

Wissenschaftliches Seminar

im Wintersemester 2017/2018

Eine Einführung in TensorFlow

bearbeitet von: Bierschneider Christian 3118760
 Pöschl Maximilian 3121342
 Maiwald Benjamin 3097528
 Studiengang: Informatik
 Schwerpunkt: Software Engineering

Betreuer: Prof. Dr. Jan Dünneweber
 OTH Regensburg

Regensburg, 14. Januar 2018

Abstract

Die hier vorliegende Arbeit befasst sich mit der Einführung in die plattformunabhängige Bibliothek TensorFlow (TF), welche seit Ende 2015 als Open-Source zur Verfügung steht und für den Bereich Maschinelles Lernen vom Google Brain Team entwickelt wurde. Das Ziel dieser Arbeit ist es, dem Leser den Einstieg in TensorFlow zu erleichtern und einen Überblick über die vorhandenen Möglichkeiten dieses umfangreichen Frameworks zu geben.

Zu Beginn wird kurz die Geschichte des Maschinellen Lernens vorgestellt. Hierbei wird auf einige wichtige Meilensteine eingegangen, sowie auf Bereiche in denen der Mensch mit künstlicher Intelligenz in Kontakt kommt. Darauf folgend werden im Kapitel 2 die Grundlagen der Neuronalen Netze erläutert, da diese im Umgang mit TensorFlow am häufigsten verwendet werden. Hierbei wird als erstes ein vorwärtsgerichtetes Netzwerk mit seinem grundlegendem Aufbau aus verschiedenen Schichten, sowie die Berechnung der Gewichtsmatrix der dazwischenliegenden Verbindungen beschrieben. Danach erfolgt ein Überblick über die wichtigsten Aktivierungsfunktionen wie ReLU-, Sigmoid- und Tangenshyperbolicusfunktion, außerdem wird noch auf die Kostenfunktionen eingegangen, die als Maß dienen, wie gut ein Neuronales Netz lernt. Das Kapitel 3 beschreibt das Framework TensorFlow näher, wie der Beginn der Entwicklung und die angesprochene Zielgruppe. Ein weiterer Punkt ist die Durchführung der Berechnungen mit Hilfe der Darstellung der Graphen als Datenflüsse. Hierbei leitet sich auch der Name TensorFlow, den Google dafür gewählt hat ab. Ebenso werden auch die Hard- und Software Anforderungen in diesem Kapitel beschrieben, um TF ausführen zu können und die Berechnungen auf CPU und GPU zu verteilen. Zum Abschluss wird noch der Aufbau der Architektur erklärt, sowie die Nutzung von Keras, eine speziell für Neuronale Netze geschriebene High-Level Bibliothek mit TensorFlow. Das letzte Kapitel dient der Beurteilung des Modells, zum Einen durch Aufteilung der vorhandenen Datensätze in Trainings-, Validierungs- und Testdaten und zum Anderen durch die Visualisierung mit TensorBoard, eine in TensorFlow enthaltene Webanwendung. Hierbei wird dann auf die einzelnen Visualisierungsmöglichkeiten, die sich in Skalare, Bilder, Graphen, Histogramme, Verteilungen, Projektor, Audio und Text aufteilen eingegangen, sowie die in TF nötigen Definitionen für den Programmcode.

Abkürzungsverzeichnis

API	Application Programming Interface.
DNN	Deep Neural Network.
KI	Künstliche Intelligenz.
ML	Machine Learning, bzw. Maschinelles Lernen.
MNIST	Mixed National Institute of Standards and Technology.
MSE	Mean Squared Error.
NN	Neurones Netz, bzw. Neural Network.
PCA	Principal Component Analysis.
ReLU	Rectified Linear Unit.
t-SNE	t-Distributed Stochastic Neighbor Embedding.
TF	TensorFlow.
TPU	Tensor Processing Unit.

Inhaltsverzeichnis

Abkürzungsverzeichnis	iii
1 Eine Einführung in Maschinelles Lernen	1
1.1 Meilensteine des Maschinellen Lernens	1
2 Neuronale Netze in TensorFlow	4
2.1 Feedforward Netzwerk	4
2.2 Der Input-Tensor	5
2.3 Gewichte	6
2.4 Aktivierungsfunktionen	7
2.4.1 Rectified linear unit function	7
2.4.2 Sigmoidfunktion	8
2.4.3 Tangenshyperbolicus	9
2.4.4 Softmax	9
2.5 Kostenfunktionen	10
2.5.1 Mean Squared Error	10
2.5.2 Cross Entropy	10
2.6 Der Lernprozess	11
3 Das Framework TensorFlow	12
3.1 Die Entwicklung von TensorFlow	12
3.2 Angesprochene Zielgruppe	13
3.3 Datenflussgraphen in TensorFlow	13
3.4 Hard- und Software Anforderungen	15
3.4.1 Betriebssysteme	15
3.4.2 Hardware Unterstützung	15
3.4.3 Software Anforderungen	16
3.5 Architektur von TensorFlow	16
3.6 Nutzung von TensorFlow mit Keras	18

4	Der Allgemeine Workflow in TensorFlow	19
4.1	Vorgehensweise beim Trainingsprozess	19
4.2	Die Visualisierung mit TensorBoard	20
4.3	Die einzelnen Visualisierungsmöglichkeiten im Detail	21
4.3.1	Skalare	21
4.3.2	Bilder	22
4.3.3	Graphen	23
4.3.4	Histogramme	24
4.3.5	Verteilungen	24
4.3.6	Projektor	25
4.3.7	Audio und Text	28
4.4	Die Graphenelemente im Datenfluss	29
	Abbildungsverzeichnis	30
	Anhang	31
	Literaturverzeichnis	33

1

Kapitel 1

Eine Einführung in Maschinelles Lernen

Da Vorwissen in den Bereichen Künstliche Intelligenz (KI) und Machine Learning (ML) selbst bei Studierenden der Informatik nicht generell vorausgesetzt werden kann, geben dieses und das nächste Kapitel eine kurze Einführung in das Thema. Es wird über die Grundlagen im Bereich des Maschinellen Lernens mit dem Schwerpunkt auf Neuronale Netze (NN) informiert, die zum Verständnis der Arbeit benötigt werden. Sollte sich der Leser bereits mit dem Thema auseinander gesetzt haben und die Bedeutung von Begriffen wie „Label“, „Schichten (Layers)“ und „Gewichte“ schon bekannt sein, können dieses und das nächste Kapitel auch übersprungen werden.

1.1 Meilensteine des Maschinellen Lernens

In den letzten Jahren (seit ca. 2010) hat sich das Thema KI zu einem regelrechten Hype-Thema entwickelt. Denn gerade durch Anwendungen in den Bereichen Bildverarbeitung, Spracherkennung, sogenannter „Recommender Systems“ oder auch des automatisierten Fahrens, gab es enorme Fortschritte. Diese machten KI für die breite Masse salonfähig und ermöglichten die Entwicklung von Produkten wie Siri, den Skype Translator, Filmempfehlungen auf Netflix oder die Fahrerassistenzsysteme im Tesla Model S, die heute von Millionen von Menschen täglich genutzt werden.

Allen voraus liegt das Hauptaugenmerk vieler Informatiker und Forscher gerade auf den sogenannten „Künstlichen Neuronale Netzen“ (Artificial Neural Networks). Manche dieser Neuronale Netze erlangten durch das Gewinnen von großen Wettbewerben weltweite Aufmerksamkeit. Ein Beispiel hierfür ist „AlphaGo“, das von Google entwickelt wurde und 2016 den damaligen Vize-Weltmeister Lee Sedol in vier von fünf Runden im Brettspiel „Go“ besiegte [1].

Dabei sind die meisten Grundlagen auf diesem Gebiet bereits Jahrzehnte alt. Schon in den 1940er Jahren und somit unmittelbar nach der Erfindung des modernen Computers, begannen die ersten Forscher damit, ein Modell für Künstliche Neuronale Netze zu entwickeln

und behaupteten sogar bereits, dass entsprechend definierte Netze auch lernfähig seien [2]. In demselben Artikel, in dem Alan Turing 1950 die Idee des weltbekannten Turing-Tests — in [3] „The Imitation Game“ genannt — veröffentlichte, schrieb er außerdem zum ersten Mal über „lernende Maschinen“ und philosophierte darüber, wie man einer Maschine beibringen könne, im Imitation Game zu bestehen. In dieser Niederschrift formulierte Turing auch die Grundideen zum heute als „Reinforcement Learning“ bezeichneten Lernen durch Bestrafung und Belohnung.

1956 war schließlich offiziell das Geburtsjahr der Künstlichen Intelligenz. Am Dartmouth College (Hanover, New Hampshire) veranstalteten McCarthy, Minsky, Shannon und Rochester (allesamt Größen in der Entwicklung der KI) ein „Summer Research Project on Artificial Intelligence“ zusammen mit weiteren Forschern aus ganz Amerika. Hier wurde der Begriff „Artificial Intelligence“ zum ersten Mal überhaupt benutzt. Ziel des Workshops war es, in zwei Monaten einen signifikanten Fortschritt bei der Entwicklung einer intelligenten Maschine zu erreichen. Dieses Ziel konnte zwar nicht erfüllt werden, jedoch sorgte das Treffen dafür, dass sich die wichtigsten Personen kennenlernten, die in den darauffolgenden 20 Jahren die größten Neuerungen auf diesem Gebiet entwickelten [2].

Wie auch heute gab es schon einmal in den 1980er Jahren einen großen Boom in der KI-Industrie. Die Investitionen stiegen von einigen Millionen Dollar im Jahr 1980 auf mehrere Milliarden Dollar im Jahr 1988. Viele der KI-Firmen konnten ihre Versprechen jedoch nicht halten, weshalb der Markt in den 90er Jahren zusammenbrach. In Folge dessen ging auch die Forschung auf dem Gebiet zurück und es kam zu keinen nennenswerten Erkenntnisgewinnen in den 90er Jahren. Deshalb wird dieses Jahrzehnt auch als „KI-Winter“ bezeichnet.

Wenn alle diese Entwicklungen im Bereich der KI aber bereits so lange zurück liegen, warum hat es dann bis heute gedauert, dass es die ersten Anwendungen zum Endkunden schaffen?

Wir leben heute in einer spannenden Zeit, denn einige wichtige Faktoren, die für den Erfolg der KI wichtig sind, wurden nahezu zeitgleich verfügbar.

Zum einen stieg die Leistung von Computern seit deren Erfindung stetig an. Für die meisten Berechnungen im Bereich des Maschinellen Lernens wird eine sehr hohe Rechenleistung benötigt. Vor allem die Verwendung von GPUs zur parallelen Berechnung von allgemeinen Aufgaben beschleunigt das Trainieren von Neuronalen Netzen um ein Vielfaches. Rechenvorgänge, die noch vor zehn Jahren große Rechencluster für viele Monate beanspruchten, können heute in Stunden, maximal aber wenigen Tagen abgeschlossen werden – und das auf kompakten Rechnern, die sogar für Privatpersonen erschwinglich sind. Dies gibt den Forschern und Softwareingenieuren die Möglichkeit, schon nach kurzer

Zeit ein Feedback zu erhalten, ob die von Ihnen gewählten Ansätze richtig sind und falls nicht, Anpassungen vorzunehmen.

Zum anderen leben wir im Zeitalter von „Big Data“. Für ML ist es extrem wichtig, dass große Datenmengen zur Verfügung stehen, die für den Trainingsprozess verwendet werden können. Firmen wie Google, Facebook, Apple oder Microsoft (und natürlich vielen anderen) steht ein schier unerschöpflicher Pool an Informationen zur Verfügung. Diese maschinell generierten Daten eignen sich aufgrund ihres großen Umfangs perfekt dazu, neuronale Netze zu trainieren und das wird von diesen Firmen natürlich auch genutzt, um neue Geschäftsideen zu entwickeln und die angebotenen Dienste für ihre Kunden kontinuierlich zu verbessern.

2 Kapitel 2

Neuronale Netze in TensorFlow

Neuronale Netze sind das am häufigsten benutzte Werkzeug in TF. Im Folgenden werden die Grundlagen für Neuronale Netze vorgestellt, sowie deren Umsetzung in TF. Dabei wird mit [4]

```
1 import tensorflow as tf
```

TF in Python importiert. TF-Befehle können dann über das Kürzel `tf` aufgerufen werden. Diverse Codeauschnitte in diesem und allen nachfolgenden Kapiteln sind Implementierungen in der Programmiersprache Python.

2.1 Feedforward Netzwerk

Die häufigste in TF implementierte Version Neuronaler Netze ist das Feedforward Network oder ein vorwärtsgerichtetes Netzwerk. Der Name stammt von den Informationen, die innerhalb des Netzes immer weiter nach "vorne" wandern, von den Eingabeneuronen über die versteckten Schichten zu den Ausgabeneuronen [5]. Es gibt keine Rückkopplungen, wie es zum Beispiel beim Hopfieldnetz der Fall ist [6]. Sie werden in der Regel für Deep Learning verwendet [5]. Ein Feedforward Netz besteht aus einer Eingabeschicht, einer Ausgabeschicht und n versteckten Schichten dazwischen.

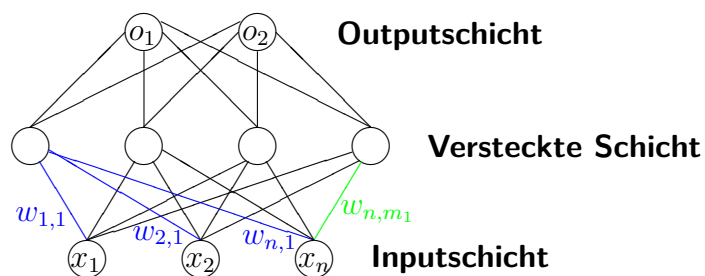


Abbildung 2.1: Feedforward Netz

Hierbei ist jedes Neuron mit jedem Neuron der nachfolgenden Schicht durch Gewichte verbunden.

Die Abbildung zeigt beispielhaft ein Feedforward Netzwerk mit 3 Input-Neuronen in der Eingabeschicht, einer versteckten Schicht mit 4 versteckten Neuronen und einer Ausgabeschicht mit 2 Ausgabe-Neuronen [7].

Ein Feedforward Netzwerk kann beliebig viele versteckte Schichten enthalten, aber nur eine Eingabe- und eine Ausgabeschicht. Die Anzahl der Neuronen in den versteckten Schichten kann frei gewählt werden. Eine möglichst sinnvolle Anzahl dieser, genauso wie eine möglichst passende Anzahl der versteckten Schichten lassen sich für das jeweilige Problem nur experimentell bestimmen [8]. Man sollte darauf achten, nicht zu wenige Neuronen zu verwenden, da sonst die Lernkapazität möglicherweise zu eingeschränkt ist. Auch zu viele Neuronen können problematisch sein, da der Zeitaufwand, jedes der vielen Neuronen zu trainieren, sehr groß wird und die Effizienz des Netzwerks darunter leidet [9]. Man kann bereits mit einer versteckten Schicht und einer ausreichend großen Anzahl von Neuronen jedes Problem simulieren, tendenziell ist es aber besser die Anzahl der Schichten zu erhöhen, anstatt die Anzahl der Neuronen pro Schicht [8].

2.2 Der Input-Tensor

Vektoren und Matrizen werden in TF als Tensoren bezeichnet. Der Input eines Neuronalen Netzes ist ein n-dimensionaler Vektor, der für jedes Input-Neuron einen Wert enthält. Die Anzahl der Input-Neuronen richtet sich nach dem untersuchten Problem. Das Einführungsbeispiel für TF, das in etwa dem "Hello World" für Programmiersprachen entspricht, behandelt ein Klassifikationsproblem [8]. In diesem Problem sollen mithilfe der Mixed National Institute of Standards and Technology (MNIST) Datenbank, die tausende von handgeschriebenen 28×28 Bilder der Zahlen von 0-9 enthält, das Neuronale Netz lernen handgeschriebene Ziffern zu unterscheiden. Für jedes dieser Bilder existiert eine 28×28 Matrix mit Grauwerten. Mit dem Befehl `reshape(-1, ...)` [8] lässt es sich in einen eindimensionalen Vektor mit $28 * 28 = 784$ Input-Neuronen verwandeln.

Input-Tensoren werden in TF mit Platzhaltern angelegt, da während des Lernprozesses der Input-Tensor für jedes zu lernende Beispiel aktualisiert wird. Um den Platzhalter anzulegen, verwendet man den Befehl [4]

```
1 input = tf.placeholder(dtype, [None, Input_Neuronen])
```

Mit `dtype` wird der Datentyp des Inputs für die einzelnen Neuronen gewählt. Die Variable `Input_Neuronen` gibt die Anzahl der Input-Neuronen an. `None` steht für die Anzahl der Trainingsdaten, die später noch dynamisch eingefügt wird. So muss man sich zunächst nicht festlegen, wie viele Trainingsdaten man benutzen will [8].

2.3 Gewichte

Man kann alle Gewichte zwischen Inputschicht und der ersten versteckten Schicht als Gewichtsmatrix

$$W^{(1)} := \begin{pmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & \dots & w_{1,m_1}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & \dots & . \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & \dots & . \\ \vdots & \vdots & \vdots & \vdots \\ w_{n,1}^{(1)} & w_{n,2}^{(1)} & \dots & w_{n,m_1}^{(1)} \end{pmatrix} \quad (2.1)$$

auffassen.

Die Zeilen der Matrix $W^{(1)}$ entsprechen allen ausgehenden Verbindungen für ein Neuron aus der Inputschicht. Die Spalten entsprechen den eingehenden Verbindungen für ein Neuron aus der versteckten Schicht.

Der TF Befehl, um die Gewichte zwischen zwei Schichten zu initialisieren, lautet [8]

```
1 init = tf.truncated_normal([n_inputs, n_neurons])
2 W = tf.Variable(init,name="weights")
```

Damit werden die Gewichte mit zufälligen Werten belegt und es wird eine Matrix bzw. Tensor der Dimension (input_neuronen×versteckte_neuronen) erzeugt.

Um den Output des Neuronalen Netzes zu berechnen, führt man mithilfe des Befehls `tf.matmul(input,W)` eine Matrixmultiplikation zwischen Input-Tensor und Gewichtsmatrix durch [8].

So erhält man einen weiteren Tensor, der für jedes versteckte Neuron einen Wert besitzt. Auf diese Werte wendet man nun eine Aktivierungsfunktion an. Das Ergebnis nennt man Aktivität [6] der versteckten Schicht.

Dieses Verfahren kann man nun für beliebig viele versteckte Schichten wiederholen. Dabei nutzt man jeweils die Aktivität der aktuellen Schicht und die Gewichtsmatrix der nächsten versteckten Schicht, führt eine Matrixmultiplikation durch und wendet eine Aktivierungsfunktion an. So wird Stück für Stück die Aktivität aller Schichten berechnet. Die Aktivität der Outputschicht ist der eigentliche Output, den das Netz produziert.

2.4 Aktivierungsfunktionen

Aktivierungsfunktionen werden verwendet, um den Wertebereich, den die Neuronen annehmen können, einzugrenzen. So liegen manche Aktivierungsfunktionen nur zwischen den Werten -1 und 1 oder bilden alle negativen Werte auf Null ab. Exemplarisch werden 3 häufig verwendete Aktivierungsfunktionen beschrieben. Während die Rectified Linear Unit (ReLU) Funktion mittlerweile als Standardaktivierungsfunktion dient, wurden vorher oft die Sigmoid- und die Tangenshyperbolicusfunktion eingesetzt, da ihre Ableitungen leicht zu berechnen sind. Deshalb eignen sie sich gut zur Berechnung des Gradienten der Fehlerfunktion.

2.4.1 Rectified linear unit function

Eine sehr häufig genutzte Variante einer Aktivierungsfunktion ist die ReLU Funktion. Für die ReLU Funktion gilt in der parametrisierten Form [5]

$$\sigma(x) = \begin{cases} 0 & \text{falls } x \leq 0 \\ ax & \text{sonst,} \end{cases}$$

wobei a frei wählbar ist und an das jeweilige Beispiel angepasst werden kann. Obwohl die ReLU Funktion eine sehr einfache, fast lineare, Funktion ist, erweist sie sich als sehr leistungsfähig. Sie dient als Standardaktivierungsfunktion, die für die meisten Feedforward Netzwerke empfohlen wird [5]. Die ReLU Funktion ist stetig, aber im Nullpunkt nicht differenzierbar [4]. Während der linksseitige Grenzwert der Ableitung 0 ist, ist der rechtsseitige Grenzwert 1.

In der Praxis spielt das kaum eine Rolle, da während des Rechnens meist Werte verwendet werden, die nur sehr nahe an der Null sind, jedoch nicht zu Null werden, und dabei den links- bzw. rechtsseitigen Wert für die Ableitung liefern [5]. In TF ist die ReLU Funktion auf verschiedene Arten implementiert, die oben beschriebene Standard ReLU Funktion allerdings nur für den Parameterwert $a=1$. Sie wird mit dem Befehl [4]

```
1 tf.nn.relu(features, name=None)
```

eingebunden. Eine andere in TF verwendete Versionen der ReLU Funktion ist [4]

```
1 tf.nn.relu6(features, name=None)
```

Für diese gilt:

$$\sigma(x) = \begin{cases} 0 & \text{falls } x \leq 0 \\ x & \text{falls, } 0 < x < 6 \\ 6 & \text{sonst} \end{cases}$$

Sie kann schneller berechnet werden und hat den Vorteil, dass weder Werte nahe der Null verschwinden, noch die Werte zu groß werden können [4].

2.4.2 Sigmoidfunktion

Eine andere Aktivierungsfunktion ist die Sigmoidfunktion. Beliebte wurde die Sigmoidfunktion in den 1980er Jahren, um die davor verwendeten Stufenfunktionen zu ersetzen. Die Sigmoidfunktion hatte den Vorteil, dass sie der Stufenfunktion ähnelte, dabei aber auf dem ganzen Wertebereich differenzierbar war. Sie wurde bis in die 2000er Jahre verwendet. Mittlerweile wird sie häufig durch die ReLU Funktion ersetzt [5].

Die in TF verwendete Sigmoidfunktion hat folgende Form: [4]

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

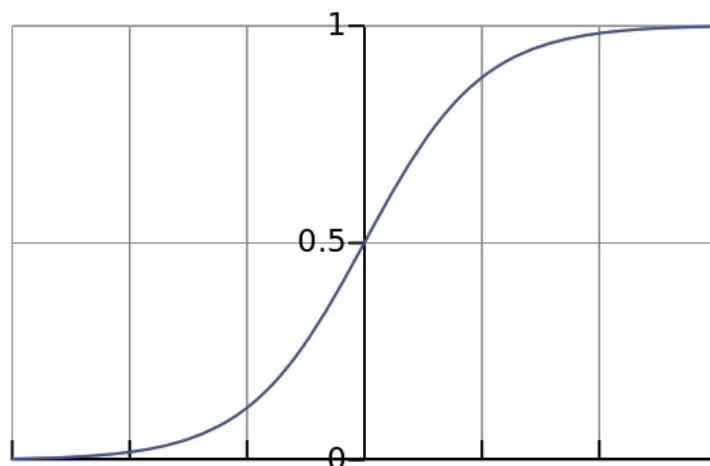


Abbildung 2.2: Sigmoidfunktion [10]

Der Wertebereich der Sigmoidfunktion liegt zwischen 0 und 1, daher findet sie häufig Anwendung in binären Klassifikationsproblemen [5]. Ein Problem der Sigmoidfunktion ist, dass ihre Ableitung geringer wird je größer bzw. kleiner die eingesetzten Werte sind. Da für das Lernen die Ableitung der Sigmoidfunktion verwendet wird, wird der Lernprozess für sehr große Werte bzw. sehr kleine Werte sehr langsam [5]. In TF ist sie mit dem Befehl `tf.sigmoid(x, name=None)` [10] definiert.

2.4.3 Tangenshyperbolicus

Die Tangenshyperbolicusfunktion ähnelt in seiner Struktur und Verwendung stark der Sigmoidfunktion. Sie ist definiert als [7]

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \quad (2.3)$$

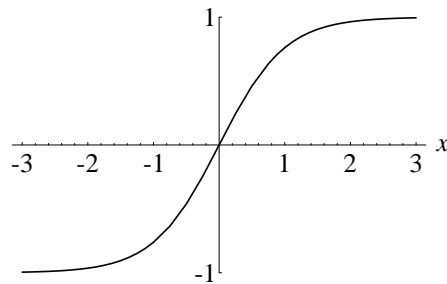


Abbildung 2.3: Graph des Tangenshyperbolicus [11]

Wie in der Abbildung ersichtlich wird, befindet sich der Wertebereich des Tangenshyperbolicus im Intervall von $[-1,1]$. Es wird mit `tf.tanh(x, name=None)` [10] aufgerufen. Der Tangenshyperbolicus eignet sich vor allem, wenn man den Wertebereich auf negative Zahlen ausdehnen möchte [4].

2.4.4 Softmax

Eine weitere nützliche Aktivierungsfunktion ist die Softmaxfunktion. Diese kommt vor allem in Klassifikationsproblemen zum Einsatz und wird dort auf die Outputschicht angewandt um eine Wahrscheinlichkeitsverteilung zu erzeugen [5].

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}. \quad (2.4)$$

Die Werte der Softmaxfunktion liegen zwischen Null und Eins. Summiert man alle Werte auf erhält man Eins, wie man es von einer Wahrscheinlichkeitsverteilung erwartet. Um den Output des Netzes zu bestimmen nimmt man nun den Wert mit der größten Wahrscheinlichkeit [8]. Die Softmaxfunktion wird meistens in Verbindung mit der Cross Entropy Kostenfunktion verwendet, die sich sehr gut für Klassifikationsprobleme eignet und im nächsten Kapitel näher erläutert wird.

2.5 Kostenfunktionen

Kosten- oder Fehlerfunktionen sind ein Maß dafür, wie gut ein NN lernt. Es stellt eine berechenbare Formel bereit, die den durch das Netz berechneten Output für ein Trainingsbeispiel mit dem zu lernenden Output vergleicht. Außerdem werden sie für das Lernen des NN benötigt, da aus den Ableitungen der jeweiligen Kostenfunktion für das zugehörige Gewicht die neuen Gewichte gebildet werden [5].

2.5.1 Mean Squared Error

Eine mögliche Kostenfunktion ist der Mean Squared Error (MSE) [11].

$$MSE = \frac{1}{N} \sum_{i=1}^N (\hat{o}_i - o_i)^2. \quad (2.5)$$

\hat{o} steht für den gewünschten Output und o für den vom Netz für den zugehörigen Input-Vektor generierten Output. Der MSE wird mit [4]

```
1 Kosten = tf.losses.mean_squared_error(labels,predictions)
```

eingebunden und summiert die Quadrate der Abweichungen auf, d.h. je geringer der MSE wird, desto genauer hat das Netz die Trainingsdaten gelernt.

2.5.2 Cross Entropy

Eine andere Möglichkeit ist die sogenannte Cross Entropy Funktion [12].

$$CE = -\frac{1}{N} \sum_{i=1}^N \hat{o}_i \ln o_i + (1 - \hat{o}_i) \ln(1 - o_i) \quad (2.6)$$

Der Vorteil der Cross Entropy Funktion besteht darin, dass sie umso schneller lernt je größer der anfängliche Fehler ist. Beginnt man also bei sehr ungünstig gewählten zufälligen Gewichten mit dem Lernprozess, wird Cross Entropy schneller bessere Ergebnisse liefern als der MSE [12]. Welche man letztendlich verwendet, hängt vom zugrundeliegenden Problem ab. Während der MSE bei Regressionen beliebt ist, findet Cross Entropy bei Klassifikationsproblemen häufig Anwendung. Cross Entropy wird in TF mit dem Befehl

```
1 Kosten = tf.nn.softmax_cross_entropy_with_logits()
```

verwendet [4]. Wie man sieht wird in dieser Version des Befehls bereits eine Softmax-funktion automatisch mit angewandt.

2.6 Der Lernprozess

Damit das Netz lernt müssen nach und nach die Gewichte angepasst werden. Zu diesem Zweck berechnet man die Ableitung der Kostenfunktion und benutzt sie, um die Gewichte abzuändern. Dieses Verfahren wird „Backpropagation of Error“ mittels „Gradient Descent“ - dem Gradientenabstiegsverfahren - genannt.

Das Update der Gewichte funktioniert nach folgender Regel: [11]

$$w_{i,j} = w_{i,j} - \frac{\partial \text{Kosten}}{\partial w_{i,j}} \eta, \quad (2.7)$$

wobei η die Lernrate darstellt, welche frei wählbar ist. Üblicherweise liegt sie im Bereich von 0.01 und 0.5.

Die optimale Lernrate für das gegebene Problem muss experimentell bestimmt werden, da dazu im Vorfeld keine genauen Vorhersagen gemacht werden können. Trainiert wird das Netzwerk letztendlich mit dem Befehl [10]

```
1 tf.train.GradientDescentOptimizer(Lernrate).minimize(Kosten)
```

Das Gradientenabstiegsverfahren ist dabei eine der einfachsten Optimierungsalgorithmen für das Lernen. Es gehört zur Gruppe der sogenannten Optimierungsalgorithmen erster Ordnung [5]. Darauf aufbauend gibt es eine Reihe von Erweiterungen wie zum Beispiel den „Stochastic Gradient Descent“ [5].

Kapitel 3

3 Das Framework TensorFlow

TF ist eine plattformunabhängige Bibliothek für ML, die darauf optimiert wurde, auch große und variable Hardware-Architekturen effizient zu nutzen und sehr gut zu skalieren [13]. Sie wurde Ende 2015 veröffentlicht [14]. Um die einzelnen Verarbeitungsschritte der Daten darzustellen, werden von TF sogenannte Datenflussgraphen verwendet. Diese bieten auch die Möglichkeit für alle Operationen festzulegen, von welcher Hardware sie berechnet werden sollen. TF unterstützt dabei CPUs, GPGPUs¹ und eigens für ML entwickelte Hardware. Besonders umfangreich unterstützt das Framework Arbeiten im Bereich der Deep Neural Networks (DNNs). Schnittstellen zu den Hochsprachen Python und C++ sollen den Einstieg erleichtern und sicherstellen, dass die verfügbare Hardware immer bestmöglich genutzt werden kann. Mit Tensorboard bringt TF außerdem eine Weboberfläche mit, die ohne großen Aufwand für den Entwickler viele relevante Informationen ausgibt und teilweise auch grafisch aufbereitet [15].

3.1 Die Entwicklung von TensorFlow

Bereits 2011 begann das Google Brain Projekt damit, den Nutzen von sehr großen DNN zu erforschen. Einen Teil davon bildete der Aufbau von „DistBelief“, ein System, das Training und Vorhersage im Bereich des ML verteilt und skalierbar ermöglichte. Neben einigen Forschungsprojekten wurde DistBelief auch bereits produktiv in einigen Google-Produkten, wie Google Search, Google Translate oder Youtube eingesetzt [15]. 2012 wurde von Google-Mitarbeitern ein Paper veröffentlicht, das über die Nutzung von zehntausenden CPU-Kernen durch DistBelief berichtete, wodurch auch sehr große Modelle in absehbarer Zeit trainiert werden konnten [16]. Auf Basis der Erfahrungen beim Einsatz von DistBelief arbeitete Google dann am System der zweiten Generation für großskalierende ML-Modelle und veröffentlichte im November 2015 TF [14]. TF ist seitdem auf GitHub unter der Apache License 2.0 verfügbar [17]. Mit der Veröffentlichung von Version 1.0 im Februar 2017 wurde schließlich ein verlässliches Application Programming Interface (API) eingeführt, die auch in Zukunft sicherstellen soll, dass der geschriebene

¹general-purpose graphics processing units

Code mit neuen Versionen von TF kompatibel ist [18]. Des Weiteren wurde die Leistung weiter verbessert und die Einführung eines neuen Moduls, ermöglicht seitdem die Nutzung von TF mit Keras.

3.2 Angesprochene Zielgruppe

Im Gegensatz zu DistBelief, welches für viele Forschungsgebiete zu unflexibel war, ist TF sowohl für die Forschung als auch für den produktiven Einsatz in großen Softwareprojekten geeignet. Abbildung 3.1 wurde beim TensorFlow Dev Summit 2017 [19] gezeigt und veranschaulicht die Abdeckung der Zielgruppen beider Systeme.

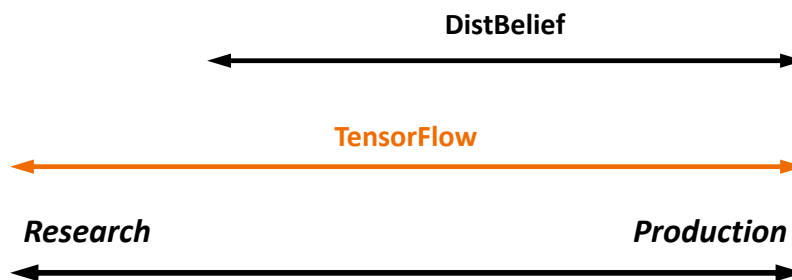


Abbildung 3.1: Zielgruppenabdeckung von DistBelief und TF [19]

TF bietet zum einen genug Flexibilität für Forschungsprojekte, um mit neuen Modellen zu experimentieren, ist gleichzeitig aber hochperformant und robust, was beim Einsatz in Produktivsoftware wichtig ist. Modelle können somit also oft aus der Forschung direkt in Produktivumgebungen übernommen werden [15].

3.3 Datenflussgraphen in TensorFlow

Einer der Hauptbestandteile für die Durchführung von Berechnungen mit TF sind Datenflussgraphen [20], im Original „Dataflow Graph“ genannt, welche aus einem oder mehreren Knoten bestehen [13]. Diese werden im Quellcode meist zu Beginn über eine der verfügbaren Client-Schnittstellen (Python oder C++) definiert und können bei Bedarf von TensorBoard dargestellt werden (Unterabschnitt 4.3.3). Listing 3.1 zeigt den Beispiel-Code für die Definition eines sehr einfachen Datenflussgraphen, bei dem zunächst in Zeile zwei ein Vektor b von der Größe 100 mit Nullen initialisiert wird. Anschließend wird in Zeile drei eine Matrix w mit den Dimensionen 625x100 mit Zufallswerten zwischen -1 und 1 belegt und in Zeile vier ein variabler Platzhalter x für die spätere Eingabe definiert. Der eigentliche Graph entsteht in Zeile fünf: Zunächst wird der Input x mit der Matrix w

multipliziert und anschließend der Vektor b aufaddiert. Das Ergebnis wird abschließend einer ReLU-Funktion (Unterabschnitt 2.4.1) übergeben. Diese Verkettung kann beliebig lange weitergeführt werden, was im Code in Zeile sechs und in der grafischen Darstellung in Abbildung 3.2 durch den Platzhalter symbolisiert wird.

Listing 3.1: Code-Beispiel zur Definition eines Graphen [13]

```
1 import tensorflow as tf
2 b = tf.Variable(tf.zeros([100]))
3 W = tf.Variable(tf.random_uniform([625,100],-1,1))
4 x = tf.placeholder(name="x")
5 relu = tf.nn.relu(tf.matmul(W, x) + b)
6 C = [...]
```

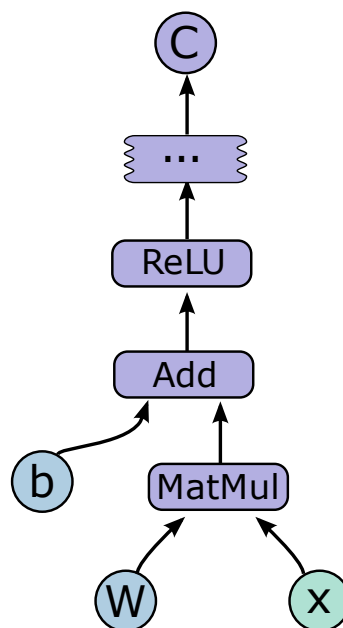


Abbildung 3.2: Datenflussgraph zu Listing 3.1 [13]

Jeder Knoten hat eine beliebige Anzahl an Ein- und Ausgängen und stellt eine Operation dar. Werte, die entlang normaler Kanten im Graph fließen (von Aus- zu Eingängen) werden „Tensoren“ genannt, welche beliebig dimensionierten Arrays mit fest definierten Datentypen entsprechen. Des Weiteren gibt es noch spezielle Kanten, genannt „control dependencies“ über die keine Daten laufen. Diese indizieren, dass die Berechnungen im Quellknoten abgeschlossen sein müssen, bevor der Zielknoten starten kann. Somit können diese control dependencies verwendet werden um „happens before“ Relationen zu implementieren, was zum Beispiel genutzt werden kann um den maximalen Speicherverbrauch zu kontrollieren.

3.4 Hard- und Software Anforderungen

TensorFlow wurde von Grund auf so gestaltet, dass es eine möglichst große Bandbreite an Hardware zur Berechnung der NN nutzen kann und auch größtenteils unabhängig von dem System ist, auf dem es ausgeführt wird.

3.4.1 Betriebssysteme

TF ist seit seinen Anfängen auf Linux und MacOS lauffähig. Mit Version 0.12 kam die Unterstützung für Windows hinzu [21]. Die reine Ausführung eines bereits trainierten Netzes (Inference) ist außerdem auch auf Android und iOS möglich [13][15].

3.4.2 Hardware Unterstützung

Lernalgorithmen basieren oft auf aufwändigen Rechenoperationen, die zwar hoch parallelisierbar sind, meist jedoch auch große Abhängigkeiten beim Datenzugriff aufweisen. Ein Beispiel hierfür sind Matrix-Multiplikationen. Mit dem Aufkommen von sogenannten „General Purpose GPUs“ die eine große Anzahl an Recheneinheiten und schnellen lokalen Speicher bieten, konnten die NN stark beschleunigt werden.

TF bietet hier den Vorteil, dass es unabhängig von der vorhandenen Hardware eingesetzt werden kann. Es werden sowohl CPUs als auch eine Vielzahl von NVIDIA GPUs unterstützt. Im Moment können GPUs anderer Hersteller noch nicht genutzt werden, da TF die CUDA-API von NVIDIA nutzt. Diese „CUDA Compute Capability“ muss hardwareseitig auch mindestens in Version 3.0 implementiert sein. Eine Liste mit allen unterstützten Karten findet sich auf der [NVIDIA Homepage](#) [22].

Neben diesen allgemein gehaltenen Recheneinheiten kann TF auch spezielle Hardware für die Berechnungen nutzen. So gibt es von Google selbst sogenannte Tensor Processing Units (TPUs), die eine enorme Beschleunigung von Inference ermöglichen und dabei vor allem wesentlich energieeffizienter sind, als CPUs oder auch GPUs [23]. Da es sich hierbei jedoch um proprietäre Hardware handelt, können diese TPUs im Moment nur als Service in der Google Cloud genutzt werden [24]. Außerdem kann der sogenannte „Neural Compute Stick“ von Movidius (inzwischen Intel) [25] verwendet werden, der es auch auf leistungsschwacher Hardware wie zum Beispiel einem Raspberry Pi möglich macht, Inference in NN durchzuführen.

In TF kann genau definiert werden, welche Schritte im Programmcode von welcher Recheneinheit ausgeführt werden sollen. So wird in Zeile zwei von Listing 3.2 angegeben, dass die dritte GPU im System verwendet werden soll² [26].

Listing 3.2: Festlegung des Geräts, auf dem die Berechnung getätigt werden soll

```
1 # Creates a graph.
2 with tf.device('/device:GPU:2'):
3     a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
4     b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
5     c = tf.matmul(a, b)
6 # Creates a session with log_device_placement set to True.
7 sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
8 # Runs the op.
9 print(sess.run(c))
```

In den Zeilen drei und vier werden zwei konstante Arrays definiert und in Zeile fünf wird die durchzuführende Rechenoperation festgelegt. In Zeile sieben wird eine TensorFlow Session erstellt und mit „sess.run“ in Zeile neun wird schließlich die Berechnung gestartet.

3.4.3 Software Anforderungen

TF ist sowohl unter Python 2.7 als auch unter Python 3 ab Version 3.4 [22] lauffähig. Alternativ sind aber auch Docker Images verfügbar.

Für die Verwendung von NVIDIA GPUs muss außerdem das CUDA® Toolkit in Version 8.0 sowie cuDNN v6.0 installiert werden [22].

3.5 Architektur von TensorFlow

TF wurde als erweiterbare, plattformübergreifende Bibliothek konzipiert. Abbildung 3.3 zeigt, wie dies durch Modularisierung erreicht wurde. Dabei wird alles oberhalb der C-API als „user-level“ bezeichnet und alles darunter als „core library“ [15].

Die meisten Nutzer von TF werden sich der bereitgestellten Client-Schnittstellen bedienen. Diese sind für die beiden Sprachen „Python“ und „C++“ vorhanden. Java- und Go-Schnittstellen befinden sich aber in der Entwicklung und können bereits getestet werden. Alle Schnittstellen kommunizieren mit einer C-API, die den Zugriff auf die

²TF beginnt bei null zu zählen

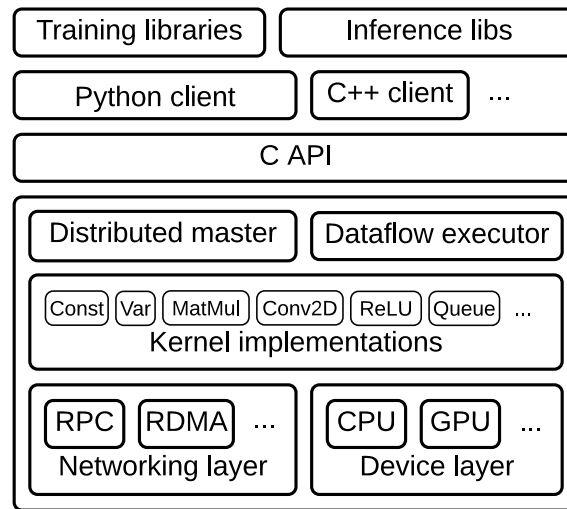


Abbildung 3.3: Die Architektur von TF [15]

core-library von TF ermöglicht. Diese wurde in C++ implementiert um Portabilität und Leistungsfähigkeit zu gewährleisten. Im user-level befinden sich außerdem noch Training- und Inference-Bibliotheken, in denen häufig genutzte Funktionen wie Backpropagation oder das Gradientenabstiegsverfahren aus Kapitel 2.6 bereits implementiert wurden.

Der Kern von TF besteht aus einer Vielzahl an Komponenten. Der sogenannte „distributed master“ teilt den Berechnungsgraphen in mehrere Sub-Graphen auf, die dann auf den einzelnen Recheneinheiten vom „dataflow executor“ ausgeführt werden. Des weiteren sind mehr als 200 Standard-Operationen für zum Beispiel Array Manipulation oder Zustandsmanagement enthalten. Dabei wurde an vielen Stellen auf bereits bestehende Open-Source-Software aufgebaut, die eine effiziente Parallelberechnung auf Mehrkern-CPUs und GPUs sicherstellt.

Im „Networking layer“ wurden erweiterte Netzwerkprotokolle wie „gRPC over TCP“ oder „RDMA over Converged Ethernet“ integriert, die eine schnelle Kommunikation zwischen entfernten Rechnern sicherstellen sollen. Doch auch innerhalb eines Rechners wurden zusätzliche Funktionen implementiert, die einen schnellen Datenaustausch von beispielsweise zwei GPUs ermöglichen sollen.

3.6 Nutzung von TensorFlow mit Keras

Keras ist eine in Python geschriebene Bibliothek speziell zum Erstellen von NN, die TF, CNTK ³ oder Theano ⁴ als Backend für die Berechnungen verwenden kann. Der Fokus liegt dabei auf der schnellen Umsetzung von experimentellem Code und Unterstützung beim Prototyping [27]. Hauptsächlich wird Keras von Google entwickelt, Teile jedoch auch von Microsoft, Amazon, NVIDIA und Weiteren. Keras wurde mit Version 1.4 auch direkt in TF integriert [21].

In Listing 4.3 (Anhang) ist ein Beispiel-Code für das Trainieren eines Modells auf dem MNIST-Datenset zu finden. Besonders interessant sind hier die Zeilen 21–27, in denen das Modell definiert wird. Zunächst wird in Zeile 21 festgelegt, dass ein Sequentielles Modell erstellt werden soll. Nachfolgend kann man die einzelnen Layer jeweils mit der `.add` Funktion hinzufügen. So definieren die Zeilen 22–24 drei sogenannte Dense- oder Fully-Connected-Layer, bei denen jeweils jeder Knoten eine Kante zu jedem Knoten der übergeordneten Schicht besitzt. Bei den ersten beiden wird als Aktivierungsfunktion wieder die ReLU Funktion verwendet, bei der letzten kommt eine Softmax-Funktion zum Einsatz, die Wahrscheinlichkeiten auf die einzelnen Klassen aufteilt und somit bestimmt, welches Ergebnis ausgegeben wird. Der Code zeigt, wie einfach und vor allem intuitiv ein Modell mit Hilfe von Keras erstellt werden kann.

³CNTK ist ein Deep-Learning Toolkit von Microsoft

⁴Theano ist eine Bibliothek für numerische Berechnungen und wurde zu großen Teilen von der Universität Montreal entwickelt

Kapitel 4

4 Der Allgemeine Workflow in TensorFlow

Zu Beginn dieses Kapitels wird die Methodik der Aufteilung der Datensätze in Trainings-, Validierungs- und Testdaten erläutert, welches für die Beurteilung der Ergebnisse eines Modells unerlässlich ist. Anschließend erfolgt eine Einführung in das Visualisierungstool TensorBoard, mit dessen Hilfe sich die hier vorgestellten und noch weitere Vorgänge anschaulich abbilden lassen. Am Schluss werden noch die einzelnen Graphenelemente näher betrachtet, wodurch sich umfangreiche Graphabbildungen erzeugen lassen.

4.1 Vorgehensweise beim Trainingsprozess

Eines der entscheidenden Kriterien über die Qualität des Modells ist die Prognose der Zielgröße auf zuvor ungesehenen Daten. Die Genauigkeit auf den Daten, mit denen das Modell trainiert wurde, geben keine grundlegende Aussage über die zu erwartende Genauigkeit auf zukünftige Daten. Deshalb ist es essentiell wichtig ungenutzte Testdaten zur Beurteilung mit einzubeziehen. Häufig müssen Entscheidungen über Metaparameter wie Neuronenanzahl, Stärke der Regularisierung, Art des Kernels getroffen werden. Daher unterteilt man die Datensätze in Trainings-, Validierungs- und Testdaten [28].

- **Training**

Trainingsdaten bilden die Grundlage für das überwachte Lernen. Hierbei werden durch eine Reihe von Beispielen Parameter wie die Gewichte der Verbindungen zwischen den Neuronen angepasst. Häufig besteht der Trainingsdatensatz aus Paaren von Eingangsvektoren und der dazugehörigen Antwortvektoren.

- **Validierung**

Validierungsdaten dienen der Festlegung der optimalen Struktur des Modells, insbesondere zur Einstellung der optimalen Parameter des Lernalgorithmus wie die

Lernrate oder die Anzahl der Trainingsepochen. Validierungsdatensätze werden auch für die Regularisierung von "early stopping" und bei der Zunahme des Fehlers im Validierungsdatensatz, da dies ein Zeichen für "overfitting" ist, angewandt.

- **Testen**

Testdaten werden verwendet, um eine Bewertung eines endgültigen Modells zu ermöglichen, welche auf ungesehene Daten angewandt wurde.

Cross-Validation

Die Unterteilung des Datensatzes in einen festen Trainingssatz und einen festen Testsatz kann problematisch sein, wenn der Testsatz zu klein ist. Dies impliziert statistische Unsicherheit im Bereich des geschätzten Testfehlers. Abhilfe schafft die k-fache Kreuzvalidierung (Cross-Validation), welche die Daten in k disjunkte Teilmengen unterteilt, auf denen k Tests durchgeführt werden, wobei jeweils k-1 Teilmengen für das Training und die verbliebene Teilmenge zum Testen verwendet wird [28].

4.2 Die Visualisierung mit TensorBoard

TensorBoard ist eine in TensorFlow enthaltene Webanwendung zur Visualisierung der Abläufe in einem Neuronalen Netz. TensorBoard stellt eine Vielzahl an graphischen Elementen zur Verfügung, welche dem Entwickler vorallem das Debuggen oder die Optimierung eines erstellten Modells erleichtern. Ebenso können damit insbesondere komplexe Datenstrukturen zum besseren Verständnis anschaulich visualisiert werden.

Bevor mit TensorBoard gearbeitet werden kann, muss nachfolgender Befehl in die Kommandozeile eingegeben werden:

```
1 C:\> tensorboard --logdir=path/to/log_directory
```

wobei vorher im spezifizierten Log Ordner die gewünschten Event Daten mit der Klasse

```
1 tf.summary.FileWriter('path/to/log_directory')
```

abgespeichert werden müssen. Nachdem TensorBoard gestartet wurde, navigiert man mit dem Browser zu folgender Seite:

```
1 http://localhost:6006
```

Falls keine Fehlermeldungen aufgetreten sind, sollte folgender Bildschirminhalt Abbildung 4.1 angezeigt werden. Um die Visualisierungen in den einzelnen Reiter zu erhalten, müssen diese vorher im Programm mit speziellen TensorFlow Klassen abgespeichert werden. In dem nachfolgenden Kapitel werden die einzelnen Reiter und der dazugehörigen Klasse vorgestellt.

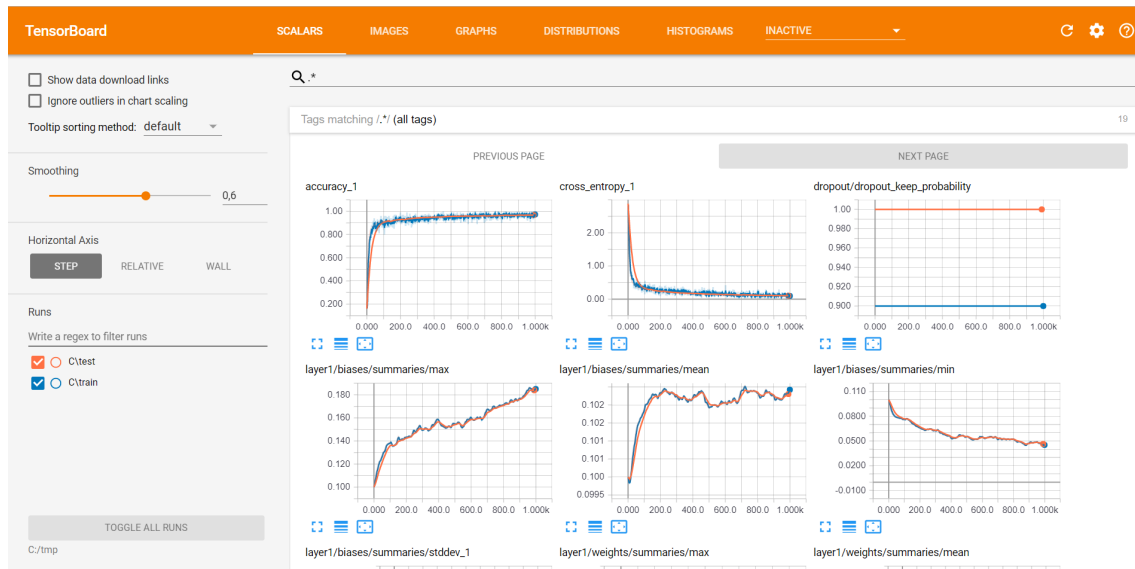


Abbildung 4.1: Die Startseite von TensorBoard

4.3 Die einzelnen Visualisierungsmöglichkeiten im Detail

4.3.1 Skalare

Unter dem Reiter Skalare, welche mit der Klasse

```
1 tf.summary.scalar(name, tensor, collections=None, family=None)
```

abgespeichert werden, können verschiedenste Statistiken während eines Trainingsprozesses visualisiert werden. Ein Name und ein Tensor müssen bei der Klassendeklaration angegeben werden, im Gegensatz zu den Schlüsselwortparametern *collections* (Zusammenfassung

mehrerer Operationen in einer Sammlung) und *family* (Zusammenfassung der Skalare durch verschachtelte Tag-Tags) welche nicht zwingend erforderlich sind. Dies könnten zum Beispiel die Genauigkeit (Accuracy) oder Kreuzentropie (Cross Entropy) sein, welche in Abbildung 4.2 dargestellt sind. Hierbei wird die Genauigkeit über den einzelnen Trainingsschritten aufgetragen. Wählt man mit der Maus einen bestimmten Datenpunkt aus, so werden zahlreiche weitere Informationen angezeigt. Ebenso ist hierbei ersichtlich, dass auch Trainings- und Testdaten gleichzeitig angezeigt und miteinander verglichen werden können [29].

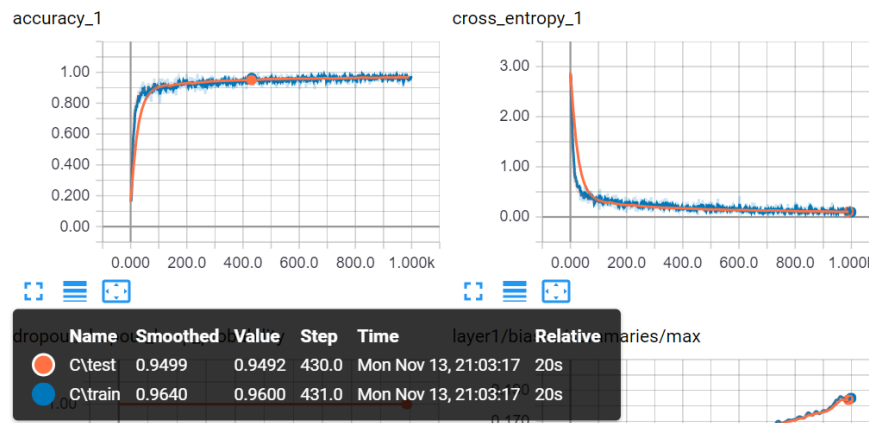


Abbildung 4.2: Visualisierung der Accuracy und Cross Entropy über die einzelnen Trainingsschritte

Der in Abbildung 4.2 eingestellte Glättungsfaktor beträgt 0.6, wodurch der veränderte Wert 0.9499 bei *Smoothed* zustande kommt. Bei einem Glättungsfaktor von 0 sind die Werte *Smoothed* und *Value* identisch.

4.3.2 Bilder

Innerhalb des Reiters Bilder, welche mit der Klasse

```
1 tf.summary.image(name, tensor, max_outputs=3, collections=None, family=
    None)
```

abgespeichert werden, können zur genaueren Analyse die Test- und Trainingsbilder eingesehen werden. Wie in Abbildung 4.3 ersichtlich, ist über den Bildern eine Scrollbar vorhanden mit dieser können einzelne Test- und Trainingsschritte ausgewählt werden, wodurch genau ersichtlich wird, welches Bild zum aktuellen Durchlauf gehört [29].

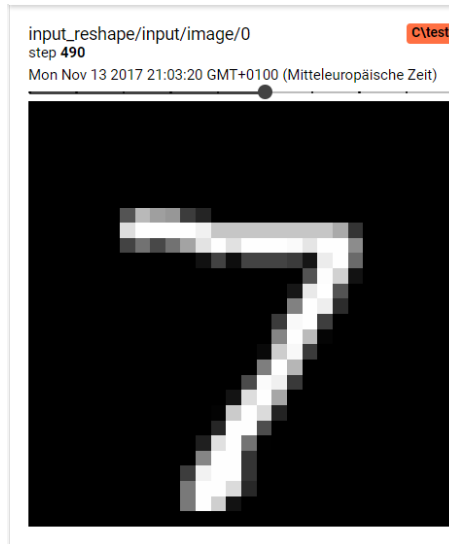


Abbildung 4.3: Bild des Testschrittes 490

4.3.3 Graphen

Unter dem Reiter Graphen befindet sich das komplette TensorFlow Model als Graph, wie in Abbildung 4.4 zu sehen ist. Nur wenn im vorher erstellten Programm Name scopes zu den gewünschten Operationen definiert wurden, werden diese in TensorBoard als Graph angezeigt. Mit nachfolgendem Befehl in Zeile 1 erhält man einen *cross_entropy* Name scope:

Listing 4.1: Erstellen verschachtelter Name scopes

```
1 # create a 'cross_entropy' name scope
2 with tf.name_scope('cross_entropy'):
3     diff = tf.nn.softmax_cross_entropy_with_logits(targets=y_, logits=y)
4     # name scope 'total' within name scope 'cross_entropy'
5     with tf.name_scope('total'):
6         cross_entropy = tf.reduce_mean(diff)
7 tf.summary.scalar('cross_entropy', cross_entropy)
```

Der Name scope *cross_entropy* in Abbildung 4.5 kann natürlich wiederum in mehrere Untergruppierungen aufgeteilt werden. So erhält man eine übersichtliche Visualisierung komplexer TensorFlow Modelle [29].

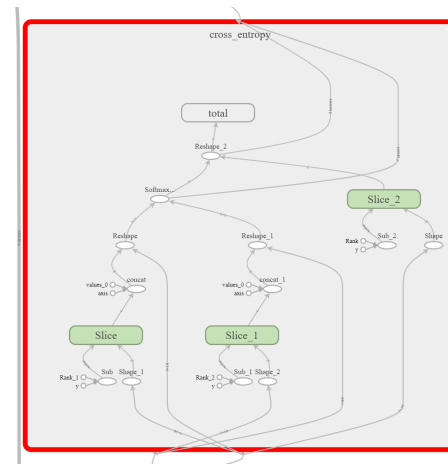
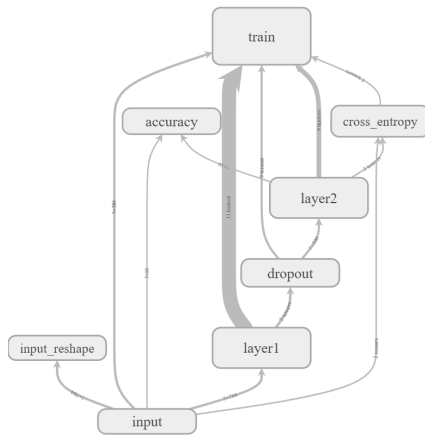


Abbildung 4.4: TensorFlow Graph mit definierten Name scopes

Abbildung 4.5: Name 'cross_entropy' unterteilt in weiteren Operationen

4.3.4 Histogramme

Unter dem Reiter Histogramme, welche mit der Klasse

```
1 tf.summary.histogram(name, values, collections=None, family=None)
```

abgespeichert werden, wird die statistische Verteilung eines Tensors über der Zeit dargestellt. Im Histogramm sind zeitliche "Slices" der Daten visualisiert, wobei jeder einzelne Slice ein Histogramm des Tensors in einem einzelnen Schritt darstellt. In Abbildung 4.6 ist ein einzelner Slice schwarz markiert [29].

4.3.5 Verteilungen

Unter dem Reiter Verteilungen, welche ebenfalls mit der Klasse

```
1 tf.summary.histogram(name, values, collections=None, family=None)
```

abgespeichert werden, befindet sich eine weitere Möglichkeit der Visualisierung der statistischen Verteilung. Hierbei repräsentiert die oberste Linie den über die Zeit veränderten maximalen Wert, die unterste Linie den minimalen Wert und die mittlere Linie den veränderten Median über der Zeit [29].

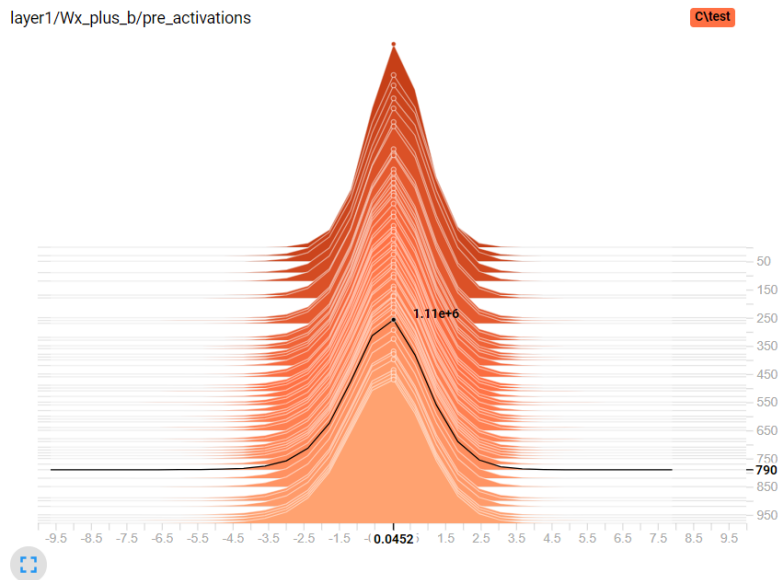


Abbildung 4.6: Visualisierung eines Histogramms über die einzelnen Trainingschritte

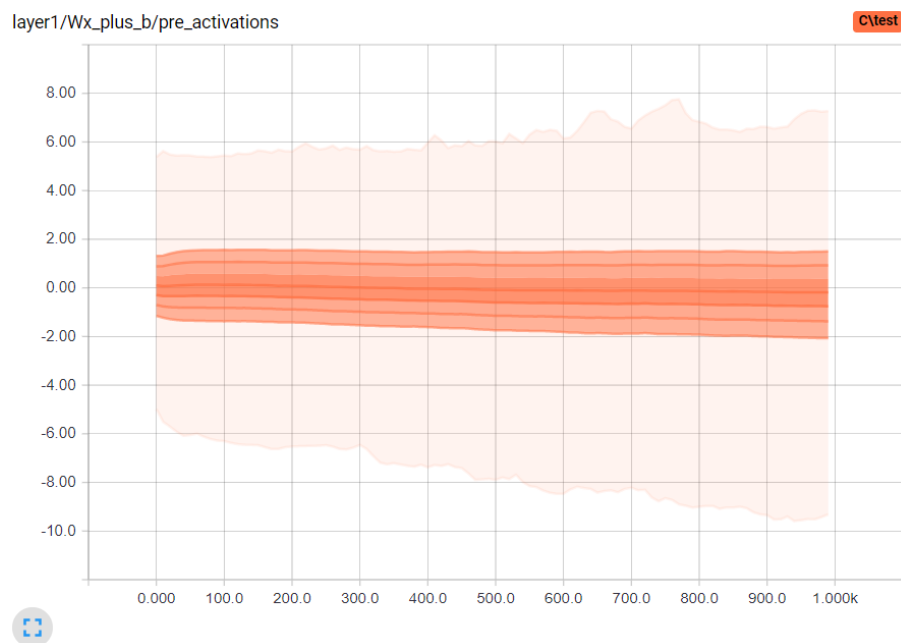


Abbildung 4.7: Eine weitere Möglichkeit der Visualisierung der statistischen Verteilung

4.3.6 Projektor

Die Auswahl des Reiters Projektors ermöglicht die höherdimensionale Visualisierung der Eingabedaten. Nachfolgend eine mögliche minimale Konfiguration des Projektors:

Listing 4.2: Konfiguration eines Projektors in TensorFlow [30]

```
1 from tensorflow.contrib.tensorboard.plugins import projector
2
3 # Use the same LOG_DIR where you stored your checkpoint.
4 summary_writer = tf.summary.FileWriter(LOG_DIR)
5
6 # Format: tensorflow/contrib/tensorboard/plugins/projector/
   projector_config.proto
7 config = projector.ProjectorConfig()
8
9 # You can add multiple embeddings. Here we add only one.
10 embedding = config.embeddings.add()
11 embedding.tensor_name = embedding_var.name
12
13 # Link this tensor to its metadata file (e.g. labels).
14 embedding.metadata_path = os.path.join(LOG_DIR, 'metadata.tsv')
15
16 # Saves a configuration file that TensorBoard will read during startup.
17 projector.visualize_embeddings(summary_writer, config)
```

Hierbei werden zwei wesentliche Darstellungen unterschieden.

- Principal Component Analysis (PCA)

Ein häufiges Problem bei multivariaten Daten ist, dass diese nicht im zweidimensionalen Raum dargestellt werden können. Hierbei werden bei der Hauptkomponentenanalyse (PCA) die Daten so auf eine zweidimensionale (bzw. dreidimensionaler) Ebene projiziert, mit der Erwartung, dass diese neue Darstellung eventuell vorhandenes Rauschen herausfiltert und versteckte Strukturen zum Vorschein bringt [31].

- t-Distributed Stochastic Neighbor Embedding (t-SNE)

Diese Technik der Dimensionsreduktion eignet sich besonders gut, um hochdimensionale Daten in einen Raum von zwei oder drei Dimensionen zu projizieren. Hierbei wird jedes hochdimensionale Objekt durch einen zwei- oder dreidimensionalen Punkt modelliert, sodass ähnliche Objekte durch nahe gelegene Punkte und ungleiche Objekte durch entfernte Punkte modelliert werden [32].

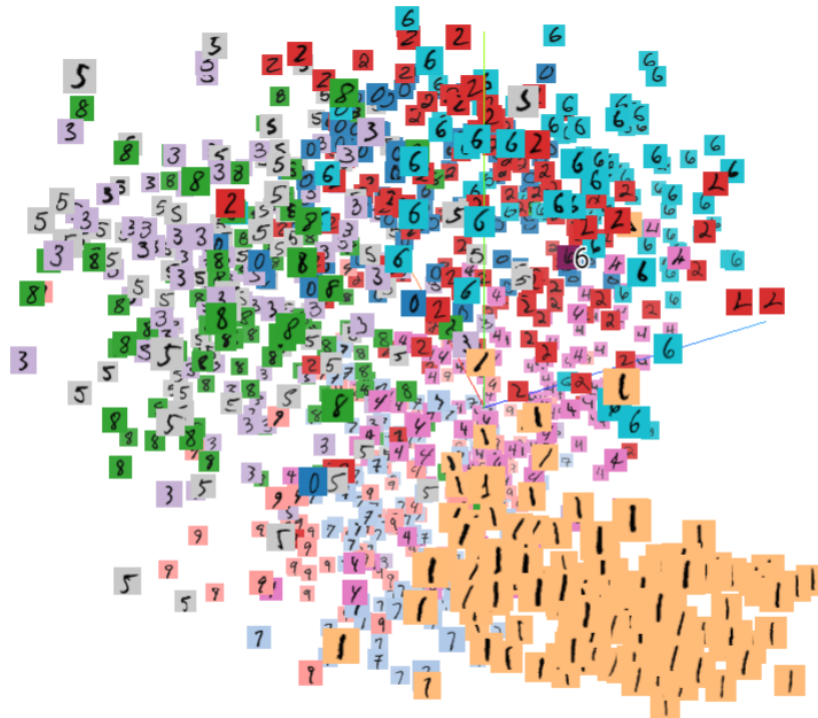


Abbildung 4.8: Visualisierung der PCA

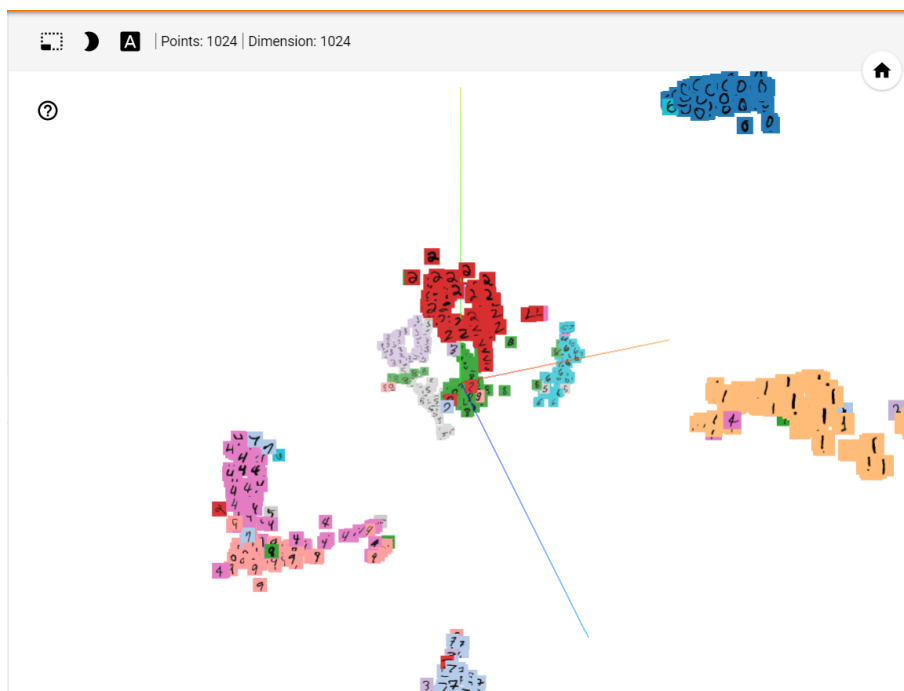


Abbildung 4.9: Visualisierung der t-SNE

4.3.7 Audio und Text

Unter dem Reiter Audio können mit der Klasse

```
1 tf.summary.audio(name, tensor, sample_rate, max_outputs=3, collections=  
    None, family=None)
```

abspielbare Audio-Widgets eingebettet werden. Die Deklaration dieser Klasse folgt analog der vorherigen Klassen, nur das zusätzlich eine *sample_rate* angegeben werden muss.

Ebenso können unter dem Reiter Text mit der Klasse

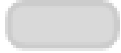
```
1 tf.summary.text(name, tensor, collections=None)
```

Textausschnitte abgespeichert werden. Zusätzliche Funktionen wie Hyperlinks, Listen und Tabellen werden unterstützt [29].

4.4 Die Graphelemente im Datenfluss

Dafür wurden zahlreiche unterschiedliche Elemente definiert, welche nachfolgend kurz erläutert werden [33].

Namespace



High-level Knoten repräsentiert einen definierten Name scope.

Unconnected series



Nummerierte Knoten, die nicht miteinander verbunden sind.

Connected series



Nummerierte Knoten, die miteinander verbunden sind.

Operation node



Ein Knoten der eine einzelne Operation darstellt.

Constant



Repräsentiert eine Konstante im Programm.

Summary node



Dieser Knoten stellt eine Zusammenfassung dar.

Dataflow edge



Durchgezogener Pfeil zeigt den Datenfluss zwischen den Operationen an.

Control edge



Gepunkteter Pfeil zeigt die Steuerungsabhängigkeit zwischen den Operationen an.

Reference edge



Gelber Pfeil bedeutet, dass die ausgehende Operation die ankommende mutieren kann

Abbildungsverzeichnis

2.1	Feedforward Netz	4
2.2	Sigmoidfunktion [10]	8
2.3	Graph des Tangenshyperbolicus [11]	9
3.1	Zielgruppenabdeckung von DistBelief und TF [19]	13
3.2	Datenflussgraph zu Listing 3.1 [13]	14
3.3	Die Architektur von TF [15]	17
4.1	Die Startseite von TensorBoard	21
4.2	Visualisierung der 'Accuracy' und 'Cross_entropy' über die einzelnen Trainingsschritte	22
4.3	Bild des Testschrittes 490	23
4.4	TensorFlow Graph mit definierten Name scopes	24
4.5	Name scope 'cross_entropy' unterteilt in weiteren Operationen	24
4.6	Visualisierung eines Histogramms über die einzelnen Trainingsschritte	25
4.7	Eine weitere Möglichkeit der Visualisierung der statistischen Verteilung	25
4.8	Visualisierung der PCA	27
4.9	Visualisierung der t-SNE	27

Anhang

Listing 4.3: Beispiel für MNIST mit Keras

```
1  [...]
2
3  batch_size = 128
4  num_classes = 10
5  epochs = 20
6
7  # the data, shuffled and split between train and test sets
8  (x_train, y_train), (x_test, y_test) = mnist.load_data()
9
10 x_train = x_train.reshape(60000, 784)
11 x_test = x_test.reshape(10000, 784)
12 x_train = x_train.astype('float32')
13 x_test = x_test.astype('float32')
14 x_train /= 255
15 x_test /= 255
16
17 # convert class vectors to binary class matrices
18 y_train = keras.utils.to_categorical(y_train, num_classes)
19 y_test = keras.utils.to_categorical(y_test, num_classes)
20
21 model = Sequential()
22 model.add(Dense(512, activation='relu', input_shape=(784,)))
23 model.add(Dense(512, activation='relu'))
24 model.add(Dense(num_classes, activation='softmax'))
25 model.summary()
26
27 model.compile(loss='categorical_crossentropy',
28               optimizer=RMSprop(),
29               metrics=['accuracy'])
30 history = model.fit(x_train, y_train,
31                     batch_size=batch_size,
32                     epochs=epochs,
33                     verbose=1,
34                     validation_data=(x_test, y_test))
35 score = model.evaluate(x_test, y_test, verbose=0)
```

Literaturverzeichnis

- [1] LEE, C. S. ; WANG, M. H. ; YEN, S. J. ; WEI, T. H. ; WU, I. C. ; CHOU, P. C. ; CHOU, C. H. ; WANG, M. W. ; YAN, T. H.: Human vs. Computer Go: Review and Prospect [Discussion Forum]. In: *IEEE Computational Intelligence Magazine* 11 (2016), Aug, Nr. 3, S. 67–72. <http://dx.doi.org/10.1109/MCI.2016.2572559>. – DOI 10.1109/MCI.2016.2572559. – ISSN 1556–603X
- [2] RUSSELL, Stuart ; NORVIG, Peter ; KIRCHNER, Frank: *Künstliche Intelligenz: Ein moderner Ansatz*. 3., aktualisierte Aufl. München : Pearson Higher Education, 2012 (Always learning). – ISBN 978–3–86894–098–5
- [3] TURING, A. M.: I.—COMPUTING MACHINERY AND INTELLIGENCE. In: *Mind* LIX (1950), Nr. 236, S. 433–460. <http://dx.doi.org/10.1093/mind/LIX.236.433>. – DOI 10.1093/mind/LIX.236.433. – ISSN 0026–4423
- [4] MCCLURE, N.: *TensorFlow Machine Learning Cookbook*. Packt Publishing, 2017 <https://books.google.de/books?id=LVQoDwAAQBAJ>. – ISBN 9781786466303
- [5] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. MIT Press, 2016. – <http://www.deeplearningbook.org>
- [6] ERTEL, Wolfgang: *Grundkurs Künstliche Intelligenz - Eine praxisorientierte Einführung*. Berlin Heidelberg New York : Springer-Verlag, 2013. – ISBN 978–3–834–82157–7
- [7] BISHOP, Christopher M.: *Neural Networks for Pattern Recognition*. Oxford : Clarendon Press, 1995. – ISBN 978–0–198–53864–6
- [8] GÉRON, A.: *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017 <https://books.google.de/books?id=bRpYDgAAQBAJ>. – ISBN 9781491962244
- [9] RASHID, Tariq: *Make Your Own Neural Network*. 1st. USA : CreateSpace Independent Publishing Platform, 2016. – ISBN 1530826608, 9781530826605
- [10] BONNIN, R.: *Building Machine Learning Projects with TensorFlow*. Packt Publishing, 2016 <https://books.google.de/books?id=pZ3cDgAAQBAJ>. – ISBN 9781786466822

- [11] ROJAS, Raul ; VARGA, Peter: *Neural Networks - A Systematic Introduction*. Berlin Heidelberg : Springer Science, Business Media, 1996. – ISBN 978–3–540–60505–8
- [12] MICHAEL, Nielsen: Improving the way neural networks learn. (2017), Aug. <http://neuralnetworksanddeeplearning.com/chap3.html>
- [13] ABADI, Martín ; AGARWAL, Ashish ; BARHAM, Paul ; BREVDO, Eugene ; CHEN, Zhifeng ; CITRO, Craig ; CORRADO, Greg S. ; DAVIS, Andy ; DEAN, Jeffrey ; DEVIN, Matthieu ; GHEMAWAT, Sanjay ; GOODFELLOW, Ian ; HARP, Andrew ; IRVING, Geoffrey ; ISARD, Michael ; JIA, Yangqing ; JOZEFOWICZ, Rafal ; KAISER, Lukasz ; KUDLUR, Manjunath ; LEVENBERG, Josh ; MANÉ, Dan ; MONGA, Rajat ; MOORE, Sherry ; MURRAY, Derek ; OLAH, Chris ; SCHUSTER, Mike ; SHLENS, Jonathon ; STEINER, Benoit ; SUTSKEVER, Ilya ; TALWAR, Kunal ; TUCKER, Paul ; VANHOUCKE, Vincent ; VASUDEVAN, Vijay ; VIÉGAS, Fernanda ; VINYALS, Oriol ; WARDEN, Pete ; WATTENBERG, Martin ; WICKE, Martin ; YU, Yuan ; ZHENG, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. <https://www.tensorflow.org/>. Version: 2015. – Software available from tensorflow.org
- [14] DEAN, Jeff ; MONGA, Rajat: *TensorFlow - Google's latest machine learning system, open sourced for everyone*. https://research.googleblog.com/2015/11/tensorflow-googles-latest-machine_9.html, . – [letzter Zugriff: 28. Nov 2017]
- [15] ABADI, Martín ; BARHAM, Paul ; CHEN, Jianmin ; CHEN, Zhifeng ; DAVIS, Andy ; DEAN, Jeffrey ; DEVIN, Matthieu ; GHEMAWAT, Sanjay ; IRVING, Geoffrey ; ISARD, Michael ; KUDLUR, Manjunath ; LEVENBERG, Josh ; MONGA, Rajat ; MOORE, Sherry ; MURRAY, Derek G. ; STEINER, Benoit ; TUCKER, Paul ; VASUDEVAN, Vijay ; WARDEN, Pete ; WICKE, Martin ; YU, Yuan ; ZHENG, Xiaoqiang: *TensorFlow: A system for large-scale machine learning*. <https://www.tensorflow.org/>. Version: 2016. – Software available from tensorflow.org
- [16] DEAN, Jeffrey ; CORRADO, Greg ; MONGA, Rajat ; CHEN, Kai ; DEVIN, Matthieu ; MAO, Mark ; RANZATO, Marc'aurelio ; SENIOR, Andrew ; TUCKER, Paul ; YANG, Ke ; LE, Quoc V. ; NG, Andrew Y.: Large Scale Distributed Deep Networks. Version: 2012. <http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf>. In: PEREIRA, F. (Hrsg.) ; BURGESS, C. J. C. (Hrsg.) ; BOTTOU, L. (Hrsg.) ; WEINBERGER, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, 1223–1231

- [17] *tensorflow*, *Computation using data flow graphs for scalable machine learning*. <https://github.com/tensorflow/tensorflow>, . – [letzter Zugriff: 28. Nov 2017]
- [18] SANDJIDEH, Amy M.: *Announcing TensorFlow 1.0*. <https://developers.googleblog.com/2017/02/announcing-tensorflow-1.0.html>, . – [letzter Zugriff: 29. Nov 2017]
- [19] JEFF DEAN, Rajat M. ; KACHOLIA, Megan: *Keynote (TensorFlow Dev Summit 2017)*. <https://www.youtube.com/watch?v=4n1AHvDvVvw>, . – [letzter Zugriff: 01. Dez 2017]
- [20] ARVIND ; ; CULLER, D E.: Dataflow Architectures. In: *Annual Review of Computer Science* 1 (1986), Nr. 1, 225-253. <http://dx.doi.org/10.1146/annurev.cs.01.060186.001301>. – DOI 10.1146/annurev.cs.01.060186.001301
- [21] *TensorFlow Release*. <https://github.com/tensorflow/tensorflow/blob/master/RELEASE.md>, . – [letzter Zugriff: 06. Jan 2018]
- [22] *Installing TensorFlow on Ubuntu*. https://www.tensorflow.org/install/install_linux, . – [letzter Zugriff: 19. Dez 2017]
- [23] JOUPPI, Norman P. ; YOUNG, Cliff ; PATIL, Nishant ; PATTERSON, David ; AGRAWAL, Gaurav ; BAJWA, Raminder ; BATES, Sarah ; BHATIA, Suresh ; BODEN, Nan ; BORCHERS, Al ; BOYLE, Rick ; CANTIN, Pierre-luc ; CHAO, Clifford ; CLARK, Chris ; CORIELL, Jeremy ; DALEY, Mike ; DAU, Matt ; DEAN, Jeffrey ; GELB, Ben ; GHAEMMAGHAMI, Tara V. ; GOTTIPATI, Rajendra ; GULLAND, William ; HAGMANN, Robert ; HO, Richard C. ; HOGBERG, Doug ; HU, John ; HUNDT, Robert ; HURT, Dan ; IBARZ, Julian ; JAFFEY, Aaron ; JAWORSKI, Alek ; KAPLAN, Alexander ; KHAITAN, Harshit ; KOCH, Andy ; KUMAR, Naveen ; LACY, Steve ; LAUDON, James ; LAW, James ; LE, Diemthu ; LEARY, Chris ; LIU, Zhuyuan ; LUCKE, Kyle ; LUNDIN, Alan ; MACKEAN, Gordon ; MAGGIORE, Adriana ; MAHONY, Maire ; MILLER, Kieran ; NAGARAJAN, Rahul ; NARAYANASWAMI, Ravi ; NI, Ray ; NIX, Kathy ; NORRIE, Thomas ; OMERNICK, Mark ; PENUKONDA, Narayana ; PHELPS, Andy ; ROSS, Jonathan ; SALEK, Amir ; SAMADIANI, Emad ; SEVERN, Chris ; SIZIKOV, Gregory ; SNELHAM, Matthew ; SOUTER, Jed ; STEINBERG, Dan ; SWING, Andy ; TAN, Mercedes ; THORSON, Gregory ; TIAN, Bo ; TOMA, Horia ; TUTTLE, Erick ; VASUDEVAN, Vijay ; WALTER, Richard ; WANG, Walter ; WILCOX, Eric ; YOON, Doe H.: In-Datacenter Performance Analysis of a Tensor Processing Unit. In: *CoRR* abs/1704.04760 (2017). <http://arxiv.org/abs/1704.04760>

- [24] JEFF DEAN, Urs H.: *Build and train machine learning models on our new Google Cloud TPUs*. <https://blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/>, . – [letzter Zugriff: 21. Dez 2017]
- [25] *Introducing: TensorFlow™ Support for Neural Compute Stick*. <https://www.movidius.com/news/introducing-tensorflow-support-for-neural-compute-stick>, . – [letzter Zugriff: 19. Dez 2017]
- [26] *Using GPUs / TensorFlow*. https://www.tensorflow.org/tutorials/using_gpu, . – [letzter Zugriff: 21. Dez 2017]
- [27] CHOLLET, François u. a.: *Keras*. <https://github.com/keras-team/keras>, 2015
- [28] HOFFMANN, F. ; HÜLLERMEIER, E.: *Proceedings. 22. Workshop Computational Intelligence, Dortmund, 6. - 7. Dezember 2012*. Karlsruher Institut für Technologie, 2014 <https://books.google.de/books?id=93djoc03yVgC>. – ISBN 9783866449176
- [29] *TensorBoard*. <https://github.com/tensorflow/tensorboard>. – Zuletzt besucht am 04.01.2018
- [30] *Embedding Visualization*. https://www.tensorflow.org/versions/r1.1/get_started/embedding_viz. – Zuletzt besucht am 04.01.2018
- [31] *Principal Component Analysis*. http://www.statistics4u.info/fundstat_germ/cc_pca.html. – Zuletzt besucht am 04.01.2018
- [32] *T-distributed Stochastic Neighbor Embedding*. https://www.tensorflow.org/programmers_guide/embedding. – Zuletzt besucht am 04.01.2018
- [33] *Graph Visualization*. https://www.tensorflow.org/versions/r1.1/get_started/graph_viz. – Zuletzt besucht am 04.01.2018
- [34] *TensorFlow v0.12.0 RC0*. <https://github.com/tensorflow/tensorflow/releases/tag/0.12.0-rc0>, . – [letzter Zugriff: 19. Dez 2017]
- [35] BARRAT, James: *Our Final Invention: Artificial Intelligence and the End of the Human Era*. THOMAS DUNNE BOOKS, 2013 http://www.ebook.de/de/product/20253628/james_barrat_our_final_invention_artificial_intelligence_and_the_end_of_the_human_era.html. – ISBN 0312622376
- [36] JEFFREY DEAN, Rajat Monga Kai Chen Matthieu Devin Quoc V. Le Mark Z. Mao Marc'Aurelio Ranzato Andrew Senior Paul Tucker Ke Yang Andrew Y. N.

Greg S. Corrado C. Greg S. Corrado: *Large Scale Distributed Deep Networks*
. <https://static.googleusercontent.com/media/research.google.com/de//pubs/archive/40565.pdf>, . – [letzter Zugriff: 28. Nov 2017]

- [37] RUSSELL, Stuart ; DEWEY, Daniel ; TEGMARK, Max: Research Priorities for Robust and Beneficial Artificial Intelligence. (2015), jan. http://futureoflife.org/data/documents/research_priorities.pdf