Universität Leipzig

Fakultät für Mathematik und Informatik

Studiengang: Informatik, Master of Science

Seminararbeit

TensorFlow

Eine Open-Source Software-Bibliothek für maschinelles Lernen

Forschungsseminar Deep Learning

Abteilung Datenbanken

WS 2017/2018

Eingereicht von: Matthias Täschner

Matrikel-Nr.: 3723445

Betreuender Professor: Prof. Dr. Erhard Rahm

Betreuer: M.Sc. Markus Nentwig

Leipzig, den 8. Januar 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ALU	arithmetisch-logische Einheit, engl. *Arithmetic Logic Unit*
ASIC	anwendungsspezifische integrierte Schaltung, engl. *Application-Specific Integrated Circuit*
CISC	*Complex Instruction Set Computer*
CNN	gefaltete neuronale Netze, engl. *Convolutional Neural Network*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
DNN	tiefe künstliche neuronale Netze, engl. *Deep Neural Network*
GPU	*Graphics Processing Unit*
KNN	künstliche neuronale Netze
ML	maschinelles Lernen, engl. *Machine Learning*
MXU	*Matrix Multiplier Unit*
RDD	*Resilient Distributed Dataset*
RDMA	*Remote Direct Memory Access*
RISC	*Reduced Instruction Set Computer*
SGD	*Stochastic Gradient Descent*
TCP	*Transmission Control Protocol*
TPU	*Tensor Processing Unit*

1 Einleitung

Die Zunahme sehr großer Datensätze in Rechenzentren sowie die Leistungssteigerungen bei der zu deren Verarbeitung eingesetzten Hardware haben dem maschinellen Lernen, engl. *Machine Learning* (ML), und künstlichen neuronalen Netze (KNN) in den letzten Jahren eine Renaissance ermöglicht. Insbesondere tiefe künstliche neuronale Netze, engl. *Deep Neural Network* (DNN), haben hier nicht nur zu Durchbrüchen in der Sprach- und Bilderkennung beigetragen. Ein solches kam auch in der Software *AlphaGo* aus Googles Projekt *DeepMind* zur Anwendung, um damit im Jahr 2016 einen der weltbesten Profispieler im komplexen, strategischen Brettspiel Go zu schlagen.

KNN bestehen aus miteinander verbundenen künstlichen Neuronen - einer auf gewichteten Eingabedaten angewandten Aktivierungsfunktion - welche in Schichten angeordnet sind. DNN basieren auf der Aneinanderreihung sehr vieler dieser Schichten, welche zudem von unterschiedlicher Art sein können. Die zwei Phasen bei der Anwendung von KNN aller Art sind das Training des Netzes und die anschließende Vorhersage (Inferenz) durch das Netz. Das Training bestimmt dabei die ideale Gewichtung der einzelnen Eingabedaten, um eine vorliegende Aufgabe zu lösen. Bei der Entwicklung werden der Typ, der Aufbau und die Anzahl der Schichten mittels eines dafür geeigneten Modells festgelegt. Für die Erstellung dieses Modells und der zugehörigen Algorithmen existieren inzwischen verschiedene Software-Bibliotheken und Frameworks.

Thema dieser Arbeit soll das im Jahr 2015 für diesen Zweck von Google veröffentlichte TensorFlow sein. Dazu betrachtet Abschnitt 2 die Entwicklung dieser Bibliothek sowie verwandter ML-Bibliotheken. Das spezielle Datenfluss-Modell von TensorFlow wird in Abschnitt 3 analysiert. Abschnitt 4 zeigt die Art der Implementierung der einzelnen Komponenten. Neben der Forschung an KNN existiert auch die Entwicklung diesbezüglicher anwendungsspezifischer integrierter Schaltungen, engl. *Application-Specific Integrated Circuit* (ASIC), um diese anstelle der bisher zum Training und zur Inferenz genutzten *Central Processing Units* (CPUs) oder *Graphics Processing Units* (GPUs) zu nutzen. Google setzt in diesem Bereich die im Jahr 2016 vorgestellte *Tensor Processing Unit* (TPU) erfolgreich in Kombination mit TensorFlow ein. Abschnitt 5 widmet sich deren Entwicklung und Aufbau. Abschließend erläutert Abschnitt 6 die Nutzung von TensorFlow anhand von Beispielen. Eine Zusammenfassung erfolgt in Abschnitt 7.

2 Verwandte Arbeiten und Entwicklung

Um eine Einordnung von TensorFlow vornehmen zu können, soll im Folgenden ein Überblick über die Entwicklung und die Anwendung ähnlicher Bibliotheken für ML und KNN gegeben werden. Zudem wird DistBelief, der Vorgänger von TensorFlow, näher beschrieben, um damit die Entwicklung von TensorFlow nachvollziehen zu können.

2.1 Bibliotheken für maschinelles Lernen und Vorgänger

Vorgestellt werden hier zunächst zwei Bibliotheken für die allgemeine Nutzung bei ML und einfachen KNN, welche in erster Linie für statistische Analysen, Clustering und ähnliche Aufgaben entwickelt wurden.

MLC++ wurde bereits 1994 vorgestellt und ist eine der ältesten Bibliotheken für ML. Sie ist in C++ geschrieben und soll bei der Wahl und Entwicklung von Algorithmen für überwachtes Lernen unterstützen [Koh+94]. Der Fokus lag dabei anfangs auf Klassifikation-Aufgaben. Eine erweiterte Version von MLC++ wurde zusammen mit der Originalversion bis Ende 2017 von der Silicon Graphics International Corp. verwaltet.

Scikit-learn ist eine Python-Bibliothek für ML-Algorithmen des überwachten und unüberwachten Lernens [Ped+11]. Sie baut auf den Python-Modulen *NumPy*, *SciPy* und *Cython* auf und legt den Fokus auf eine imperative Programmierweise. Unterstützt werden unter anderem Algorithmen für Klassifikation, Regression, Clustering und Dimensionsreduktion.

Im Gegensatz dazu werden folgende Bibliotheken und Frameworks speziell dafür eingesetzt, um Modelle für DNN, also Netze mit zahlreichen Schichten (*Hidden Layers*) zwischen Ein- und Ausgabeschicht, zu trainieren.

Torch wurde 2002 als eine der ersten Bibliotheken für DNN vorgestellt [CBM02]. Es bündelt in einem C/CUDA-Framework [1] algebraische und numerische Optimierungsroutinen sowie Algorithmen und Modelle für KNN. Zudem verwendet es n-dimensionale Arrays als Datenobjekte. Als Skriptsprache kommt LUA zum Einsatz und die Auslagerung von Rechenoperationen auf die GPU wird unterstützt [2].

Caffe ist ein performantes C++ - Framework, um insbesondere gefaltete neuronale Netze, engl. *Convolutional Neural Network* (CNN), auf Multi-Core-CPU/GPU-Systemen zu trainieren. Es unterstützt allerdings keine verteilten Systeme. Parallelisierbare Rechenoperationen werden mittels CUDA auf die GPU ausgelagert. Caffe ermöglicht es, ML-Modelle sehr leicht aus vorgefertigten Schichten zu erstellen und trennt deren Repräsentation als gerichteter azyklischer Graph von der eigentlichen Ausführung. Genutzt wird Caffe hauptsächlich im Multimedia-Bereich, hier speziell für die Bilderkennung [Jia+14].

Theano erlaubt dem Nutzer, mathematische Ausdrücke über Symbole zu definieren und diese als Datenfluss-Graph mit Variablen und Operationen zu speichern [Al-+16]. Dieser wird vor der Ausführung auf verschiedene Arten optimiert. Das genutzte Interface ist in Python geschrieben und orientiert sich am Python-Modul *NumPy*. Theano ist kein Framework für das Training von KNN sondern wurde eher für die effektive Berechnung mathematischer Ausdrücke entwickelt. Trotzdem ist es TensorFlow sehr ähnlich, insbesondere hinsichtlich der Flexibilität durch den verwendeten Datenfluss-Graphen.

Neon ist ein von Nervana Systems, einer Tochterfirma von Intel, entwickeltes Framework für DNN [3]. Der Nutzer wird durch vorgefertigte Modelle, Schichten und Funktionen für eine Vielzahl von KNN unterstützt. Deren Schichten nutzen zweidimensionale Tensoren als internes Datenformat. Das Python-Interface ist für Linux und MacOS verfügbar. Mittels CUDA können Berechnungen auf die GPU ausgelagert werden.

SparkNet ist ein Framework für das Training von DNN auf Cluster-Systemen [Mor+15]. Spark als Framework für die Stapelverarbeitung auf solchen Clustern ist für die stark asynchrone und kommunikationsintensive Arbeitsweise von DNN wenig geeignet. SparkNet bündelt daher Interfaces für die Arbeit mit Sparks *Resilient Distributed Datasets* (RDD), für das Einbinden des Caffe-Frameworks und die Nutzung multidimensionaler Tensoren. Ein bleibender Nachteil der Stapelverarbeitung ist die Anforderung, dass Eingabedaten für Berechnungen unveränderlich bleiben und die Berechnungen selbst deterministisch sein müssen, um diese im Falle eines Cluster-Ausfalls wiederholen zu können. Dies erschwert insbesondere die Aktualisierung der Parameter eines ML-Modells.

DistBelief wurde im Rahmen des Google Brain Projekts im Jahr 2011 entwickelt, um die Nutzung von hochskalierbaren DNN zu erforschen und stellt den Vorgänger von TensorFlow dar [Dea+12]. Zum Einsatz kam es bei vielen Arbeiten des unüberwachten Lernens, der Bild- und Spracherkennung und unter anderem auch bei der Evaluation von Spielzügen

im Brettspiel Go. DistBelief hatte trotz der erfolgreichen Nutzung einige Einschränkungen. Die Definition der Schichten eines KNN geschieht im Gegensatz zum genutzten Python-Interface aus Gründen der Effizienz mit C++. Die Anpassung der Gradientenfunktion zur Minimierung des Fehlers erfordert eine Anpassung der Implementierung des integrierten Parameter-Servers. Algorithmen können konstruktionsbedingt lediglich für vorwärtsgerichtete KNN entwickelt werden - das Training von Modellen für rekurrente KNN oder verstärkendes Lernen ist nicht möglich. Zudem wurde DistBelief für die Anwendung auf großen Clustern von Multi-Core-CPU-Servern entwickelt und unterstützte den Betrieb auf verteilten GPU-Systemen nicht. Ein 'herunterskalieren' auf andere Umgebungen erweist sich daher als schwierig [Aba+16].

2.2 Entwicklung von TensorFlow

Die mit DistBelief gesammelten Erfahrungen und das fortgeschrittene Verständnis von KNN flossen in das Nachfolgesystem TensorFlow ein. DistBeliefs Einschränkungen wurden bei dessen Entwicklung ebenfalls adressiert. Im Jahr 2015 als Open-Source veröffentlicht, bietet TensorFlow eine plattformübergreifende Bibliothek für die Entwicklung und Ausführung von Modellen für hochskalierbares ML und insbesondere DNN [Aba+15]. ML-Algorithmen werden in TensorFlow über ein zustandsorientiertes Datenfluss-Modell beschrieben und können auf einer Vielzahl von Systemen abgebildet werden - von mobilen Plattformen wie Android oder iOS über einfache Single-CPU/GPU-Rechner bis hin zu großen, verteilten Systemen. Im Vergleich zu DistBelief arbeitet TensorFlow flexibler, signifikant performanter und unterstützt die Entwicklung und das Training von mehr ML-Algorithmen auf heterogener Hardware.

3 Datenfluss-Modell

TensorFlow nutzt einen Datenfluss-Graphen, um sowohl die Berechnungen als auch die Zustände in einem ML-Algorithmus zu repräsentieren. Dies umfasst auch dessen mathematische Operationen, die verwendeten Parameter und deren Aktualisierungen. Durch dieses Konzept ermöglicht TensorFlow mehrere nebenläufige Ausführungen von sich überlappenden Subgraphen. Veränderliche Zustände von Knoten können über verschiedene Ausführungen eines Graphen hinweg geteilt werden.

3.1 Berechnungsgraph

Der genutzte Berechnungsgraph ist ein gerichteter Graph, dessen Knoten Operationen darstellen und dessen Kanten den Datenfluss zwischen diesen Knoten repräsentieren. Die Berechnung von zum Beispiel $y = W * x + b$ mit $W = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $x = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$ und $b = \begin{bmatrix} 7 \\ 8 \end{bmatrix}$ lässt sich als Graph wie in Abbildung 1 beschreiben.

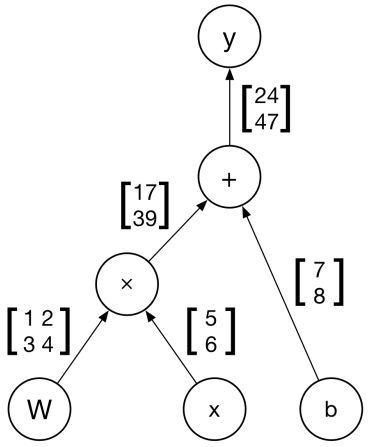


Abbildung 1: Berechnungsgraph für $y = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 7 \\ 8 \end{bmatrix}$

3.1.1 Operationen als Knoten

Die Operationen drücken eine Kombination oder Transformation von durch den Graphen fließende Daten aus. Dabei kann eine Operation keine oder mehrere Eingaben haben und keine oder mehrere Ergebnisse liefern. Daher kann ein Knoten im Graph zum Beispiel eine

mathematische Gleichung, eine Variable, eine Konstante oder auch eine Dateioperation sein. Eine bestimmte Implementierung einer Operation, entwickelt für die Ausführung auf einer bestimmten Recheneinheit (CPU, GPU, ...) wird als *Kernel* bezeichnet. Tabelle 1 zeigt eine Auswahl an möglichen Operationen.

Tabelle 1: Typen von Operationen in TensorFlow, in Anlehnung an [Aba+15]

Kategorie	**Beispiel**
Mathematische Operationen	Add, Sub, Mul, Div, Exp, ...
Operationen auf Arrays	Concat, Split, Shuffle, ...
Matrix-Operationen	MatMul, MatrixInverse, ...
Zustandsorientierte Operationen	Variable, Assign, ...
Bausteine für neuronale Netze	SoftMax, Sigmoid, ReLU, ...
CheckPoint-Operationen	Save, Restore, ...
Operationen zur Flusssteuerung	Merge, Switch, NextIteration, ...

3.1.2 Tensoren als Kanten

Kanten repräsentieren den Datenfluss zwischen den einzelnen Operationen und werden als Tensoren bezeichnet. Ein Tensor ist ein mathematisches Objekt der linearen Algebra und ist in TensorFlow gleichbedeutend mit einem multidimensionalen Array aus homogenen Werten statischen Typs. Die Anzahl der Dimensionen beschreibt dabei den Rang (*Rank*) des Tensors. Der Begriff *Shape* bezeichnet ein Tupel aus der maximalen Anzahl von Elementen je Dimension. Tabelle 2 zeigt einige Beispiele von Tensoren verschiedenen Ranges.

Tabelle 2: Beispiele für Tensoren (*Rank, Shape*), in Anlehnung an [Vö17a]

Tensor	**Rank**	**Shape**
3	Rank 0	Shape[]
[1.0, 2.0, 3.0]	Rank 1	Shape[3]
[[1.0, 2.0, 3.0],[1.0, 2.0, 3.0]]	Rank 2	Shape[2,3]
[[[1.0, 2.0, 3.0]],[[1.0, 2.0, 3.0]]]	Rank 3	Shape[2,1,3]

In TensorFlow werden Tensoren als Generalisierung für mehrdimensionale Matrizen, Vektoren und Skalare genutzt - ein Skalar ist hier ein Tensor des Ranges 0. Im Berechnungsgraph fungieren Tensoren als Identifikatoren für die Ausgaben von Operationen und bilden Interfaces für den Zugriff auf die damit referenzierten Werte im Speicher. Für die

Tensoren im oben aufgeführten Beispiel gilt: W vom Rang 2 mit Shape[2,2], x, b und y jeweils vom Rang 2 mit Shape[2,1].

3.2 Session

Eine *Session* dient als Schnittstelle zwischen dem genutzten Client-Interface und dem TensorFlow-System und bietet unter anderem zwei grundsätzlichen Methoden. Über die Methode `Extend` kann der initial leere Berechnungsgraph um Knoten und Kanten erweitert werden. Die Methode `Run` führt den Berechnungsgraphen entsprechend der übergebenen Argumente aus, wobei ein gewünschter Graph selbst auch ein Argument sein kann. Bei Beginn der Ausführung betrachtet TensorFlow die Ausgabe-Knoten und durchläuft den Graphen rückwärts, um unter Beachtung aller Abhängigkeiten der Knoten untereinander die Menge aller auszuführenden Knoten zu berechnen. Diese werden abschließend über einen Platzierungs-Algorithmus (siehe Abschnitt 4.1.1) auf die verfügbaren Recheneinheiten verteilt.

3.3 Variablen - zustandsorientierte Operatoren

Eine Variable, als spezielle Art der Operation, schafft eine über mehrere Berechnungsdurchläufe des Graphen hinweg persistente Referenz zu einem veränderbaren Tensor. Die zugehörigen Operationen `Assign` oder `AssignAdd` können diese Tensoren verändern. Ein Beispiel für die Anwendung von Variablen sind Gewichtungen von Eingabedaten eines ML-Modells, welche bei jedem Durchlauf des Trainings angepasst werden.

4 Implementierung

Die Anbindung der in C++ programmierten Kernkomponenten an die verschiedenen Client-Sprachen erfolgt über eine C-API - priorisiert werden derzeit Python und C++ (siehe Abbildung 2). Die Kernkomponenten sollen in diesem Abschnitt näher betrachtet werden, der Python-Client wird für die Erläuterungen an Anwendungsbeispielen in Abschnitt 6 genutzt.

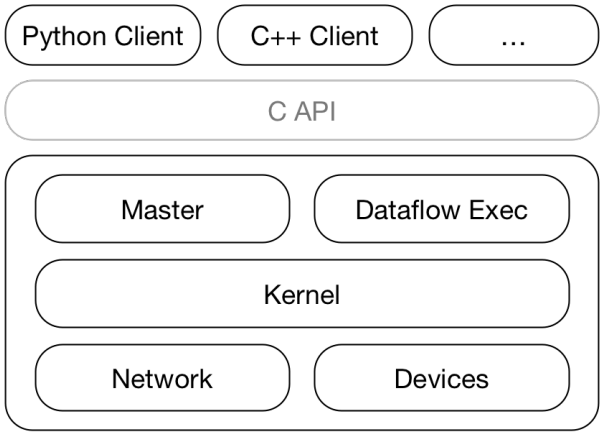


Abbildung 2: TensorFlows geschichtete Implementierung, in Anlehnung an [Aba+16]

Generell kommunizieren die Kerkomponenten *Client*, *Master* und *Worker* über das *Session*-Interface miteinander. Jedem *Worker* obliegen dabei die Zugriffsverhandlungen für eine oder mehrere Recheneinheiten eines Systems (CPU, GPU, ...) auf denen die vom *Master* übergebenen Subgraphen des Berechnungsgraphen ausgeführt werden.

Laufen alle Komponenten innerhalb des Kontextes eines einzigen Betriebssystems auf einem System, findet die lokale Implementierung Anwendung. Im einfachsten Fall eines Systems mit einer einzigen verfügbaren Recheneinheit werden die Knoten des Graphen, entsprechend ihrer Abhängigkeiten untereinander, nacheinander auf dieser Recheneinheit ausgeführt.

Die Implementierung für verteilte Systeme erweitert die lokale Implementierung dahingehend, dass *Client*, *Master* und *Worker* verschiedene Prozesse auf verschiedenen Systemen sein können. Abbildung 3 veranschaulicht die beiden Prinzipien.

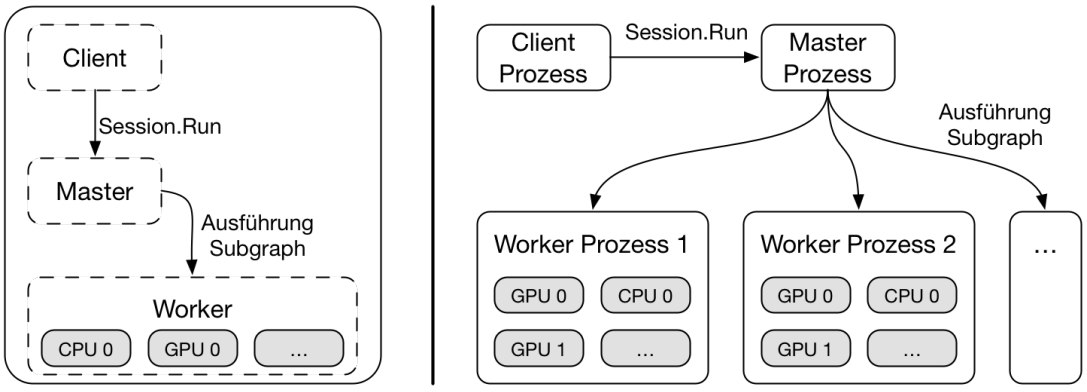


Abbildung 3: Struktur für die Implementierung auf einer lokalen Maschine (links) und für ein verteiltes System (rechts), in Anlehnung an [Aba+15]

4.1 Nutzung mehrerer Recheneinheiten

Steht in einem System mehr als eine Recheneinheit zur Verfügung, sind zwei Probleme zu lösen. Die einzelnen Knoten des Berechnungsgraphen müssen auf die Recheneinheiten verteilt werden und es gilt, die Kommunikation für den Datenaustausch zwischen den genutzten Recheneinheiten zu regeln. TensorFlow stellt hierfür die im Folgenden aufgeführten Lösungen bereit [Aba+15].

4.1.1 Knoten-Platzierung

Der für die Knoten-Platzierung genutzte Algorithmus baut auf einem Kostenmodell auf. Dieses zieht die geschätzte Größe der Eingabe- und Ergebnis-Tensoren eines Knotens und die geschätzte Berechnungszeit der Operation in Betracht. Hierbei ist ein Zugriff auf gemessene Größen früherer Berechnungsdurchläufe möglich. Vor der eigentlichen Ausführung des Berechnungsgraphen erfolgt deren Simulation. Die Knoten werden dabei mittels einer *Greedy*-Heuristik auf diejenigen Recheneinheiten verteilt, welche einen entsprechenden Kernel für die Operation implementiert haben und die bestmögliche Berechnungszeit bieten würden. Die eigentliche Ausführung entspricht dem Ergebnis der Simulation. Der Berechnungsgraph wird in Subgraphen für die einzelnen Recheneinheiten aufgeteilt.

4.1.2 Kommunikation

Findet eine Recheneinheit-übergreifende Kommunikation statt, werden in den beteiligten Subgraphen neue Knoten für die Regelung der Datenübertragung eingefügt - *Send* und *Receive*. Dabei wird der Datenaustausch mit mehr als zwei beteiligten Knoten möglichst zusammengefasst, um Tensoren nicht redundant zu übertragen. Über die neu eingefügten Knoten kann der Datenaustausch zudem dezentralisiert von den für die jeweilige Recheneinheit verantwortlichen Worker-Prozess geregelt und synchronisiert werden. Abbildung 4 zeigt die Schritte dieser Umstrukturierung. Für den lokalen Datentransfer kommen dabei NVIDIAs CUDA Runtime API (CPU zu GPU) oder *Direct Memory Access* (DMA) (GPU zu GPU) zum Einsatz [Aba+16].

Die Kommunikation bei der Implementierung für verteilte Systeme ähnelt dem Ansatz für die Nutzung mehrerer Recheneinheiten. Für den Worker-übergreifenden Datenaustausch werden ebenfalls *Send*- und *Receive*-Knoten eingesetzt, welche über *Transmission Control Protocol* (TCP) oder *Remote Direct Memory Access* (RDMA) kommunizieren [Aba+15].

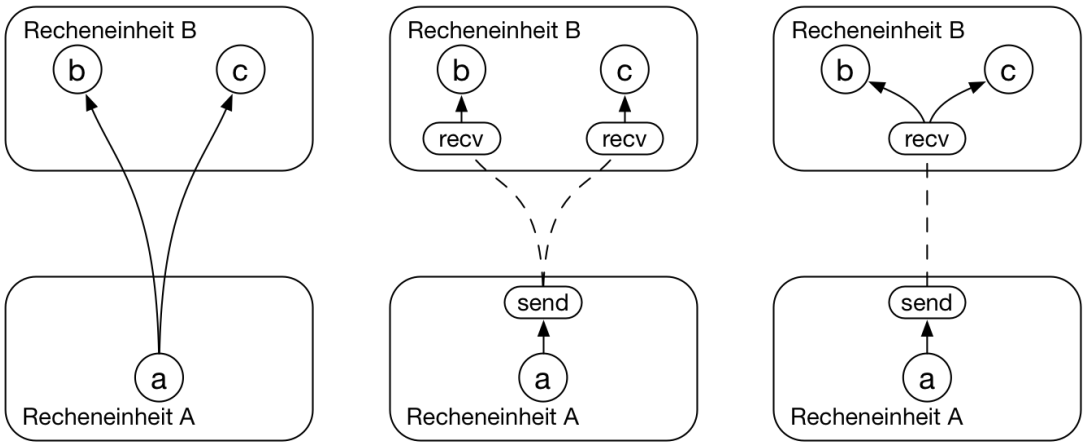


Abbildung 4: Die Umstrukturierung der Kommunikation zwischen mehreren Recheneinheiten, in Anlehnung an [Aba+15]

4.2 Fehlertoleranz

Das Auftreten eines Fehlers während der Ausführung des Berechnungsgraphen führt zum Abbruch und Neustart der gerade laufenden Graph-Berechnung. Fehler können dabei zum Beispiel in der Worker-übergreifenden Kommunikation entstehen oder bei den regelmäßigen Health-Checks zwischen Master und Worker entdeckt werden. Allerdings bleiben dabei die Ergebnis-Tensoren von gewünschten Knoten, zum Beispiel von Variablen im Graphen, bestehen. Über das Setzen konsistenter Checkpoints und der Möglichkeit deren Wiederherstellung bei einem Neustart sind die damit referenzierten Daten über den abgebrochenen Berechnungsdurchlauf hinweg verfügbar. Dafür werden die Knoten mit einem Speicher- und Wiederherstellungsknoten verknüpft. Ersterer schreibt die aktuellen Ergebnis-Tensoren in den persistenten Speicher - zum Beispiel ein verteiltes Dateisystem. Der zweite liest diese bei Neustart der Graph-Berechnung erneut ein und weist diese dann zum Beispiel mit `Assign` der entsprechenden Variable im Graphen zu.

4.3 Erweiterungen

Zusätzlich zur ausgeführten grundlegenden Implementierung verfügt TensorFlow über weitere Funktionalitäten. Einige davon werden nachfolgend erläutert [Aba+15].

4.3.1 Automatisierte Gradientenberechnung

Gängige ML-Algorithmen nutzen Gradientenverfahren (zum Beispiel *Stochastic Gradient Descent* (SGD)), um eine Kostenfunktion für einen Ergebnis-Tensor in Bezug auf die Ein-

gabedaten des Algorithmus zu berechnen und durch deren Anpassung das Minimum dieser Kostenfunktion zu ermitteln. TensorFlow bietet hierfür eine automatisierte Gradientenberechnung via *Back Propagation*. Diese erweitert den Berechnungsgraph, indem dieser rückwärts durchlaufen und zu jeder an der Gradientenberechnung beteiligten Operation ein Knoten eingefügt wird. Dieser berechnet den zur jeweiligen Operation zugehörigen partiellen Gradienten.

Abbildung 5 veranschaulicht die Gradientenberechnung für den Ergebnis-Tensor C in Abhängigkeit der Eingabe-Tensoren b, W und x. Der Graph wird von C ausgehend rückwärts durchlaufen und für jede im Pfad zu b, W und x liegende Operation wird ein Knoten für deren partiellen Gradienten erstellt. Die roten Pfeile zeigen die möglichen Eingabewerte für die partiellen Gradientenfunktionen. Für die Gradientenberechnung und deren Ausgabe im Python-Client genügt folgender Befehl:
`[db,dW,dx] = tf.gradients(C, [b,W,x])`.

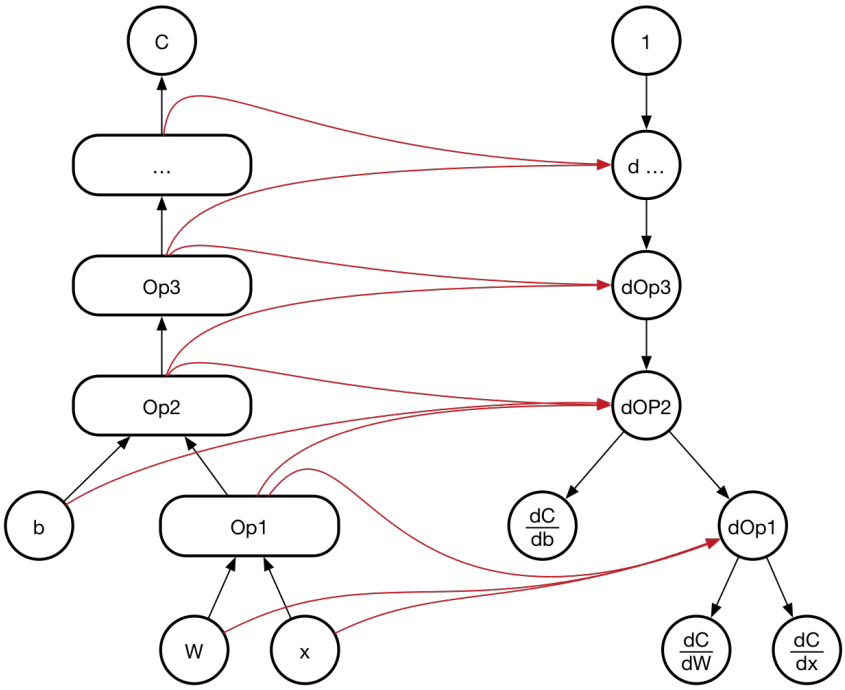


Abbildung 5: Die automatisierte Berechnung der Gradienten, nach einer Darstellung von [Aba+15]

4.3.2 Teilweise Ausführung des Graphen

TensorFlow erlaubt es, einen beliebigen Subgraphen des Berechnungsgraphen auszuführen, diesem dabei beliebige Eingabedaten zu präsentieren und die Ergebnis-Tensoren einer beliebigen Kante auszulesen. Dafür werden die Methoden `Session.partial_run_setup` und `Session.partial_run` zur Verfügung gestellt.

4.3.3 Bedingungen für die Auswahl von Recheneinheiten

Die Art der Verteilung der Knoten auf die einzelnen Recheneinheiten im Rahmen der Knotenplatzierung ist über Vorgaben beeinflussbar. So kann zum Beispiel die Einschränkung getroffen werden, Knoten eines bestimmten Typs nur auf einer GPU zu platzieren oder einen Knoten einer beliebigen Recheneinheit zuzuweisen, solange diese von einem bestimmten Worker-Prozess verwaltet wird.

4.3.4 Flusssteuerung

TensorFlow beinhaltet eine Anzahl von Operatoren zur Flusssteuerung. Damit lassen sich zyklische Datenfluss-Graphen erstellen, die ML-Algorithmen präziser und effizienter repräsentieren können. Mit den in Tabelle 3 erläuterten Operationen können bedingte Verzweigungen und Schleifen kompiliert werden.

Tabelle 3: Operationen zur Flusssteuerung in TensorFlow; übersetzt aus dem Englischen, nach einer Darstellung von [Aba+15]

Name	**Anwendung**
Switch	Demultiplexer, für bedingte Verzweigung erzeugt aus Steuerungswert und Tensor einen Tensor und einen „toten“ Wert
Merge	Multiplexer, für bedingte Verzweigung erzeugt aus einem Tensor und einem „toten“ Wert einen Tensor
Enter	Beginn einer Schleife
Leave	Abbruch einer Schleife
NextIteration	Schleifenzähler

Für den Einsatz auf verteilten Systemen nutzt TensorFlow einen Mechanismus zur Koordination der Flusssteuerung. Sind zum Beispiel zu einer Schleife gehörige Knoten des Graphen auf unterschiedliche Recheneinheiten verteilt, werden bei der Partitionierung des Graphen zur Knotenplatzierung ebenfalls Steuerungsknoten eingefügt. Diese organisieren den Beginn und Abbruch einer Iteration beziehungsweise der gesamten Schleife.

4.4 Softwareseitige Optimierungen

Für eine verbesserte Performance und Speichernutzung wurden in TensorFlow folgende Optimierungen implementiert [Aba+15].

4.4.1 Vereinheitlichung gleicher Subgraphen

Insbesondere bei großen Graphen für DNN mit vielen Schichten und bei der Nutzung von Abstraktionen können redundante Operationen auftreten. Redundant heißt in diesem Zusammenhang, dass gleiche Operationen mit den gleichen Eingangsdaten mehrfach im Graph verwendet werden. Diese zu vereinheitlichen, optimiert sowohl die Performance als auch die Speichernutzung, da die betreffende Operation nur noch einmal auszuführen ist und Redundanzen aus dem Graph entfernt werden können. Durch die Umstrukturierung der Kanten steht der Ergebnis-Tensor allen beteiligten Operationen zur Verfügung. Dieses Prinzip kommt ebenfalls bei Compilern zum Einsatz und ist in Abbildung 6 veranschaulicht. TensorFlow nutzt hierfür eine Variante von *Global Code Motion* [Cli95].

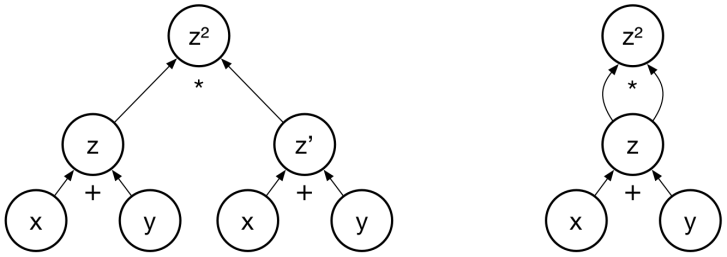


Abbildung 6: Die Vereinheitlichung gleicher Subgraphen mit gleichen Eingabedaten, in Anlehnung an [Gol16]

4.4.2 Steuerung von Datentransfer und Speichernutzung

Ein weiterer Schritt zur Optimierung ist die Reduzierung der Dauer, in der Zwischenergebnisse einer Berechnung im Speicher gehalten werden müssen. Dies ist insbesondere bei der Nutzung von GPUs bedeutend, da deren Speicher beschränkt ist. Zudem soll die gezielte Steuerung der Recheneinheit-übergreifenden Kommunikation dabei helfen, die Auslastung von Netzwerk-Ressourcen zu minimieren. Um dies umzusetzen steuert TensorFlow den Zeitpunkt, an dem *Receive*-Knoten die für sie bestimmten Werte lesen. Eine *as-soon-as-possible*- und *as-late-as-possible*-Analyse der kritischen Pfade im Graph berechnet den idealen Zeitpunkt für den Datenempfang.

4.4.3 Optimierte Kernel

Für die Implementierung der Kernel werden, wo immer möglich, bereits existierende, hochoptimierte numerische Bibliotheken genutzt. Dies betrifft zum Beispiel Matrix-Operationen oder Bibliotheken für die Nutzung der GPU als Recheneinheit. Bei vielen Kernel wird die Eigen-Bibliothek für Operationen der linearen Algebra verwendet.

5 Optimierte Hardware - Tensor Processing Unit

Neben den softwareseitigen Optimierungen kann insbesondere der Einsatz spezialisierter Hardware das Training und den Einsatz von KNN beschleunigen. Im Folgenden wird die von Google entwickelte TPU näher betrachtet.

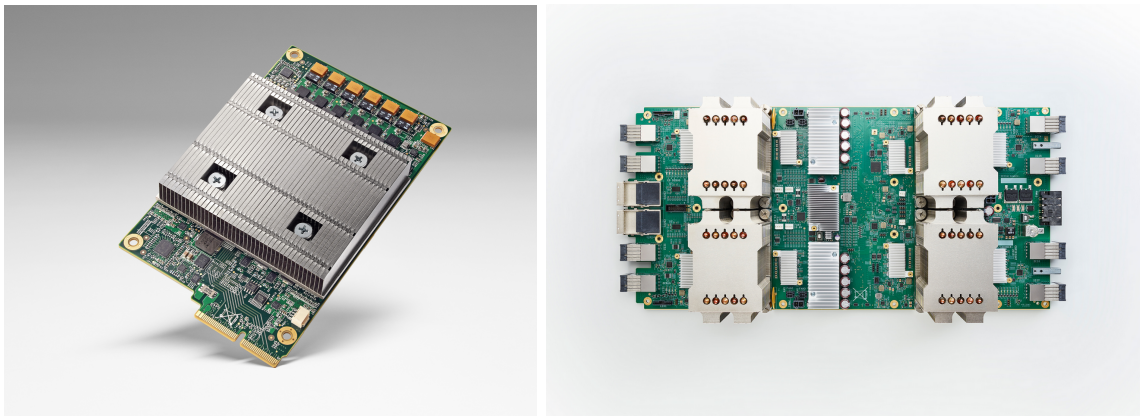
5.1 Hintergrund und Entwicklung

Die gestiegene Rechenlast durch die Fortschritte im ML veranlasste Google bereits im Jahr 2006 dazu, an ASIC zu forschen. 2013 wurden diese Bemühungen intensiviert, da die - auch intern - wachsende Nachfrage nach Rechenkapazität im Bereich ML voraussichtlich eine Verdoppelung der benötigten Kapazität an Rechencentern nach sich gezogen hätte [4]. Im Jahr 2016 gab Google die erfolgreiche Entwicklung und den Einsatz der TPU bekannt [5], einen ASIC speziell für Anwendungen des ML und maßgeschneidert für TensorFlows Nutzung von Tensoren als Datenformat. Das Design einer TPU orientiert sich dabei an den grundlegenden mathematischen Operationen innerhalb eines KNN. Diese sind für jedes Neuron:

- die Matrix-Multiplikation aller Eingabedaten mit deren Gewichtungsmatrix,
- das Addieren all dieser Ergebnisse und
- die Anwendung einer Aktivierungsfunktion darauf.

Die Matrix-Multiplikation ist eine der rechenintensivsten Operationen. Die TPU wurde entwickelt, um die Ausführung dieser Operation zu optimieren und zu beschleunigen.

Die erste Version davon - dahingehend konzipiert, mit einem bereits trainierten ML-Modell effizient zu inferieren - wurde als externe Beschleuniger-Karte in bestehenden Servern genutzt (siehe Abbildung 7a). Das Training selbst erforderte noch immer performante CPU- und GPU-Cluster. Auf der aktuellen und weiterentwickelten Version lassen sich ML-Modelle sowohl effizient ausführen als auch trainieren (siehe Abbildung 7b). Zudem verfügt sie über eigene Hochgeschwindigkeits-Netzwerkadapter. 64 Stück dieser weiterentwickelten Version lassen sich zu einem Pod zusammenschließen [6].



(a) Tensor Processing Unit - Version 1 (b) Tensor Processing Unit - Version 2

Abbildung 7: Versionen der Tensor Processing Unit, Quelle: Google, `https://cloudplatform.googleblog.com`

5.2 Aufbau und Performance

Im Folgenden soll der Aufbau und die Funktionsweise der ersten Version der TPU erläutert werden. Technische Details zur zweiten Version waren zum Zeitpunkt der Erstellung dieser Arbeit noch nicht verfügbar.

Dem Blockschaltplan in Abbildung 8 zeigt den Aufbau der TPU. Als Kernkomponente kommt eine *Matrix Multiplier Unit* (MXU) mit 256x256 arithmetisch-logischen Einheiten, engl. *Arithmetic Logic Units* (ALUs) für die Multiplikation von 8-Bit Integer zum Einsatz. Hierfür werden die bei den Berechnungen verwendeten Gleitkommazahlen in 256 Stufen quantisiert. Dies spart im Vergleich zu Operationen mit 32- oder 16-Bit Gleitkommazahlen sowohl Speicherressourcen als auch Rechenleistung. Die verminderte Genauigkeit ist für das Inferieren durch das trainierte KNN trotzdem ausreichend. Die MXU ist zudem als systolisches Array konzipiert. Im Vergleich zu den in CPUs anzutreffenden Zyklen aus „Lesen aus dem Register“, „Verarbeiten in der ALU“ und „Schreiben in das Register“ lassen sich mittels der MXU einmal aus dem Register gelesene Werte durch mehrere ALUs hintereinander leiten, ohne Zwischenergebnisse in das Register schreiben zu müssen. Die multiplizierten Werte werden nach dem Addieren in den Akkumulatoren an die gewünschte festverdrahtete Aktivierungsfunktion übertragen.

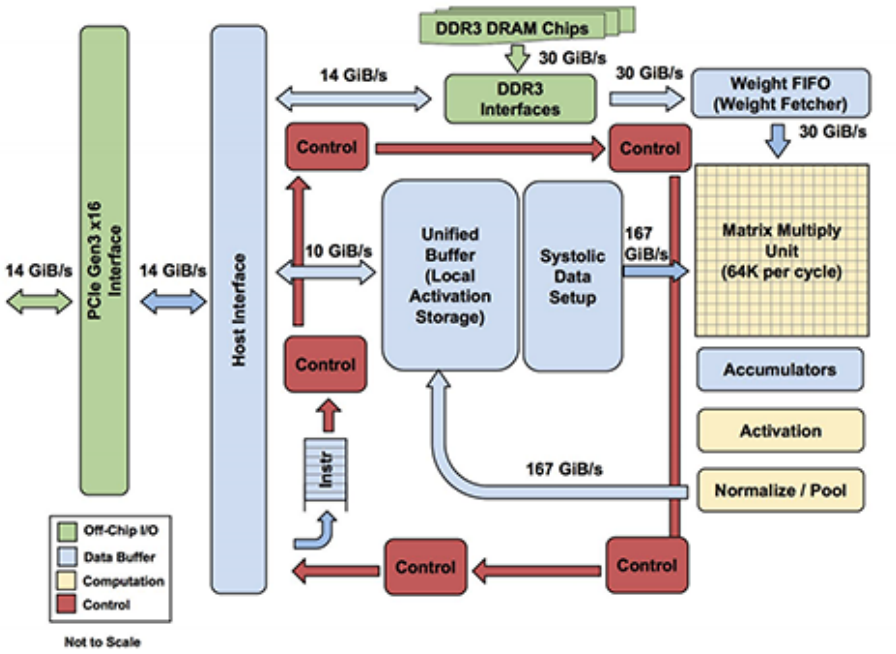


Abbildung 8: Der Blockschaltplan einer Tensor Processing Unit, Quelle: [4]

Durch die Programmierbarkeit der TPU ist diese für mehrere Arten von neuronalen Netzen geeignet. Im Gegensatz zu dem bei CPUs genutzten *Reduced Instruction Set Computer* (RISC) - Ansatz wird hier ein *Complex Instruction Set Computer* (CISC) - Ansatz verwendet. Dieser High-Level Befehlssatz mit seinen im Vergleich abstrakteren Anweisungen ermöglicht die Bearbeitung komplexer Aufgaben. Dadurch fokussiert sich die TPU auf die oben genannten, für KNN typischen mathematischen Operationen. Für den Einsatz von TensorFlow mit der TPU wurde von Google ein Compiler entwickelt, der API-Anfragen von TensorFlow direkt in die entsprechenden Anweisungen übersetzt.

Der Vorteil liegt somit in der Verarbeitung von hunderttausenden Matrix-Operationen in einem einzigen Takt. Im Vergleich dazu verarbeiten CPUs und GPUs lediglich Skalarwerte und Vektoren. Dies macht die TPU im Anwendungsbereich der KNN durchschnittlich 15x bis 30x schneller als aktuell verwendete GPUs (Nvidia K80) oder CPUs (Server-CPU Intel Haswell mit 18 Kernen)[Jou+17].

6 Anwendung von TensorFlow

Eine umfangreiche Beschreibung der Installation und der Anwendung von TensorFlow bietet Google über eine eigene Website [7]. Hier stehen unter anderem Installationsdateien für Windows, Mac OS und Ubuntu zur Verfügung. Zudem werden neben der standardmäßigen Python API noch Bibliotheken für die Nutzung von TensorFlow in den Programmiersprachen Java, C++ und Go angeboten. Diese sind zum Zeitpunkt der Erstellung dieser Seminararbeit allerdings noch nicht so umfangreich wie die Python API. Deren Anwendung soll hier zunächst bei einer arithmetischen Berechnung erläutert werden. Nachfolgend wird die Nutzung eines CNN anhand eines gängigen Beispiels der Handschrifterkennung gezeigt.

6.1 Installation und Nutzung der Python API

Es wird empfohlen, TensorFlow in einer virtuellen Python-Umgebung zu verwenden, um Beeinflussungen durch das beziehungsweise mit dem restlichen System zu vermeiden. Über Pythons Paketverwaltungsprogramm `pip` werden darin TensorFlow und alle benötigten Abhängigkeiten installiert: `pip install --upgrade tensorflow`.

Der grundlegende Aufbau und die Ausführung eines Graphen mit der Python API sollen für die Berechnung $y = W * x + b$ mit $W = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $x = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$ und $b = \begin{bmatrix} 7 \\ 8 \end{bmatrix}$ aus Abschnitt 3.1 gezeigt werden. Der vollständige Programmtext dazu ist Listing 1 zu entnehmen. Um TensorFlow in Python nutzen zu können, muss die Bibliothek in das Programm eingebunden werden:

```
import tensorflow as tf
```

Nun werden die Knoten des Berechnungsgraphen erstellt, beginnend mit einer Variablen für W und einen Platzhalter für x. Die Angabe des Datentyps und eines Namens ist optional. Ein Platzhalter bleibt während der Ausführung des Berechnungsgraphen unverändert und wird zu Beginn der Berechnung von Python an TensorFlow übergeben. Die Variable kann im Gegensatz dazu auch während der Verarbeitung geändert werden. Über den mathematischen Operator `tf.matmul` wird die Matrix-Multipliaktion durchgeführt:

```
node_W = tf.Variable([[1, 2], [3, 4]], tf.int32, name='W')
node_x = tf.placeholder(tf.int32, name='x')
node_matmul = tf.matmul(node_W, node_x)
```

Als nächstes wird eine Konstante für b erzeugt und mit dem vorliegenden Tensor des Zwischenergebnisses addiert. Neben den mathematischen Operatoren von TensorFlow können auch die Operatoren von Python auf die Knoten des Graphen angewandt werden:

```
node_b = tf.constant([[7], [8]], tf.int32, name='b')
node_y = node_matmul + node_b
```

Um den Berechnungsgraphen ausführen zu können muss ein `tf.Session`-Objekt angelegt werden. Innerhalb dieser Session werden als erster Schritt alle erzeugten Variablen initialisiert und damit mit ihrem Anfangswert belegt. Die Platzhalter beziehen ihre Werte über ein Dictionary, welches zusammen mit dem erstellten Graph an die Methode `tf.Session().run()` übergeben wird. Das Ergebnis ist dann die korrekte Matrix $\begin{bmatrix}24\\47\end{bmatrix}$:

```
with tf.Session() as session:
    tf.global_variables_initializer().run()
    print(session.run(node_y, {node_x: [[5], [6]]}))
```

6.2 Anwendung am Beispiel der Handschrifterkennung

Ein umfangreicheres Beispiel für die Anwendung bei der Erkennung von handgeschriebenen Zahlen mittels CNN soll nun weiterführende Möglichkeiten von TensorFlow aufzeigen. Bezug wird hierbei auf ein Beispiel aus [Vö17b] und Tutorials von TensorFlow genommen [8],[9]. die Erläuterungen der theoretischen Grundlagen zu CNN basieren auf [DV16]. Der vollständige Programmtext dazu ist Listing 2 zu entnehmen.

CNN eignen sich besonderes für das Verarbeiten von Bildern. Bei der Operation *Convolutional* läuft der sogenannte Filterkernel, eine Matrix von festgelegter Größe in Bildpunkten, über ein Eingabebild und erzeugt durch das Zusammenfassen der Bildinformationen eine komprimierte Ergebnismatrix. Hierbei werden die Elemente des Filterkernel nacheinander mit den jeweils abgedeckten Elementen des Eingabebildes multipliziert, diese Werte werden anschließend addiert und ergeben ein Element der Ergebnismatrix. Der Filterkernel wandert danach weiter, die Schrittweite beeinflusst, wie stark das Eingabebild zusammengefasst wird. Dieses Prinzip wird in Abbildung 9 veranschaulicht. In der Praxis

laufen mehrere Filterkernel über ein Eingabebild, um unterschiedliche Informationen wie zum Beispiel Farb- und Helligkeitswerte zu erfassen.

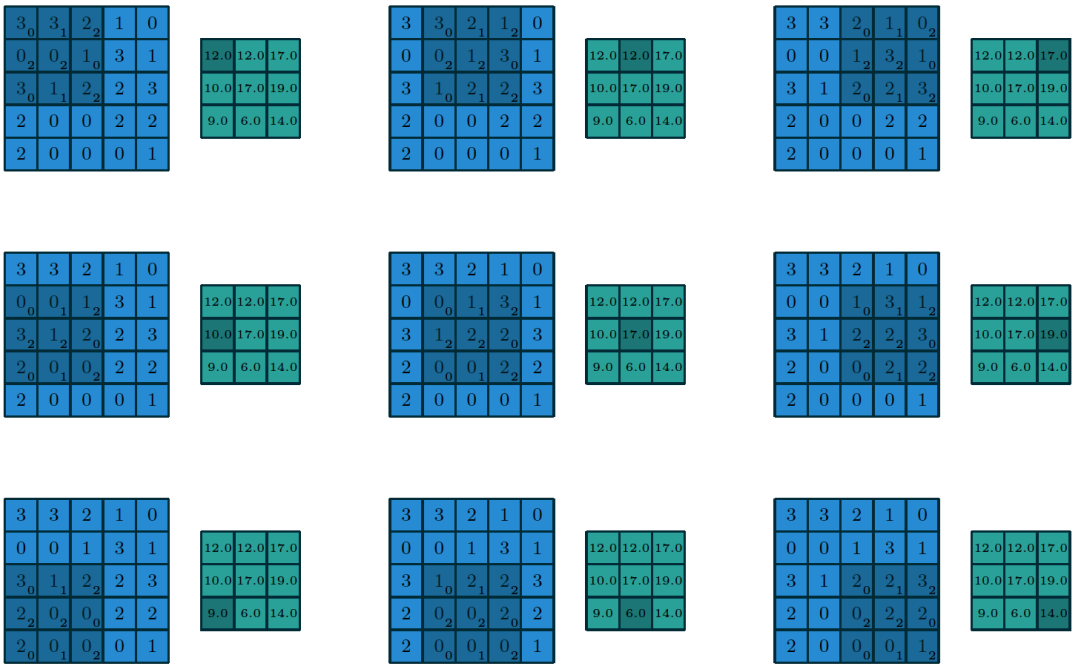


Abbildung 9: Ablauf der Operation *Convolutional* bei CNN mit einem Filterkernel von $\begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$, aus dem Eingabebild (blau) wird die Ergebnismatrix (grün) erzeugt, Quelle: [DV16]

Die benötigte große Anzahl an Eingabebildern von handgeschriebenen Zahlen ist über die *Modified National Institute of Standards and Technology database* (MNIST) verfügbar. TensorFlow bietet hierfür eine integrierte Möglichkeit auf diese Daten zuzugreifen:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data/', one_hot=True)
```

Ein Bild besteht dabei aus 28x28 Bildpunkten in Graustufen. Jeder Bildpunkt lässt sich damit durch eine Zahl zwischen 0 und 1 beschreiben, die 784 Bildpunkte jedes Bildes werden in einem eindimensionalen Array gespeichert. Eine dargestellte Zahl wird über ein zehnstelliges Array beschrieben, dessen der Zahl entsprechende Position im Array auf 1 gesetzt wird. Die Zahl 4 entspricht somit `[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]`.

Für die Gewichtungen (Weight) der Eingabedaten und die Abweichung (Bias) wird eine jeweilige Methode erstellt. Die Variablen für die Gewichtung werden mit normalverteilten Werten initialisiert:

```
def weight_variable(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.1))

def bias_variable(shape):
    return tf.Variable(tf.constant(0.1, shape=shape))
```

Die als eindimensionales Array und in unbekannter Anzahl vorhandenen Eingabebilder werden über einen Platzhalter an den Berechnungsgraphen übergeben und dann in eine zweidimensionale Matrix mit 28x28 Bildpunkten und einem Farbkanal umgewandelt:

```
with tf.name_scope('model'):
    x = tf.placeholder(tf.float32, shape=[None, 784])
    x_image = tf.reshape(x, [-1, 28, 28, 1])
```

Für die erste *Convolutional*-Schicht des KNN werden die Filterkernel als Gewichtungsvariable definiert. Die Größe eines Filterkernel beträgt 5x5 Bildpunkte, es soll ein Farbkanal ausgewertet werden. Zudem sollen 32 Filterkernel je Eingabebild genutzt werden. Für jeden davon wird eine Abweichung errechnet. Der Parameter `strides` definiert die Schrittweite der Filterkernel, `padding` definiert deren Verhalten am Rand des Berechnungsfensters. Auf die Ergebnismatrix wird eine ReLU-Aktivierungsfunktion angewandt:

```
    # Layer Conv1
    W_conv1 = weight_variable([5, 5, 1, 32])
    b_conv1 = bias_variable([32])
    output_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1,
                                           strides=[1, 1, 1, 1],
                                           padding='SAME') + b_conv1)
```

Die Ergebnisse jeder *Convolutional*-Schicht werden mittels einer *Pooling*-Schicht komprimiert. Beim genutzten Max-Pooling wird der Maximalwert eines 2x2 Bildpunkte großen Bereiches weitergegeben, welcher sich um 2 Bildpunkte weiterbewegt:

```
    # Layer Pool1
    output_pool1 = tf.nn.max_pool(output_conv1, ksize=[1, 2, 2, 1],
                                  strides=[1, 2, 2, 1], padding='SAME')
```

Für die Ergebnisverbesserung werden beide Schichten wiederholt, diesmal mit 64 Filterkernel. Es folgt eine vollvernetzte Schicht (*Dense*) mit 1024 Neuronen und ReLU als Aktivierungsfunktion, um die Zwischenergebnisse in die für die Ausgabe benötigte Form zu bringen. Die Dimension der Gewichtungsvariable entspricht der Bildhöhe und -breite nach dem zweiten *Pooling* (7x7) multipliziert mit der Anzahl der Kernelfilter der zweiten *Convolutional*-Schicht (64) und der Neuronenzahl:

```
    # Layer Dense
    W_dense = weight_variable([7 * 7 * 64, 1024])
    b_dense = bias_variable([1024])
    output_pool2_flat = tf.reshape(output_pool2, [-1, 7 * 7 * 64])
    output_dense = tf.nn.relu(tf.matmul(output_pool2_flat, W_dense) + b_dense)
```

Über eine *Dropout*-Schicht wird nur ein bestimmter Prozentsatz an Ergebnissen zur Ausgabe mit 10 Neuronen weitergeleitet, um Overfitting zu vermeiden:

```
    # Layer Dropout
    keep_prob = tf.placeholder(tf.float32)
    output_dense_drop = tf.nn.dropout(output_dense, keep_prob)

    # Layer Output
    W_output = weight_variable([1024, 10])
    b_output = bias_variable([10])
    y_conv = tf.nn.softmax(tf.matmul(output_dense_drop, W_output) + b_output)
```

Trainiert wird dann mit der in TensorFlow integrierten Gradientenberechnung für die Fehlerminimierung, für die Evaluation wird die Genauigkeit ausgegeben:

```
with tf.name_scope('train'):
    y_ = tf.placeholder(tf.float32, shape=[None, 10])
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    labels=y_, logits=y_conv))
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

with tf.name_scope('evaluate'):
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

with tf.Session() as session:
    tf.global_variables_initializer().run()
    for i in range(2000):
        batch = mnist.train.next_batch(100)
        session.run(train_step, feed_dict={x: batch[0], y_: batch[1],
                                           keep_prob: 0.4})

    print(session.run(accuracy, feed_dict={x: mnist.test.images,
                                           y_:mnist.test.labels,
                                           keep_prob: 1.0}))
```

Bei 2000 Trainingsschritten und einem *Dropout* von 40% erreicht das aufgebaute CNN bei der Evaluation eine Genauigkeit bei der Erkennung der handgeschriebenen Zahlen von 97,55%. Durch mehr Trainingsschritte und durch Anpassung der Anzahl der Testdaten je Trainigsschritt sowie der *Dropout*-Rate lässt sich hier eine Genauigkeit von über 99% erzielen.

6.3 Visualisierung mit TensorBoard

In der Praxis erreichen die Modelle von KNN häufig eine so hohe Komplexität, dass Werkzeuge zur Visualisierung notwendig sind, um den Überblick zu behalten oder effektiv Fehlersuche betreiben zu können. Mit TensorBoard [10] liefert TensorFlow hierfür ein integriertes Web-Interface zur Graph-Visualisierung. Hier sollen die Möglichkeiten und die Anwendung dieses Werkzeuges anhand des CNN aus Abschnitt 6.2 erläutert werden.

Die Kernfunktion von TensorBoard ist die detaillierte Darstellung des Berechnungsgraphen, die auch eine Gruppierung von Knoten erlaubt. Weiterhin können über sogenannte *Summaries* einzelne Tensoren und deren Werte über die Berechnungsdurchläufe hinweg ausgewertet werden. Möglich ist dies sowohl für Skalarwerte, wie zum Beispiel die Genauigkeit eines Modells, als auch für Wertverteilungen in Form von Histogrammen. Zudem können Bilder angezeigt werden, um zum Beispiel die Filterkernel eines CNN zu visualisieren. TensorBoard schreibt die für die Visualisierung benötigten Daten in eine Log-Datei, welche im Programmtext angegeben wird. Diese wird nach dem Start von TensorBoard in der Konsole (`tensorboard --logdir=path/to/log-directory`) über den Webbrowser geöffnet (`localhost:6006`). Was neben dem Berechnungsgraphen zusätzlich dargestellt werden soll, wird über Befehle im Programmtext geregelt. Der vollständige Programmtext dazu ist Listing 2 zu entnehmen.

Zuerst sollten Operationen entsprechend ihrer Funktion in sogenannten *Name Scopes* gruppiert werden, um dem Graphen eine logische Gliederung zu geben - hier am Beispiel für die Operationen der Evaluation. Für die Nachverfolgung und Auswertung der erzielten Genauigkeit wird zudem eine *Summary* für deren Skalarwert angelegt:

```
with tf.name_scope('evaluate'):
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy',accuracy)
```

Abschließend wird ein *SummaryWriter* benötigt, der die gesammelten *Summaries* innerhalb der *Session* in die festgelegte Log-Datei schreibt. Dabei können verschiedene *SummaryWriter* erstellt werden. Im vorliegenden Fall für das Training und die abschließende

Evaluation. Beide schreiben unterschiedliche Werte, dem ersten wird der Berechnungsgraph für dessen Darstellung in TensorBoard übergeben:

```
with tf.Session() as session:
    train_writer = tf.summary.FileWriter('./mnist-cnn-log/train', session.graph)
    test_writer = tf.summary.FileWriter('./mnist-cnn-log/test')
    tf.global_variables_initializer().run()
    for i in range(2000):
        batch = mnist.train.next_batch(100)
        summary_train, _ = session.run([tf.summary.merge_all(), train_step],
                                       feed_dict={x: batch[0],y_: batch[1],
                                                  keep_prob: 0.4})
        train_writer.add_summary(summary_train, global_step=i)

    summary_test, acc = session.run([tf.summary.merge_all(), accuracy],
                                    feed_dict={x: mnist.test.images,
                                               y_:mnist.test.labels,
                                               keep_prob: 1.0})
    test_writer.add_summary(summary_test)
```

In TensorBoard kann dann die erzielte Genauigkeit des Modells und der Berechnungsgraph betrachtet werden (siehe Abbildung 10 und Abbildung 11).

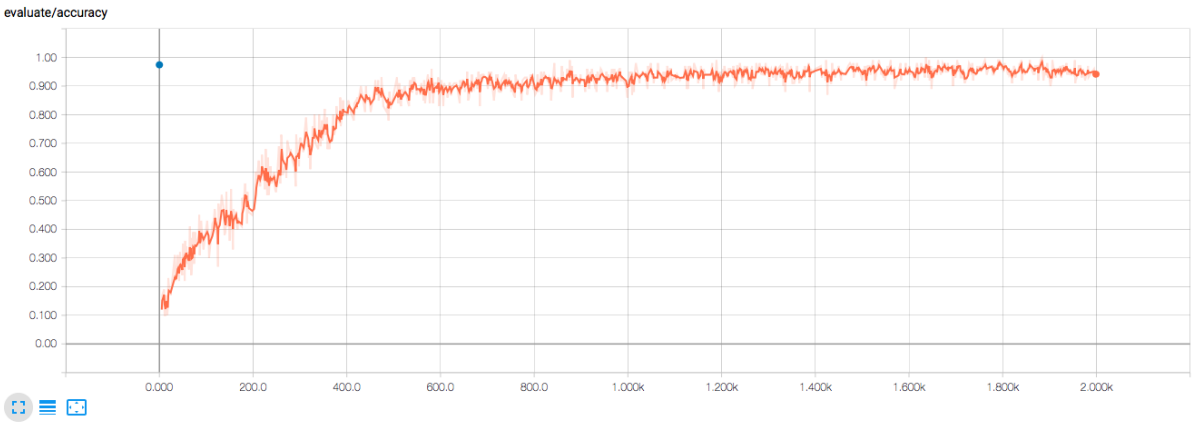
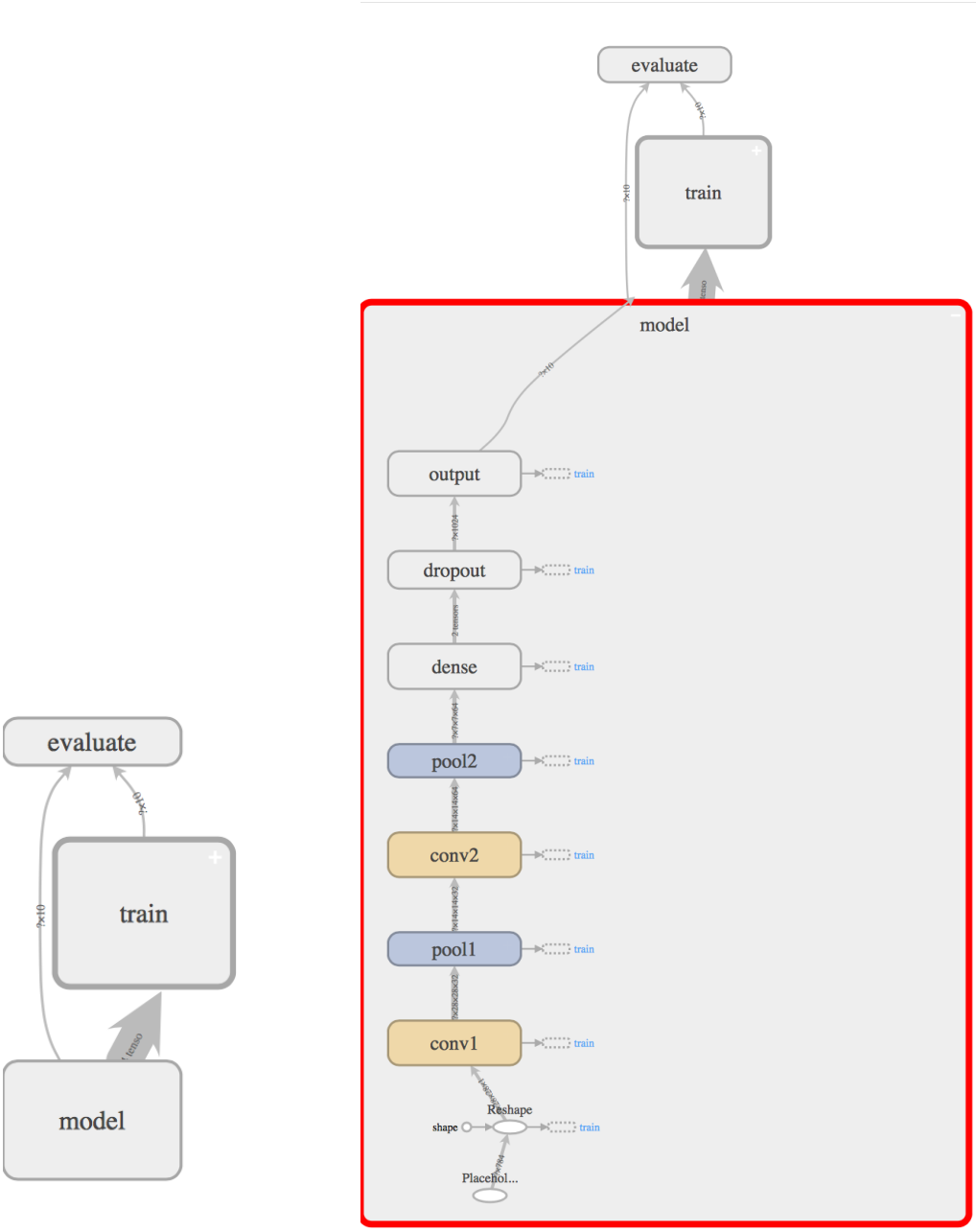


Abbildung 10: Die Verlaufs-Visualisierung der erreichten Genauigkeit während des Trainings mit 2000 Trainingsschritten (orange Kurve) und bei der Evaluierung (blauer Punkt)



(a) Grobübersicht des gesamten Graphen

(b) Expandierte Schichten des erstellten CNN-Models

Abbildung 11: Visualisierung des Berechnungsgraphen in TensorBoard

7 Bewertung und Fazit

Die in Abschnitt 2 vorgestellten Frameworks und Bibliotheken sind mit TensorFlow in einigen Punkten vergleichbar. *Theano* ähnelt TensorFlow dabei am meisten - über ein Python-Frontend wird ein Berechnungsgraph deklarativ definiert. Allerdings unterstützt *Theano*, wie alle anderen genannten Bibliotheken, keine native Ausführung von ML-Modellen auf verteilten Systemen. Dies wiederum ist eine Kernfunktion von TensorFlow. *Torch* verwendet im Gegensatz zu TensorFlow keinen Berechnungsgraph für die Modellierung von Algorithmen und nutzt dafür einen imperativen Ansatz. *Caffe*, fokussiert auf CNN, unterstützt keine zyklischen Modelle und entfällt damit für den Einsatz bei rekurrenten KNN.

Für einen Vergleich der Performance wird in Tabelle 4 Bezug auf einen Benchmark von Soumith Chintala [11] genommen. Dieser wurde auf einem Einzelplatzrechner durchgeführt und vergleicht mehrere Bibliotheken für ML beim Einsatz vorgegebener CNN zur Bilderkennung.

Tabelle 4: Benchmark-Ergebnisse von ML-Bibliotheken beim Einsatz von CNN zur Bilderkennung, gemessen wird die durchschnittliche Zeit für einen Trainingsschritt in Millisekunden

	AlexNet	**Overfeat**	**OxfordNet**	**GoogleNet**
Caffe	324	823	1068	1935
Neon	87	**211**	**320**	**270**
Torch	**81**	268	529	470
Chainer	177	620	885	687
TensorFlow	**81**	279	540	445

Neon übertrifft TensorFlow bei drei der eingesetzten CNN. Die ist auf die bei dieser Bibliothek angewandten manuellen Optimierung der CNN-Kernel in Assemblersprache zurückzuführen [Aba+16]. Abgesehen davon platziert sich TensorFlow auch auf einem Einzelplatzrechner unter den schnellsten Bibliotheken.

TensorFlows Ansatz eines Datenfluss-Graphen in Kombination mit seinen hochkompatiblen Abstraktionen für die Rechenoperationen und der integrierten Visualisierung des Berechnungsgraphen bietet Anwendern demnach die Möglichkeit, Modelle für ML effizient zu entwickeln sowie performant und hochskalierbar auf heterogener Hardware auszuführen.

Quellcode

Listing 1: Einfacher Aufbau und Ausführung eines Berechnungsgraphen

```
1  import tensorflow as tf
2  node_W = tf.Variable([[1, 2], [3, 4]], tf.int32, name='W')
3  node_x = tf.placeholder(tf.int32, name='x')
4  node_matmul = tf.matmul(node_W, node_x)
5
6  node_b = tf.constant([[7], [8]], tf.int32, name='b')
7  node_y = node_matmul + node_b
8
9  with tf.Session() as session:
10     tf.global_variables_initializer().run()
11     print(session.run(node_y, {node_x: [[5], [6]]}))
```

Listing 2: Erkennung handgeschriebener Zahlen mittels *Convolutional Neural Networks*

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data/', one_hot=True)

def weight_variable(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.1))

def bias_variable(shape):
    return tf.Variable(tf.constant(0.1, shape=shape))

with tf.name_scope('model'):
    x = tf.placeholder(tf.float32, shape=[None, 784])
    x_image = tf.reshape(x, [-1, 28, 28, 1])

    with tf.name_scope('conv1'):
        # Layer Conv1
        W_conv1 = weight_variable([5, 5, 1, 32])
        b_conv1 = bias_variable([32])
        output_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1,
                                               strides=[1, 1, 1, 1],
                                               padding='SAME') + b_conv1)

    with tf.name_scope('pool1'):
        # Layer Pool1
        output_pool1 = tf.nn.max_pool(output_conv1, ksize=[1, 2, 2, 1],
                                      strides=[1, 2, 2, 1], padding='SAME')

    with tf.name_scope('conv2'):
        # Layer Conv2
        W_conv2 = weight_variable([5, 5, 32, 64])
        b_conv2 = bias_variable([64])
        output_conv2 = tf.nn.relu(tf.nn.conv2d(output_pool1, W_conv2,
                                               strides=[1, 1, 1, 1],
                                               padding='SAME') + b_conv2)

    with tf.name_scope('pool2'):
        # Layer Pool2
        output_pool2 = tf.nn.max_pool(output_conv2, ksize=[1, 2, 2, 1],
                                      strides=[1, 2, 2, 1], padding='SAME')

    with tf.name_scope('dense'):
        # Layer Dense
```

```
        W_dense = weight_variable([7 * 7 * 64, 1024])
        b_dense = bias_variable([1024])
        output_pool2_flat = tf.reshape(output_pool2, [-1, 7 * 7 * 64])
        output_dense = tf.nn.relu(tf.matmul(output_pool2_flat, W_dense) + b_dense)

    with tf.name_scope('dropout'):
        # Layer Dropout
        keep_prob = tf.placeholder(tf.float32)
        output_dense_drop = tf.nn.dropout(output_dense, keep_prob)

    with tf.name_scope('output'):
        # Layer Output
        W_output = weight_variable([1024, 10])
        b_output = bias_variable([10])
        y_conv = tf.nn.softmax(tf.matmul(output_dense_drop, W_output) + b_output)

with tf.name_scope('train'):
    y_ = tf.placeholder(tf.float32, shape=[None, 10])
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_,
    logits=y_conv))
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

with tf.name_scope('evaluate'):
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy',accuracy)

with tf.Session() as session:
    train_writer = tf.summary.FileWriter('./mnist-cnn-log/train', session.graph)
    test_writer = tf.summary.FileWriter('./mnist-cnn-log/test')
    tf.global_variables_initializer().run()
    for i in range(2000):
        batch = mnist.train.next_batch(100)
        summary_train, _ = session.run([tf.summary.merge_all(), train_step],
                                       feed_dict={x: batch[0],y_: batch[1],
                                                  keep_prob: 0.4})
        train_writer.add_summary(summary_train, global_step=i)

    summary_test, acc = session.run([tf.summary.merge_all(), accuracy],
                                    feed_dict={x: mnist.test.images,
                                               y_:mnist.test.labels,
                                               keep_prob: 1.0})
    test_writer.add_summary(summary_test)
```

Literatur

[Aba+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Whitepaper, Software available from tensorflow.org. 2015. URL: `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45166.pdf`.

[Aba+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis u. a. „TensorFlow: A System for Large-Scale Machine Learning“. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, S. 265–283. ISBN: 978-1-931971-33-1.

[Al-+16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau u. a. „Theano: A Python framework for fast computation of mathematical expressions“. In: *CoRR* abs/1605.02688 (2016). arXiv: `1605.02688`.

[CBM02] Ronan Collobert, Samy Bengio und Jhonny Mariéthoz. *Torch: a modular machine learning software library*. 2002.

[Cli95] Cliff Click. „Global Code Motion / Global Value Numbering“. In: *SIGPLAN Not.* 30.6 (Juni 1995), S. 246–257. ISSN: 0362-1340. DOI: `10.1145/223428.207154`.

[Dea+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Chen Kai, Matthieu Devin u. a. „Large Scale Distributed Deep Networks“. In: *Advances in Neural Information Processing Systems 25*. Hrsg. von F. Pereira, C. J. C. Burges, L. Bottou und K. Q. Weinberger. Curran Associates, Inc., 2012, S. 1223–1231.

[DV16] Vincent Dumoulin und Francesco Visin. „A guide to convolution arithmetic for deep learning“. In: *arXiv eprints* abs/1603.07285 (2016). arXiv: `1603.07285`.

[Gol16] Peter Goldsborough. „A Tour of TensorFlow“. In: *CoRR* abs/1610.01178 (2016).

[Jia+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long u. a. „Caffe: Convolutional Architecture for Fast Feature Embedding“. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14.

Orlando, Florida, USA: ACM, 2014, S. 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889.

[Jou+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal u. a. „In-Datacenter Performance Analysis of a Tensor Processing Unit“. In: *CoRR* abs/1704.04760 (2017).

[Koh+94] Ron Kohavi, George John, Richard Long, David Manley und Karl Pfleger. „MLC++ A machine learning library in C++“. In: *Sixth IEEE International Conference on Tools with Artificial Intelligence*. 1994, S. 234–245. DOI: 10.1109/TAI.1994.346412.

[Mor+15] Philipp Moritz, Robert Nishihara, Ion Stoica und Michael I. Jordan. „SparkNet: Training Deep Networks in Spark“. In: *CoRR* abs/1511.06051 (2015).

[Ped+11] Fabian Pedregosa, Gaél Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion u. a. „Scikit-learn: Machine Learning in Python“. In: *J. Mach. Learn. Res.* 12 (Nov. 2011), S. 2825–2830. ISSN: 1532-4435.

[Vö17a] Gerhard Völkl. „Python-Tutorial, Teil 1: Maschinelles Lernen mit TensorFlow“. In: *iX* 9 (2017), S. 42–50. ISSN: 0935-9680.

[Vö17b] Gerhard Völkl. „Python-Tutorial, Teil 2: Neuronale Netze und Deep Learning“. In: *iX* 9 (2017), S. 52–59. ISSN: 0935-9680.

Webadressen

[1] NVIDIA: *Parallele Berechnungen mit CUDA*, http://www.nvidia.de/object/cuda-parallel-computing-de.html, abgerufen am 21.12.2017

[2] Torch, http://torch.ch, abgerufen am 02.01.2018

[3] Intel Nervana Neo, http://neon.nervanasys.com/index.html/, abgerufen am 06.01.2018

[4] Kaz Sato, Cliff Young und David Patterson: *An in-depth look at Googles first Tensor Processing Unit*, https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, abgerufen am 20.12.2017

[5] Norm Jouppi: *Google supercharges machine learning tasks with TPU custom chip*, https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, abgerufen am 20.12.2017

[6] Jeff Dean und Urs Hölzle: *Build and train machine learning models on our new Google Cloud TPUs*, https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/, abgerufen am 20.12.2017

[7] Google: *TensorFlow*, https://www.tensorflow.org, abgerufen am 20.11.2017

[8] Google: *Deep MNIST for Experts*, https://www.tensorflow.org/get_started/mnist/pros, abgerufen am 31.12.2017

[9] Google: *A Guide to TF Layers: Building a Convolutional Neural Network*, https://www.tensorflow.org/tutorials/layers, abgerufen am 31.12.2017

[10] Google: *TensorBoard*, https://www.tensorflow.org/get_started/summaries_and_tensorboard, abgerufen am 01.01.2018

[11] Soumith Chintala, https://github.com/soumith/convnet-benchmarks, abgerufen am 02.01.2018