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**

Vorwort **vi**

Kurzfassung **vii**

Abstract **viii**

1 Einleitung **1**
1.1 Hintergrund 1
1.2 Motivation 1
1.3 Zielsetzung 1

2 Begriffe im Maschinellen Lernen **2**
2.1 Data Science 2
2.2 Machine Intelligence 3
2.3 Machine Learning 3
2.4 Neuronale Netzwerke 3
2.5 Neuron 4
2.6 Ebenen/Layer 5
2.7 Informationen Merken und wieder Erkennung 6
2.8 Konvergieren im Maschinellen Lernen 6
2.9 Backpropagation 6
2.10 Allgemeine Probleme 7
2.11 Trainieren 8
2.12 Domänenklassen 8
2.12.1 Clustering 8
2.12.2 Regression 9
2.12.3 Classification 9
2.12.4 Predict 9
2.12.5 Robotics 9
2.12.6 Computer Vision 10
2.12.7 Optimierung 10
2.13 Neuronale Netzwerktypen 10

2.13.1 FeedForward 10
2.13.2 Self-Organizing Map 11
2.13.3 Hopfield Neuronal Network 11
2.13.4 Boltzmann Machine 11
2.13.5 Deep FeedForward 12
2.13.6 NEAT 12
2.13.7 Convolutional neural network 12
2.13.8 Recurrent Network 13
2.14 Domänen und Typen Matrix 13
2.15 Optimierung 14

3 TensorFlow **16**
3.0.1 Graphs / Dataflowgraph 17
3.0.2 Operation 18
3.0.3 Sessions 18
3.0.4 Tensor 18
3.1 Bibliotheksinhalt 18
3.1.1 Datentypen 18
3.1.2 Operationen 19
3.1.3 Probleme 22
3.1.4 TensorBoard 22

4 Facial Keypoints Detection **23**
4.1 Ausgangssituation 23
4.2 Vorbereitung 23
4.2.1 Daten vorbereiten und normalisieren 23
4.2.2 Evaluation- und Errorfunktion 23
4.3 Neuronale Ebenen vorbereiten 23
4.4 Neuronale Ebenen verknüpfen 23
4.5 Trainieren 23
4.6 Validierungsresultate 23

Quellenverzeichnis **24**
Literatur 24

Vorwort

Kurzfassung

Abstract

Kapitel 1

Einleitung

Hintergrund - Motivation - Zielsetzung
Warum - wieso - wesshalb?

1.1 Hintergrund

1.2 Motivation

1.3 Zielsetzung

Kapitel 2

Begriffe im Maschinellen Lernen

Diese Erklärung der Begriffe und Elemente verfolgt zwei Ziele. Zum einen stellt dies 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 dieser Begriffe erläutert, welche noch häufig zum Einsatz kommen (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, mit 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 die 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 größer Gruppe an 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 in dieser Form noch nicht definiert worden ist. Einige namhafte Unternehmen wie Google Inc. und Microsoft Corporation bieten jeweils unterschiedliche Definitionen oder Beschreibungen. Die Definitionen dieser Firmen weicht nur unwesentlich von einander ab. Dieser wird als Überbegriff über das gesamte Gebiet mit Machine Learning, Künstlicher Intelligenz, Konversationsintelligenz und alle Themen 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 umfasst 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 welchen man einen rein mathematischen Ansatz anstrebt. [4]

2.4 Neuronale Netzwerke

Neuronale Netzwerke sind im Jahr 2016 auch bekannt unter dem Begriff 'Deep Learning'.

Die Theorie und die ersten Grundlagen wurden im Jahre 1943 von Warrn McCulloch und Walter Pitts geschaffen, die ein Modell entwickelten, jedoch nicht die technischen Möglichkeiten hatten dieses umzusetzen. Dieses führte zur 'Threshold Logik'. Durch den Grundstein des 'Backpropagation'-Algorithmus im Jahre 1975 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.Die Verbindungen sind nicht fest vorgegeben, sondern können auch zum Beispiel Schleifen bilden.
[3]

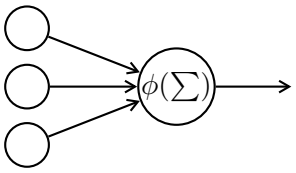


Abbildung 2.1: Neuron mit Eingang, Kernfunktion, Aktivierungsfunktion

2.5 Neuron

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

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.

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 und führt dazu, dass ein Ergebnis noch stärker ausgeprägt weitergegeben wird oder minimiert wird. Diese Aktivierungsfunktion ist meist die Sigmoid-Funktion oder eine lineare Funktion.

TODO FORMAT Diagramm

Die einfachste Repräsentation eines Neurons lässt sich mathematisch folgendermaßen darstellen. Im Kern wird eine Summenberechnung durchgeführt. Dabei werden die Eingangswerte und deren Gewichtung miteinander multipliziert, sowie diese Ergebnisse aufsummiert. Der griechische Buchstabe ϕ (phi) steht für die Aktivierungsfunktion des Neurons und stellt damit die Ausgabe des Neurons dar.

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

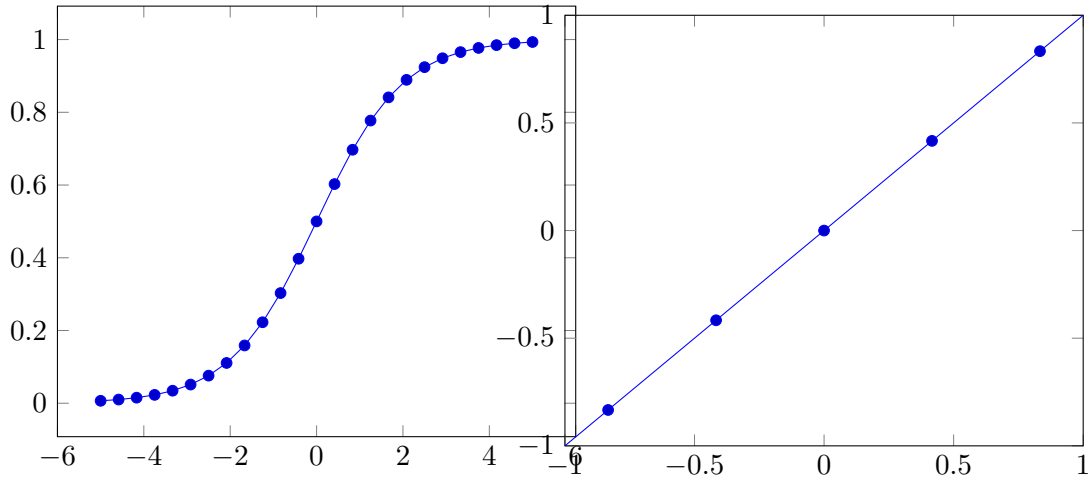


Abbildung 2.2: Basisaktivierungsfunktionen

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.

2.6 Ebenen/Layer

Ebenen sind Zusammenschlüsse von Neuronen, welche sich auf der selben 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 stellen 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 außer weiter zu geben anstatt an das darauffolgende Netzwerk. 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

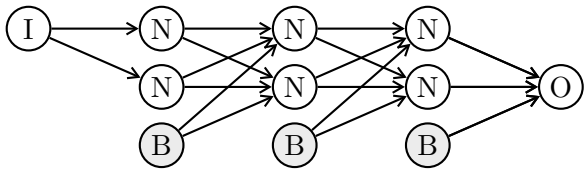


Abbildung 2.3: Einfaches Neuronales FeedFordward Netzwerk

und einer davon von dem Inputlayer eingenommen wird.

2.7 Informationen Merken und wieder Erkennung

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 einmal vorhanden war, dann sollte dieser ähnlich behandelt werden. Dieser kann möglicherweise zu derselben Kategorie gehören, wie der zuvor schon bekannte gemachte und gelernte Dateneingang.

2.8 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. Eine 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-Sqare-Error'-Methode.

2.9 Backpropagation

Bis zum Jahre 1986 gab es kein automatisierte Möglichkeit, die Gewichtungen in einem Netzwerk anzupassen. In diesem Jahre entwickelten Rumelhart, Hinton & Williams eine mögliche Lösung, welche sehr ähnlich zu anderen Ansätzen von früher war (Werbos, 1974; Parker, 1985; Cun, 1985). Die zentrale Idee in ihrer Lösung liegt darin, die Abweichung des produzierten Ergebnisses zum wirklichen erwarteten Ergebnis zu bestimmen. Aufgrund dieses Fehlers lassen sich im Anschluss die Gewichtungen im Netzwerk von dem Ende zum Anfang nachjustieren. Diese Technik ermöglichte damit auch

die Möglichkeit tiefe Netzwerke zu konstruieren und auch zu Trainieren.

Lernrate skaliert die Steigung des Lernprozesses, mit der Auswirkung ob schneller oder langsamer Gelernt wird. Eine Lernrate unter 0 würde die Lerngeschwindigkeit stark verlangsamen. Ein Wert über 1 würde eine hohen Lerngeschwindigkeit zur Folge haben. Eine zu hohe Rate würde nicht zum Konvergieren (2.8) führen, sondern zum springen.

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

2.10 Allgemeine Probleme

Nach dem aktuellen Stand der Dinge können Neuronale Netzwerke nicht jede Frage dieser Welt beantworten. Das Entwickeln eines neuen Netzwerks ist ein sehr schwierige und eine lang andauernde Aufgabe. Dabei können Fehler aufdrehten, welche natürlicher Natur sein können aber auch durch den 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 Machine Learning betrifft, sondern auch Menschen und andere Lebewesen. Zum Beispiel ein Student lernt auf 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 werden.

Daten die zum Trainieren von Netzwerken verwendet werde stellen selbst Probleme dar. Eine zu geringe Menge an Daten stellt den Entwickler vor das Problem, dass er für diese Daten ein akkurates Netzwerk entwickeln kann. Dieses Netzwerk würde aber möglicherweise nicht die gewünschten Resultate liefern, da die zur Verfügung gestellten Daten nur einen kleinen Teil des gesamten Repräsentieren. Zusätzlich kann es sein, dass die zur Verfügung gestellten Daten selbst nicht vollständig sind und somit wieder nur ein Subset verwendet werden kann.

Datensätze können ähnlich sein zu anderen, sind aber trotzdem wieder *TODO: eigen | juniek*. Dies bedeutet, dass ein Netzwerk welches für eine Problemstellung entwickelt und trainiert wurde, nicht eins zu eins übernommen werden kann. Für diese Daten muss wieder ein Netzwerk entwickelt werden.

2.11 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 welchem das Netzwerk kommen sollte. Der bekannte Wert wird meistens als Lable bezeichnet. In diese Situation liefert das Netzwerk ein Ergebnis welches mit dem erwarteten Wert verglichen werden kann. Dieser Unterschied wird zum feststellen des Fehlers verwendet, welche besagt wie inkorrekt das Ergebnis ist. Des weiteren wird dieser Fehlerwert für die Backpropagation (2.9) 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 (2.12.1) verwendet. Dabei sollen nicht bekannte Gruppen von zusammengehörende Beispiele identifiziert werden. Self-Organizing Maps (2.13.2) entdecken zusammen gehörende Muster und geben dies in einer Grafik zum Beispiel weiter zur weiteren Interpretation.

2.12 Domänenklassen

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

In dieser Arbeit werde sieben Hauptdomänen erklärt und beschrieben, welche von Heaton [**1**]AI3 definiert wurden. Im speziellen wird auf die Neuronalen Netzwerke eingegangen, welche öfter zum Einsatz kommen und eingesetzt werden.

2.12.1 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.

2.12.2 Regression

Regression beschreibt den Fall in welchem Daten generiert werden und das Kontinuierlich. Ein Anwendungsfall ist das Finden einer Zugrundeliegenden Funktion, wo nur Resultate dieser Funktion vorliegen. So mit werden aus Daten weitere Daten erzeugt. So gibt es Abläufe in der Natur welche Approximiert werden um sie für weitere Systeme möglicherweise zu verwenden. [2]

2.12.3 Classification

Das Classification Problem ist in gewisser Weise ähnlich zu Regression Problemen. 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. [3]

Ergebnisse einer Classification sagt aus, zu wieviel Prozent etwas auf den gegebenen Input zutrifft. Das gesamt Ergebnis ergibt immer 100 Prozent. Das Ergebnis bei einer Regression wird dabei nicht in Prozent angegeben sondern stellt einen konkreten Wert dar.

2.12.4 Predict

Predict Problemstellungen kommen im Kontext von Business beziehungsweise von E-Business zur Anwendung vor. Hier muss anhand von meist zeitgesteuerten Ereignisse 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, diesen zu Folgen. Im Falle der Börse sind zeitlich Kurse aus der Vergangenheit verfügbar. Diese können als Trainingsdaten für ein Netzwerk verwendet, um den nächsten Tag möglicherweise vorherzusagen. Es kann somit als eine Spezialisierung von Regression und Classification angesehen werden, da Daten generiert werde diese aber mit einer Wahrscheinlichkeit.

2.12.5 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 Dreidimensionales in einer höhere Dimension berechnet werden muss. Aus diesen Daten muss zur selben Zeit gelernt werden aber auch Aktionen eingeleitet werden, wie das steuern von Motoren, um zum Beispiel nicht Umzufallen.

2.12.6 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 ein geordnet nach diversen Kategorien wie zum Beispiel Gesichtsgefühlszustände. Das Dienste werden auch kommerziell eingesetzt und auch angeboten. In autonom gesteuerten Fahrzeugen findet diese Technologie auch bereits Verwendung um Objekte zu erkennen und zu verstehen. So muss zum Beispiel ein Verkehrszeichen von einem Passanten unterschieden werden können.

2.12.7 Optimierung

Optimierung bezieht sich auf eines der Grundprobleme der Informationstechnologie. So werden immer bessere schnellere Algorithmen entwickelt, welcher konkrete Probleme noch effizienter lösen können. Durch das Thema BigData entstand ein Performanz Problem, sodass selbst sehr effiziente Algorithmen einige Problemstellungen nicht mehr in konstanter oder adäquater Zeit lösen können. Das 'Salesman' Problem gehört zu diesen Problemen. Durch Optimierung wird in konstanter Zeit eine Lösung ermittelt, welche nicht die beste Lösung repräsentiert. Diese Lösung liegt aber im Rahmen von einer bestimmten definierten Toleranz und kann als Lösung verwendet werden. [3]

2.13 Neuronale Netzwerktypen

In den letzten Jahren heraus haben sich diverse gut funktionierende Neuronale Netzwerktypen gebildet, beziehungsweise sind entwickelt und erforscht worden. Diese Netzwerktypen definieren Richtlinien oder Ansätze zu möglichen Netzwerken, welche aber nicht komplett übernommen werden müssen, sonder einen kreativen Spielraum ermöglichen.

2.13.1 FeedForward

FeedForward Netzwerke (FFN) waren bis for 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 (Neuron 2.5), sowie dem ausbauen dieses zu Ebenen mit mehreren Neuronen (Layer 2.6). So ein Netzwerk besitzt einen Inputlayer, einen Hiddenlayer sowie einen Outputlayer. Sobald das Netzwerk mehrere Hiddenlayer aufweist wird es als Deep FeedForward Netzwerk (2.13.5) bezeichnet. Das FFN weist dabei eine Charakteristik auf, indem dass

der Datenfluss eindeutig definiert ist. Der Datenfluss beginnt bei dem Inputlayer und endet bei dem Outputlayer, ohne dass ein Datenrückfluss zum Beispiel von dem Hiddenlayer in den vorhergehenden Hiddelayer vorhanden ist. Dies würde eine Rekursion oder einem Kurzzeitgedächtnis entsprechen. Die einzelnen Ebenen müssen dabei aber nicht voll verbunden sein, sondern die Vernetzung kann selbst bestimmt werden.

2.13.2 Self-Organizing Map

Self-Organizing Map (SOM) findet vor allem im Bereich der Classification (2.12.3) Verwendung und wurden 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 es zu den Systemen, welche unsupervised trainiert werden. Außerdem besitzen sie die Möglichkeit, sich auch nach der Trainingsphase auf sich ändernde Eingangsdaten anzupassen. Kohonen entwarf die SOM mit zwei Ebenen, ein Inputlayer und ein Outputlayer ohne Hiddenlayer. Der Inputlayer propagiert Muster an den Outputlayer, wo der Dateneingang gewichtet wird. In dem 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 (siehe 2.5) aus und es kommen ausschließlich Lineare Aktivierungsfunktionen zur Verwendung.

2.13.3 Hopfield Neuronal Network

Ein Hopfield Neuronal Network (HNN) 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. Die Neuronen selbst nehmen dabei den Zustand 1 für Wahr und -1 für Falsch an. Das Problem bei diesem Type ist, dass jedes Neuron auf dem Status der 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.

2.13.4 Boltzmann Machine

Im Jahre 1985 stellten Hinton & Sejnowski 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. Des 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 in dem Outputlayer verbunden. Die Verbindungen zwischen den Neuronen in dem selben Layer entfallen. Der alte uneingeschränkte Type der Boltzmann Maschinen eignet sich gut für Optimierungsprobleme sowie für Mustererkennungen.

2.13.5 Deep FeedForward

Deep FeedForward Netzwerke unterscheiden sich zu normalen FeedForward Netzwerken in dem, dass sie mehrere Hiddenlayers beinhalten anstatt nur einem.

2.13.6 NEAT

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

Compositional pattern-producing network (CPPN) ist ein Netzwerk das andere Strukturen entwickelt und basiert dabei auf der Theorie von NEAT. Dies können Bilder aber auch andere Netzwerke sein, wobei meist Bilder generiert und weiter entwickelt werden. CPPN können im Gegensatz zu NEAT Netzwerken mit verschieden Aktivierungsfunktionen verwendet werden. Ein erzeugtes finalisiertes Netzwerke resultiert aber immer in einem regulären NEAT Netzwerk.

HyperNEAT (*VL weglassen*) ist eines der bekanntesten CPPN Netzwerke welches keine Bilder produziert sondern neue Netzwerke erstellt. Mit der Fähigkeit andere Netzwerke zu kreieren, welche wiederum für ihre Aufgabe gute Ergebnisse liefern, ermöglicht es schneller Netzwerke zu kreieren und sich auf ändernde Probleme schneller anzupassen.

2.13.7 Convolutional neural network

Werden selbst nicht als komplettes eigenes Netzwerk verwendet, sondern in FeedForward Netzwerken verwendet. Im Speziellen wen es sich um Bilderkennung geht. Dabei werden entweder zwei Ebenen nicht voll vernetzt sondern nur teilweise und somit Gewichtungen eingespart. Im zweiten Fall

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

2.13.8 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 endlos Schleifen, welche behandelt werden müssen. So können Kontext Neuronen verwendet werden aber auch eine definierte Anzahl an Iterationen durchlaufen werden, wo 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 zwischen gespeichert wird. Die Informationen die in einem solchen Neuron gespeichert werden, werden bei diesem Speichervorgang nicht Gewichtet sondern erst wenn diese Information an das Netzwerk zurück gegeben werden. Diese Rekursionen werden vor allem in Fällen verwendet, wenn es sich um zeitliche Abläufe und Änderungen geht. Wie zum Beispiel mit der Temperatur für den nächsten Tag, wo man Jahre an Daten zur Verfügung hat.

Elman Network wurden im Jahre 1990 vorgestellt und verwenden Rekursionen mit Kontext Neuronen. Dabei existieren zwei Hiddenlayers mit einem Layer für normale Neuronen und eines mit Kontext Neuronen. Die Kontext Neuronen sind dabei voll verbunden mit dem Hiddenlayer 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 wurden 1993 der Öffentlichkeit präsentiert und sind zu den Elman Netzwerken sehr ähnlich. Es werden wieder Kontext Neuronen für das zwischen Speichern 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.

2.14 Domänen und Typen Matrix

Wie in der Abbildung 2.4 erkennbar ist, existiert kein Netzwerk Grundtyp, der für alle Problemdomänen geeignet ist. Dies führt zu der Schlussfolge-

	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 [3]

rung, 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 einem Typ. Meistens sind es einige mehrere Netzwerke unterschiedlicher Typen die hintereinander und parallel geschallten sind und so ein ganzen System darstellen. Dabei übernimmt jedes Teilnetzwerk eine kleine Aufgabe des gesamten und zwar 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 um so tiefer das Netzwerk ist um so aufwändiger ist es zu Trainieren.

2.15 Optimierung

Optimierungen beinflussen das Lernverhalten und das Speicherverhalten eines Netzwerkes.

Lernrate skaliert die Lerngeschwindigkeit. Wie im Punkt Backpropagation 2.9 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. Hinten (2012) beschreibt DropOuts als eine effektive Art, um Overfitting (siehe 2.10) 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 zu dem nächsten Layer. Dies hat zur Folge, dass nur ein geringerer Teil an Informationen aus dem vorhergehenden Layer in den Nächsten übergehen. Durch diesen Prozess, des künstlichen gering halten der Informationen während dem Training führt dazu, dass das Netzwerk trotz dieser Einschränkung trotzdem 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 möchte.

L1 und L2 Regularisierung sind auch Techniken die zur Verhinderung von Overfitting beitragen. Im Gegensatz zu der DropOut Strategie sind diese zwei Regulationstechniken, Teil der Backpropagation oder Teil einer Funktion. Beide Techniken arbeiten mit Strafen welche verteilt werden und so die Gewichtungen in ein Muster zwingen. Im Falle von L1 ähnelt dieses Muster einer Gaussian Glockenkurve und bei L2 einer Laplace Kurve. Durch das Bestrafen der Gewichtungen im Netzwerk werden diese Werte gering gehalten, sodass sie nicht ausarten. 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 als Resultat hat, dass ein Rauschen in den Daten möglicherweise erkannt wird und ignoriert wird.

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

Die Funktion 2.2 bildet die Berechnung der Strafe in der Backpropagation ab. Das λ definiert wie stark die Regulierung den Error-Wert des Netzwerkes beeinflussen soll. Ein Wert von 0 führt dazu, dass die Regulierung 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.

GPU - GPGPU

Batch Learning

Kapitel 3

TensorFlow

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

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

TensorFlow steht für mehrere Programmiersprachen zur Verfügung welche Offiziell unterstützt werden, wobei es noch mehr durch die Open Source Gemeinschaft unterstützte Sprachen gibt. Den Hauptbereich stellt die Python API dar, welche auch die vollständigste Implementierung darstellt. 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, um einen Graphen zu erstellen, zu trainieren und zu testen. Durch die Verwendung von Python besteht die Möglichkeit sehr schnell Änderungen am Graphen durchzuführen und nicht erst ganze Applikationsstrukturen zu übersetzten damit ein Ergebnis der Änderung ersichtlich wird. Dieser Graphen wird nach seiner Trainingsphase exportiert und beinhaltet alle Knoten sowie die dazugehörigen Gewichtungen. Die C++ API sowie die Java API und GO API zielen

```
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
```

Abbildung 3.1: TensorFlow Codefragment zur Definition eines Teils des Graphen

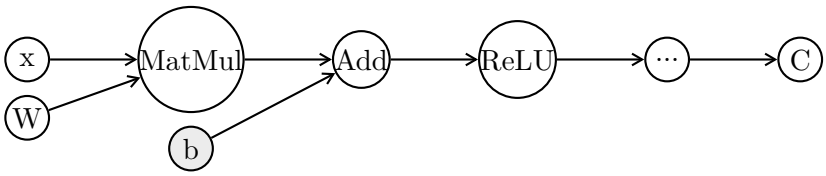


Abbildung 3.2: Der resultierenden Teilgraph aus dem Codefragment aus Abbildung 3.1 nach dem Beispiel in [1]

auf eine sehr effiziente Ausführung ab. Durch die Verwendung des trainierten Graphen kann dieser auch auf mobiles Plattformen eingesetzt werden.

3.0.1 Graphs / Dataflowgraph

Ein TensorFlow Graph kann wie in Abbildung 3.1 beschrieben werden. Dieser wurde zum Beispiel mit der Python API erstellt. Im Gesamten mit den Knoten und den Verbindungen ergibt sich ein Datenfluss, diese beinhaltet alle erforderlichen Komponenten auch für das per sistieren 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 besitzt eine Kernfunktion. Zu den 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 benützt um Abhängigkeiten zu definieren, um zum Beispiel eine Ausführung in einem anderen Knoten vor einem anderen zu definieren. So muss der Quellknoten mit der Ausführung abgeschlossen haben bevor der darauf wartende mit der Ausführung beginnt. [1]

3.0.2 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 eine Operation Polymorph für Datentypen zu ermöglichen.* Der Kernel selbst ist die Implementierung der Operation selbst. Dieser kann auf verschiedenen Geräten ausgeführt werden wie CPU oder GPU. Die Operationen und die dazugehörigen Kernel werden über einen Registrierungsmechanismus zur Verfügung gestellt. Diese Sammlung an Operationen kann auch Erweitert werden. [1]

3.0.3 Sessions

Die Session repräsentiert die Laufzeit für einen Graphen. Dieser Session wird ein Graphen übergeben, welcher erst initialisiert werden muss. Ohne die Initialisierung ist der Knoten und Verbindungen würde die weitere Ausführung mit diesem nichts produzieren, da alle Werte 0 sind. Diese stellt eine weitere Funktion zur Verfügung *Run*. Der Run-Funktion wird eine Liste Endknoten übergeben welche berechnet werden sollen und die zu dem initialisierten Graphen gehören. Die Platzhalter Tensoren werden mit Daten verknüpft und so in den Graphen gereicht. In den Meisten fällen wird ein Graphen einmal erstellt und mehrfach ausgeführt. [1]

3.0.4 Tensor

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

3.1 Bibliotheksinhalt

3.1.1 Datentypen

TensorFlow besitzt eine große Anzahl an Datentypen die verwendet werden können. Dies reicht von Grunddatentypen wie 'Boolean' und 'String' bis hinzu verschiedene Integer Datentypen. Diese stehen in verschiedene Wertebe-

reichen zur Verfügung. So gibt es Gleitkommazahlen mit unterschiedlicher Genauigkeit, wie 16-bit was für halbe Genauigkeit steht aber auch bis zu 64-bit Genauigkeit reicht, was einer doppelten Genauigkeit entspricht. 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 diese möglicherweise nie benötigen. In diesem Fall können die Wertebereiche reduziert werden, auf zum Beispiel 32-bit signierte Integer und somit die Berechnungen hochperformanter ausgeführt werden. [5]

3.1.2 Operationen

Konstanten und Zufallswerte

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

- *tf.zeros* erstellt einen Tensor mit angegebenen Dimension bestehend aus 0 und von einem Datentypen.
- *tf.zeros_like* gibt einen Tensor zurück, welcher die selbe Dimensionen wie der gegeben besitzt. Alle Werte in diesem Tensor sind aber auf 0 gesetzt. In diesem Zuge kann der Datentyp mit angepasst werden, wenn nur die Dimensionen übernommen wenden sollen.
- *tf.ones* agiert genau wie der Tensor *tf.zeros* mit dem unterschied dass alles mit 1 gefüllt ist.
- *tf.ones_like* repräsentiert das selbe wie *tf.zeros_like* nur mit 1.
- *tf.fill* wird zu der Dimension noch ein Skalar mit gegeben, für die Werte die ausgefüllt werden sollen.
- *tf.constant* liefert einen Tensor mit selbst definierbaren Werten. Diese Werte können eine Liste sein sowohl als auch eine einzelner Wert welcher überall 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 einfließen zu lassen.

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

Zufallswerte werden im Bereich von maschinellen Lernens sehr häufig benötigt. So werden meist der Startzustand mithilfe von Zufallszahlen hergestellt.

- *tf.random_normal* liefert einen Tensor mit Zufallswerten anhand einer Normalverteilung (Gaussian). Die Dimension des Ergebnistensors muss spezifiziert werden, der Meridian, Standardabweichung sowie der resultierende Datentyp können angegeben werden.
- *tf.truncated_normal* verhält sich gleich zu *tf.random_normal* mit dem unterschied, dass Werte die größer sind als 2-mal die Standardabweichung, 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 exklusive der oberen Grenze ist, wie zum Beispiel '$[0, 1)$'.
- *tf.random_shuffle* erstellt selber keine neuen Werte sondern, mischt einen Tensor anhand seiner ersten Dimension durch.
- *tf.random_crop* liefert einen zufälligen Teil eines Tensors mit der selben Anzahl an Dimensionen und aber mit der spezifizierten Größe.

Einige dieser Funktionen benötigen sogenannte Seed-Werte, welche den Startwert der Zufallszahlen zerstreuen sollen sowie die Folge selbst. Im Falle von TensorFlow beruht dies auf zwei Werten, einer wird für den Graphen spezifiziert, der zweite wird für die Operation selbst spezifiziert. Der Wert für den Graphen kann mit *tf.set_random_seed* gesetzt werden. Für weiter Informationen steht die online Dokumentation zur Verfügung. [1]

Variables

Variablen geben bei jedem Durchlauf einen Tensor ab. Dieser Wert ändert sich nicht, außer ihm wird eine neuer Wert zugewiesen.

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 *TypeError* resultiert.

Shapes und Shaping liefert die Gestalt eines Tensors, bietet aber auch die Möglichkeit diese zu ändern.

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

- *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 ergibst sich aus den konkreten Werten.
- *tf.rank* verhält sich ähnlich zu *tf.size* mit dem unterschied, dass die Anzahl der Felder Vertiefung gezählt wird.
- *reshape* wird verwendet um Tensoren in eine neue Struktur zu bringen. Dabei kann für das einebnen der Dimensionen eine Kurzschreibweise verwendet werden mit -1 als Zielausführung der Gestalt.
- *tf.squeeze* entfernt ganze Dimensionen aus dem gegebenen Tensor. Ohne Achsen Angabe werden alle Dimensionen mit der Größe 1 entfernt oder es werden die spezifizierten Dimensionen herausgenommen.
- *tf.expand_dims* gliedert wider um Dimensionen in einen Tensor ein. Im Standard an der Indexstelle 0, außer es wurde spezifiziert.

Slicing und Joining wie in diversen Programmiersprachen unterstützt auch TensorFlow das Teilen und Zusammenfügen von Daten und aber hier im Speziellen mit Tensoren. Diese Operationen reichen von einfachen Slicing Operationen über Transponieren bis hin zu dem Verketten von Tensoren, dabei kann definiert werde 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. Diese 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 Absolutwerten in einem Tensor sowie eine Exponentialfunktion.

Matrizen Funktionen werden am häufigsten benötigt, da Tensoren im Grunde Matrizen sind und somit 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.

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

- *tf.eye* erzeugt eine Identitätsmatrix, in welcher alle Werte an der Diagonale 1 sind und alle anderen 0.

Zu diesen Funktionen existieren noch weitere 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 de Graphen eingepflegt werden.

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

- *tf.reduce_sum* berechnet die Summe aller Werte in einem Tensor.
- *tf.reduce_mean* berechnet die Summe aller Werte an der Diagonale eines Tensors.
- *tf.reduce_max* reduziert einen Tensor auf die maximal Werte in der letzten Dimension und reduziert dabei den Rang um eins.

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

Die Anzahl an mathematischen Funktionen ist um einiges sehr viel Größer als die hier erwähnten. Diese hier repräsentieren lediglich die meist verwendeten Operationen. Für weiter Informationen steht die online Dokumentation zur Verfügung. [3]

Flusskontrolle

Images / FFmpeg

Input und Readers

Neural Network

Running Graphs

Training

3.1.3 Probleme

NaN Problem

3.1.4 TensorBoard

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

Kapitel 4

Facial Keypoints Detection

4.1 Ausgangssituation

4.2 Vorbereitung

4.2.1 Daten vorbereiten und normalisieren

4.2.2 Evaluation- und Errorfunktion

4.3 Neuronale Ebenen vorbereiten

4.4 Neuronale Ebenen verknüpfen

4.5 Trainieren

4.6 Validierungsresultate

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. 17, 18).

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

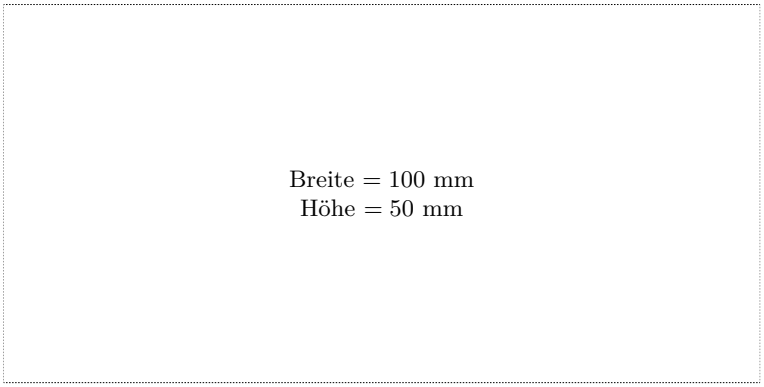
[3] Jeff Heaton. *Artificial Intelligence for Humans. Volume 3: Deep Learning and Neural Networks*. 2015 (siehe S. 3, 9, 10, 14).

[4] Aristomenis S Lampropoulos und George A Tsihrintzis. *Machine Learning Paradigms*. Springer, 2015 (siehe S. 3).

[5] *TensorFlow*. URL: http://www.tensorflow.org (siehe S. 19).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —