensor *tf.zeros*, mit dem Unterschied, dass alles mit 1 gefüllt wird.
- *tf.ones_like* repräsentiert dasselbe wie *tf.zeros_like*, jedoch mit dem Wert 1.
- *tf.fill* fügt eine neue Dimension in einen Tensor ein, mit dem gegebenen Skalar, der für die Werte eingesetzt werden soll.
- *tf.constant* liefert einen Tensor mit selbst definierbaren Werten. Diese Werte können sowohl eine Liste sein, als auch ein einzelner Wert, welcher beliebig eingefügt werden soll.

Sequenzen können verwendet werden, um einen Wertebereich in eine bestimmte Anzahl an Werte zu zerteilen und diese als Tensor in das System wieder einfließen zu lassen.

- *tf.lin_space* generiert einen eindimensionalen Tensor vom Datentypen 32 oder 64-bit Gleitkommazahl, mit einer bestimmten Folge. Diese beginnt mit einem Startwert und endet mit dem Endwert. Die Werte, welche innerhalb dieses Bereiches liegen, werden gleichmäßig verteilt.
- *tf.range* erstellt wie *tf.lin_space* einen eindimensionalen Tensor mit Skalarwerten. Die Folge beginnt mit einem Startwert und erweitert sich um ein bestimmtes Delta bis zum Endwert, welcher nicht Teil der Folge ist.

Zufallswerte werden im Bereich von maschinellem Lernen sehr häufig benötigt. So wird der Startzustand oft mithilfe von Zufallszahlen hergestellt.

- *tf.random_normal* liefert einen Tensor mit Zufallswerten anhand einer Normalverteilung (Gaussian). Die Dimension des Ergebnistensors muss spezifiziert werden, der Median, die Standardabweichung sowie der resultierende Datentyp können angegeben werden.
- *tf.truncated_normal* verhält sich gleich zu *tf.random_normal* mit dem Unterschied, dass bei Werten die größer als 2-mal die Standardabweichung sind, diese ignoriert werden und ein neuer Wert ausgewählt wird.
- *tf.random_uniform* generiert einen Tensor, in welchem Werte gleich wahrscheinlich vorkommen. Die Werte werden aus dem spezifizierten Wertebereich genommen, wobei diese exklusiv der oberen Grenze entsprechen (siehe Beispiel '$[0, 1)$').

- *tf.random_shuffle* erstellt eigenständig keine neuen Werte, sondern mischt einen Tensor anhand seiner ersten Dimension durch.
- *tf.random_crop* liefert einen zufälligen Teil eines Tensors mit derselben Anzahl an Dimensionen jedoch mit der spezifizierten Größe.

Einige dieser Funktionen benötigen sogenannte Seed-Werte, welche für die zufällige Verteilung der Startwert benötigt werden. Im Falle von TensorFlow beruht dies auf zwei Werten, einer wird für den Graphen spezifiziert, der andere für Operationen selbst. Der Wert für den Graphen kann mit *tf.set_random_seed* gesetzt werden. Für weitere Informationen steht die online Dokumentation zur Verfügung. [1]

Variables

Variablen geben bei jedem Durchlauf einen Tensor ab. Dieser Wert ändert sich solange nicht, bis ihm ein neuer Wert zugewiesen wird.

Transformationen

Casting bietet die Möglichkeit wie in anderen Programmiersprachen Typen zu konvertieren. Diese Operation muss in den Graphen eingepflegt werden, da keine impliziten Konvertierungen durchgeführt werden. Es kann jeder Tensor konvertiert werden, sowie eine Zeichenfolge in eine Zahl. Bei diesem Vorgang kann ein Fehler entstehen, welcher in einem *TypeError* resultiert.

Shapes und Shaping liefern die Gestalt eines Tensors, bieten jedoch auch die Möglichkeit an, diese zu ändern.

- *tf.shape* liefert eine genaue Aufschlüsselung des Tensors mit der Dimension und der Tiefe.
- *tf.size* repräsentiert die Anzahl an Elementen in einem Tensor. Diese Anzahl ergibt sich aus den konkreten Werten.
- *tf.rank* verhält sich ähnlich zu *tf.size* mit dem Unterschied, dass die Anzahl der Dimensionen gezählt werden.
- *tf.reshape* wird verwendet, um Tensoren in eine neue Struktur zu bringen. Dabei steht für das Einebnen der Struktur auf eine Ebene eine Kurzschreibweise zur Verfügung, mit -1 als Zieldefinition.
- *tf.squeeze* entfernt ganze Dimensionen aus dem gegebenen Tensor. Ohne Achsenangabe werden alle Dimensionen mit der Größe 1 entfernt oder es werden die spezifizierten Dimensionen herausgenommen.

[1] Online Dokumentation: Constants, Sequences, and Random Values www.tensorflow.org/api_guides/python/constant_op

- *tf.expand_dims* gliedert eine Dimension in einen Tensor wieder ein. Standardmäßig an der Indexstelle 0, außer es wurde spezifiziert.

Slicing und Joining wird wie in diversen Programmiersprachen auch von TensorFlow unterstützt, im Speziellen mit Tensoren. Diese Operationen reichen von einfachen Slicing Operationen über Transponieren bis hin zu dem Verketten von Tensoren. Dabei kann definiert werden, anhand welcher Achse der Dimensionen die Operation ausgeführt werden soll.
Weiter Informationen befinden sich in der online Dokumentation. [2]

Mathematik

Arithmetische Operationen stellen die mathematischen Grundoperationen dar. Diese können teilweise in Kurzschreibweisen verwendet werden, wie zum Beispiel die Addition. Sie kann entweder als explizite Operation *tf.add(x, y)* verwendet werden aber auch implizit bei der Addition + von einem Tensor mit einem Bias-Tensor.

Basis Funktionen ergänzen die arithmetischen Operationen um Standardfunktionen. Zu diesen Funktionen zählen die Berechnung der Absolutwerte in einem Tensor sowie eine Exponentialfunktion.

Matrizen Funktionen werden am häufigsten benötigt, da Tensoren im Grunde aus Matrizen bestehen und diese geändert werden können.

- *tf.matmul* führt eine Matrizenmultiplikation aus. Diese Operation findest meist in voll vernetzten Neuronen Verwendung, wenn der übergebene Tensor mit der Gewichtung multipliziert wird.
- *tf.eye* erzeugt eine Identitätsmatrix, in welcher alle Werte entlang der Diagonale 1 sind und alle anderen den Wert 0 bekommen.

Zu diesen Funktionen existieren noch weitere Ansätze, die zur Lösung von Gleichungen verwendet werden können. Diese Gleichungen müssen in Matrizenschreibweise im Tensor abgebildet sein.

Komplexe Zahlen können verwendet werden und Operationen mit ihnen in die Graphen eingepflegt werden.

Reduzierungsoperationen kommen dann zum Einsatz, wenn der Unterschied zwischen dem erzielten und dem erwarteten Ergebnis festgestellt werden soll.

- *tf.reduce_sum* berechnet die Summe aller Werte in einem Tensor.

[2] Online Dokumentation: Tensor Transformations www.tensorflow.org/api_guides/python/array_ops

- *tf.reduce_mean* berechnet das arithmetische Mittel eines Tensors.
- *tf.reduce_max* reduziert einen Tensor auf die maximalen Werte in der letzten Dimension und reduziert dabei den Rang um eins.

Zu diesen gibt es noch weitere, welche in diversen Fällen benötigt werden, zum Beispiel, wenn Wahrheitswerte reduziert werden sollen.

Die Anzahl an mathematischen Funktionen ist sehr viel größer, als die hier Erwähnten. Die hier Angeführten repräsentieren lediglich die meist verwendeten Operationen. Für weitere Informationen steht die online Dokumentation zur Verfügung. [3]

Flusskontrolle

Flusskontrollen sind Operationen, die den Ablauf im Graphen beeinflussen. Dies können Bedingungen, wie im Sinne von *if (Bedingung){...} else {...}* aber auch *switch (Term) { case '0': ...; break;}* Bedingungen sein. In beiden Fällen müssen die auszuführenden Verzweigungen als Funktionen vorliegen. Zusätzlich gibt es noch eine *While* und eine *For* Schleife. Zu beachten ist, dass diese Operationen den Fluss durch den Graphen stark beeinträchtigen können.

Logik Operatoren können verwendet werden, um Vergleiche zwischen Tensoren durchzuführen. Diese werden aber als Logik Operationen ausgeführt und liefern immer Wahrheitswerte, wie eine logische Und-Verknüpfung auf Binärebene.

Vergleichsoperatoren sind neben den logischen Operatoren weitere Vergleichsoperatoren, welche zur Verfügung stehen. Hierzu zählen *tf.equal* sowie die verneinte Variante *tf.less* und *tf.greater* mit jeweils einer gleichen Version. Diese Operatoren geben wiederum einen Tensor mit Wahrheitswerten aus.

Debugging Operationen ermöglichen es in den Graphen Kontrollstrukturen einzubauen, welche auf diverse Bedingungen reagieren. So kann überprüft werden, ob ein Tensor Werte mit undefinierten Zustand beinhaltet. Die Funktion *tf.Print* ermöglicht es Tensoren auszugeben, wenn diese als Funktion im Graphen evaluiert werden. Aktuell sind die Möglichkeiten einen Graphen zu debuggen relativ eingeschränkt. Grund dafür ist, dass der Graph, in seiner rohen Darstellung schwer zu verstehen ist. [4]

[3] Online Dokumentation: Math www.tensorflow.org/api_guides/python/math_ops

[4] Online Dokumentation: Control Flow www.tensorflow.org/api_guides/python/control_flow_ops

Images

Encodieren und Decodieren von Bilddateien wird in TensorFlow direkt unterstützt. Dabei können Bilder der Datentypen Gif, Jpeg und PNG gelesen werden, sowie das Erstellen von Bildern in diese Datentypen, ausgenommen Gif. In allen Fällen wird das Bild als Zeichenkette mit Pfad angegeben.

Größenänderung von Bildern ist erforderlich, da im Laufe der Zeit sehr wahrscheinlich größere Bilder verwendet werden, diese aber nicht in das fixe Raster des Netzwerkes passen. Für die Größenänderung stehen mehrere Implementierungen mit unterschiedlichen Algorithmen zur Verfügung.

Beschneiden wird dann benötigt, wenn aus einem Bild ein Teil herausgenommen werden soll. Zum Herausnehmen stehen wiederum mehrere Operationen zur Verfügung, welche mit umschließenden Boxen arbeiten oder wie viel Prozent von der Mitte des Bildes aus verwendet werden soll.

Flippen, Rotieren und Transponieren ermöglicht es, Bilder zu verändern, sodass es für einen Menschen mehr oder weniger noch dieselbe Bedeutung hat, jedoch nicht mehr für einen Computer. Für diesen stellt ein rotiertes oder gespiegeltes Bild ein neues Bild dar. Diese Technik wird beim Trainieren von Bilderkennungen eingesetzt, um zum Beispiel aus geringen Datenmengen, die zum Trainieren verfügbar sind, mehrere zu generieren. Zusätzlich gibt es noch die Möglichkeit die Farbkanäle des Bildes zu ändern sowie das Bild nachzujustieren. [5]

Input und Readers

Platzhalter werden benötigt, um einen Graphen zu erstellen. Ohne Platzhalter können keine Daten in den Graphen von außerhalb geladen werden. Diese müssen zur Ausführungszeit durch echte Daten ersetzt werden. Dies erfolgt mit Hilfe eines Schlüssen-Wert-Paars in der Run-Methode der Session.

Readers ermöglichen es, aus dem Dateisystem Daten direkt zu laden. Dabei stehen spezifizierte Reader zur Verfügung, welche Tensoren direkt ausliefern, sowie Zeile für Zeile oder ganze Dateiinhalte liefern.

Konvertierungsoperationen ermöglichen es, Dateien, die mit TensorFlow Readers gelesen wurden weiter zu verarbeiten, so kann zum Beispiel eine CSV Datei decodiert verwendet werden.

[5] Online Dokumentation: Images www.tensorflow.org/api_guides/python/image

Des Weiteren sind Protokoll Buffer sowie Queues implementiert, die zum Vorverarbeiten von Daten dienen. [6]

Neuronale Netzwerke

Neuronale Netzwerke sind eine Spezialisierung im Gebiet des maschinellen Lernens. TensorFlow bietet eine breite Unterstützung beziehungsweise eine große Implementierungsvielfalt für diesen Typ an.

Aktivierungsfunktionen repräsentieren den Ausgang eines Neurons, dabei existieren aus der Vergangenheit einige Ansätze für diesen Bereich eines neuronalen Netzwerkes.

- *tf.sigmoid* ist eine der bekanntesten und ältesten diesen Typs. Diese Funktion besitzt im Punkt 0 einen Aktivierungswert von 0.5 und hat zwei Beschränkungen. Im negativen Zahlenbereich auf der X-Achse wird der Grenzwert der Funktion mit 0 definiert und im positiven Zahlenbereich mit dem Grenzwert von maximal 1. Ein negativer Wert führt somit zu einem geringen Aktivierungswert, welcher sich im Negativen an 0 annähert sowie im Positiven an 1. In der Abbildung 3.2 befindet sich diese Funktion mit ihren Grenzwerten.

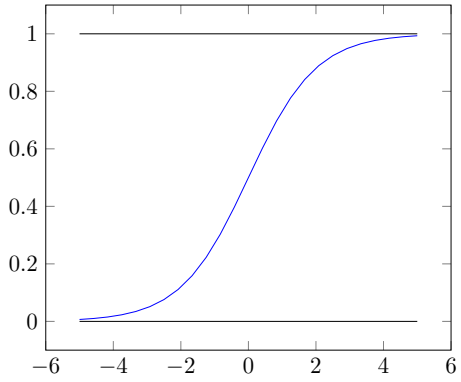


Abbildung 3.2: Sigmoide Aktivierungsfunktion mit den Grenzwerten an den Stellen 1 und 0

- *tf.relu* ersetzt mittlerweile immer mehr die sigmoiden Funktionen. Ein Grund dafür ist, dass die Berechnung eines Wertes in einer sigmoiden Funktionen Ressourcen intensiver ist und dass negative Werte meist nicht gewollt sind. Die rektifiziert lineare Funktion ist sehr viel einfacher, da Werte unter 0 als 0 weitergegeben werden und Werte darüber linear sind. Somit resultiert ein Eingangswert von -0.1 in einer 0 und

[6] Online Dokumentation: Input und Readers www.tensorflow.org/api_guides/python/io_ops

ein Wert von 0.5 in 0.5. Wie in der Abbildung 3.3 ersichtlich ist, führt dies bei einem negativen Wert dazu, dass sich eine Multiplikation mit der Gewichtung in der nächsten Ebene ebenfalls in einer 0 sich repräsentiert und somit in der Addition ignoriert wird.

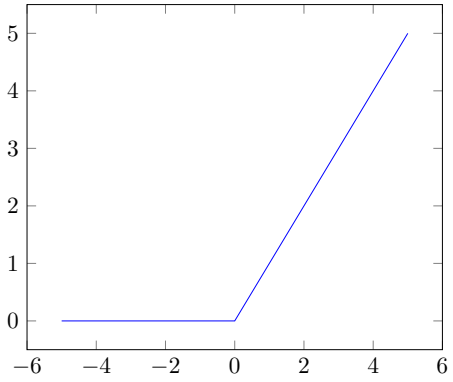


Abbildung 3.3: rektifiziert lineare Aktivierungsfunktion

- *tf.tanh* genannt Hyperbolic Tangent gehört ebenfalls zu den grundlegenden Aktivierungsfunktionen. Der Unterschied zwischen dieser Funktion und der sigmoiden Aktivierungsfunktion ist, dass der untere Grenzwert nicht bei 0 liegt sondern bei -1. In der Abbildung 3.4 lässt sich erkennen, wo sich der Wendepunkt befindet und liegt im Falle des Hyperbolic Tangent in der Koordinate $x = 0, y = 0$.

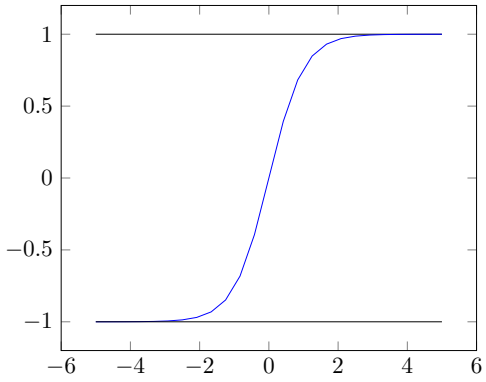


Abbildung 3.4: Hyperbolic Tangents Aktivierungsfunktion mit den Grenzwerten an den Stellen 1 und -1

Zu diesen Aktivierungsfunktionen stehen noch einige weitere zur Verfügung, die ausführlich getestet werden sollten. Im Grunde kann jede Funktion angewendet werden, doch besitzt jede eine Eigenheit und beeinflusst so den gesamten Graphen.

Faltung Operationen werden bei Bilderkennungen unter anderem deshalb verwendet, da eine Operation mit unterschiedlichen Daten in einem Schritt auf mehrere Daten angewendet werden kann. Dabei wird ein Fenster über ein Bild geschoben und auf jedem Bild wird im selben Fenster die Operation durchgeführt. Diese Operation generalisiert die darunterliegenden Daten so, als ob sie auf etwas reagiert hätten. Dies entspricht in etwa dem, als ob ein Auge auf etwas reagiert hätte.

- *tf.nn.conv2d* steht für zweidimensionale Bilder zur Verfügung.
- *tf.nn.conv3d* ermöglicht es mit dreidimensionalen Objekten zu arbeiten.

Des Weiteren stehen noch andere spezialisierte Versionen implementiert zur Verfügung.

Bündelung wird verwendet, um Daten zu vereinfachen, wie zum Beispiel „MaxPool“in Convolutional Netzwerken. Eine Faltungsoperation führt dazu, dass aus einem Bild viele mit unterschiedlichen Filtern erzeugt werden. Eine Bündelung ermöglicht eine Vereinfachung der Daten, wobei die Schlüsselinformationen dennoch erhalten bleiben sollen. TensorFlow bietet mehrere Umsetzungen, so kann sowohl ein Maximalwert aus der Filtermatrix, als auch der Mittelwert übernommen werden.

Verluste beschreiben, wie weit ein Ergebnis vom erwarteten Ergebnis entfernt liegt. Diese Art der Verlustfeststellung wird bei Regressionsproblemen benötigt, sie haben generell auch die Funktion des Regulierens.

- *tf.nn.l2_loss* berechnet die Hälfte der L2 Norm für den gegebenen Tensor. Im Falle dieser Implementierung wird keine Wurzel des Quadrats berechnet, sondern das Ergebnis der Summierung durch 2 dividiert.
- *tf.nn.log_poisson_loss* berechnet den logarithmischen Wahrscheinlichkeitsverlust zwischen dem Ergebnis und einem erwarteten Ergebnis. Diese Methode liefert nicht den exakten Verlust, dies stellt in Bezug auf Optimizer (3.1.2) aber kein Problem dar. Sollte trotzdem ein genauerer Wert bentötigt werden, muss die aufwändige Stirling Approximation [5] aktiviert werden.

Klassifizierungen repräsentieren einen großen Bereich des maschinellen Lernens. TensorFlow besitzt deshalb mehrere Hilfsfunktionen, welche das Arbeiten mit Klassifizierungen erleichtert.

- *tf.nn.softmax* bildet alle Ergebnisse auf einen prozentualen Bereich ab. So ergeben alle möglichen Ausgänge in Summe 100%, was so viel bedeutet, dass ein Ergebnis eine gewisse Wahrscheinlichkeit besitzt.
- *tf.nn.softmax_cross_entropy_with_logits* bietet die Möglichkeit, den Fehlerwert für Diskret-Klassifikationen zu berechnen, wobei jedes Er-

gebnis genau einer Klasse zugeordnet werden muss. Diese Funktion kann zum Trainieren verwendet werden, benötigt die unskalierten Werte des Netzwerkes und liefert für jeden Eintrag im Batch einen Fehlerwert.

Zu diesen existieren noch weitere Implementierungen mit weiteren Eigenheiten, welche in diversen Situationen möglicherweise einen Vorteil bieten.

Des Weiteren gibt es Implementierungen für rekursive neuronale Netzwerke und noch mehr. [7]

Running Graphs

Session stellt eine Hauptklasse des TensorFlow-Systems dar, mit der TensorFlow Engine im Hintergrund. In ihr werden alle Operationen ausgeführt und alle Tensoren evaluiert. Dieser Session wird ein Graphen mitgegeben, in dem der Endpunkt des Graphen angeben wird. Zur Ausführungszeit führt die Engine alle Operationen bis zum definierten Endpunkt des Graphen durch und evaluiert die Tensoren in diesem. Die Engine führt dabei alles bis zum gegebenen Punkt aus, welcher als Endpunkt übergeben wurde. Sollte der Graphen weiterführen, so wird dieser nicht mehr durchlaufen. Dies bietet eine eingeschränkte Möglichkeit, um das aufgebaute System zu testen. Eine Session wird mit *tf.Session* erstellt und stellt die Funktionalität zum Ausführen, sowie die Möglichkeit, diese zu schließen zur Verfügung. Mit *tf.InteractiveSession* wird ebenfalls eine Session erstellt, diese wird aber zugleich als Basissession installiert. Dies bietet die Möglichkeit, interaktiv in einer Kommandozeile Operationen auszuführen, ohne die Session explizit zu übertragen und anzusprechen. Die Tensoren und Operatoren bieten in diesem Fall die Option sich und den Graphen auszuführen, indem die Methoden *TensorVariable.eval* sowie *OperationsVariable.run* in diesen aufgerufen werden. [8]

Training

Optimizers stellen einen weiteren Kernteil des Systems dar. TensorFlow stellt eine Menge an implementierten Optimierungsalgorithmen zur Verfügung. Diese Operationen trainieren den Graphen mit der gewählten Technik des gewählten Algorithmus. Diese Implementierungen versuchen die gegebenen Kosten eines Graphen zu minimieren. Bei der Verwendung von *minimize* führt die Operation zwei Schritte in einem aus. In diesem wird der Gradient berechnet und dieser wird direkt auf die Variablen adaptiert. Diese Schritte können in einzelne zerlegt werden, wenn mit den berechneten Gradienten

[7] Online Dokumentation: Neural Network www.tensorflow.org/api_guides/python/nn
[8] Online Dokumentation: Running Graphs www.tensorflow.org/api_guides/python/client

noch etwas zusätzlich durchgeführt werden soll. Die Berechnung wird dabei mit *opt.compute_gradients* ausgelöst, was eine Liste mit Paaren liefert. Diese Liste kann bearbeitet werden, aber auch zu Testzwecken mit protokolliert werden. Die Gradienten werden im dritten Schritt mit *opt.apply_gradients* auf die Variablen angewendet. Jeder Optimierungsalgorithmus verfügt über Eigenheiten und spezielle Verhalten, welche bei der Auswahl des Optimierers berücksichtigt werden sollten.

Gradient Computation umfasst Methoden, die das Verhalten des Graphen und der Optimierung beeinflussen. Diese Methoden ermöglichen es, Einfluss auf die Gradientenberechnung sowie auf dessen Evaluierung zu nehmen. In diesem Sinne sind diese mit Vorsicht zu verwenden.

Verteilte Ausführung stellt eine der Stärken von TensorFlow dar, da diese Technologie schon im System integriert ist und somit keine manuelle Verteilung der Aufgaben entwickelt werden muss. Dadurch besteht die Option, die Berechnungen auf mehrere Geräte zu verteilen und so die zur Verfügung stehenden Ressourcen besser auszunützen.

Einige Komfortmethoden ermöglichen es, einfacher eine Session zu erstellen und alle Variablen zu initialisieren sowie im Anschluss zu trainieren, wobei eine Stoppbedingung mitdefiniert werden kann. In diesem Zuge können Hooks einfach in das System integriert werden, welche aufgerufen werden.
Im Weiteren kann Threading, sowie der Verfall der Lernrate beeinflusst werden. [9]
TensorFlow beinhaltet noch sehr viele weitere Komponenten und Möglichkeiten. Dies würde aber den Rahmen und den ersten Einblick in die Materie des maschinellen Lernens und im Speziellen von TensorFlow sprengen. Im Grunde kann mit diesen Grundlagen ein Netzwerk entwickelt werden und damit gearbeitet werden. Seit der Offenlegung kommen immer mehr Erweiterungen aus der Community dazu, was auch dazu führt, dass Teile die sehr oft benötigt werden und aus mehreren Komponenten bestehen, als Modul oder Funktion zur Verfügung stehen. Im Zuge dessen besteht die Möglichkeit, sich einen bestehen Graphen zu nehmen, welcher zum Teil schon vortrainiert worden ist. Im Zuge dessen werden nur mehr die letzten Ebenen des Graphen trainiert und auf die konkrete Aufgabe hin ausgelegt. Dies hat zur Folge, dass ein verwendbarer Graphen schneller vorhanden ist, dieser aber sehr wahrscheinlich nicht den Anforderungen entspräche.

[9] Online Dokumentation: Training www.tensorflow.org/api_guides/python/train

3.1.3 TensorBoard

TensorBoard stellt eine Erweiterung des TensorFlow-Systems dar, im Sinne einer Toolerweiterung. Jeder Graph kann in ein File serialisiert werden, welches als Event-File bezeichnet wird. Dies hat zur Folge, dass dieser auch wieder geladen werden kann. Bei dieser Serialisierung werden alle Informationen des Graphen inklusive der Gewichtungen in die definierte Datei gespeichert. TensorBoard bietet nun die Möglichkeit, diesen Graphen zu laden und zu visualisieren. Zu den Graph-Informationen kann jeder Tensor mitgespeichert werden und als Diagramm visualisiert werden, mit einer zeitlichen Komponente. Dies ermöglicht es, den Verlauf des Trainings zu analysieren. Aus einem Graphen können mehrere dieser Event-Files erzeugt werden sowie Zustände festgehalten werden. Beim Laden eines Graphen in die TensorFlow- sowie TensorBoard-Umgebung kann spezifiziert werden, zu welchen Zeitpunkt geladen werden soll. Damit wird ermöglicht, viele Trainingsdurchläufe zu durchlaufen und bei einer Verschlechterung der Präzision zu einem früheren Zustand zurückzuspringen. Dieses Tool ermöglicht es einem in das Verhalten eines Graphen ein wenig Einsicht zu nehmen und so die sogenannte Black Box zu durchleuchten.

Namensbereiche (*tf.name_scope*) stellen eine Hilfe für die Darstellung und die Lesbarkeit des visualisierten Graphen in TensorBoard dar. Durch die Verwendung des Python-Schlüsselwortes *with* wird eine Ressource verwaltet und wieder freigegeben. In Verwendung mit *tf.name_scope* werden alle Operationen und Tensoren in diesem Block in der Visualisierung in einen benannten Block zusammengefasst.

```
1 import tensorflow as tf
2
3 with tf.name_scope("func"):
4   b = tf.Variable(tf.zeros([100]))
5   W = tf.Variable(tf.random_uniform([784,100],-1,1))
6   x = tf.placeholder(name="x")
7   relu = tf.nn.relu(tf.matmul(W, x) + b)
8
9 C = [...]
10 s = tf.Session()
11 for step in xrange(0, 10):
12   input = ...construct 100-D input array ...
13   result = s.run(C, feed_dict={x: input})
14
15   print step, result
```

Listing 3.2: TensorFlow Codefragment zur Namescope Verwendung in Graphen

Wie im Codefragment 3.2 beschrieben, werden die Tensoren und Operatoren *b, W, x, relu* in einem Block zusammengefasst. In diesem Beispiel gibt es keinen Tensor, welcher in den Block übergeben wird, da die Daten in der

Ausführung von außerhalb des Systems in dieses gelangen. Die Operation *relu* und der daraus resultierende Tensor bilden den Ausgang des Blockes. Diese Technik der Namensbereiche ermöglicht es einem, den Graphen zu strukturieren, da nicht wie in *tf.zeros([100])* viele einzelne Knoten dargestellt werden, sondern abstrahiert werden, aber weiterhin einsehbar sind.

Graph bildet den Punkt zum Visualisieren des Graphen selbst. Hierbei werden aus dem Event-File alle Informationen zum Aufbau des Graphen geladen und visualisiert. Durch die Verwendung der Namensbereiche werden Gruppen gebildet, was dazu führt, dass die Gruppierungen sich möglicherweise in Ebenen widerspiegeln. Der dargestellte Graph kann nach dem Einlesen und Generieren interaktiv analysiert werden. So können Bereiche vergrößert und geöffnet werden und die definierten Tensoren betrachtet werden. Dieses Tool bietet zusätzliche Funktionalitäten, wie das Darstellen, an welchem Punkt die meiste Rechenzeit benötigt wurde, sowie auch welche Berechnungen auf welchem Gerät ausgeführt worden sind. Alle diese zusätzlichen Funktionalitäten benötigen Daten, welche beim Erstellen des Graphen mit definiert werden müssen und auch mit in das Event-File serialisiert werden müssen.

Scalars repräsentiert den Bereich, in welchem die Lernergebnisse dargestellt werden können. Dies umfasst die Präzision sowie die Verluste. Das Ziel des Graphen ist im Grunde immer, die Präzision zu erhöhen und die Verluste zu minimieren. Aus diesem Grund sollte sich die Genauigkeit an 1 annähern, außer die Definition dieser Berechnung liefert andere Werte oder besitzt einen anderen Grenzwert. Der Verlust sollte sich im Laufe des Trainings an 0 annähern, denn dadurch spiegelt sich die Fehlerquote wider. Dies hängt aber wieder vom entwickelten Graphen ab und kann sich somit einem anderen Wert annähern.
TensorBoard bietet die Möglichkeit, mehrere Graphen und ihre Eventdaten zu visualisieren. In diesem Fall werden alle gelesenen Events in einer Liste aufgeführt, in der ausgewählt werden kann, welche Ausführungen in den Diagrammen dargestellt werden sollen. Im Zuge dessen können diese Diagramme zusammengeführt werden und so die Ergebnisse direkt verglichen werden. Dies hat den Vorteil, dass ein Netzwerk mit Hyperparametern automatisch getestet werden kann und jede Kombination ein eigenes Event-File erzeugt. Solche Testdurchläufe benötigen mehr Zeit, abhängig von den definierten Kombinationen an Parametern, muss $x * y * ...$ durchgetestet werden.

Die Informationen für die Lernrate sowie des Verlustes werden in diesem Fall am besten festgehalten. Dies wird ermöglicht, indem die Tensoren, welche die Lernrate sowie den Verlust beinhalten, in die Methode *tf.summary.scalar* jeweils gefüttert werden. Bei jedem Schreibzyklus in das Event-File werden

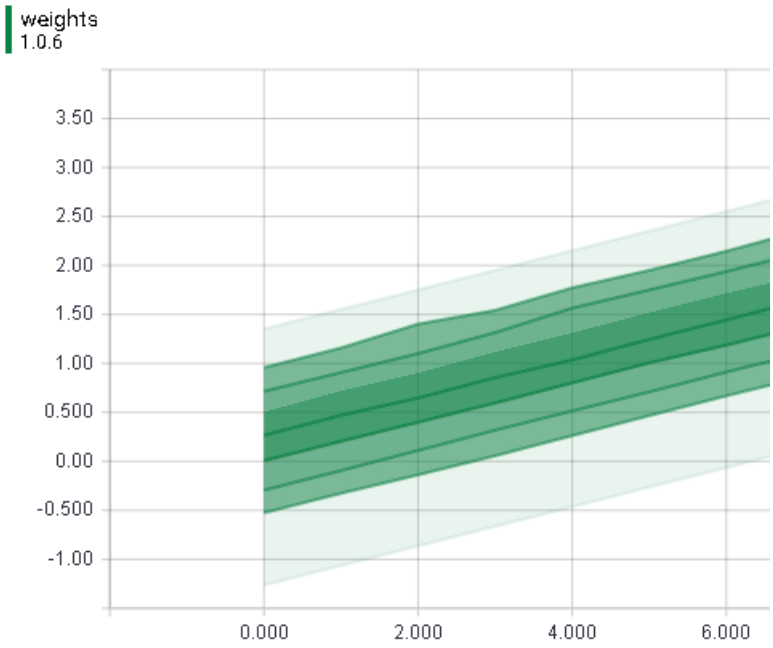


Abbildung 3.5: Verteilung der Werte in einem Tensor über die Zeit

diese Informationen dann mit übernommen und stehen daraufhin in TensorBoard zur Verfügung.

Distributions stellt eine weiter Funktionalität von TensorBoard dar. Diese Funktionalität war in den Versionen vor 'r1.0' noch unter dem Punkt Histogramm. Im Allgemeinen werden Tensoren mit der Methode *tf.summary.histogram* wieder in das Event-File serialisiert. Das Ergebnis stellt eine Verteilung für die Werte dar, die im Tensor vorkommen. Das Diagramm repräsentiert auf der X-Achse die Anzahl der Schritte, die durchgeführt worden sind. Die Y-Achse gibt die konkreten Werte wieder, welche sich im Tensor über die Zeit befinden. Im Diagramm 3.5 wird ein Tensor mit 100 Werten dargestellt. Dieser Tensor wurde mit einer Normalverteilung initialisiert, wobei die Standardabweichung bei 0.5 liegt und der Median zu Beginn bei 0.2, mit einer geringen Abweichung. Die Linien in diesem Diagramm und ihre Einfärbungen präsentieren die Verteilung der Werte im beobachteten Tensor. Die Verteilung muss von unten nach oben gelesen werden, dabei ergibt die unterste Linie den Minimalwert, der vorgekommen ist. Die nächste Linie besagt, dass sich 7% der Werte im Bereich zwischen dem geringsten und der zweiten Linie befinden, was inklusiv des geringsten Wertes ist. Der nächste Bereich definiert, wie in einer gaußschen Normalverteilung, dass sich bis zum Ende dieses Bereiches 16% darin befinden. Im Gesamten sind dies 9 Markierungen mit 8 Bereichen, welche zusammen alle Werte

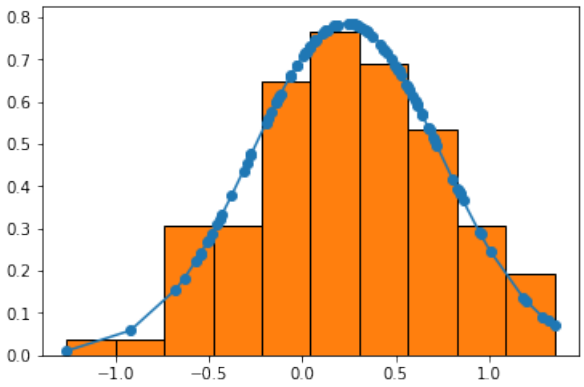


Abbildung 3.6: Verteilung der Werte in dem Tensor zu dem Diagramm 3.5

im Tensor widerspiegeln. Diese Folge an prozentualen Anteilen lauten wie folgend: $min, 7\%, 16\%, 31\%, 50\%, 69\%, 84\%, 93\%, max$. Im Diagramm 3.6 ist diese Verteilung besser ersichtlich, zusätzlich befinden sich in der Abbildung 3.3 die Rohdaten der Diagramme.

```
 1  -1.2645409,  -0.92252451, -0.68004417,  -0.63273954,  -0.57005012,
 2  -0.5477351,  -0.54554129, -0.51146084,  -0.50482351,  -0.4872272,
 3  -0.45631331, -0.44180638, -0.43258488,  -0.38066232,  -0.31675094,
 4  -0.29567719, -0.27768314, -0.27437925,  -0.19503899,  -0.18343721,
 5  -0.16653274, -0.13992153, -0.13946836,  -0.12969615,  -0.12044857,
 6  -0.11908005, -0.06734778, -0.062724337, -0.032598898, -0.02885592,
 7   0.006216079, 0.015204117, 0.018379062,  0.036883533,  0.041039094,
 8   0.063002124, 0.068820029, 0.072805718,  0.11137276,   0.11735194,
 9   0.12555882,  0.12613684,  0.13053563,   0.13633718,   0.17283598,
10   0.18271323,  0.18530971,  0.18671049,   0.24375655,   0.25207496,
11   0.27566099,  0.27588493,  0.27921408,   0.28581429,   0.29526407,
12   0.30613232,  0.32309669,  0.33705187,   0.34577289,   0.34687665,
13   0.37553167,  0.41834235,  0.43759531,   0.4376972,    0.45076531,
14   0.47984695,  0.49715465,  0.50634104,   0.51550949,   0.5168677,
15   0.53031796,  0.5579083,   0.56285316,   0.57165861,   0.59320259,
16   0.60513371,  0.61539149,  0.61814398,   0.63975775,   0.64333171,
17   0.67751783,  0.67795348,  0.68242437,   0.70252627,   0.70793462,
18   0.72128826,  0.80693412,  0.83029318,   0.83635086,   0.84400082,
19   0.84558558,  0.86151552,  0.95068389,   0.95598722,   1.0072051,
20   1.1837469,   1.1992682,   1.285683,     1.3168017,    1.3521272
```

Listing 3.3: Sortierter Ergebnistensor zum Verteilungsdiagramm 3.6 und 3.5 im Schritt 0

Histogram passiert auf denselben Daten wie die Verteilungsansicht. Im Grunde präsentiert diese Ansicht die Daten nur auf eine andere Art und Weise. Wie auch in der anderen Darstellung werden die Schritte direkt auf einer Achse dargestellt. Im Falle des Histogramm 3.7 ist dies die Achse, welche

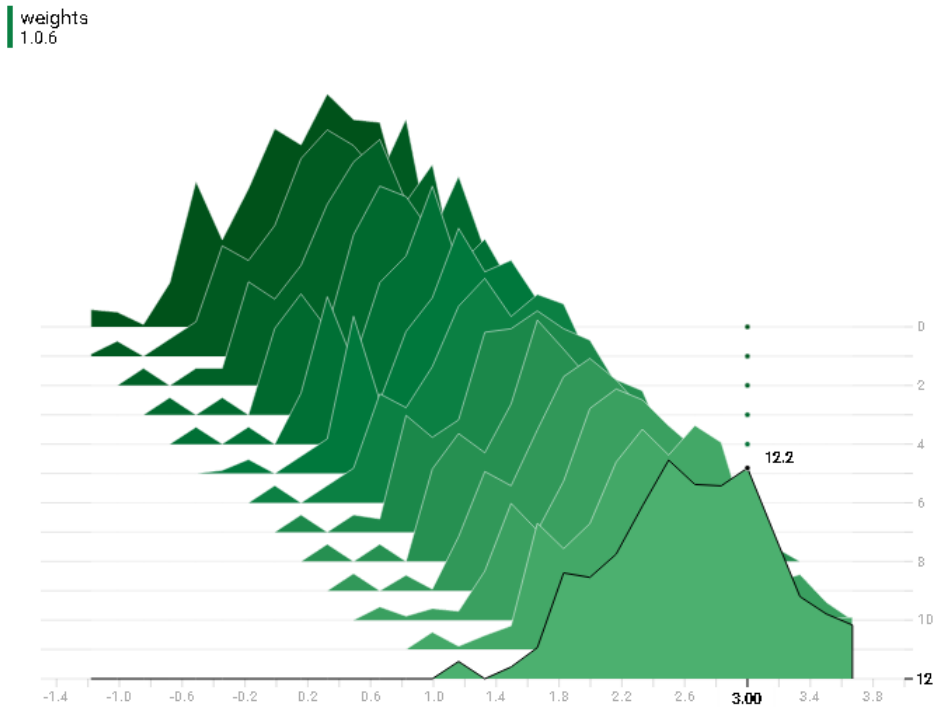


Abbildung 3.7: Verteilung der Werte in dem Tensor in einem Histogramm

sich dreidimensional aus dem Hintergrund des Bildes in den Vordergrund zieht. Die horizontalen Achsen, welche zu jedem Schritt gezeichnet werden, projizieren ihre Wertebereiche immer auf die vorderste Achse. Auf dieser wird der Wertebereich abgebildet, in welchem sich die Werte im Tensor befinden. Die Erhebungen und die sich darunter bildenden Flächen geben die Verteilung der Werte wieder. So wird beim Überfahren eines Schrittes mit der Maus dieser aktiviert, wie im Histogramm 3.7 ersichtlich ist. Im Falle dieses Diagramms und dieser Stelle bedeutet das, dass sich in der Nähe des Wertes 3.00 ungefähr 12.2 Einträge im Tensor befinden. Anders ausgedrückt haben 12.2 konkrete Werte im Tensor den Wert 3.00 oder einen naheliegenden. Durch die Verteilung der Werte entsteht nun der Fall, dass ein Teil der Einträge auch zu einem Wertebereich vorher oder nachher gehören kann. Dieses Diagramm stellt die Verteilung in Wertebereiche dar, wobei alle vertikalen Werte in einem Schritt die Anzahl der Werte im Tensor ergeben müssen. Im Falle dieses Beispiels ergeben sie aufsummiert einen Wert von 100.044, was gerundet die 100 Einträge im Tensor bestätigt. Die Aufteilung der Werte in Wertebereiche mit Teilzuweisungen erklärt auch den Schrittverlauf im Histogramm 3.7. Hier ist ersichtlich, dass sich die Verteilungen und Zugehörigkeiten immer ein wenig vom vorhergehenden abweichen, obwohl in diesem Beispiel in jedem Schritt konstant 0.2 zu jedem Wert hinzu

addiert worden ist. Dies lässt sich bei einer geringen Anzahl an Werten, wie hier mit 100 leichter beobachten, als bei einer sehr viel höheren.
TensorBoard bietet noch weitere Möglichkeiten, wie Bilder- oder Soundinhalte mit in das Event-File zu geben, um diese dann in Tensorboard weiter zu verwenden. So können diese Inhalte durch den Graphen gesendet werden und dabei beobachtet werden. Die letzte Erweiterung in Tensorboard ist der Punkt mit 'Embedding' [10], wo gelernte Informationen, so wie sie vom Graphen gruppiert worden sind, dargestellt werden können.
Dieses Kapitel repräsentiert die grundsätzliche Funktionalität des TensorFlow-Systems. Dabei wurde auf die Grundlagen und die am meisten benötigten Methoden eingegangen. Ihr Verständnis stellt die Grundlage für das nächste Kapitel dar. In diesem wird ein praktisches Beispiel mit TensorFlow erläutert.

[10] Online Dokumentation: Emeddings www.tensorflow.org/get_started/embedding_viz

Kapitel 4

Facial Keypoints Detection

4.1 Ausgangssituation

Gesichtserkennung spielt im 21. Jahrhundert eine immer größer werdende Rolle. So existieren Herausforderungen mit den Schlüsselpunkten im Gesicht eines Menschen, welche wieder für Gesichtserkennungen verwendet werden können. Im Gesicht eines Menschen wird zum Beispiel die Iris in beiden Augen als Schlüsselpunkt definiert, aber auch die Nasenspitze und die Mundwinkel sowie weitere. Diese Schlüsselpunkte variieren sehr stark von einem Individuum zum Nächsten, jedes Individuum hat eine Menge an Variationen. So spielt hier die Größe, die Position, die Neigung sowie die Beleuchtung eine Rolle und erzeugt eine fast unendliche Menge an Möglichkeiten. Mit Computer Vision konnten sehr viele Verbesserungen in diesem Bereich erzielt werden, wobei noch sehr viel Raum für Forschung und Verbesserungen bleibt.

Die Aufgabenstellung wird auf der Online-Plattform Kaggle [1] gehostet, wo auch die Trainings- und Testdaten zur Verfügung gestellt werden. Diese Aufgabe war im Jahr 2016 eine Herausforderung, an der sich jeder beteiligen konnte, um den ersten Platz zu erreichen. Das Ziel für jeden Teilnehmer war es ein System zu entwickeln, welches mit den Trainingsdaten trainiert wird. Im Anschluss sollte dieses System dann mit den Testdaten getestet werden. Das Ergebnis musste im Anschluss eingereicht werden, welches dann überprüft und bewertet wurde.

[1] Kaggle: Facial Keypoints Detection https://www.kaggle.com/c/facial-keypoints-detection

4.2 Vorbereitung

Die Daten, die zur Verfügung gestellt werden, sind auf mehreren Dateien aufgeteilt. In diesem Fall beinhaltet die Datei mit den Trainingsdaten die meisten Daten. Sie beinhaltet die Bilder der Gesichter, sowie die Koordinaten der Schlüsselpunkte, gespeichert in einer CSV-Notation. Im gesamten Gesicht werden in diesem Beispiel 15 Schlüsselpunkte berücksichtigt. In der Auflistung 4.1 sind die Bezeichnungen der Spalten mit den Werten ersichtlich, wobei zu jeder Bezeichnung ein „X“und ein „Y“existiert. Ein Beispiel dazu ist in der Auflistung 4.2 zu finden.

```
1 left_eye_center, right_eye_center,
2 left_eye_inner_corner, left_eye_outer_corner, right_eyee_inner_corner,
      right_eye_outer_corner,
3 left_eyebrow_inner_end, left_eyebrow_outer_end, right_eyebrow_inner_end,
       right_eyebrow_outer_end,
4 nose_tip,
5 mouth_left_corner, mouth_right_corner,
6 mouth_center_top_lip, mouth_center_bottom_lip
```

Listing 4.1: 15 Schlüsselpunkte im Gesicht eines Menschen

Jeder diese Schlüsselpunkte besteht aus einer X und einer Y Koordinate. Diese Datei besitzt zusätzlich in der 31. Spalte das Bild mit dem dazugehörigen Gesicht. Das Bild ist encodiert abgelegt und besteht aus $96 * 96$ Werten. In diesem Sinne stehen nur Bilder in Graustufen zur Verfügung mit einer Auflösung von $96 * 96$ Pixel und einer Farbtiefe von $[0, 255]$, wie in der Abbildung 4.1 zu erkennen ist. Für diese Aufgabe werden im Grunde auch nur die Konturen benötigt, was somit den einen Farbkanal erklärt, aber auch die Aufgabe schwieriger macht, denn mehrere Farbkanäle könnten mit Filter bearbeitet werden und so noch weitere Konturen hervorheben.

```
1  # left_eye_center, right_eye_center
2  66.0335639098,39.0022736842,30.2270075188,36.4216781955
3  # left_eye_inner_corner, left_eye_outer_corner
4  59.582075188,39.6474225564,73.1303458647,39.9699969925
5  # right_eye_inner_corner, right_eye_outer_corner
6  36.3565714286,37.3894015038,23.4528721805,37.3894015038
7  # left_eyebrow_inner_end, left_eyebrow_outer_end
8  56.9532631579,29.0336481203,80.2271278195,32.2281383459
9  # right_eyebrow_inner_end, right_eyebrow_outer_end
10 40.2276090226,29.0023218045,16.3563789474,29.6474706767
11 # nose_tip
12 44.4205714286,57.0668030075
13 # mouth_left_corner, mouth_right_corner
14 61.1953082707,79.9701654135,28.6144962406,77.3889924812
15 # mouth_center_top_lip, mouth_center_bottom_lip
16 43.3126015038,72.9354586466,43.1307067669,84.4857744361
17 # image
```

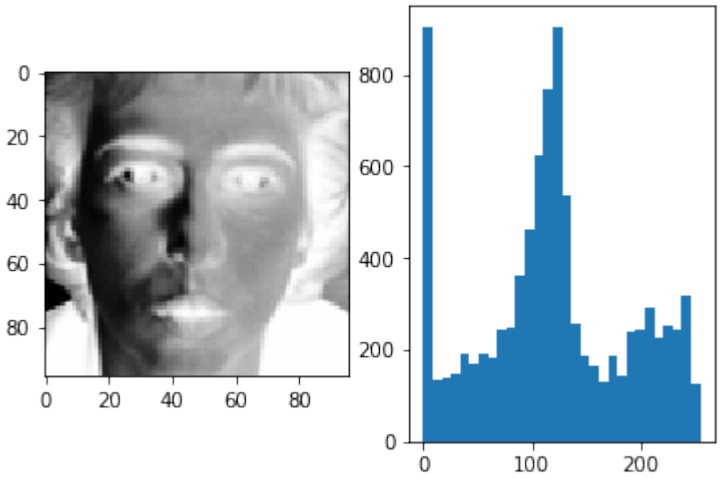


Abbildung 4.1: Ein Gesicht aus dem Datenbestand mit der Verteilung der Graustufenwerten

```
18 238 236 237 238 240 240 239 241 241 243 240 239 231 212 ...
```

Listing 4.2: Ein gesamter Datensatz aus den Trainingsdaten mit den X und Y Werten pro Schlüsselpunkt

4.2.1 Daten vorbereiten und normalisieren

Zu Beginn diese Daten müssen vorbereitet und normalisiert werden. Um die Problemgröße zu verringern, empfiehlt es sich, die Aufgabe auf mehrere Netzwerke aufzuteilen. Dies hat einen zusätzlichen Grund, da die Trainingsdaten nicht komplett sind und somit nicht zu allen Bildern alle 15 Schlüsselpunkte vorhanden sind. Sollte dies ignoriert werden, würde sich die Anzahl der zur Verfügung stehenden Datensätze von 7049 auf 2140 reduzieren. Ein weiterer Grund für die Auftrennung der Problemstellung ist, dass Aufgaben leichter verteilt werden können und die Netzwerke noch genauer angepasst werden könnten.

Unter der Zuname von Pandas[2] und NumPy[3] besteht die Möglichkeit, sehr einfach und effizient auf die Datensätze zuzugreifen und diese zu verwenden. Wie im Codebeispiel 4.3 ersichtlich ist, werden die Daten mit Hilfe von Pandas geladen und unter Zunahme von NumPy normalisiert und neu strukturiert.

```
1 import pandas as pd
```

[2]Pandas: Python Data Analysis Library http://pandas.pydata.org/
[3]NumPy: Scientific Computing http://www.numpy.org/

```
2 import numpy as np
3
4 # konstanten Definition
5 IMAGE_SIZE = 96
6
7 # Daten einlesen
8 df = pd.read_csv('~/training.csv')
9
10 # Bilder um konvertieren in eine List von Zahlen
11 df['Image'] = df['Image'].apply(lambda im: np.fromstring(im, sep=' '))
12
13 # die Aktuell benötigten Spalten herausnehmen
14 df = df[['left_eye_center', 'right_eye_center', 'Image']]
15
16 # entfernen unvollständiger Datensätze
17 # Verringerung der Datensätze von 7049 auf 7033
18 df = df.dropna()
19
20 # normalisieren der Bilder in einen Wertebereich von [0, 1]
21 # und überführen in eine 96 mal 96 Matrix
22 # Variable X beinhaltet alle Bilder des Datensatzes welche Vollständig sind
23 X = np.vstack(df['Image']) / 255.
24 X = X.reshape(-1, IMAGE_SIZE, IMAGE_SIZE, 1)
25
26 # explizites definieren des Datentyps für die Werte der Bilder
27 X = X.astype(np.float32)
28
29 # narmalisieren der Y Koordinaten in einen Wertebereich von [0, 1]
30 # Variable Y beinhaltet die Labels zu allen Bildern
31 Y = df[df.columns[:-1]].values
32 Y = Y / 96.0
```

Listing 4.3: Daten einlesen und einschränken

4.2.2 Evaluations- und Errorfunktion

Die Ergebnisse des Netzwerkes müssen immer verglichen und validiert werden. In diesem Beispiel handelt es sich nicht um eine Klassifizierungsaufgabe, sondern um eine Regressionsproblemstellung. Deshalb kann nicht einfach eine 'Cross Entropy' Funktionen verwendet werden, um die Daten zu evaluieren und zu adaptieren. Aus diesem Grund muss dies manuell durchgeführt werden und selbst eine Berechnung aufgestellt werden, welche diese Werte liefert, damit diese einem Optimierer übergeben werden können. Der Verlust, beziehungsweise die Differenz zwischen Ergebnis des Netzwerkes und dem bekannten Ergebnis, kann durch eine Subtraktion sowie einer Quadrierung berechnet werden. Diese ist in der Gleichung 4.1 zu erkennen, wobei *graph* eine Matrix (Tensor) an Ergebnissen ist, mit der Anzahl an Zeilen wie in der Konstante *BATCH_SIZE* definiert. Die Variable *train_labels_node* beinhaltet die bekannten Ergebnisse zu Bildern mit denselben Dimensionen wie in der Matrix *graph*. *tf.subtract* führt eine Subtraktion auf jeden einzelnen Wert

der beiden Matrizen aus, was auch dazu führt, dass diese dieselben Dimensionen haben müssen. *tf.square* quadriert die berechneten Differenzen um negative Werte zu entfernen. Zum Abschluss werden alle Ergebnisse in der Ergebnismatrix mit *tf.reduce_sum* aufsummiert, was zu einem Skalar führt. Für diese konkrete Implementierung wurde als Optimierungsalgorithmus ein *Adam*-Algorithmus [9] verwendet. Um das Verlustergebnis für einen Leser lesbarer zu machen, muss der Verlustwert durch die Anzahl der Batchgröße dividiert werden, was die Differenz in einem Datensatz als Durchschnitt ergibt, ohne die Quadrierung zu berücksichtigen. Die gesamte Umsetzung ist im Codebeispiel 4.4 zu finden.

$$Verlust := \sum (R - L)^2 \tag{4.1}$$

```
1 import tensorflow as tf
2
3 # Konstanten Definition
4 BATCH_SIZE = 20
5
6 # Verlustberechnung
7 with tf.name_scope("loss"):
8     # sollte sich im Laufe der Trainingsphasen an 0 annähern
9     loss = tf.reduce_sum(
10           tf.square(
11             tf.subtract(graph, train_labels_node)))
12
13 # Auswahl eines konkreten Optimierungsalgorithmuses in der Kurzschreibweise
14 # mit einer Lernrate von 0.00001
15 with tf.name_scope("train"):
16     train = tf.train.AdamOptimizer(learning_rate=1e-5).minimize(loss)
17
18 # Verlustwert durch die Anzahl der Bilder im Batch da diese
19 # in der loss-Berechnung zusammen summiert werden
20 with tf.name_scope("accuracy"):
21     accuracy = loss / BATCH_SIZE
```

Listing 4.4: Verlustberechnung, konkreter Opitimierungsalgorithmus, Genauigkeitsberechnung

4.3 Neuronale Ebenen vorbereiten

Damit die Ebenen einfacher verwendet werden können, können diese als konfigurierbare Muster definiert werden. Dadurch wird erzielt, dass gleiche Ebenen im visualisierten Graphen dieselbe Farbe besitzen und zum anderen alle Inhalte darin zusammengefasst werden. Im Grunde existieren zwei verschiedene Haupttypen an Ebenen. Zum einen die Convolutional-Ebenen und zum anderen die Vollvernetzten-Ebenen. Wie im Code 4.5 ersichtlich ist, besitzen beide Hauptgruppen an Ebenen jeweils eine Datenquelle, beschrieben als $x\_$ und Gewichtungen und Biaseswerte. Die Bias-Werte werden dabei

erst nach der Kernfunktion an das Ergebnis angefügt und somit erst in der Aktivierungsfunktion berücksichtigt.

```
def conv_layer(x_, size_in, size_out, name="conv"):
    with tf.name_scope(name):
        weights = tf.Variable(tf.truncated_normal(
                    [3, 3, size_in, size_out],
                    dtype=tf.float32, stddev=1e-1),
              trainable=True, name='weights')
        conv = tf.nn.conv2d(x_, weights, [1, 1, 1, 1], padding='SAME')
        biases = tf.Variable(tf.constant(0.0,
                 shape=[size_out], dtype=tf.float32),
                             trainable=True, name='biases')
        bias = tf.nn.bias_add(conv, biases)
        conv = tf.nn.relu(bias, name="act")

    maxPool = tf.nn.max_pool(conv,
                             ksize=[1, 2, 2, 1],
                             strides=[1, 2, 2, 1],
                             padding='VALID')
    return conv

def fc_layer(x_, size_out, name="fc", act=None):
    with tf.name_scope(name):
        size_in = x_.get_shape()[1].value
        weights = tf.Variable(tf.truncated_normal([size_in, size_out],
                dtype=tf.float32, stddev=1e-2),
              trainable=True, name='weights')
        biases = tf.Variable(tf.constant(0.0, shape=[size_out],
                dtype=tf.float32),
              trainable=True, name='biases')
        mul = tf.nn.xw_plus_b(x_, weights, biases)

        if act is not None:
            mul = act(mul)
        return mul
```

Listing 4.5: Definition der Convolutional- und Vollvernetzten-Ebenen

4.4 Neuronale Ebenen verknüpfen

Diese Definitionen der Ebenen aus dem Codefragment 4.5 müssen im nächsten Schritt zu einem Netzwerk zusammengesetzt werden. Ein neuronales Netzwerk, welches relativ leichtgewichtig ist und diese Problematik relativ brauchbar lösen kann, ist im Grunde ein sehr vereinfachtes 'VGG16' Netzwerk[4]. Für diesen Fall besteht dieses aus 3 Convolutional-Ebenen und 3 Vollvernetzten-Ebenen, wie im Codefragment 4.6 zu sehen ist. Das originale VGG16 Netzwerk besitzt im Gegensatz zum aktuell verwendeten, 5

[4] **VGG16: https://arxiv.org/pdf/1409.1556.pdf**.

Convolutional-Ebenen mit je 2 integrierten Convolutional-Ebenen mit denselben Dimensionen, die Hauptebenen 3 bis 5 zusätzlich eine 3 Convolutional-Ebene. Die letzte Ebene der Vollvernetzten-Ebenen wird dabei ohne Aktivierungsfunktion ausgeführt, um die Roh-Ergebnisse zu bekommen. Im aktuellen Fall werden für die 2 Iris-Positionen im Gesicht 4 Ergebnisse benötigt. An diesem Beispiel lässt sich erkennen, dass hier Variablen wiederverwendet werden, dies ist durch Python möglich und durch die Beschreibung des Graphen, welcher im Hintergrund aufgebaut und verbunden wird.

```
def model(data):
    net = conv_layer(data, 1, 32, "conv1")
    net = conv_layer(net, 32, 64, "conv2")
    net = conv_layer(net, 64, 128, "conv3")

    # Transformieren in eine flache Struktur
    dims = net.get_shape()[1:]
    k = dims.num_elements()
    with tf.name_scope('flatten'):
        net = tf.reshape(net, [-1, k])

    net = fc_layer(net, 256, "fc1", tf.nn.relu)
    net = fc_layer(net, 256, "fc2", tf.nn.relu)
    net = fc_layer(net, 4, "fc3")

    return net

# Definition der Platzhalter für die Datenübergabe
train_data_node = tf.placeholder(tf.float32,
            shape=(BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, 1))
train_labels_node = tf.placeholder(tf.float32,
            shape=(BATCH_SIZE, 4))

# erstellen eines Graphen mit dem definierten Model
graph = model(train_data_node)
```

Listing 4.6: Modelldefinition des Graphen

4.5 Trainieren

Um das erzeugte Netzwerk aus 4.6 verwenden und trainieren zu können, muss dieses zuerst initialisiert werden. Wie im Code 4.7 zu erkennen ist, wird eine globale Initialisierung verwendet, welche alle Konstanten und Variablen der aktuellen Umgebung initialisiert. In der For-Schleife werden fast alle Datensätze einmal durch den Graphen gesendet und entweder zum Trainieren oder Evaluieren verarbeitet. Der Session wird in der *Run*-Methode eine Liste an Punkten des Graphen mitgegeben, welche evaluiert werden sollen. In der Trainingsphase ist dies der Optimierungsendpunkt und in der Evaluierungsphase die Punkte *accuracy* und *graph. accuracy*, damit ein vergleichbarer Wert zur Verfügung steht, an welchem der Lernfortschritt erkennbar

ist und der Hauptendpunkt des Graphen selbst, damit die Ergebnisse direkt in Bilder gezeichnet werden können. Dies ermöglicht eine visuelle Verifikation durch einen Supervisor. Dieses Codefragment ergibt eine sogenannte Epoche, in der alle Datensätze einmal vom Netzwerk verarbeitet werden.

```
# erstellen einer Session
sess = tf.Session()

# erstellen einer globalen Initialisierungsroutine
init_op = tf.global_variables_initializer()
# initialisieren aller Konstanten und Variablen
sess.run(init_op)

# durchlaufen aller Datensätze -> 1 Epoche
runing = train_data.shape[0] // BATCH_SIZE
for i in range(runing):
    offset = i * BATCH_SIZE

    # laden eines Datenbatches aus den Datensätzen
    batch_data = train_data[offset:(offset + BATCH_SIZE), ...]
    batch_labels = train_labels[offset:(offset + BATCH_SIZE)]

    # ersetzen der Platzhalter durch die konkreten Daten
    feed = {train_data_node: batch_data,
            train_labels_node: batch_labels}

    # ausführen des Graphen zur Evaluierung
    if i % 5 == 0:
        [train_accuracy, data] = sess.run([accuracy, graph],
                          feed_dict=feed)
        print data[0:4]
        print batch_labels[0:4]

        print i, train_accuracy
    # ausführen einer Trainingsiteration
    else:
        sess.run(train, feed_dict=feed)
```

Listing 4.7: Initialisierung des Graphen und Durchführen einer Epoche

4.6 Validierungsergebnisse

Durch längeres Trainieren kann sich das Netzwerk entwickeln und im Lauf der Epochen bessere Ergebnisse liefern. Dies führt natürlich auch zu der Möglichkeit, dass das Netzwerk beginnt, Muster zu speichern anstatt zu lernen, auch bekannt als Overfitting. In der Abbildung 4.8 sind die Ergebniszustände am Ende der 100. Epoche sowie am Ende der 150. Epoche zu sehen. Im Gesamten kann festgestellt werden, dass das Netzwerk seine Arbeit relativ korrekt erledigt, wenn man die Größe und Dimension des Netzwerkes berücksichtigt.

```
.... Epochen 100 ....
# Ist-Ergebnis
[[ 0.66568404  0.38626809  0.35157766  0.4218266 ]
 [ 0.68138432  0.40295351  0.27889865  0.40295351]
 [ 0.676839    0.4047336   0.3197888   0.397911  ]
 [ 0.66469301  0.41260486  0.29902736  0.42368389]]
# Soll-Ergebnis
[[ 0.69634622  0.37385067  0.32604855  0.38231322]
 [ 0.70251364  0.34271669  0.32569841  0.3908942 ]
 [ 0.67929983  0.36809221  0.2629534   0.36637166]
 [ 0.71011567  0.39950868  0.3037816   0.3843804 ]]
# Trainingsgenauigkeit
0.0001787
.... Epochen 150 ....
# Ist-Ergebnis
[[ 0.6832875   0.381525    0.32855     0.40945625]
 [ 0.67460902  0.36388947  0.33187218  0.38741128]
 [ 0.68836184  0.38701776  0.29144079  0.38995789]
 [ 0.6741519   0.37355443  0.32906962  0.40749367]]
# Soll-Ergebnis
[[ 0.67965472  0.3785463   0.32667899  0.40375352]
 [ 0.67585701  0.35977855  0.33285627  0.38183641]
 [ 0.68490797  0.38329497  0.29059213  0.3844119 ]
 [ 0.67040122  0.3710691   0.32904905  0.40210161]]
# Trainingsgenauigkeit
5.38928e-05
```

Listing 4.8: Ergebnisse am Ende der 100. Epoche und am Ende der 150. Epoche

Zum besseren Feststellen des Ergebnisses befindet sich in der Abbildung 4.2 ein direkter Vergleich. Dabei wird der Istzustand durch blaue Punkte gekennzeichnet.

4.7 Ergebnis

In der Abbildung 4.3 sind die gesamten Trainingsergebnisse in Form der Verluste aufgezeichnet. Diese wiederum ergeben sich aus dem quadrierten Abstand zwischen Soll- und Ist-Punkt. In diesem Fall nähert sich der Verlustwert der 0-Grenze, trotzdem scheint es ein Probleme mit einigen Daten zu geben. Eine mögliche Interpretation wäre, dass die Lernrate mit '$1e-5$' noch zu hoch definiert wurde und sich das Netzwerk so zu schnell einem lokalen Minima genähert hat, obwohl es möglicherweise ein globales Minima gegeben hätte. Eine andere Möglichkeit wäre, dass die Datensätze Bilder beinhalten, in welchem das Gesicht nicht vollständig zu erkennen ist oder dass diese stärker rotiert sind. Sollte es nicht so viel Datensätze mit diesen Eigenheiten geben, so könnte dies ein weiterer Grund sein warum das Netzwerk springt. Dies bedeutet, dass ein nicht stabiler Zustand vorhanden ist und möglicherweise ein Rauschen in den Daten nicht ignoriert wurde, sondern darauf reagiert wird. In der Abbildung 4.3 lässt sich erkennen, dass

Abbildung 4.2: Ist-Visualisierung des Ergebnisses

die Sprünge nicht regelmäßig sind, was auf ein Mischen der Datensätze am Ende jeder Epoche zurückgeführt werden kann.

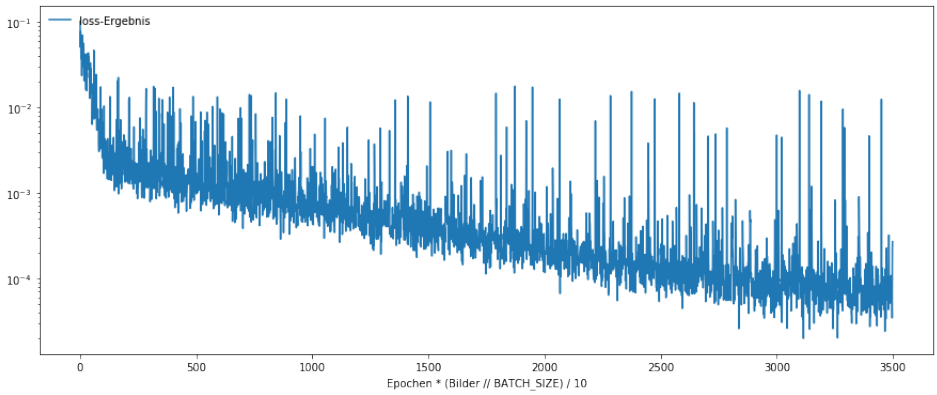


Abbildung 4.3: Trainings-Verlustergebnisse in 100 Epochen mit 350 Tests pro Epoche

Durch einen Test aller Datensätze und Herausfinden des besten und des schlechtesten Ergebnisses, lässt sich feststellen warum das Netzwerk diverse Probleme mit Bildern besitzt. Im Konkreten beinhalten die Daten einige wenige dubiose Bilder. Ein Beispiel dazu befindet sich in der Abbildung 4.4. Hier ist erkennbar, dass in den zwei oberen Bildern mehrere Bilder auf einem

WhiteBoard vorhanden sind. Zum einen ist hier das Problem, dass sich hier mehr als ein Gesicht im Bild befindet und das Netzwerk nicht für so einen Fall ausgelegt und trainiert worden ist. Zum anderen lässt sich erkennen, dass das Netzwerk hier ein anderes Bild bevorzugt, auf welchem ebenfalls Menschen abgebildet sind. Dies führt zu der Annahme, dass das Netzwerk sich hier nicht die Positionen durch das Trainieren gemerkt hat und dies als Rauschen wahrnimmt, sowie sich an die sonst zentralisierten Gesichter orientiert und trotzdem versucht hier zwei Augen zu finden, welche mehr oder weniger nebeneinander liegen sollten. In den zwei unteren Bildern kann festgestellt werden, dass das Netzwerk akkurate Ergebnisse liefert.

Abbildung 4.4: Schlechtester Fall in der 150. Epoche; blauer Punkt = Soll-, roter Punkt = Ist-Wert

Im besten Fall erzielt das Netzwerk eine Abweichung von $5.1624589e-05$, was in der 150. Epoche als sehr positiv gewertet werden kann. Das beste Ergebnis der 150. Epoche ist in der Abbildung 4.5 ersichtlich. An diesem Ergebnis lässt sich erkennen, dass das Netzwerk mehr oder weniger keine Probleme mit diversen Hautfarben und Kontrasten hat. Desweiteren werden Brillen ignoriert und auch leichte Neigungen nach links und rechts stellen keine Schwierigkeit dar.
Im Langzeittest bis zu der 800. Epoche lässt sich feststellen, dass das Netzwerk noch genauer wird, wobei hier der zeitliche Aufwand für das Trainieren

Abbildung 4.5: Bester Fall in der 150. Epoche; blauer Punkt = Soll-, roter Punkt = Ist-Wert

sehr viel Zeit beansprucht und zusätzlich das Problem des Overfittung akut werden könnte. In der Tabelle 4.1 sind zwei Systeme aufgeführt auf welchen das Netzwerk trainiert wurde, wo der Berechnungsvorteil der GPU deutlich ersichtlich ist.

System	1 Epoche	100 Epochen
Amazon EC2 Instanz, c3.8xlarge Intel Xeon E5-2680 v2, 32 Kerne 60 GiB Ram keine GPU	466	46609
Dell Optiplex 790 Intel Core 2 Duo CPU E8400, 2 Kerne, 8 GiB Ram Nvidia TitanX, 12 GiB	87	8700

Tabelle 4.1: Aufschlüsselung der ungefähr benötigten Berechnungszeiten für eine und 100 Epochen in Sekunden

Zur genaueren Feststellung der Funktionalität des Netzwerkes befindet sich in der Abbildung 4.6 ein Test mit einem noch komplett unbekannten Gesicht. Dieses wurde für diesen Test manuell aufgenommen und konvertiert sowie

im Weiteren in das Netzwerk als Testdatensatz eingefügt. Wie zu erkennen ist, befindet sich das Netzwerk in der 150. Epoche bereits in einem Zustand, welcher als verwendbar einstuft werden kann.

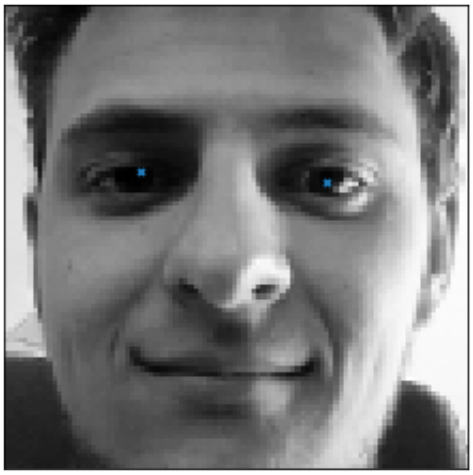


Abbildung 4.6: Testaufnahme mit Ergebnis des Netzwerkes

4.8 Graphenvisualisierung

Der Graphen wurde in diesem Beispiel in ein Event-File serialisiert, dabei wurde aus Performanzgründen die Speicherung von Veränderungen in den Ebenen sowie anderer Skalar-Werten weggelassen. In der Abbildung 4.7 ist der aktuelle Graph des Beispiels ersichtlich. Dieser beinhaltet die 3 Convolutional-Ebenen, die Transformation in eine flache Struktur sowie die Vollvernetzten-Ebenen und den Optimizer sowie die Verlustberechnung. Im Gesamten lässt sich ein Graph erkennen, in welchem der Datenfluss am Ende beginnt und nach oben läuft. Am Rande der Abbildung befinden sich noch zwei zusätzliche Objekte, welche Verbindungen zu allen Objekten haben. Diese werden automatisch durch TensorFlow hinzugefügt und werden für die Ausführung benötigt.

4.9 Verbesserungen

Dieses Netzwerk stellt lediglich einen möglichen Lösungsansatz dar, mit welchem die Problemstellung verstanden werden soll. Im Grund existieren mehrere Möglichkeiten, um das Ergebnis des Netzwerks zu verbessern. Diese Möglichkeiten führen aber dazu, dass mehr Ressourcen benötigt werden.

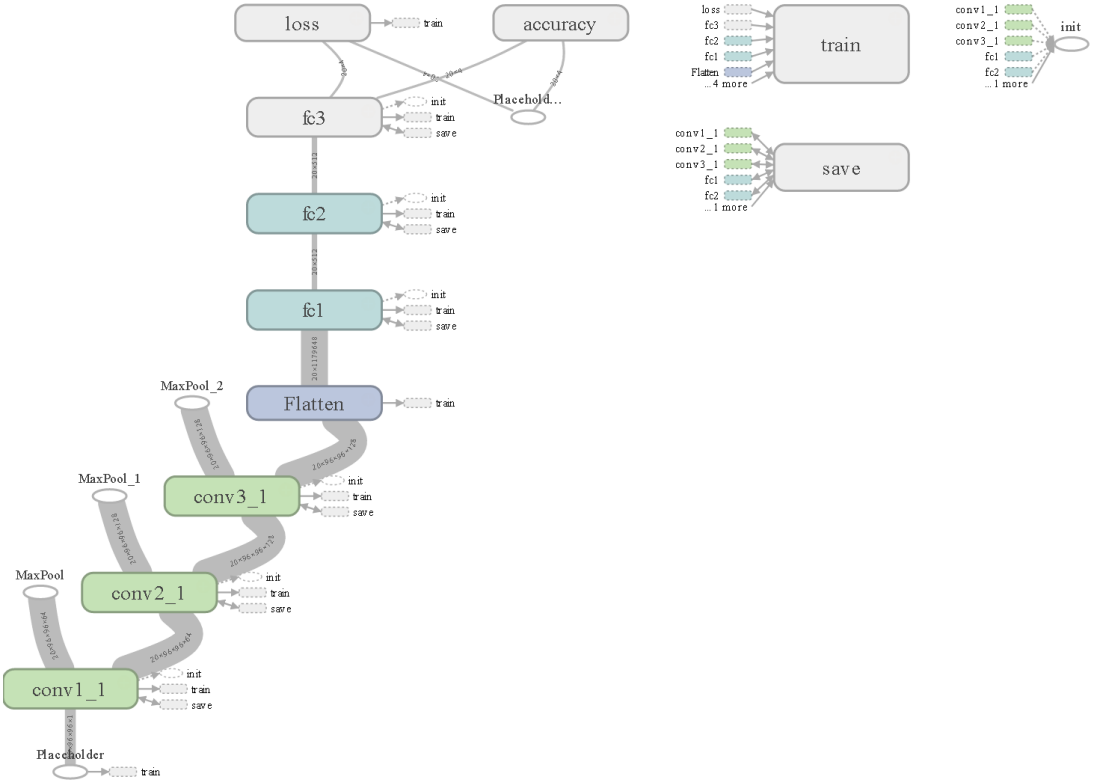


Abbildung 4.7: Aktueller Graphen des Beispiels

- Verbreitern der Vollvernetzten-Ebenen, damit mehr Muster gespeichert werden können
- Convolutional-Ebenen doppelt ausführen
- Grundanzahl an Convolutional-Ebenen erhöhen
- Anzahl der Datensätze durch Spiegeln verdoppeln

Kapitel 5

Zusammenfassung & Ausblick

5.1 Zusammenfassung

Maschinelles Lernen bietet sich in sehr vielen Fällen an, eingesetzt zu werden. Es steht aber fest, dass neuronale Netzwerke nichts Außergewöhnliches erschaffen oder gar von sich aus etwas Unvorhersehbares produzieren. Im Kern kann jeder Zustand eines Netzwerkes festgestellt werden und somit auch nachvollzogen werden, beziehungsweise mathematisch nachgerechnet werden. Grundsätzlich kann jedes Problem, welches sich in irgendeiner Art und Weise als Funktion beschreiben lässt, in einem neuronalen Netzwerk abgebildet werden kann.

Das Gebiet des maschinellen Lernens bietet einen nicht abschätzbare Umfang an Möglichkeiten. Trotzdem kann es auf einfache Grundregeln der Mathematik und Informatik herabgebrochen werden und somit auch erlernt werden. Gesamt wird es aber praktisch nie möglich sein, das gesamte Gebiet komplett zu verstehen und zu kennen. Es wird eine Möglichkeit geben, sich weiterzubilden und das Thema zu vertiefen. Sollte trotzdem der Punkt erreicht werden, an dem nichts Neues mehr gelernt werden kann, dann sollte diese Möglichkeit dazu führen, die Forschung voranzutreiben und so für eine Weiterentwicklung zu sorgen.

Ein Nachteil im Bereich des Machine Intelligence ist, dass sehr viel Zeit in das Entwickeln, Trainieren und Testen gesteckt werden muss. Aus diesem Grund entwickelte Google einen eigenen Prozessor, welcher nur für solche Berechnungen ausgelegt worden ist. In diesem Fall ist dies eine Tensor Processing Unit (TPU)[1], welche auch im AlphaGO Projekt zum Beispiel zum

[1] TPU: https://cloudplatform.googleblog.com

Einsatz kommt. So wurde der Großteil der Berechnungen für das Beispiel auf einer TitanX von Nvidia durchführt, welche für dieser Arbeit zur Verfügung gestellt worden ist. Trotzdem ist die Zeit, in der eine algorithmische Lösung entwickelt wird, meist bei weitem größer, was auch erklärt, warum in diesem Gebiet seit einiger Zeit sehr viel Forschung betrieben wird.

5.2 Ausblick

Machine Intelligence und im Speziellen TensorFlow sind Techniken und Tools, welche sehr weitläufig eingesetzt werden können. Im Detail kann TensorFlow oft zu Problemen führen, da diese Bibliothek praktisch keine Einschränkungen besitzt. Da dies aber auf einem zu geringen, beziehungsweise technisch hohen Level agiert, wo sich der Benutzer sehr gut mit der Materie auskennen muss, existieren zu diesem Zweck Abstraktionen, wie zum Beispiel Keras[2]. Die Möglichkeit von TensorFlow nicht nur CPU's und GPU's in einer Recheneinheit zu verwenden, sondern die Entwicklung auch auf mehrere Recheneinheiten zu verteilen und dies mit Unterstützung aus der Bibliothek, macht es zu einem sehr vielfältigen und einsatzfähigen Tool. Zusätzlich besteht die Möglichkeit, ein System direkt in Produktion zu nehmen und dies mit wenig Aufwand, dies stellt einen weiteren Vorteil dar.

Die Idee, etwas nicht explizit zu programmieren, sondern das System die Muster oder die Lösung selber finden zu lassen, wird in Zukunft sehr wahrscheinlich noch sehr viel öfter zu sehen sein. In diesem Fall wird es ein Austauschformat geben müssen, in welchem solche Systeme ausgetauscht werden können, wie heutzutage Daten mit Protokollen ausgetauscht werden und Logik mit Mathematik und Programmiersprachen abgebildet wird. Eine Mischform existiert hier bereits von Microsoft und der Universität Cambridge [2], in dem ein Machine Intelligence System eine Problemstellung entgegennimmt und mit Hilfe von bestehenden Programmcode automatisch ein Programm entwickelt, das die gegebenen Problemstellung löst. In diesem Fall werden Codeteile zusammen kopiert und so zu einem Programm ausgebaut.

Ein Gebiet, das zurzeit noch sehr stark erforscht wird, ist das unüberwachte Lernen. Es stellt eine Möglichkeit dar, das menschliche Hirn und auch die Natur noch besser zu verstehen, birgt aber selbst sehr viel Unbekanntes. So arbeiten Forscher auf der gesamten Welt daran, diese Technik zu erklären und zu verstehen.

[2] Keras: https://keras.io/

Quellenverzeichnis

Literatur

[1] Martin Abadi u. a. „Tensorflow: Large-scale machine learning on heterogeneous distributed systems“. *Google Research Whitepaper, http://research.google.com/pubs/archive/45166.pdf* (2015) (siehe S. 18–20).

[2] Matej Balog u. a. „DeepCoder: Learning to Write Programs“. In: *Proceedings of ICLR'17*. März 2017. URL: https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/ (siehe S. 52).

[3] Christopher M Bishop. „Pattern recognition“. *Machine Learning* 128 (2006), S. 1–58 (siehe S. 9, 20).

[4] Howard B Demuth u. a. *Neural network design*. Martin Hagan, 2014 (siehe S. 11).

[5] William Feller. *An introduction to probability theory and its applications: volume I*. Bd. 3. John Wiley & Sons New York, 1968 (siehe S. 28).

[6] Jeff Heaton. *Artificial Intelligence for Humans. Volume 3: Deep Learning and Neural Networks*. 2015 (siehe S. 4, 9, 10, 14, 16).

[7] Robert Hecht-Nielsen u. a. „Theory of the backpropagation neural network.“ *Neural Networks* 1.Supplement-1 (1988), S. 445–448 (siehe S. 7).

[8] G. E. Hinton. „Boltzmann machine“. *Scholarpedia* 2.5 (2007). revision #91075, S. 1668 (siehe S. 12).

[9] Diederik P. Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980 (siehe S. 41).

[10] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems*. 2012, S. 1097–1105 (siehe S. 15).

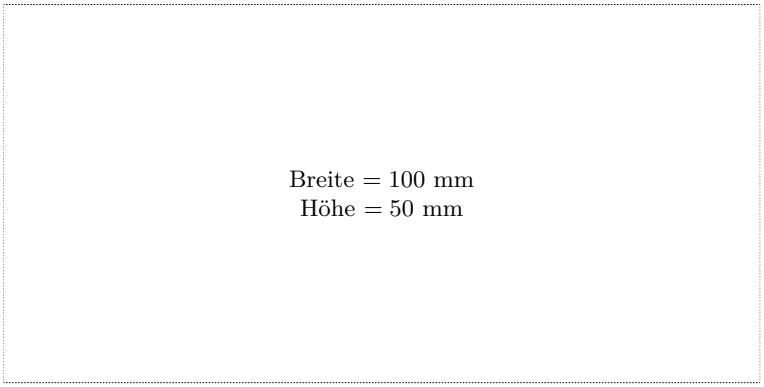
[11] Aristomenis S Lampropoulos und George A Tsihrintzis. *Machine Learning Paradigms*. Springer, 2015 (siehe S. 4).

[12] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013 (siehe S. 4).

[13] *TensorFlow*. URL: http://www.tensorflow.org (siehe S. 20).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —