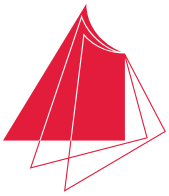


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Horizontales Skalieren von Deep Learning Frameworks

Bachelor Thesis von

Sebastian Jäger

an der Fakultät für Informatik und Wirtschaftsinformatik
Fachrichtung Verteilte Systeme (VSYS)

Erstgutachter:	Prof. Dr. rer. nat. Christian Zirpins
Zweitgutachter:	Prof. Dr. Sulzmann
Betreuer:	Hans-Peter Zorn (inovex GmbH)

01. Oktober 2017 – 31. Januar 2018

Hochschule Karlsruhe Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik
Moltkestr. 30
76133 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 31.01.2018

. . . . . .

(Sebastian Jäger)

Zusammenfassung

In dieser Arbeit werden generische Deep Learning Frameworks für die Programmiersprache Python auf deren horizontale Skalierbarkeit beim Training von Neuronalen Netzen experimentell evaluiert.

Hierzu werden zuerst Kriterien definiert, die dafür von Nutzen sind, um *TensorFlow* und *MXNet* für die Evaluation auszuwählen. Um eine möglichst umfangreiche Bewertung zu erreichen, wurde mit jedem der Frameworks sowohl ein Convolutional Neural Network, als auch ein Recurrent Neural Network implementiert. Für die Experimente werden die zwei gängigen Datensätze *Fashion-MNIST* und *PTB* verwendet, um den Durchsatz in *Batches pro Sekunde* zu messen. Die Skalierbarkeit wird mithilfe der Metrik *Speedup* berechnet. Weiter wird für die Experimente die Kubernetes Engine der Google Cloud Platform genutzt, deren Knoten jeweils mit einer CPU des Typs *2.5 GHz Intel Xeon E5 v2* bestückt sind.
Dabei wurde gezeigt, dass die Implementierung von Neuronalen Netzen mit MXNet effizienter ist, TensorFlow jedoch bessere Skalierungseigenschaften aufweisen kann. Trotzdem gelang es TensorFlow lediglich in einem der Experimente einen höheren Durchsatz zu erzielen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Der *Hype Cycle* des Unternehmens Gartner zeigt die verschiedensten Technologien in den unterschiedlichen Phasen auf ihrem Weg zur Produktivität. Ein Begriff, der im vergangenen Jahr 2017 auf der Spitze seines Hypes abgebildet wurde, ist *Deep Learning*[1].

Für den Begriff Deep Learning existieren diverse Definitionen. Einige Experten dieses Gebiets, darunter auch Ian Goodfellow, Yoshua Bengio und Aaron Courville, beschreiben Deep Learning als das Erlernen von diversen Merkmalen auf unterschiedlich abstrakten Ebenen. Dabei werden die Merkmale der jeweils niedrigeren Abstraktionsschicht verwendet und zu komplexeren aggregiert. [2, 11] Andere hingegen beschreiben Deep Learning konkreter als das Verwenden von Neuronalen Netzwerken, wobei der Begriff deep dafür steht, dass die eingesetzten Netze aus vielen Schichten bestehen [7].
Die Definitionen teilen, dass Deep Learning ein Bereich des Maschinellen Lernens und somit auch Teil des Forschungsgebiets der Künstlichen Intelligenz ist. Der Bereich des Maschinellen Lernens befasst sich damit, Algorithmen zu entwerfen, die selbständig und ohne explizite Implementierung von Trainingsdatensätzen lernen, wie sie sich bei zukünftigen Daten verhalten sollen oder versuchen, Vorhersagen über zukünftige Daten zu treffen.

Der Aufschwung von Deep Learning lässt sich ebenfalls an vielen Produkten und Services festmachen, die Neuronale Netze einsetzen, um das Benutzererlebnis zu verbessern. Beispielsweise ist es bei Facebook seit 2015 möglich, beim Hochladen von Bildern Freunde automatisch vom System markieren zu lassen. Das dahinter verborgene Projekt nennt sich *DeepFace*, das mithilfe von älteren bereits markierten Bildern lernt, Gesichter zu unterscheiden und diese den Personen korrekt zuzuordnen. [27]
Ende des Jahres 2016 hat das Unternehmen Google angekündigt[2], dass der Service *Google Translate* zukünftig die Übersetzung zwischen englischen und chinesischen Texten mithilfe von Neuronalen Netzen erstellt [32].
Auch bei Empfehlungssystemen, wie sie beispielsweise bei YouTube oder Google Play verwendet werden, wurden Verbesserungen durch die Verwendung von Neuronalen Netzen erzielt [4, 5].

1.1 Motivation

Weil das Trainieren vor allem bei großen Neuronalen Netzen, wie sie in Projekten eingesetzt werden, die im vorherigen Abschnitt beschrieben sind, sehr viel Rechenleistung und ebenso viel Zeit in Anspruch nimmt, wurde deren Code schon früh optimiert. Zu diesen Optimierungen gehören nicht nur effizientere Algorithmen und Implementierungen, sondern auch das Auslagern von rechenintensiven Aufgaben auf verfügbare Grafikkarten. Um

[1] Gartner Hype Cycle 2017: `https://goo.gl/MrQazW`
[2] Blog-Artikel über maschinelle Übersetzungen mit Neuronalen Netzen: `https://goo.gl/jRlrL6`

Entwicklern das Verwenden von Neuronalen Netzen zu erleichtern, wurden *Deep Learning Frameworks* programmiert, die die Komplexität der Optimierungen abstrahieren.

Frühe Entwicklungen von Deep Learning Frameworks waren besonders darauf ausgelegt, die Verwendung Neuronaler Netze zu erleichtern. Durch das Wachstum der Netze, aber auch durch die Weiterentwicklung hin zu komplexeren Netztypen, stieg der benötigte Bedarf an Rechenleistung zum Trainieren weiter, bis über die physikalischen Grenzen der Rechenleistung einer Maschine hinaus. Somit wurde es zunehmend wichtiger, dass Frameworks das verteilte Trainieren von Neuronalen Netzen unterstützen. Auf dieser Basis sind neue Deep Learning Frameworks entstanden, die bei deren Entwicklung auch die horizontale Skalierbarkeit fokussierten und daher Mechanismen bieten können, um das Training über mehrere Maschinen zu verteilen.

Es zeigt sich, dass derzeit eine Vielzahl von Deep Learning Frameworks zur Verfügung steht. Für Entwickler, die Neuronale Netze einsetzen, stellt sich die Frage: Welches Framework ist das beste um eine gegebene Problemstellung zu bearbeiten. Ein wichtiger Punkt, beim Beantworten dieser Frage, ist die Trainingsperformanz, die bei modernen Neuronalen Netzen ausschlaggebend von der horizontalen Skalierbarkeit des Frameworks mit beeinflusst wird.

In Zeiten diverser Cloudangebote unterschiedlichster Anbieter ist eine naheliegende Idee, diese zum Trainieren der Neuronalen Netze zu verwenden. Der große Vorteil hierbei ist, dass die enorme Rechenleistung, die zum Trainieren nötig ist, nicht gekauft werden muss. Kosten, die durch nicht genutzte Hardware entstehen, können, dank sekundengenauer Abrechnungen bei Cloudangeboten, minimiert werden, da lediglich für die genutzte Rechenzeit gezahlt wird.

1.2 Zielsetzung

Das Ziel der vorliegenden Arbeit ist die experimentelle Evaluation von generischen Deep Learning Frameworks für die Programmiersprache Python in Hinblick auf das horizontale Skalieren.
Generisch bedeutet, dass ein Framework nicht auf eine bestimmte Anwendungsdomäne beschränkt ist, sondern in möglichst vielen Bereichen eingesetzt werden kann. Das heißt, die Frameworks sollten mindestens die gängigsten Netzarchitekturen unterstützten. Der Grund für die Verwendung der Programmiersprache Python ist deren weite Verbreitung im Bereich Data Science[3]. Dieser ist das Hauptanwendungsgebiet von Maschinellem Lernen und Neuronalen Netzen. Durch die Verwendung von Python soll Entwicklern der Einstieg in das Implementieren Neuronaler Netze erleichtert werden.

Dabei soll evaluiert werden, wie sich diverse Frameworks beim Trainieren unterschiedlicher Neuronaler Netzarten verhalten. Speziell die Veränderung der Performanz in Abhängigkeit der Worker-Maschinen Anzahl und des Netztyps ist hierbei von Interesse.

[3]Die Data Science befasst sich mit dem Generieren von Wissen aus Daten.

1.3 Aufbau dieser Arbeit

Zunächst werden in Kapitel 2 die Grundlagen von Neuronalen Netzen erläutert. Anschließend wird beschrieben, wie Neuronale Netze trainiert werden. Bevor auf die horizontalen Skalierungsmöglichkeiten des Trainings mit unterschiedlichen Parallelisierungsarten eingegangen wird, werden gesondert sowohl Convolutional Neural Networks, als auch Recurrent Neural Networks erläutert.

Das dritte Kapitel beschäftigt sich mit der Definition unterschiedlicher Kriterien, die aus der Zielsetzung des vorherigen Abschnitts abgeleitet werden. Anschließend sind diverse Frameworks und deren Eigenschaften beschrieben, mit deren Hilfe eine Auswahl für die experimentelle Evaluation getroffen wird.

In Kapitel 4 ist der Versuchsaufbau der experimentellen Evaluation beschrieben. Dabei wird zuerst diskutiert, welche Aufgabengebiete heran gezogen werden, um einen möglichst umfangreichen Vergleich der Frameworks zu erzielen. Anschließend werden die zur Evaluation verwendeten Metriken und Datensätze vorgestellt und die Umgebungsbedingen des Versuchsaufbaus erläutert.

Im fünften Kapitel werden eingangs die verwendeten Technologien beschrieben und anschließend aufgezeigt, wie mit diesen die infrastrukturelle Umgebung der Versuche implementiert ist. Im zweiten Teil des Kapitels sind die Implementierungen der Neuronalen Netze, sowie frameworkabhängigen Besonderheiten des verteilten Trainings erläutert.

Das Kapitel 6 beschreibt unterschiedliche Experimente und schließt mit einer Bewertung der Messergebnisse und der Frameworks selbst.

2 Theoretische Grundlagen von Deep Learning Verfahren mit Neuronalen Netzen

In diesem Kapitel wird das grundlegende Wissen über Deep Learning Verfahren mithilfe von Neuronalen Netzen vermittelt. Zunächst wird der Aufbau eines künstlichen Neurons und die mathematischen Grundlagen von Neuronalen Netzen und deren Training erläutert. Bevor diverse horizontale Skalierungsmöglichkeiten beschrieben und erklärt werden, wird auf unterschiedliche Netzarten und deren Einsatzgebiete eingegangen.

2.1 Neuronale Netzwerke

Die künstlichen Neuronalen Netzwerke, auch Netze, wurden, wie schon viele technische Neuerungen vor ihnen, von der Natur inspiriert. Zur besseren Lesbarkeit werden diese im Folgenden als *Neuronale Netzwerke* oder kurz als *Neuronale Netze* bezeichnet. Die ersten künstlichen Neuronen, die bereits im Jahr 1943 entwickelt wurden, folgten noch sehr stark ihrem natürlichem Vorbild, dem Gehirn. Auch sie werden im weiteren Verlauf kurz als *Neuronen* bezeichnet.
Trotz ihrer sehr einfachen Funktionsweise: "Sie aktivieren ihren Ausgang, wenn eine fest definierte Schwelle von aktiven Eingängen überschritten wird", konnten McCulloch und Pitts zeigen, dass durch das Kombinieren dieser Neuronen jeder beliebige boolsche Ausdruck berechnet werden kann [21].

Neuronen, wie sie heute eingesetzt werden und im folgenden Abschnitt beschrieben sind, wurden erst einige Zeit später im Jahr 1958 von Frank Rosenblatt [25] entwickelt.

2.1.1 Aufbau von künstlichen Neuronen

Wie bereits im Abschnitt 2.1 erwähnt, bestehen Neuronale Netze aus vielen einzelnen Neuronen, die miteinander verbunden werden. Der Aufbau eines einzelnen Neurons kann schematisch, wie in Abbildung 2.1 zu sehen, dargestellt werden. Sie bestehen aus einem oder mehreren *Eingängen* und aus genau einem *Ausgang*. Seine Aufgabe ist es, die Eingänge $x_1, x_2, \ldots, x_n$, welche als Vektor $\hat{\mathbf{x}}$ darstellbar sind, auf einen skalaren *Aktivierungswert* a abzubilden. Dabei besitzt jeder Eingang ein dediziertes *Gewicht* $w_1, w_2, \ldots, w_n$, das bestimmt, wie stark dieser in die Berechnung des Aktivierungswertes mit einfließt. Auch die Gewichte können als Vektor $\hat{\mathbf{w}}$ dargestellt werden. Für die Berechnung des Aktivierungswertes muss zuerst die *Netto-Eingabe net* berechnet werden. Hierfür wird die sogenannte *gewichtete Summe* errechnet, n entspricht der Anzahl Eingänge des Neurons.

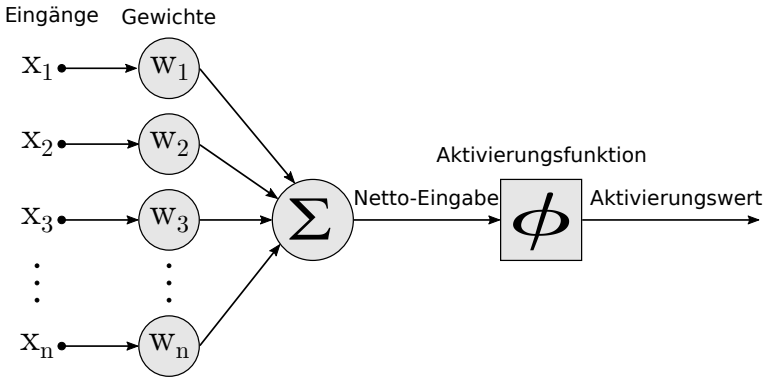


Abbildung 2.1: Aufbau eines künstlichen Neurons, angelehnt an [22].

Die Operation $\cdot$ angewendet auf zwei Vektoren, steht in dieser Arbeit stellvertretend für das Skalarprodukt zweier Vektoren.

$$net = \hat{\mathbf{x}} \cdot \hat{\mathbf{w}} = \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

Die gewichtete Summe *net* wird anschließend zum Berechnen des Aktivierungswertes a mithilfe einer *Aktivierungsfunktion* ϕ verwendet. Eine Auswahl diverser Aktivierungsfunktionen und welche Auswirkungen diese auf das Verhalten des Netzes haben, wird im Abschnitt 2.1.4 diskutiert. Der Aktivierungswert eines Neurons kann, wie in Gleichung 2.2 zu sehen, berechnet werden. [22]

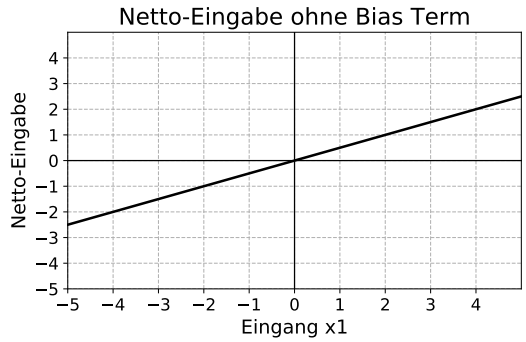
$$a = \phi(net) = \phi(\hat{\mathbf{x}} \cdot \hat{\mathbf{w}}) = \phi(\sum_{i=1}^{n} x_i w_i) \tag{2.2}$$

Damit ein Neuron möglichst flexible Berechnungen ausdrücken kann, wird häufig ein *Bias* hinzugefügt. Dieser wird als Eingang mit konstantem Wert dargestellt und fließt in die Gleichungen als Variable b mit ein. Die obigen Berechnungen werden dadurch, wie in Gleichung 2.3 und 2.4 zu sehen ist, verändert.

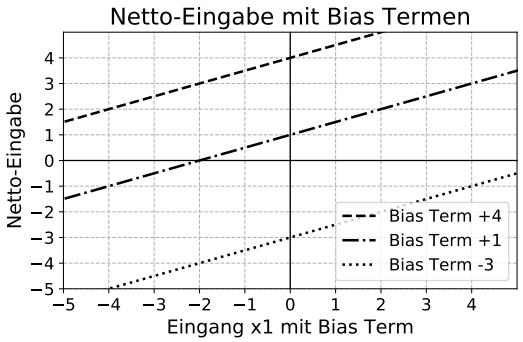
$$net = \hat{\mathbf{x}} \cdot \hat{\mathbf{w}} + b = \sum_{i=1}^{n} x_i w_i + b \tag{2.3}$$

$$a = \phi(net) = \phi(\hat{\mathbf{x}} \cdot \hat{\mathbf{w}} + b) = \phi(\sum_{i=1}^{n} x_i w_i + b) \tag{2.4}$$

Das Beispiel der Abbildung 2.2a zeigt die Berechnung der gewichteten Summe *net* eines Neurons mit einem Eingang x_1. Das einzige Gewicht w_1 wurde mit einem Wert von $0,5$ belegt. Durch das Hinzufügen eines Bias Terms können die Funktionswerte der gewichteten Summe somit entlang der Y-Achse verschoben werden, wie es Abbildung 2.2b zeigt. Durch die Verschiebung von *net* ist es implizit möglich, den Aktivierungswert des Neurons bei gleichem Eingangsmuster zu erhöhen oder erniedrigen. [22]



(a) Netto-Eingabe eines Neurons mit einem Eingang und Gewichtung 0, 5 ohne Bias Term.



(b) Verschiebung der Netto-Eingabe, des selben Neurons, durch das Hinzufügen diverser Bias Terme.

Abbildung 2.2: Darstellung der Auswirkung von Bias Termen auf die Funktionswerte der Netto-Eingabe.

An diesem Punkt stellt sich die Frage, wie man die Gewichte, beziehungsweise Bias Terme, wählt, um eine möglichst effektive Arbeitsweise eines Neurons, beziehungsweise eines kompletten Netzes, zu erzielen. Weder die Gewichte, noch die Bias Terme müssen manuell bestimmt werden, denn das Wählen der besten Werte geschieht während des Trainings des Netzwerks, das im Abschnitt 2.1.3 beschrieben ist.

Im folgenden Abschnitt ist beschrieben, wie mehrere künstliche Neuronen, die in diesem Abschnitt beschrieben wurden, zu komplexeren Netzen verbunden werden.

2.1.2 Aufbau von Neuronalen Netzen

Da die Natur als Vorbild für ein einzelnes Neuron herangezogen wurde, liegt es nahe, dass sie auch als Vorbild für die Verbindung von mehreren und somit für das Bilden von Neuronalen Netzen fungiert. Denn wie auch im Gehirn[1], werden viele Neuronen in Schichten miteinander verbunden. Ein Neuronales Netzwerk besteht mindestens aus einer *Eingangsschicht* und einer *Ausgangsschicht*, in der Regel besitzt es jedoch noch eine oder mehrere *versteckte Schichten*. In diesem Falle werden sie auch *tiefe Neuronale Netzwerke* genannt.

Die Eingangsschicht ist die Schnittstelle, die zum Eingeben der Daten in das Netzwerk dient. Dazu wird üblicherweise keine Aktivierungsfunktion, beziehungsweise die Identitätsfunktion $\phi(x) = x$ verwendet, die die Eingangswerte unverändert an die nächste Schicht weitergibt. Ebenso ist es für diese üblich, exakt so viele Neuronen zu verwenden wie die Eingangsdaten *Merkmale* (engl. *features*) besitzen. Bei einem Graustufen Bild mit 28×28 Pixeln würden typischerweise 784 Neuronen für die Eingangsschicht verwendet werden, für jedes Pixel genau ein Neuron.

Mit der Anzahl der versteckten Schichten, beziehungsweise der Anzahl der Neuronen, die in dieser Schicht verwendet werden, kann die Performanz des Neuronalen Netzwerks

[1] Informationen zur Großhirnrinde: `https://goo.gl/k7OWmC`

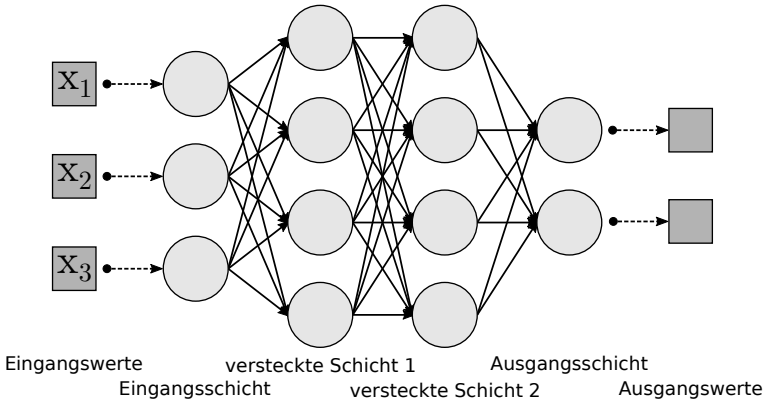


Abbildung 2.3: Geschichteter Aufbau eines modernen Neuronalen Netzes, angelehnt an [22].

sehr stark beeinflusst werden. Wird die Anzahl der Neuronen oder Schichten erhöht, stehen mehrere Gewichte zur Verfügung, die während des Trainings angepasst werden können. Damit kann das Neuronale Netzwerk komplexere Modelle ausdrücken, es benötigt gleichzeitig jedoch ebenfalls mehr Zeit, um diese zu trainieren.
Mithilfe der Ausgangsschichten drückt das Neuronale Netz sein Ergebnis aus. Typischerweise hängt dessen Struktur stark von der Aufgabe des Netzwerks ab. Dabei wird vor allem darauf geachtet, dass die Aktivierungswerte der Neuronen der Ausgangsschicht einfach weiterverarbeitet werden können. Soll das Neuronale Netz zum Beispiel eine Klassifizierung vornehmen und jede Eingabe in eine von zehn exklusiven Klassen einordnen, ist ein weitverbreiteter Ansatz zehn Neuronen in der Ausgangsschicht zu verwenden. [22]

Die Netzarchitektur, die in der Abbildung 2.3 dargestellt ist, wird *voll verbundenes vorwärts Netzwerk* (engl. *fully connected feed-forward network*) genannt. Diese Bezeichnung beschreibt gleich mehrere Eigenschaften des Netzes. Die durchgezogenen Pfeile visualisieren die Verbindungen zwischen den Neuronen. Dabei ist gut zu erkennen, dass diese nur in eine Richtung, nämlich von der Eingangsschicht über die versteckten Schichten hin zur Ausgangsschicht "vorwärts" miteinander verbunden sind. Die zweite Eigenschaft ist, dass in diesem speziellen Netz jedes Neuron einer Schicht i mit jedem Neuron der Schichten $i-1$ und $i+1$ "voll verbunden" ist. Jede dieser Verbindungen besitzt ein Gewicht, das während des Trainings angepasst werden kann. Daher ist diese Netzart besonders flexibel und kann für viele Aufgaben eingesetzt werden. [22]
Das heißt jedoch nicht, dass es für alle Aufgaben eine gleich gute Performanz liefern kann. Für unterschiedliche Probleme wurden darüber hinaus auch speziellere Netzarchitekturen entwickelt, die, in diesem speziellen Bereich, deutlich bessere Ergebnisse liefern können.

2.1.3 Training von Neuronalen Netzwerken

Beim Trainieren wird versucht, die Gewichte und Biases des Neuronalen Netzes so anzupassen, dass das Netz zukünftig für möglichst alle Eingangsdaten eine annähernd optimale Lösung findet. Die trainierbaren Gewichte und Biases werden im Folgenden als

Trainingsparameter bezeichnet, die mit dem griechischen Buchstaben Θ in Gleichungen dargestellt sind.

In sehr vielen Wiederholungen werden dem Neuronalen Netz sogenannte *Trainingsdaten* übergeben. Das Neuronale Netz berechnet die Ausgabe und der Trainingsalgorithmus vergleicht anschließend die errechneten Ergebnisse mit dem *Label* des Trainingsdatensatzes. Weicht das Ergebnis des Neuronalen Netzes von dem erwarteten Wert ab, müssen die Trainingsparameter angepasst werden, um beim nächsten Trainingsdatensatz möglichst ein besseres Ergebnis zu erreichen.

Diese Trainingsart nennt man *überwachtes Lernen* (engl. *supervised learning*). Der Name beschreibt deutlich, was die Besonderheit dieser Methode ist. In regelmäßigen Abständen wird überprüft, ob das Gelernte sich positiv auf die Aufgabe auswirkt und versucht, zielgerichtete Änderungen an den Trainingsparametern vorzunehmen. Um dies zu ermöglichen, muss ein Trainingsdatensatz vorliegen, der nicht nur aus möglichst realen, repräsentativen und gereinigten Daten [10] besteht, sondern bei dem jedes Datum zusätzlich ein Label enthält, das dieses beschreibt.

Im Bereich des maschinellen Lernens existieren noch weitere Trainingsarten, wie das *unüberwachte Lernen* (engl. *unsupervised learning*), bei denen dem Trainingsalgorithmus keine Angaben über das erwartete Ergebnis gemacht werden. Es existieren ebenfalls hybride Ansätze oder das weiter verbreitete *Reinforcement Learning*. Diese sind jedoch für diese Arbeit nicht weiter von Bedeutung.

Um Neuronale Netze zu trainieren, kommen mathematische Verfahren und Algorithmen zum Einsatz, die in den folgenden Abschnitte beschrieben sind.

2.1.3.1 Training von Neuronalen Netzen als mathematisches Optimierungsproblem

Ganz allgemein befassen sich Optimierungsprobleme der angewandten Mathematik damit, Funktionen, die meist Gegebenheiten aus anderen Wissenschaften beschreiben, zu max- oder minimieren[2]. Das Trainieren von Neuronalen Netzwerken kann daher als ein solches angesehen werden. Hierzu wird eine sogenannte *Fehlerfunktion*, oder *Kostenfunktion* aufgestellt, die beschreibt, wie weit der errechnete Wert vom erwarteten Wert abweicht. Im Englischen wird diese Funktion auch *error function* oder *loss function* genannt und daher als E bezeichnet. Beide deutschen Begriffe werden im Verlauf dieser Arbeit synonym verwendet. Durch das Minimieren dieser Funktion, wird das Neuronale Netz verbessert. [3]

Ein gängiges Verfahren, das auch in anderen Bereichen des maschinellen Lernens Anwendung findet ist das *Gradientenabstiegsverfahren* (engl. *gradient descent*). Hierbei werden in vielen Iterationen die Trainingsparameter in kleinen Schritten geändert, bis der Fehler, also der Funktionswert der Kostenfunktion E, konvergiert und einen möglichst kleinen Wert annimmt. [3]

Um zu bestimmen, wie die Trainingsparameter anzupassen sind, wird, wie der Name des Verfahrens vermuten lässt, der Gradient verwendet. Ein Gradient ist ein Operator ∇, der angewendet auf einen bestimmten Punkt einer Funktion einen Vektor angibt, der in Richtung des steilsten Anstiegs zeigt und als Länge die Steigung angibt[3]. Mithilfe eines

[2] Informationen zum Optimierungsproblem: `https://goo.gl/21YnHu`

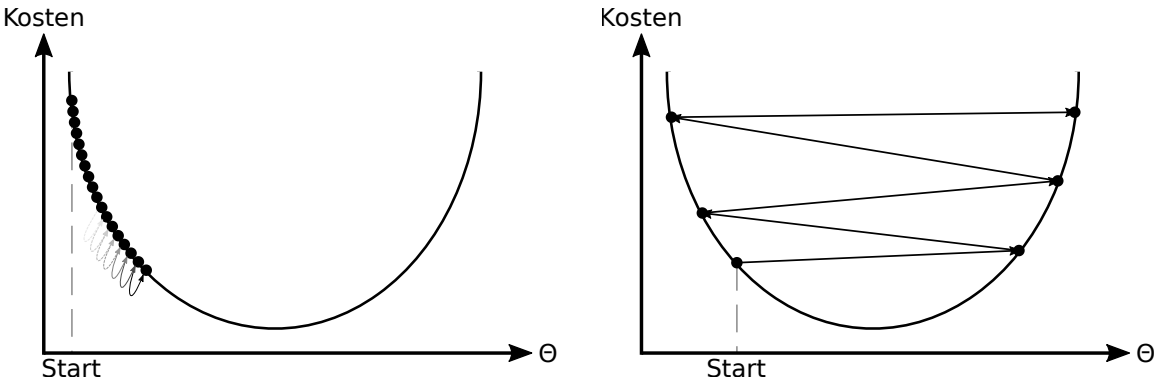
[3] Informationen zum Gradient: `https://goo.gl/2MfxT3`

Algorithmus namens *Backpropagation*, der im Abschnitt 2.1.3.2 beschrieben ist, werden die Gradienten der Trainingsparameter berechnet.

Damit der Funktionswert von E verringert wird, werden bei jeder Iteration die Trainingsparameter Θ in Richtung des negativen Gradienten geändert. Dieser Vorgang kann mathematisch nach [10] wie in Gleichung 2.5 zu sehen beschrieben werden und wird meist als *Update- Regel* bezeichnet. Der Ausdruck $\nabla_\Theta E(\Theta)$ bedeutet hierbei, dass der Gradient ∇ über alle Trainingsparameter Θ der Kostenfunktion E gebildet wird.

$$\Theta = \Theta - \eta * (\nabla_\Theta E(\Theta)) \qquad \eta \in [0, \infty) \tag{2.5}$$

Der Parameter η wird *Lernrate* genannt und dient zum Steuern der Schrittgröße. Problematisch ist hierbei, dass eine zu kleine Lernrate dazu führt, dass das Training sehr langsam konvergiert und somit sehr viele Iterationen benötigt werden, um ein gutes Ergebnis zu erhalten. Bei einer zu großen Lernrate hingegen ist es möglich, dass die Funktionswerte von E durch das Training nicht gegen einen kleinen Wert konvergieren. In Abbildung 2.4 wird dargestellt, wie sich eine schlecht gewählte Lernrate auswirkt. [10]



(a) Gradientenabstieg mit zu kleiner Lernrate. (b) Gradientenabstieg mit zu großer Lernrate.

Abbildung 2.4: Probleme mit schlecht gewählter Lernrate beim Gradientenabstiegsverfahren nach [10].

Funktionen, die mehrschichtige Neuronale Netze beschreiben, sind in der Regel sehr hochdimensional, da jedes Gewicht eine Variable in dieser Gleichung darstellt. Dadurch besitzen diese Funktionen häufig sehr viele lokale Minima. Das Gradientenabstiegsverfahren kann in der Praxis zu Schwierigkeiten führen, da es in einem lokalen Minimum stecken bleiben kann. Dieses Problem wird durch eine zu kleine Lernrate unterstützt, weil durch die sehr kleinen Schritte, die bei jedem Trainingsdurchlauf gemacht werden, die Möglichkeit verloren geht, ein solches zu überspringen. Es gibt verschiedene Verbesserungen, die weniger anfällig für lokale Minima sind, diese werden in dieser Arbeit jedoch nicht weiter betrachtet. [10]

2.1.3.2 Backpropagation Algorithmus

Der *Backpropagation* Algorithmus, was zu Deutsch so viel wie *Rückpropagierung* bedeutet, wurde 1986 von David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams zum

ersten Mal für das Trainieren von Neuronalen Netzen eingesetzt. Die grundlegende Idee ist es, zu berechnen, wie stark sich der Fehler des Neuronalen Netzes ändert, wenn der Aktivierungswert eines versteckten Neurons angepasst wird; sprich, wie stark ein Neuron zum Gesamtfehler beiträgt. Diese Fehleränderung kann weiter in die umgekehrte Richtung verfolgt und somit für jede Verbindung des Netzwerks nachvollzogen werden. [26]

Der Backpropagation Algorithmus nutzt hierzu eine Technik, die auch unter dem Begriff *automatisches Differenzieren*[4] bekannt ist, denn das Berechnen der Änderungsstärke des Fehlers entspricht dem Berechnen der Ableitung der Fehlerfunktion. Dazu wird diese als *Berechnungsgraph*, wie er beispielhaft in Abbildung 2.5 zu sehen ist, dargestellt.

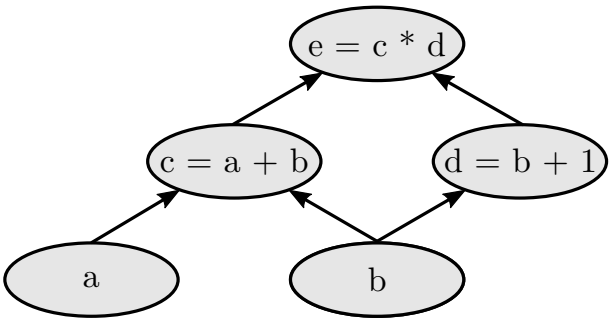


Abbildung 2.5: Darstellung eines Berechnungsgraphen der mathematischen Berechnung $(a + b) * (b + 1)$.

Das Aufbauen eines solchen Berechnungsgraphen ist gängige Praxis im Bereich von Deep Learning Frameworks und wird von sehr vielen Frameworks aus Gründen der Optimierungsmöglichkeiten und der automatischen Differenzierung eingesetzt. (siehe Abschnitt 3.2).

Mithilfe des Graphen werden zuerst alle Aktivierungswerte des Vorwärtspfades berechnet und gespeichert, denn diese werden beim automatischen Differenzieren beim Berechnen der partiellen Ableitungen erneut benötigt. Eine *partielle Ableitung* bestimmt die Steigung einer Funktion, bezogen auf die Änderung einer Variable. Beim Backpropagation Algorithmus werden die partiellen Ableitung eingesetzt, um, in Abhängigkeit jeder einzelner der Variablen der Fehlerfunktion, sprich aller gewichteter Verbindungen des Neuronalen Netzes, die Steigung, also die Geschwindigkeit der Fehleränderung, zu berechnen. Der hieraus resultierende globale Gradient wird anschließend dazu verwendet, eine Iteration des Trainings abzuschließen, indem dieser mithilfe der Update- Regel aus Gleichung 2.5 auf die Trainingsparameter angewendet wird.

2.1.4 Unterscheidung diverser Aktivierungsfunktionen

Wie im Abschnitt 2.1.1 beschrieben ist, besteht jedes Neuron aus einer Funktion, die die Netto-Eingabe und einer zweiten, die den Aktivierungswert berechnet. Im Gegensatz zum Skalarprodukt der Eingänge, ist die Aktivierungsfunktion nicht linear. Erst mithilfe dieser ist es dem Neuronalen Netz als gesamtes möglich, eine beliebige Funktion nachzubilden.

[4] Informationen zum automatischen Differenzieren: `https://goo.gl/F1Oe0Y`

Im Regelfall wird eine Aktivierungsfunktion für alle Neuronen einer Schicht verwendet. Die Wahl dieser hat dabei nicht nur starke Auswirkungen auf die Performanz des Netzes, sondern beeinflusst auch die Geschwindigkeit des Trainings. [10]
In der Tabelle 2.1 ist eine Übersicht der derzeit gängigsten Aktivierungsfunktionen zu sehen, deren Verläufe zusätzlich in der Abbildung 2.6a gezeichnet sind. Die am längsten ver-

Aktivierungsfunktion	Funktion	Wertebereich
Sigmoid	$\phi(x) = \frac{1}{1+e^{-x}}$	$[0, 1]$
Tanh	$\phi(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$[-1, 1]$
ReLU	$\phi(x) = max(0, x)$	$[0, \infty)$
Identity	$\phi(x) = x$	$(-\infty, \infty)$

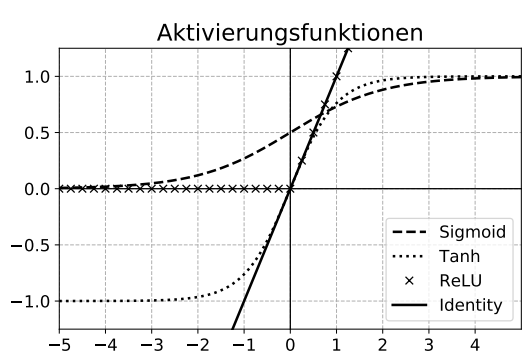
Tabelle 2.1: Auflistung diverser gängiger Aktivierungsfunktionen und deren Wertebereich.

wendete Aktivierungsfunktion ist die *Sigmoid* Funktion, denn durch ihren Wertebereich, der zwischen 0 und 1 liegt, ähnelte diese sehr gut den ursprünglichen Vorstellungen eines Neurons, das einen binären Ausgang besaß. Durch die Weiterentwicklung der künstlichen Neuronen und der zunehmenden Entfernung von ihren natürlichen Vorbildern, veränderte sich auch dessen Aktivierungsfunktion. Es stellt sich heraus, dass der *Hyperbolische Tangens*, oder kurz *Tanh*, sich als Aktivierungsfunktionen in den meist besser eignet. Dies hängt vor allem damit zusammen, dass der Wertebereich zwischen -1 und 1 ist, denn dadurch ist die Ausgabe der Schicht normalisiert, das bedeutet er ist Null-zentriert, was sich positiv auf das Training auswirkt. [10]

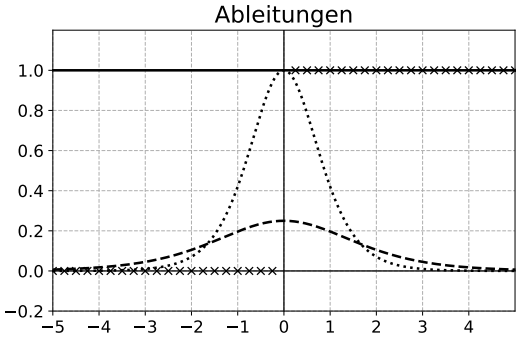
Beide, sowohl die Sigmoid Funktion als auch Tanh, besitzen eine ausgeprägte "S"- Förmige Kurve, was bedeutet, dass die Funktionen für besonders große und besonders kleine Werte in eine Sättigung übergehen. Das hat zur Folge, dass, wie in Abbildung 2.6b zu erkennen ist, die Ableitungen in diesem Bereich fast Null sind. Bei einem gradientenbasierten Training, wie es bei Neuronalen Netzen verwendet wird, führt dies dazu, dass keine oder nur extrem kleine Fortschritte möglich sind. Denn, wenn sich die Aktivierungsfunktionen während des Trainings im Bereich der Sättigung befinden, benötigt das Training viel Zeit, da nur sehr kleine Änderungen an den Gewichten vorgenommen werden. [10]

Die meist verwendete Aktivierungsfunktion ist derzeit die *rectified linear unit* oder kurz *ReLU*. Sie hat nicht nur den Vorteil, dass sie keinen Sättigungsbereich hat und somit in den häufig dafür sorgt, dass das Training deutlich schneller konvergiert, auch ist die Berechnung ihrer Funktionswerte extrem einfach, was bei sehr langer Trainingsdauer schnell zum Tragen kommt.

Die letzte, die *Identitätsfunktion*, wird meist für die Eingangsschicht verwendet, sie hat lediglich die Aufgabe die Eingangswerte in das Neuronale Netz zu speisen. Wie Eingangs erwähnt, hätte das Verwenden dieser linearen Funktion sonst keinerlei Mehrwert für das Neuronale Netz, sondern würde im Gegenteil nur den Rechenaufwand erhöhen.



(a) Kurven diverser Aktivierungsfunktionen.



(b) Kurven der Ableitungen diverser Aktivierungsfunktionen.

Abbildung 2.6: Diverse Aktivierungsfunktionen und deren Ableitungen.

2.2 Convolutional Neural Networks

Heutige CNNs sind die Weiterentwicklung des *Neocognitrons*, das 1980 von Kunihiko Fukushima vorgestellt wurde. In Abbildung 2.7 ist zu erkennen, dass bereits Fukushimas Entwurf auf der Idee basierte, dass ein Neuron nicht mehr mit allen der vorherigen Schicht verbunden sind, sondern nur noch mit wenigen, die als Kreis dargestellt sind. Wie er in seiner Veröffentlichung klärt, ist diese Architektur ebenfalls durch das Gehirn inspiriert, genauer vom *visuellen Cortex*. [9] Der visuelle Cortex ist für die visuelle Wahrnehmung verantwortlich[5], weshalb CNNs sehr häufig für das Erkennen von Inhalten in Bildern verwendet werden. Im Folgenden wird beschrieben, wie ein CNN auf Bildern arbeitet, dabei werden diese als zweidimensionale Eingangsdaten betrachtet.

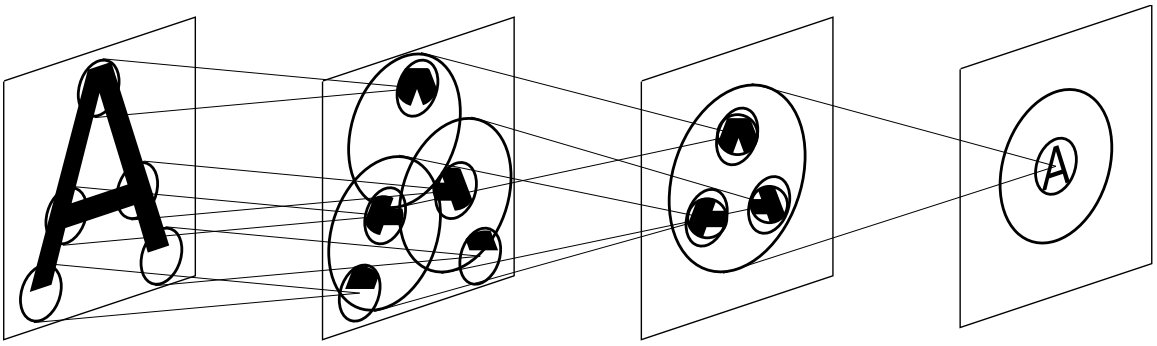


Abbildung 2.7: Funktionsweise des Neocognitron, angelehnt an [9].

Das erste CNN, wie sie heute implementiert werden, wurde 1998 von Yann LeCun, Léon Bottou, Yoshua Bengio und Patrick Haffner unter dem Namen *LeNet-5* vorgestellt. Dieses wurde weitverbreitet für das Erkennen von handgeschriebenen Zahlen eingesetzt. LeCun et al. haben bei dessen Entwicklung zwei neue Schichtarten entworfen und verwendet. [17]

[5] Informationen zum visueller Cortex: `https://goo.gl/YiGkEt`

Im Folgenden wird die Funktions- und Einsatzweise der von LeCun et al. entwickelten *Convolutional Schicht* und *Pooling Schicht* vorgestellt.

2.2.1 Erläuterung der Convolutional Schicht

Die Convolutional Schicht ist die wichtigste, die für das Bilden eines CNNs benötigt wird. Der Name dieser Schicht kommt von der mathematischen Operation *Convolution*, zu Deutsch *Faltung*, die diese Schicht ausführt. [10] Bei der Faltung werden zwei Funktionen schrittweise übereinander geschoben, mathematisch kann dieser Vorgang als Produkt der beiden Funktionen beschrieben werden[6].

Wie im Abschnitt 2.2 beschrieben und in Abbildung 2.8 zu sehen ist, ist ein Neuron nicht mit jedem der vorherigen Schicht, sondern nur mit einer bestimmten Anzahl, die als Rechteck angeordnet sind, verbunden. Dieses wird auch *aufnahmefähiges Feld* (engl. *receptive field*) genannt, seine Höhe wird mit f_h und die Breite mit f_w beschrieben. Weiter lässt sich gut erkennen, dass der Abstand zwischen den aufnahmefähigen Feldern zweier nebeneinander liegender Neuronen in diesem Beispiel genau eins beträgt, dieser Abstand nennt man *Schrittgröße* (engl. *stride*). Analog zum aufnahmefähigen Feld wird die Schrittgröße mit s_h für die vertikale Richtung und s_w für die horizontale beschrieben. Angenommen ein Neuron ist in Zeile i und Spalte j positioniert, dann ist es mit allen Neuronen, die in der Zeile $i * s_h$ bis $i * s_h + f_h - 1$ und der Spalte $j * s_w$ bis $j * s_w + f_w - 1$, der vorherigen Schicht, angeordnet sind, verbunden. [10]
Durch das Verschieben des aufnahmefähigen Felds ist es möglich, dass, zum Beispiel durch ein zu groß gewähltes Feld oder eine ungeschickte Schrittgröße, ein aufnahmefähiges Feld nicht komplett gefüllt wird und somit keine Berechnungen durchgeführt werden können. In einem solchen Fall können unterschiedliche Strategien gewählt werden, die in [10] erläutert sind. Da die Wahl der Strategie für diese Arbeit nicht weiter von Bedeutung ist, wird davon ausgegangen, dass ein solcher Fall nicht eintritt und ist daher nicht weiter beschrieben.

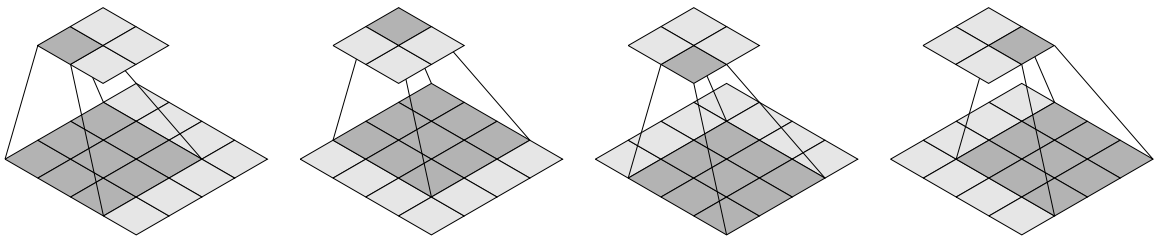


Abbildung 2.8: Funktionsweise einer Convolutional Schicht.

Die Aktivierungswerte der vorherigen Schicht stellen bei der Faltungsoperation die Werte der ersten Funktion dar, die zweite wird durch die Gewichte der Verbindungen des Neurons repräsentiert. Das Übereinanderschieben der beiden Funktionen wird bei einer Convolutional Schicht dadurch modelliert, dass alle Neuronen der selben Schicht die gleichen Gewichte verwenden.
Die Netto-Eingabe eines Neurons einer Convolutional Schicht berechnet sich durch das

[6] Informationen zur Operation Faltung: `https://goo.gl/SQJipd`

Frobenius-Skalarprodukt[7], das wie in Gleichung 2.6 zu sehen ist, berechnet wird, wobei das aufnahmefähige Feld **X** und die Gewichte **W** jeweils als Matrizen interpretiert werden.

$$net_{i,j} = \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} x_{i',j'} w_{u,v} + b \quad mit \begin{cases} i' = i * s_h + u \\ j' = j * s_w + v \end{cases} \tag{2.6}$$

Die Berechnung der Netto-Eingabe, die in Gleichung 2.6 zu sehen ist, kann man bildlich, wie in Abbildung 2.9 zu sehen, visualisieren. Die beiden Matrizen **X** und **W** werden Elementweise miteinander multipliziert und anschließend aufsummiert, wobei $x_{i',j'}$ dem Aktivierungswert des Neurons in der Zeile i und Spalte j der vorherigen Schicht entspricht.
An diesem Punkt ist auch leicht nachzuvollziehen, wieso die Gewichte einer Convolutional

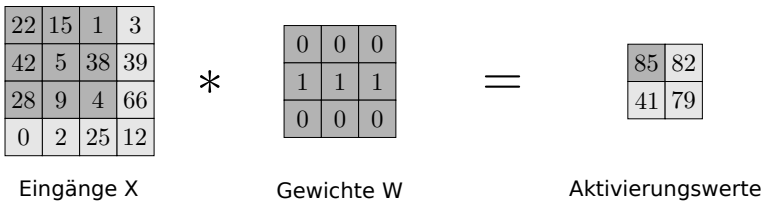


Abbildung 2.9: Visualisierung des Frobenius-Skalarprodukt.

Schicht häufig als *Filter* bezeichnet werden. Repräsentieren die Gewichte beispielsweise einen Filter, der horizontale Kanten in einem Bild erkennt, dann wird ein Neuron besonders aktiv sein, wenn sein aufnahmefähiges Feld möglichst genau seinem Filter, also einer horizontalen Kante, entspricht. Analog dazu wäre dasselbe Neuron nicht aktiv, wenn in seinem aufnahmefähigen Feld eine vertikale oder gar keine Kante liegt. Eine weiterer Bezeichnung für Filter ist *Faltungskern*, die vom Englischen Fachbegriff *convolution kernel* oder kurz *kernel*, abgeleitet ist. [10]

Es stellt sich die Frage, welche Filter für die Domäne der Eingangsdaten besonders gut geeignet sind. Da die Filter eine andere Bezeichnung für die Gewichte der Convolutional Schicht sind, werden während des Trainings des CNNs, das im Abschnitt 2.1.3 beschrieben ist, die besten Filter selbstständig erlernt.
Eine weitere Besonderheit der Convolutional Schicht ist, dass ein Filter, der ein bestimmtes Merkmal eines Bildes erkennt, unabhängig von dessen Position funktioniert. Das bedeutet, dass ein CNN robust gegenüber leichten Verschiebungen des Eingangsbildes ist. [17]

Im bisherigen Verlauf dieses Kapitels wurden die Eingangsbilder betrachtet, als würden sie aus zwei Dimensionen bestehen, einer Breite und Höhe. In den meisten Fällen besitzen Bilder noch eine dritte Dimension, den sogenannten *Kanal* (engl. *channel*), der die Farbwerte ***R**ot*, ***G**rün* und ***B**lau* zur Verfügung stellt. Betrachtet man alle Aktivierungswerte einer Convolutional Schicht, wird diese neue Repräsentation des Eingangsbildes meist als *feature map* bezeichnet, denn sie zeigt dasselbe Bild nach der Anwendung eines bestimmten Filters, nach dem bestimmte Merkmale des Bildes hervorgehoben sind. Da es sich in den meisten praktischen Anwendungsfällen als sinnvoll erwiesen hat, mehrere Filter auf das Eingangsbild, beziehungsweise auf irgendeine Zwischenrepräsentation des Bildes, anzuwenden, werden meist mehrere Neuronenschichten gestapelt, die mehrere feature

[7] Informationen zum Frobenius-Skalarprodukt: `https://goo.gl/DyWniR`

maps berechnen. Wenn die Eingangsdaten einer Convolutional Schicht dreidimensional sind, unabhängig davon, ob es sich dabei um die drei ursprünglichen RGB Kanäle oder mehrere feature maps handelt, dann ist ein Neuron mit allen Neuronen, auch die entlang der dritten Dimension, seines aufnahmefähigen Feldes verbunden. Hierdurch ändert sich die Berechnung der Netto-Eingabe, wie in Gleichung 2.7 zu sehen. [10]

$$net_{i,j,k} = \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} w_{u,v,k',k} + b_k \quad mit \begin{cases} i' = i * s_h + u \\ j' = j * s_w + v \end{cases} \tag{2.7}$$

In Gleichung 2.7 enthält die Matrix der Eingänge **X** entlang ihrer dritten Dimension die insgesamt $f_{n'}$ vielen Elemente, die den Aktivierungswerten der feature maps der vorherigen Schicht entsprechen. Ähnlich wie zuvor werden die Elementweisen Produkte zwischen den ersten beiden Dimensionen von **X** und dem Filter **W** berechnet und aufsummiert. Hier wird dieser Schritt zusätzlich für jedes Element der dritten Dimension von **X** wiederholt und somit die Gesamtsumme aller Produkte gebildet. Diese Schritte beziehen sich auf genau einen Filter k der Convolutional Schicht, in Abbildung 2.10 ist diese Berechnung ebenfalls visualisiert. Hier wird deutlich, dass Filter **W** mit allen $f_{n'}$ vielen Elementen der vorherigen Schicht verbunden ist.

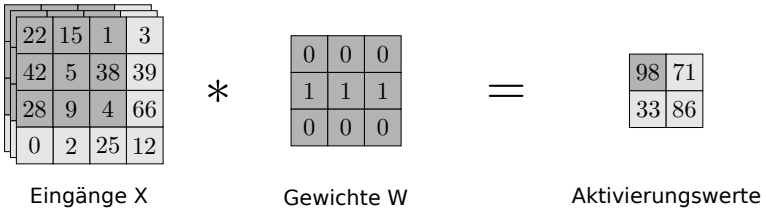


Abbildung 2.10: Visualisierung des Frobenius-Skalarproduktes mit drei Dimensionen.

Durch die Faltung wird die Höhe und Breite des Bildes nur geringfügig verringert, jedoch nimmt der Wert der dritten Dimension durch das Berechnen vieler feature maps stark zu. Angenommen eine Convolution Schicht berechnet 100 feature maps mit 5×5 Filter und einer RGB Bildgröße von 80×80 Pixeln, dann werden $(5 * 5 * 3) * 100 = 7500$ Gewichte benötigt. Im Gegensatz zu den 122,88 Millionen Gewichten, die man benötigen würde, wenn alle der 80×80 Neuronen mit jedem der $80 \times 80 \times 3$ Eingängen verbunden wären, sind dies vergleichsweise wenige.
Betrachtet man jedoch den benötigten Speicher, wird ein anderer Trend deutlich. Wenn die Aktivierungswerte der Neuronen als 32-bit floats gespeichert sind, dann benötigt die Convolutional Schicht genau $100 * 80 * 80 * 32 = 20480000$ Bit RAM, was gut 2,4 MB RAM entspricht. Eine voll verbundene Schicht hingegen würde nur $80 * 80 * 32 = 204800$ Bit RAM, was nur etwa 0,0244 MB RAM entspricht, in Anspruch nehmen.

Beim Verwenden eines CNNs mit mehreren Schichten kann der belegte RAM nach dem Errechnen der feature maps der darauffolgenden Schicht wieder freigegeben werden, wodurch maximal der Speicher der beiden größten Schichten in Summe benötigt wird.
Um eine effiziente Implementierung des Backpropagation Algorithmus zu gewährleisten müssen beim Training alle Werte der Schichten zwischengespeichert werden. Hinzu kommt, dass beim Trainieren meist mit mehreren Trainingsdaten zeitgleich gearbeitet

wird, was dazu führt, dass für mehrere Trainingsdaten alle feature maps gleichzeitig im Speicher gehalten werden müssen, was sehr schnell zu einer sehr großen Menge an Daten führt. [10]

Die Lösung für dieses Problem ist die Pooling Schicht, diese wird im nächsten Abschnitt näher erläutert.

2.2.2 Erläuterung der Pooling Schicht

Wie bereits im Abschnitt 2.2.1 angedeutet wurde, wird die Pooling Schicht vor allem zur Reduktion des benötigten Speichers verwendet. Ein weiterer positiver Effekt ist, dass zusätzlich auch die Anzahl der zu trainierenden Gewichte verringert wird. Das Verwenden einer Pooling Schicht hat noch weitere Effekte, die für diese Arbeit jedoch nicht weiter von Interesse sind. [10]

Ähnlich wie bei einer Convolutional Schicht sind auch bei einer Pooling Schicht die Neuronen mit nur einer bestimmten Anzahl Neuronen der vorherigen Schicht verbunden. Diese sind ebenfalls als Rechteck angeordnet und werden auch hier *aufnahmefähiges Feld* genannt. Ebenfalls besitzt eine Pooling Schicht eine Schrittweite, mit der bestimmt wird, um wie viel das aufnahmefähige Feld des nächsten Neurons verschoben ist.
Um den Aktivierungswert eines Neurons in der Pooling Schicht zu berechnen, wird meist entweder die *Min-*, *Max-* oder *Durchschnittsfunktion* verwendet. In der Abbildung 2.11 ist die Funktionsweise einer Max Pooling Schicht zu sehen, die ein aufnahmefähiges Feld von 2×2 und ebenfalls eine Schrittweite 2×2 besitzt. Außerdem arbeitet jede Pooling Schicht

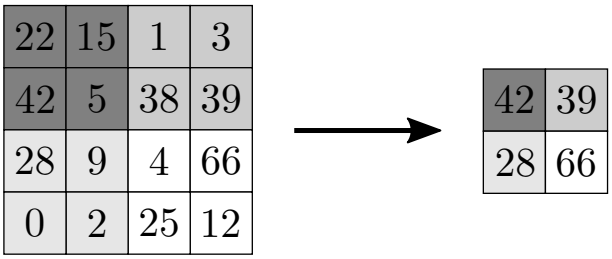


Abbildung 2.11: Visualisierung der Funktionsweise einer Pooling Schicht, angelehnt an `https://goo.gl/qDpCA8`.

auf jedem Kanal des Eingangsbildes oder auf jeder feature map separat, was dazu führt, dass nur die ersten beiden Dimension in ihrer Größe reduziert werden, die dritte jedoch vollständig erhalten bleibt. [10]

Beim Beispiel, das in Abbildung 2.11 zu sehen ist, werden drei der vier Werte je aufnahmefähigem Feld verworfen. Das bedeutet, dass die Pooling Schicht die Größe um das Vierfache verringert, oder anders ausgedrückt, es werden 75% der Eingabe verworfen.
Das Verwenden einer Pooling Schicht mit den, in diesem Beispiel verwendeten, Einstellungen ist eine sehr radikale Art, Speicher zu sparen, jedoch hat sich in der Praxis herausgestellt, dass dies, trotz der hohen Kompressionsrate, keine großen Nachteile mit sich bringt.

In diesem und dem vorherigen Abschnitt 2.2.1 wurden die beiden neuen, von LeCun et al. eingeführten [17], Schichtarten beschrieben. Im Folgenden wird erläutert wie diese in Form eines CNN zusammen spielen.

2.2.3 Aufbau von Convolutional Neuronal Networks

Das in Abbildung 2.12 dargestellte CNN zeigt den Aufbau des von LeCun et al. eingeführten Netzes LeNet-5. Auch bei moderneren CNNs ist die Verwendung der drei unterschiedlichen Schichtarten weiterhin gängig[8]. Durch das Verwenden von mehreren Convolutional und Pooling Schichtkombinationen wird versucht, unterschiedlich komplexe Merkmale auf mehrere Ebenen zu extrahieren. Die anschließenden voll verbundenen Schichten dienen zur Klassifizierung des Eingangsbildes. Mithilfe der Pooling Schichten wird, wie im Abschnitt 2.2.2 beschrieben, verhindert, dass der benötigte Speicher, besonders beim Trainieren, überhand nimmt.

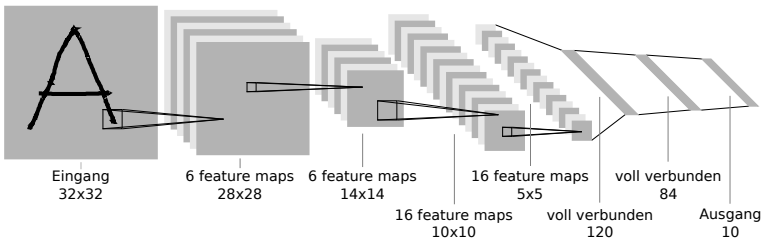


Abbildung 2.12: Aufbau des CNNs LeNet-5, angelehnt an [17].

Durch das Stapeln mehrerer Convolutional Schichten ist es dem CNN möglich, zu lernen auf unterschiedlichen Ebenen verschieden komplexe Merkmale der Eingangsbilder zu erkennen. Am Beispiel einer Gesichtserkennung, wie in der Abbildung 2.13 zu sehen, wurde zuerst gelernt, Pixelgruppierungen als verschiedene Kanten oder Ähnlichem, wie in Abbildung 2.13a, zu erkennen. Die nächst höhere Komplexitätsstufe nutzt eben diese Kanten, um Merkmale wie Augen, Nasen oder Ohren, wie in Abbildung 2.13b, zu unterscheiden. Mit steigender Anzahl der Schichten steigt auch die Komplexität der gelernten Merkmale, bis hin zu kompletten Gesichtern, die in Abbildung 2.13c zu sehen sind.

Die sehr gute Performanz, die die Convolutional Neuronalen Netze erreichen, wenn sie auf Bilder angewendet werden, lässt sich nicht auf eine Besonderheit reduzieren. Vielmehr wird diese durch das Zusammenspiel der einzelnen Schichten erreicht. Durch das Einsetzen sehr vieler unterschiedlicher Filter in einer Schicht is es möglich, dass das Netz während des Trainings die besten Filter für die jeweilige Anwendungsdomäne erlernt. Diese erlernten Filter können anschließend extrem effizient eingesetzt werden, denn durch die Convolution Operation sind die erlernten Merkmale unabhängig von ihrer Position im Bild. Durch das Stapeln mehrerer dieser Schichten können sehr effiziente Netze programmiert werden, die, wie in Abbildung 2.13 zu sehen ist, durch das Aggregieren vieler erlernter Merkmale eines Bildes sehr gute Ergebnisse erzielen können.

[8] Auflistung und Erläuterung diverser moderner CNNs: `https://goo.gl/hDYDeY`

(a) Einfache Merkmale, wie unterschiedlich gerichtete Kanten.

(b) Komplexere Merkmale, die bestimmten Teile eines Gesichtes ähneln.

(c) Sehr komplexe Strukturen, die diversen Gesichtern gleich kommen.

Abbildung 2.13: Darstellung der hierarchisch gelernten Merkmale eines CNNs, visualisiert von Lee et al. in [18].

2.3 Recurrent Neural Networks

Im Jahr 1982 veröffentlichte John Joseph Hopfield die erste Umsetzung eines Recurrent Neural Network (RNN), das heute als *Hopfield Netzwerk*[9] bekannt ist und als Vorreiter moderner RNNs gilt. Dieses besteht aus Neuronen, wie sie McCulloch und Pitts [21] entwickelten, die lediglich zwei Zustände (*aktiv*/ *inaktiv*) einnehmen können. Der Unterschied zu einem feed-forward Netz, wie es im Abschnitt 2.1.2 beschrieben ist, besteht hauptsächlich darin, dass die Neuronen nicht mit allen der nachfolgenden, sondern mit allen Neuronen der selben Schicht, außer sich selbst, verbunden sind. Die Abbildung 2.14 verdeutlicht diesen Aufbau. Hopfield zeigte, dass ein solches Netz sich wie ein Assoziativspeicher verhält und somit Informationen speichern kann. [14] Heute wird das Hopfield Netz dazu verwendet um unvollständige oder beschädigte Signale, wie zum Beispiel Bilder, wiederherzustellen [23].

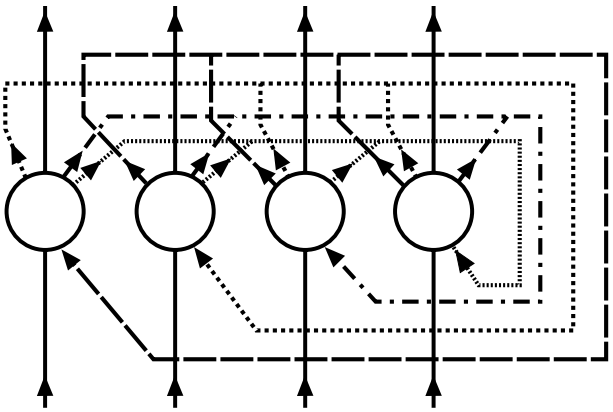


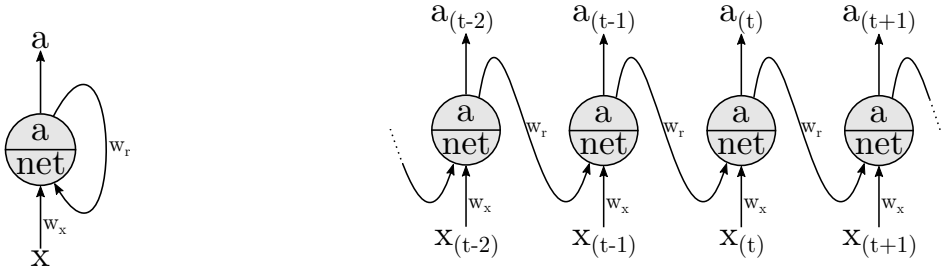
Abbildung 2.14: Aufbau eines Hopfield Netzwerks, angelehnt an: `https://goo.gl/Bk5M2G`.

In den folgenden Abschnitten wird zuerst der Aufbau und anschließend die unterschiedlichen Anwendungs- und Einsatzmöglichkeiten vorgestellt.

[9] Informationen zum Hopfield Netzwerk: `https://goo.gl/9qfkKB`

2.3.1 Aufbau und Training von Rekurrenten Neuronalen Netzwerken

Moderne RNNs und das ursprüngliche Hopfield Netzwerk unterscheiden sich durch die verwendete Aktivierungsfunktion und dadurch, dass, wie es für moderne Neuronen üblich ist, jeder Eingang ein dediziertes Gewicht besitzt. Die ursprüngliche Schwellwertfunktion, wie sie McCullochs und Pitts' Neuronen verwendeten, wurde bei modernen rekurrenten Neuronen durch eine Aktivierungsfunktion, wie sie im Abschnitt 2.1.4 beschrieben ist, ersetzt. Abbildung 2.15a zeigt ein rekurrentes Neuron mit den in diesem Abschnitt wichtigen Notationen. Durch den rückkoppelnden Aufbau bedingt, wirken sich sequenzielle Eingangsdaten auf die jeweilig nachfolgenden aus. Um diesen Vorgang besser zu veranschaulichen, wird häufig eine Darstellung verwendet, die das Neuron, wie es in Abbildung 2.15b zu sehen ist, entlang der Zeitachse ausrollt (engl. unrolling through time). [10]



(a) Ansicht eines rekurrenten Neurons.

(b) Dasselbe Neuron entlang der Zeitachse ausgerollt.

Abbildung 2.15: Darstellung eines rekurrenten Neurons und dessen ausgerollte Ansicht, angelehnt an [10].

Ein weiterer Unterschied im Vergleich zum ursprünglichen Hopfield Netzwerk ist die Tatsache, dass alle Neuronen einer Schicht zu allen, also auch zu sich selbst, rückgekoppelt sind. Mithilfe der Notationen, die in Abbildung 2.15 zu sehen sind, kann der Aktivierungswert eines Neurons mit den Gleichungen 2.8 und 2.9 berechnet werden. [10]

$$net_{(t)} = x_{(t)} * w_x + \widehat{\mathbf{a_{(t-1)}}\mathbf{w_r}} + b = x_{(t)} * w_x + \sum_{i=1}^{n} a_{(t-1)i} w_{ri} + b \tag{2.8}$$

$$a_{(t)} = \phi(net_{(t)}) = \phi(x_{(t)} * w_x + \widehat{\mathbf{a_{(t-1)}}\mathbf{w_r}} + b) = \phi(x_{(t)} * w_x + \sum_{i=1}^{n} a_{(t-1)i} w_{ri} + b) \tag{2.9}$$

Hierbei stehen die verwendeten Variablen für Folgendes:

- t entspricht dem Zeitschritt der Berechnung
- n entspricht der Anzahl Neuronen der Schicht
- $x_{(t)}$ ist der Eingangswert des Neurons zum Zeitpunkt t
- w_x ist das dedizierte Gewicht des Eingangs x

- $\widehat{\mathbf{a_{(t-1)}}}$ ist ein Vektor, der die Ausgangswerte $a_{(t-1)1}, a_{(t-1)2}, \ldots, a_{(t-1)n}$ des vorherigen Zeitschritts enthält, oder 0 wenn $t = 0$
- $\widehat{\mathbf{w_r}}$ ist ein Vektor, der die Gewichte $w_{r1}, w_{r2}, \ldots, w_{rn}$ der rekurrenten Verbindungen enthält
- b ist der Bias Wert der Schicht
- ϕ is die Aktivierungsfunktion der Schicht.

Um ein RNN zu trainieren, wird ein modifizierter Trainingsalgorithmus eingesetzt. Dieser wird *backpropagation through time (BPTT)* genannt und unterscheidet sich zum herkömmlichen Backpropagation Algorithmus lediglich darin, dass zuerst das Netzwerk ausgerollt wird. Anschließend kann der Backpropagation Algorithmus, wie er im Abschnitt 2.1.3.2 beschrieben ist, eingesetzt werden. [10]

Ein RNN kann wegen der Möglichkeit, es entlang der Zeitachse auszurollen auf unterschiedliche Arten eingesetzt werden, im folgenden Abschnitt sollen diese, sowie unterschiedliche Einsatzgebiete, dargestellt werden.

2.3.2 Anwendungsarten und Einsatzmöglichkeiten von Rekurrenten Neuronalen Netzwerken

Dadurch, dass eine Eingabe auch alle weiteren Eingaben des Netzwerks beeinflusst, ergeben sich unterschiedliche Anwendungsarten. Die vier gängigsten, wie sie Abbildung 2.16 zeigt, so wie deren Einsatzmöglichkeiten werden hier vorgestellt. Die Rechtecke stehen hierbei für das gesamte rekurrente Netz, das sowohl aus mehreren Schichten als auch aus mehreren Neuronen pro Schicht bestehen kann.

Sequenz zu Sequenz Anwendung

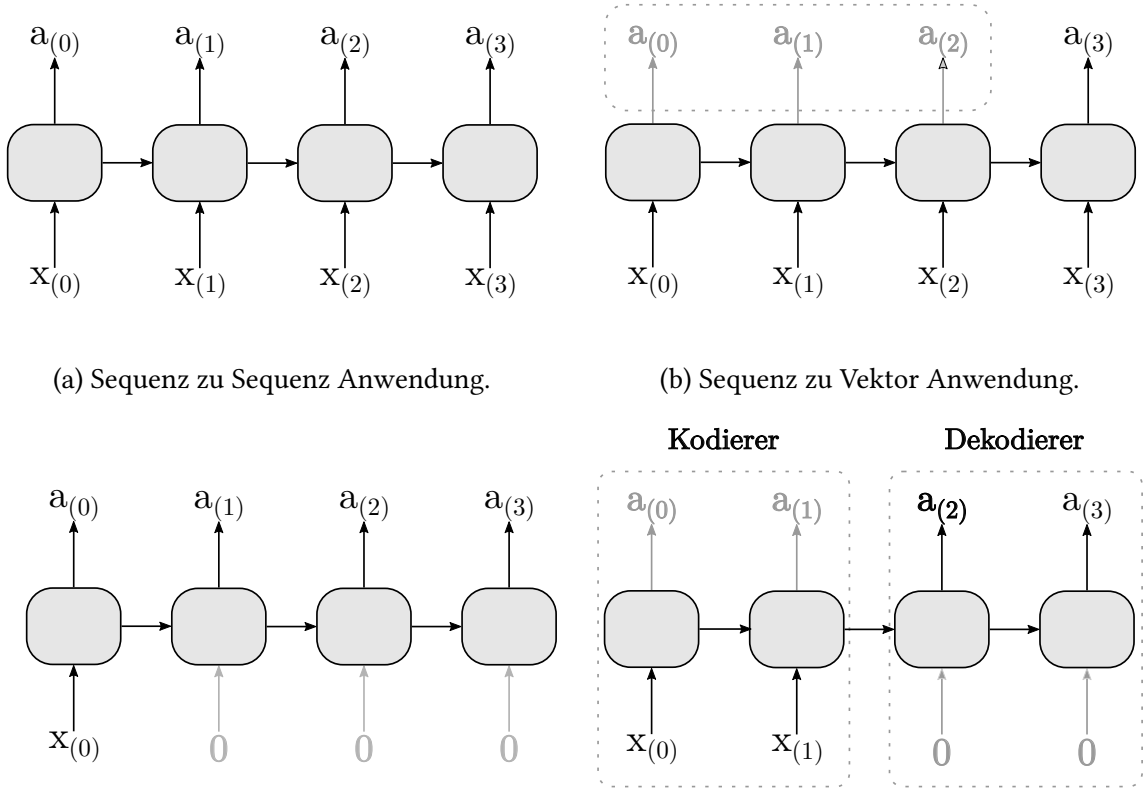
Wenn in jedem Zeitschritt sowohl ein Eingangsdatum ein-, als auch ein Ausgangsdatum ausgegeben wird, die weiter verarbeitet werden, nennt man das Netzwerk ein *Sequenz zu Sequenz* RNN. Diese Art von Verwendung kann eingesetzt werden, wenn Daten, die einer Zeitreihe angehören, vorhergesagt werden sollen. Anwendungsfälle wären beispielsweise das Vorhersagen von Aktienkursen oder Temperaturen. [10]

Sequenz zu Vektor Anwendung

Sequenz zu Vektor RNN, wie es in Abbildung 2.16b zu sehen ist, werden häufig im Bereich der Textverarbeitung eingesetzt. Zum Beispiel kann ein Text sequenziell Wort für Wort in das Netzwerk eingegeben werden. Das Netzwerk hat anschließend die Aufgabe, ein oder mehrere passende Schlagworte zu finden. Dabei wird nur die letzte Ausgabe des Netzwerks betrachtet und die, der vorherigen Zeitschritte ignoriert. [10]

Vektor zu Sequenz Anwendung

Ein *Vektor zu Sequenz* RNN, das in Abbildung 2.16c dargestellt ist, funktioniert umgekehrt. Nach der Eingabe eines Datums im ersten Zeitschritt werden in allen weiteren keine Daten



(a) Sequenz zu Sequenz Anwendung.

(b) Sequenz zu Vektor Anwendung.

(c) Vektor zu Sequenz Anwendung.

(d) Verzögerte Sequenz zu Sequenz Anwendung.

Abbildung 2.16: Unterschiedliche Anwendungsarten eines rekurrenten Neuronalen Netzwerks, angelehnt an [10].

mehr eingegeben. Ein Anwendungsfall wäre das Erstellen einer Beschreibung für ein eingegebenes Bild. Diese wird sequenziell Wort für Wort in jedem Zeitschritt ausgelesen. [10]

Verzögerte Sequenz zu Sequenz Anwendung

Die letzte Anwendungsart, die in Abbildung 2.16d zu sehen ist, wird in modernen Anwendungen eingesetzt, die ein Eingabetext in eine andere Sprache übersetzen. Diese *verzögerten Sequenz zu Sequenz* Netze funktionieren für diese Aufgabe deshalb besser als Sequenz zu Sequenz Netzwerke, weil sie den Text zuerst in eine Zwischenrepräsentation kodieren und anschließend in der zweiten Hälfte in die zu übersetzende Sprache wieder dekodieren. Das liegt vor allem an der Tatsache, dass einzelne Worte im Kontext eines oder mehreren Sätzen häufig eine andere Bedeutung annehmen. Mithilfe der Zwischenrepräsentation, die in den rekurrenten Verbindungen gespeichert ist, wird somit zuerst versucht, die Bedeutung des Satzes abzubilden und diese anschließend in einer anderen Sprache komplett neu zu modellieren.

Schon bei Anwendungsfällen, bei denen nicht in jedem Zeitschritt eine Eingabe getätigt

wird, aber weiterhin Ausgaben vom Netzwerk generiert werden, wird deutlich, dass RNNs auch nutzbar sind, um Daten zu generieren. Andere Projekte fokussieren den generativen Aspekt von RNNs deutlich. Es wurden Netze implementiert, die Texte im Stil von Shakespeare oder ähnlich dem Aufbau eines Wikipedia Artikels generieren. Weitere textbasierte Beispiele finden sich in einem Blog-Artikel[10]. Andere versuchen mithilfe von RNNs, Bilder zu malen, oder sogar ganze Musikstücke zu komponieren [12, 1].

2.4 Horizontale Skalierungsmöglichkeiten des Trainings von Neuronalen Netzen

Vor allem beim Trainieren von großen Neuronalen Netzwerken kann das Training extrem lange dauern, um eine angemessene Performanz des Netzes zu erreichen. Hierzu ist es nötig, dass das Trainieren skalierbar ist. Ganz allgemein bedeutet *Skalierbarkeit*, dass durch das Hinzufügen von weiteren Ressourcen die Leistung des Systems gesteigert werden kann. Der Fokus dieser Arbeit liegt auf dem *horizontalen Skalieren* des Trainings. Beim horizontalen Skalieren der Rechenleistung wird die Anzahl paralleler Recheneinheiten erhöht, in dieser Arbeit liegt das Hinzufügen von Recheneinheiten in unterschiedlichen Rechnern im Fokus.

Auch innerhalb eines Rechners werden unterschiedliche Mittel verwendet, um das Training zu skalieren. Wie in den vorherigen Kapiteln zu erkennen ist, werden die Zustände der Neuronalen Netze als Matrizen abgebildet. Hierdurch wird deutlich, warum zum Trainieren sehr häufig anstelle von Central Processing Units (CPUs) ein oder meherere Graphics Processing Units (GPUs) verwendet werden, denn diese sind auf höchst parallele Verarbeitung von großen Matrizen ausgelegt. Beim horizontalen Skalieren des Trainings innerhalb einer Maschine gibt es unterschiedlichste Ansätze, die ebenso unterschiedliche Vor- und Nachteile haben können, für diese Arbeit sind sie jedoch nicht weiter von Bedeutung.

Beim horizontalen Skalieren über Rechnergrenzen hinweg gibt es zwei große Ansätze, die *Modellparallelität* und den Ansatz der *Datenparallelität*. Diese sind nicht auf die Anwendung mit mehreren Maschinen beschränkt, sondern können ebenfalls verwendet werden, wenn mit mehreren Recheneinheiten innerhalb einer Maschine gearbeitet wird. [10]

In den folgenden Abschnitten sind die beiden Ansätze der Parallelisierung des Trainings von Neuronalen Netzen beschrieben.

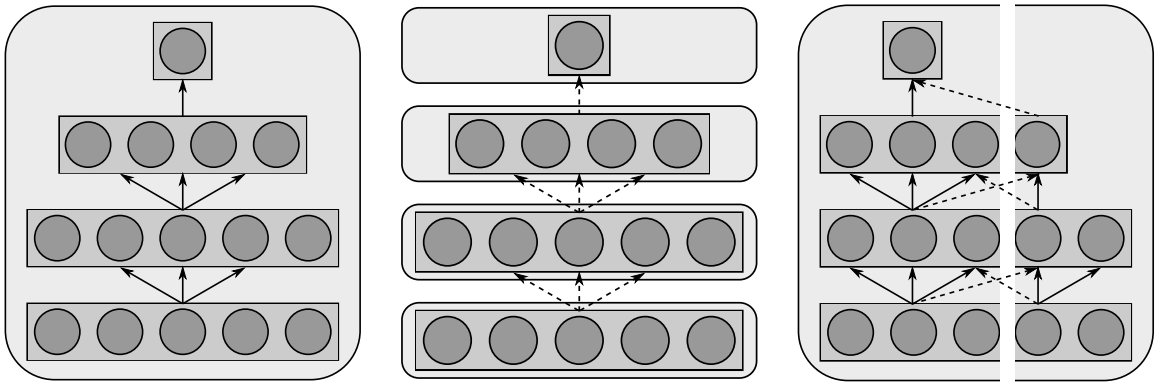
2.4.1 Ansatz und Idee der Modellparallelität

Bei den Beispielen, die in vorherigen Abschnitten verwendet wurden, sind die Neuronalen Netze bewusst überschaubar gewählt. Bei Netzwerken, die für den produktiven Einsatz konzipiert sind, werden in der Regel deutlich größere Netze, die viele Neuronen pro Schicht und meist sehr viele Schichten besitzen, verwendet. Wie im Abschnitt 2.2.1 anhand des Rechenbeispiels gezeigt ist, kann der beim Training benötigte Speicherbedarf, abhängig

[10] Blog-Artikel über die generativen Aspekte von RNNs: `https://goo.gl/FodLp5`

von der verwendeten Netzart, sehr groß werden, sogar zu groß für den Speicher eines einzigen Geräts.
Für genau diesen Fall, wenn das zu trainierenden Neuronale Netz zu groß ist, um als Ganzes im Speicher gehalten und trainiert zu werden, wird meist die Modellparallelität eingesetzt.

Die Idee der Modellparallelität ist es, das Neuronale Netz in kleinere Teile zu unter- und diese auf die zur Verfügung stehenden Maschinen zu verteilen. Wie in Abbildung 2.17 anhand der gestrichelten Pfeilen zu erkennen ist, müssen die Verbindungen des Neuronalen Netzes durch das Senden von Nachrichten zwischen den Maschinen ersetzt werden. Eben-



(a) Ursprüngliches Neuronales Netz.

(b) Horizontal zerteiltes Neuronales Netz.

(c) Vertikal zerteiltes Neuronales Netz.

Abbildung 2.17: Aufteilung eines voll verbunden Netzes, zum Verwenden der Modellparallelität, angelehnt an [10].

falls ist zu sehen, dass der Art und Weise, wie das Netzwerk zerlegt wird, keine Grenzen gesetzt sind, jedoch ist es für die Performanz des Trainings ein wichtiger Faktor. Bei einer schichtweisen Zerlegung eines voll verbundenen Netzes, wie es in Abbildung 2.17b zu sehen ist, müssen die unterschiedlichen Maschinen sehr häufig aufeinander warten, denn nach dem Berechnen der Aktivierungswerte einer Schicht müssen diese zuerst bei der Maschine, die für die darauffolgenden Schicht zuständig ist, vorhanden sein, bevor weiter gerechnet werden kann. Der Kommunikationsaufwand dieser Zerlegungsart ist jedoch sehr gering.
Durch das vertikale Zerlegen, wie es in Abbildung 2.17c zu sehen ist, können alle Maschinen zeitgleich arbeiten. Der limitierende Faktor ist hierbei der Kommunikationsaufwand, denn für jede Schicht müssen die Aktivierungswerte bidirektional ausgetauscht werden.

Bei Netzwerken, deren Schichten nicht voll verbunden sind, wie es zum Beispiel bei CNNs der Fall ist, kann meist eine sehr gute Beschleunigung erreicht werden, wenn die Zerlegung des Netzes für eine gleichmäßige Auslastung der Maschinen bei möglichst geringem Kommunikationsaufwand sorgt. Auch bei Netzwerken, deren Schichten sehr komplex sind, wie es bei den RNNs der Fall ist, kann mithilfe der Modellparallelität eine gute Beschleunigung des Trainings erreicht werden. Das Berechnen der Aktivierungswerte der einzelnen Schichten ist so rechenintensiv, dass die Latenzen der Kommunikationen meist nicht zum Tragen kommen. [10]

2.4.2 Ansatz und Idee der Datenparallelität

Bei der Horizontalen Skalierung durch die Datenparallelität ist es zwingend notwendig, dass das zu trainierenden Netzwerk als Ganzes im Speicher gehalten werden kann. Der Unterschied zur Modellparallelität besteht darin, dass jede Maschine ein von den anderen unabhängiges Netzwerk erstellt. Diese werden zeitgleich von den unterschiedlichen Maschinen mit unterschiedlichen Trainingsdaten trainiert. Damit die Parameter der einzelnen Netze nicht auseinander laufen und schlussendlich verschieden trainierte Neuronalen Netze erzeugt werden, werden in regelmäßigen Abständen die Trainingsparameter synchronisiert. Die Beschleunigung dieser Methode erfolgt dadurch, dass man in derselben Zeit mehr Trainingsdaten nutzen kann um die Änderungen der Trainingsparameter zu berechnen. [10]

In Abbildung 2.18 ist der schematische Ablauf des Trainings mit Datenparallelität zu sehen. Jede Maschine berechnet über eine Menge Trainingsdaten unabhängig von den

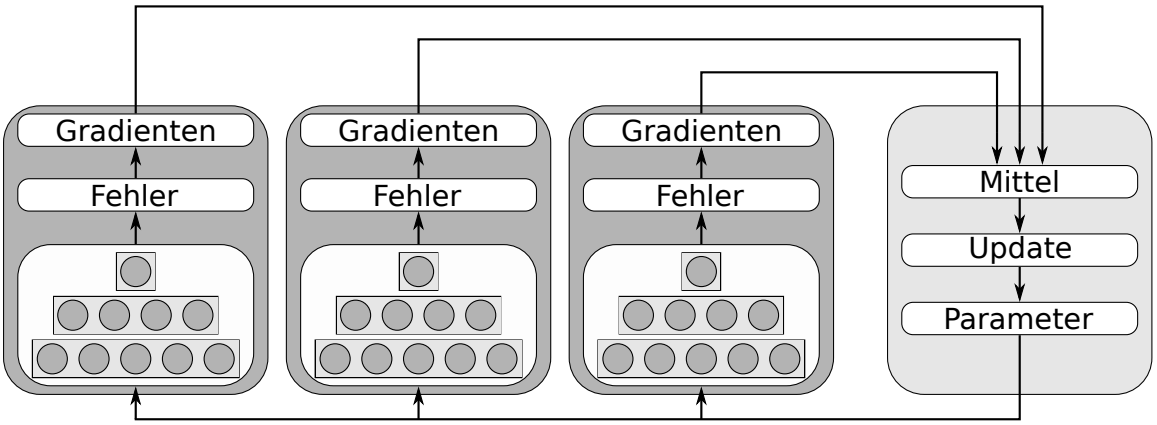


Abbildung 2.18: Schematischer Aufbau eines horizontal skalierten Trainings mit Datenparallelität, angelehnt an [10].

anderen den Fehler und die entsprechenden Gradienten, diese werden anschließend über alle Maschinen hinweg gemittelt und daraus die neuen Parameter berechnet. Dadurch, dass man sehr viel mehr Trainingsdaten zum errechnen des Gradienten verwendet, ist dieser auch sehr viel genauer. Dies hat zur Folge, dass das Training schneller konvergiert. [10]

Um die Änderungen der Parameter zu berechnen, gibt es zwei unterschiedliche Möglichkeiten, die in den folgenden Abschnitten beschrieben sind.

2.4.2.1 Synchrone Upates bei Datenparallelität

Wenn die Updates synchron ausgeführt werden, dann ist es unumgänglich, dass alle Maschinen auf die langsamste, oder auf die Nachricht, die am spätesten eintrifft, warten müssen. Erst anschließend ist es möglich, die Updates der Trainingsparameter zu berechnen.

Diese Methode hat zum Vorteil, dass die Updates immer über sehr viele Trainingsdaten berechnet werden und das Training somit in der Regel schnell konvergiert.

Dadurch, dass die Gradienten aller Maschinen gemittelt werden, senden alle Teilnehmer ihre Berechnungsergebnisse an einen zentralen Rechner, der diese Aufgabe übernimmt. Bei sehr vielen Maschinen kann daher die Netzwerkbandbreite des zentralen Rechners sehr schnell ausgelastet sein und somit die Performanz des Gesamtsystems einschränken. [10]

2.4.2.2 Asynchrone Upates bei Datenparallelität

Nachteile, die die Methode der Synchronen Updates besitzt, können durch das Ausführen von asynchronen Updates teilweise behoben werden, jedoch entsteht hierdurch ein weiterer.

Nach jedem Trainingsschritt berechnet eine Maschine unabhängig den Gradienten und sendet diesen ebenfalls an alle anderen. Obwohl hier insgesamt eine größere Menge Daten gesendet wird, finden diese Kommunikationen nicht zeitgleich statt, was dazu beiträgt, dass die Bandbreite des zentralen Rechners meist gleichmäßiger ausgelastet wird und die Performanz des Systems nicht beeinträchtigt. Ebenfalls müssen die Maschinen nicht gegenseitig aufeinander warten, was dazu führt, dass mehr Traningsschritte pro Zeit möglich sind.
Dadurch, dass alle Änderungen der Parameter an alle anderen Maschinen gesendet werden, kann es passieren, dass die Gradienten, die berechnet wurden, bei deren Anwendung auf die Trainingsparameter bereits nicht mehr korrekt sind. Diese ungültigen Gradienten nennt man *stale gradients*. In Abbildung 2.19 ist zu erkennen, dass der Gradient links berechnet wurde, als dieser, wegen zwischenzeitiger Änderungen, auf der rechten Seite angewendet wird, endet der Trainingsschritt bei einem schlechteren Ergebnis. Das Vorkommen von

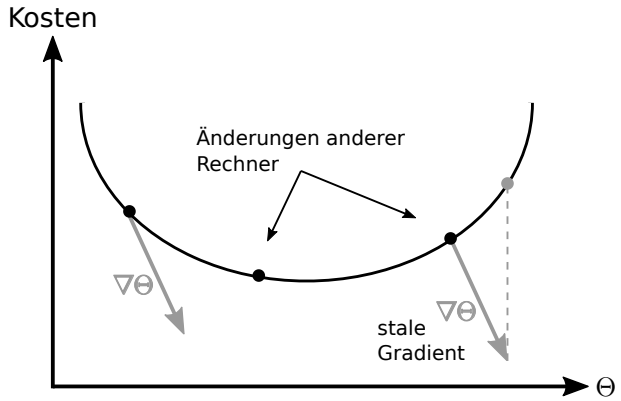


Abbildung 2.19: Stale Gradients, bei asynchronen Updates der Datenparallelität, angelehnt an [10].

stale gradients bei asynchronen Updates verlangsamen im Allgemeinen das Training. Wie Aurélien Géron in [10] zeigt, gibt es Möglichkeiten, die Auswirkungen dieser zu minimieren, was für diese Arbeit jedoch nicht weiter von Bedeutung ist. [10]

3 Vergleich aktueller Deep Learning Frameworks

In diesem Kapitel soll die Basis für den weiteren Verlauf der Arbeit geschaffen werden. Hierfür gilt es, zu definieren welche Eigenschaften die betrachteten Frameworks besitzen müssen. Nach der Vorstellungen einiger der vielversprechendsten, wird im letzten Abschnitt eine Auswahl an Frameworks getroffen, die in dieser Arbeit evaluiert werden.

3.1 Festlegung der Kriterien zur Auswahl der Deep Learning Frameworks

Um eine Auswahl der Deep Learning Frameworks für die folgende Evaluation treffen zu können, werden in diesem Abschnitt die notwendige Kriterien definiert.

Der Fokus dieser Arbeit liegt auf Frameworks, die generisch und mit der Programmiersprache Python einsetzbar sind. In diesem Kontext bedeutet generisch, dass die Frameworks Application Programming Interfaces (APIs) für mindestens Neuronale Netzwerke, CNNs und RNNs zur Verfügung stellen müssen. Zur besseren Übersicht sind die Anforderungen zusammen mit deren Priorität in Tabelle 3.1 dargestellt. Eine *hohe* Priorität bedeutet, dass es sich um ein notwendiges Kriterium handelt. Eine Anforderung die *mittel* priorisiert ist, ist keine zwingende Voraussetzung für diese Arbeit, würde sich aber positiv auswirken. Beispielsweise wäre es von Vorteil, wenn keine besondere Infrastrukturbasis zum verteilten Trainieren der Frameworks nötig ist. *Niedrig* priorisierte Anforderungen hingegen spielen für diese Arbeit eine sehr geringe Rolle, da der Fokus auf der horizontalen Skalierung über Rechnergrenzen hinweg liegt. Es besteht jedoch die Möglichkeit, dass Ergebnisse verfälscht werden könnten, wenn die zu evaluierenden Frameworks unterschiedliche Einstellungen verwenden.

3.2 Marktüberblick aktueller Deep Learning Frameworks

Die Liste der derzeit verfügbaren Deep Learning Frameworks ist lange[1], daher wird in den folgenden Abschnitten lediglich eine Auswahl der am vielversprechendsten Frameworks, mit Blick auf die im vorherigen Abschnitt definierten Anforderungen, vorgestellt.

[1] Liste diverser Deep Learning Frameworks: `https://goo.gl/6yV0ib`

Anforderungen	Priorität
Generische Einsetzbarkeit	hoch
Programmierschnittstelle für Python	hoch
Möglichkeit, mehrere Rechner zu verwenden	hoch
Datenparallelität wird unterstützt	mittel
Einfache bis keine Infrastruktur Voraussetzungen	mittel
Modellparallelität wird unterstützt	niedrig
Möglichkeit, mehrere GPUs/CPUs zu verwenden	niedrig
Auswählbares Synchronisationsmodell (sync./ async.)	niedrig

Tabelle 3.1: Priorisierte Anforderungen an die Deep Learning Frameworks.

3.2.1 Theano

Theano ist eines der ältesten Frameworks, die zum Implementieren von neuronalen Netzen geeignet ist. Es wurde vom Jahr 2008 bis im Herbst 2017[2] hauptsächlich vom *Montreal Institute for Learning Algorithms (MILA)* der Universität Montreal entwickelt. [30]

Es sieht sich selbst als Compiler, der mathematische Ausdrücke, die als gerichtete, nicht zyklische Graphen definiert wurden, effizient auf der CPU oder GPU kompiliert. Für die nutzerfreundliche Verwendung bietet Theano eine Schnittstelle für die Skriptsprache Python. Der Kern, der die rechenaufwendige Optimierungen und Kompilierung der Graphen übernimmt, ist dabei jedoch in C++ oder CUDA[3] implementiert. [30]

Offiziell unterstützt werden Neuronale Netze und CNNs. Über die Jahre sind diverse auf Theano aufbauende Frameworks entstanden, die dessen Funktionsumfang erweitern. Hierzu gehören beispielsweise Bibliotheken, die Funktionen zum Implementieren von RNNs zur Verfügung stellen[4]. Auch das Paket `platoon`[5], das die Möglichkeit bietet, das Training der neuronalen Netze mithilfe von Datenparallelität und mehreren GPUs oder CPUs zu beschleunigen, ist eine der Erweiterungen. Weiter ist ebenfalls das Verwenden von mehreren Rechnern keine Möglichkeit, die Theano zur Verfügung stellt, kann jedoch durch communitygetriebene Frameworks nachgerüstet werden[6]. [30]

[2] Information von Yoshua Bengio in einer Theano User Group: `https://goo.gl/RVw7ot`
[3] CUDA ist eine von Nvidia entwickelte Programmiertechnik, um Berechnungen auf der GPU auszuführen.
[4] Theano Erweiterung für RNNs: `https://goo.gl/LzYmUH`
[5] Theano Erweiterung für Datenparallelität: `https://goo.gl/71Cd2W`
[6] Theano Erweiterung für verteiltes Training: `https://goo.gl/4rphMN`

3.2.2 TensorFlow

Im November 2015 hat das Unternehmen Google das Framework *TensorFlow* veröffentlicht. Dieses basiert auf früheren Forschungen und Erfahrungen, die auf dem Vorgänger Framework *DistBelief* beruhen, das bereits seit 2011 eingesetzt wurde [6]. Tensorflow ist das wohl bekannteste Deep Learning Framework, das derzeit zur Verfügung steht[7]. [20]

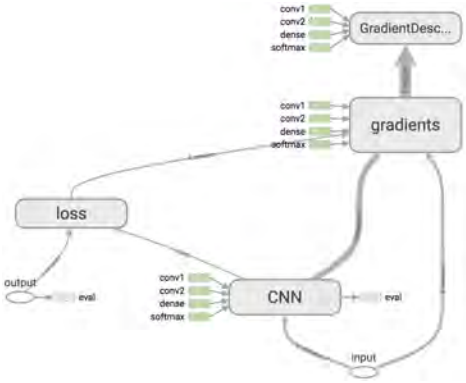
TensorFlow bietet diverse Hilfsmittel, die Programmierer beim Implementieren von neuronalen Netzen unterstützen, wie zum Beispiel Schichten in unterschiedlichster Ausprägung für CNNs oder RNNs. Weiter werden ebenfalls mehrere Optimierungsalgorithmen zur Verfügung gestellt, mit denen die Netze trainiert werden können.
Neben Algorithmen für den Bereich des Deep Learnings bietet es auch Implementierungen von gängigen Verfahren des maschinellen Lernens. [20]

Das Trainieren von Neuronalen Netzen kann sowohl auf mehrere Recheneinheiten, als auch auf mehrere Rechner verteilt werden. TensorFlow implementiert einen Ansatz namens *Parameter Server (PS)*, dabei berechnet jede Maschine unabhängig der anderen den Gradienten des aktuellen Trainingsdurchlaufs. Anschließend wird dieser an einen dedizierten Rechner, den Parameter Server, gesendet. Der PS hat die Aufgabe, die neuen Trainingsparameter zu berechnen, dies kann wie im Abschnitt 2.4.2 beschrieben, synchron oder asynchron geschehen. Bevor ein weiterer Trainingsdurchlauf gestartet wird, ist es die Aufgabe der Rechner, die derzeit aktuellen Trainingsparameter beim PS zu erfragen. Besondere Voraussetzungen sind für das verteilte Trainieren nicht nötig, alle notwendigen Bibliotheken sind in der TensorFlow Implementierung enthalten. [20, 19]

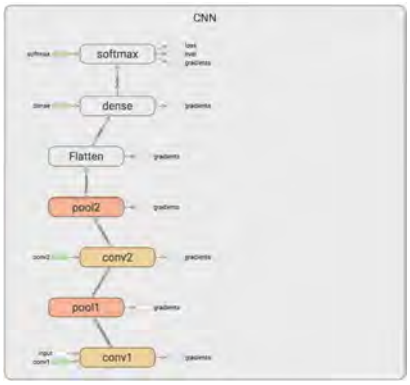
Für die Entwicklung mit TensorFlow bietet Google die Möglichkeit, auf die von dem Unternehmen entwickelten APIs für C++ oder Python zurückzugreifen. Es existieren jedoch weitere communitygetriebene[8], die dank der Tatsache, dass TensorFlows Quellcode Open Source zur Verfügung steht und durch die Aufforderung von Google selbst, entstanden sind. [20]

Eine Besonderheit und Alleinstellungsmerkmal ist das Werkzeug TensorBoard, das Google zusätzlich zu TensorFlow veröffentlicht hat. Es bietet unter anderem, wie auf Abbildung 3.1a zu sehen, die Möglichkeit, den konstruierten Graphen zu visualisieren. Dabei können bestimmte Bereiche, wie zum Beispiel das CNN sowohl in grober, als auch in feinerer Auflösung, wie in Abbildung 3.1b zu sehen ist, dargestellt werden. In Verbindung mit verschiedenen Diagrammen, die unterschiedliche Arten von Messwerten in einem zeitlichen Verlauf darstellen, wie auf Abbildung 3.1c zu erkennen ist, wird Entwicklern ein mächtiges Debugging Werkzeug angeboten. Es dient dazu, den Aufbau des neuronalen Netzes besser zu verstehen und nachzuvollziehen, wie sich das Ändern von unterschiedlichsten Parametern auf die Güte dessen auswirkt. [20]

[7] Abgelesen an den GitHub Stars TensorFlows und der Konkurrenz. Stand: 07. November 2017
[8] Auflistung der Verfügbaren APIs für TensorFlow: `https://goo.gl/LznSVy`



(a) Screenshot eines Berechnungsgraphen in grober Auflösung.



(b) Screenshot eines Teilbereichs des Berechnungsgraphen mit höherer Auflösung.

(c) Screenshot einer visualisierten Metrik im zeitlichen Verlauf des Trainings.

Abbildung 3.1: Verschiedene Visualisierungen mithilfe von TensorBoard.

3.2.3 MXNet

MXNet ist unter anderem durch die Zusammenarbeit von Entwicklern entstanden, die zuvor an diversen Deep Learning Projekten wie *CXXNET*[9], *Minerva*[10] oder *PURINE2*[11] gearbeitet haben. Auch der Name MXNet soll darauf anspielen: Er steht für ***mix*** und ***maximize***[12], denn es wurde versucht, die Erfahrungen, die durch die Entwicklung der Vorgänger Frameworks gesammelt wurden, zu verbinden.

Heute wird MXNet von unterschiedlichsten Universitäten sowie Firmen entwickelt und ist ein Teil der *Distributed (Deep) Machine Learning Community (DMLC)*. Diese ist durch diverse andere Projekte aus dem Bereich des maschinellen Lernens bekannt[13].

In der Anzahl der APIs, die MXNet zur Verfügung stellt, ist es ungeschlagen. Neben den gängigen Implementierungen für Python und C++ bietet es ebenfalls Unterstützung für Scala, R, Julia und Perl.

[9] Internetauftritt: `https://github.com/dmlc/cxxnet`
[10] Internetauftritt: `https://github.com/dmlc/minerva`
[11] Internetauftritt: `https://github.com/purine/purine2`
[12] Informationen zu MXNet: `http://dmlc.cs.washington.edu/mxnet.html`
[13] Beispielsweise die Bibliothek XGBoost, die unter anderem erfolgreich in einigen Wettbewerben eingesetzt wurde. Link: `https://xgboost.readthedocs.io/en/latest/`

Mithilfe dieser ist es möglich, auf unterschiedlichsten Ebenen Funktionen von MXNet zu verwenden. Neben sehr einfachen Funktionen, die effiziente Implementierungen von mathematischen Operationen abstrahieren, gibt es ebenfalls sehr mächtige, die zum Beispiel ganze Schichten von Neuronalen Netzen definieren. Hierbei werden neben den ursprünglichen Netzen auch allerhand spezialisierte Schichten für CNNs und RNNs zur Verfügung gestellt. Zusätzlich ist auch das Angebot an vordefinierten Optimierungsalgorithmen, Kostenfunktion oder Aktivierungsfunktionen sehr umfangreich. [31] Diese Tatsachen machen MXNet zu einem sehr universell einsetzbaren Framework.

Auch MXNet bedient sich des Graphenbasierten Ansatz und versucht anhand dessen nicht nur die Berechnungszeit, sondern auch den benötigten Speicherplatz zu optimieren. Ebenfalls wird dadurch die Verteilung der Berechnungen vereinfacht. MXNet unterstützt sowohl das Nutzen von mehreren GPUs und CPUs, als auch die Verteilung auf mehrere Rechner. Auch MXNet verwendet hierzu den Parameter Server Ansatz, jedoch ist dieser zweistufig implementiert und wird *KVStore* genannt. Die erste Stufe aggregiert die Gradienten, die von den Recheneinheiten innerhalb einer Maschine berechnet wurden, und die zweite Stufe synchronisiert über alle Rechner hinweg. Um Neuronale Netze mit MXNet verteilt trainieren zu können, muss auf den Maschinen entweder Secure Shell (SSH), oder Message Passing Interface (MPI) installiert sein. Dies wird benötigt, damit der sogenannte *Scheduler*, auf allen Maschinen den verteilten Trainingsprozess starten kann. Eine weitere Möglichkeit zum Starten des Trainings ist das Verwenden eines Scheduling Systems wie Hadoop YARN oder Sun Grid Engine (SGE)[14]. [31]

Um das Debuggen der in MXNet implementierten Neuronalen Netze zu vereinfachen, haben sich die Entwickler der Open Source Eigenschaft des Werkzeugs TensorBoard bedient und es für die Verwendung mit MXNet angepasst. Zum Zeitpunkt dieser Arbeit besitzt die MXNet Variante von TensorBoard noch nicht den vollen Funktionsumfang, es ist jedoch geplant, diesen zu erweitern[15].

3.2.4 Cognitive Toolkit

Die erste Beta Version von Microsofts *Cognitive Toolkit (CNTK)* wurde im Januar 2016 auf GitHub veröffentlicht, nur kurze Zeit später, im April 2016, folgte die erste stabile Version[16].

CNTK nutzt, wie auch alle bisher beschriebenen Frameworks, einen Berechnungsgraphen, der es ermöglicht, die Operationen zu optimieren und anschließend möglichst effizient auf unterschiedliche Recheneinheiten zu verteilen. Dabei werden nicht nur GPUs oder CPUs in einer Maschine, sondern auch die Berechnung mit mehreren Maschinen unterstützt. Für den Fall, dass das Training verteilt ausgeführt werden soll, bietet CNTK vier verschiedene Lernalgorithmen, die auf dem Gradientenabstiegsverfahren basieren. Hierbei werden beide der gängigen Ansätze der Modell- und der Datenparallelität angeboten. Zum Starten des verteilten Trainings wird eine MPI Installation benötigt. Besonders bei der kommerziellen Nutzung von CNTK muss auf die Lizenzierung geachtet werden,

[14] Informationen über die nötige Infrastrukturbasis MXNets: `https://goo.gl/NnTHxp`
[15] Blog-Artikel über das Verwenden von TensorBoard in Verbindung mit MXNet: `https://goo.gl/FFUuQ8`
[16] Information über die Veröffentlichungen von CNTK: `https://github.com/Microsoft/CNTK/releases`

denn die Trainingsalgorithmen wurden mit unterschiedlichen Lizenzen[17] veröffentlicht. [24]

CNTK bietet zur Unterstützung sowohl Operationen an, die hilfreich beim Implementieren von CNNs sind, aber auch welche, zum Implementieren von RNNs. Diese können über APIs für die Sprachen Python, C++ und C# verwendet werden. Eine weitere Sprache ist BrainScript, die speziell zum Beschreiben der Netze entwickelt wurde[18]. [24]

Ähnlich wie auch MXNet bietet CNTK die Möglichkeit, das Visualisierungswekzeug TensorBoard zu nutzen. Zum Zeitpunkt dieser Arbeit ist die Implementierung von CNTKs TensorBoard weiter fortgeschritten als die von MXNet und bietet somit einen größeren Funktionsumfang.[19]

3.2.5 Deeplearning4j

Im Gegensatz zu den bisher vorgestellten Frameworks basiert Deeplearning4j (DL4J) auf der Programmiersprache Java. Die Open Source Entwicklung wird hauptsächlich von dem Unternehmen *Skymind*[20] vorangetrieben, das ebenfalls einen kommerziellen Support anbietet.

DL4J bietet einen großen Umfang an APIs für unterschiedlichste Verwendungszwecke, wie zum Beispiel CNNs, oder RNNs. Andere Deep Learning Ansätze, wie Reinforcement Learning, werden in weiteren Projekten[21] kompatibel zu DL4J entwickelt. Auch DL4J nutzt einen Graphenbasierten Ansatz, um automatisches Differenzieren zu ermöglichen, was eine effiziente Implementierung des Backpropagation Algorithmus ermöglicht. Um mathematische Operationen effizient berechnen zu können, wird eine, ebenfalls von Skymind entwickelte, Bibliothek namens *ND4J*[22] verwendet. Diese ermöglicht zum einen den Umgang mit hochdimensionalen Matrizen, wie sie bei neuronalen Netzen sehr häufig eingesetzt werden, und zum anderen erlaubt es diese Berechnungen auf der GPU auszuführen [29]. DL4J bietet dadurch die Möglichkeit, das Trainieren von Neuronalen Netzwerken sowohl auf mehreren CPUs, als auch auf GPUs zu parallelisieren. [28]

DL4J ist nicht auf die Parallelisierung innerhalb eines Rechners begrenzt, es wird ebenfalls das horizontale Skalieren über Rechnergrenzen hinweg unterstützt. Hierzu kann entweder *Apache Spark*[23] oder *Apache Hadoop YARN*[24] verwendet werden. Beide dienen als verteilte Laufzeitumgebung, die durch entsprechende Mechanismen der DL4J API verwendet werden können. Es wird sowohl eine API für Java, als für andere Sprachen der Java Virtual Machine (JVM), wie zum Beispiel eine Implementierung für Scala und Clojure, geboten. [28]

[17] CNTK 1bit-SGD Lizenz - Link: `https://goo.gl/grGWc2`
CNTK Lizenz - Link: `https://github.com/Microsoft/CNTK/blob/master/LICENSE.md`

[18] Informationen über Brainscript: `https://goo.gl/1v7evf`

[19] Dokumentation über das Verwenden von TensorBoard mit CNTK: `https://goo.gl/UH3PCG`

[20] Internetauftritt: `https://skymind.ai`

[21] RL4J Repository: `https://github.com/deeplearning4j/rl4j`

[22] Internetauftritt: `https://nd4j.org`

[23] Internetauftritt: `https://spark.apache.org`

[24] Internetauftritt: `https://hadoop.apache.org`

3.3 Auswahl der zu evaluierenden Frameworks

In der Tabelle 3.2 sind sowohl die in den vorherigen Abschnitten vorgestellten Frameworks, als auch die priorisierten Eigenschaften, die im Abschnitt 3.1 herausgearbeitet wurden, aufgelistet. Die tabellarische Struktur erlaubt einen schnellen Überblick, welche der Frameworks die nötigen Eigenschaften besitzen, um für die weitere Evaluation in Frage zu kommen.

Priorisierte Anforderungen		**Frameworks**				
Eigenschaften	**Priorität**	**Theano**	**TensorFlow**	**MXNet**	**CNTK**	**DL4J**
Generisch	hoch	➙	✓	✓	✓	✓
Python API	hoch	✓	✓	✓	✓	✗
Multi Rechner	hoch	➙	✓	✓	✓	✓
Datenparallelität	mittel	➙	✓	✓	✓	✓
Infrastruktur-anforderungen	mittel	➙	keine	MPI/ SSH/ SGE/ Hadoop YARN	MPI	Hadoop YARN/ Spark
Synchronisations-modell	niedrig	➙	synchron/ asynchron	synchron/ asynchron	synchron/ asynchron	synchron/ asynchron
Multi GPU/ CPU	niedrig	➙	✓	✓	✓	✓
Modellparallelität	niedrig	✗	✓	✓	Keine Angabe	✗
Legende	"✓": wird unterstützt; "➙": wird nicht nativ unterstützt; "✗": wird nicht unterstützt					

Tabelle 3.2: Auflistung der Frameworks und Einordnung ihrer Eigenschaften in die priorisierten Anforderungen.

Das Framework DL4J wird nicht weiter betrachtet, da es die *hoch* priorisierte Eigenschaft der Python API nicht unterstützt.

Theano hingegen bietet die Möglichkeit, neuronale Netze mithilfe einer Python API zu implementieren. Weiter ist es durch unterschiedlichste Erweiterungen möglich, das Training auf mehrere Maschinen zu verteilen. Auch wird das Implementieren von RNNs erst durch das Verwenden von Community getriebenen Erweiterungen möglich. Unsicher ist, ob zwei oder mehrere Erweiterungen zeitgleich eingesetzt werden können. Dies, so wie die Tatsache, dass die Hauptentwickler der MILA angekündigt haben, dass sie ihre Arbeit ab der Version 1.0.0, die am 15. November 2017 veröffentlicht wurde, einstellen werden, hat schlussendlich zu dem Entschluss geführt, Theano nicht weiter zu evaluieren.

Wie die Tabelle 3.2 zeigt, besitzen MXNet und CNTK ähnliche Eigenschaften. Hinsichtlich der Infrastrukturanforderungen ist MXNet flexibler einsetzbar. Durch die Möglichkeit, ein SSH Tunnel zum Starten des Trainings zu verwenden, bietet es auch die einfachere Möglichkeit, da SSH ein weitverbreitetes Werkzeug ist. MPI hingegen ist ein Standard, der den Nachrichtenaustausch bei verteilten Computern beschreibt und häufig im Bereich von sehr großen Parallelrechnern eingesetzt wird[25].

[25]Informationen über MPI: `https://goo.gl/EEI0R1`

Da TensorFlow nicht nur alle notwendigen Eigenschaften erfüllt, sondern auch keinerlei Besonderheiten bei der Infrastruktur benötigt, wird neben MXNet auch TensorFlow im weiteren Verlauf der Arbeit evaluiert.

4 Ansatz der experimentellen Evaluation von Deep Learning Frameworks

Nach der Diskussion der Anforderungen im vergangenen Kapitel und der Auswahl der Frameworks, die im Folgenden evaluiert werden, folgt in diesem eine Erläuterung des Ansatz der experimentellen Evaluation. Im Mittelpunkt steht die Performanz der Frameworks *TensorFlow* und *MXNet* beim verteilten Trainieren mit dem Ansatz der asynchronen Datenparallelität. Mit dieser Parallelisierungsart ist es sowohl mit RNNs, als auch mit CNNs möglich, den höchsten Durchsatz an Trainingsdaten zu erreichen (vgl. Abschnitt 2.4).

In den folgenden Abschnitten wird zuerst die Wahl der Aufgaben für die experimentelle Evaluation diskutiert und anschließend die verwendeten Metriken und Datensätze vorgestellt. Zum Abschluss sind die Umgebungsbedingungen des Testaufbaus beschrieben.

4.1 Diskussion und Auswahl diverser Aufgaben zur Evaluation

Um eine möglichst umfangreiche Evaluation der Frameworks zu ermöglichen, gilt es, Aufgaben zu finden, die zum einen mithilfe von unterschiedlichen Netzarten bearbeitet werden und zum anderen eine qualitative Bewertung des Trainings erlauben. Das CNN steht hierbei stellvertretend für Netzwerke, die eine größere Anzahl Trainingsparametern besitzen. Das RNN soll das Verhalten der Frameworks beim Trainieren von Netzen verdeutlichen, deren Rechenaufwand komplexer ist.

4.1.1 Auswahl der Aufgabe für das Convolutional Neural Network

Wie bereits im Abschnitt 2.2.1 im Zuge der Erläuterung der CNNs erwähnt wurde, eignen sich diese sehr gut, um Bilder zu verarbeiten. Einer der bekanntesten und am weitest verbreiteten Anwendungsfälle ist die *Klassifizierung von Bildern*. Hierbei hat ein Neuronales Netzwerk die Aufgabe, ein Bild, abhängig von dessen Inhalt, in vordefinierte Klassen einzusortieren. Diese können, müssen aber nicht zwangsläufig, exklusiv sein.

Das in dieser Arbeit eingesetzte Convolutional Neural Network ist eine leichte Abwandlung des originalen LeNet-5. Es besteht aus zwei Convolutional Schichten mit jeweils einem aufnahmefähigen Feld von 5×5 und Schrittweite 1. Die erste berechnet 20 und die zweite 50 Filter. Beide Schichten sind gefolgt von einer Max Pooling Schicht, die ein aufnahmefähiges Feld mit 2×2 und Schrittweite 2 besitzt. Für die Klassifizierungsaufgabe folgt eine voll verbundene Schicht mit 500 und eine Aussgangsschicht mit 10 Neuronen.

Die Güte eines zur Bildklassifizierung verwendeten Convolutional Neural Networks wird mithilfe der Genauigkeit (eng. Accuracy) gemessen. Diese wird im Abschnitt 4.2.2.1 näher erläutert.

4.1.2 Auswahl der Aufgabe für das Recurrent Neural Network

Die Einsatzgebiete von RNNs und die daraus resultierenden Aufgaben sind im Vergleich zu CNNs deutlich größer (vgl. Abschnitt 2.3.2), was die Auswahl einer Aufgabe, die den Anforderungen entspricht, erschwert. Aufgaben wie das Verschlagworten von Texten, oder das Erstellen von Bildunterschriften sind schwer qualitativ zu bewerten, denn meist sind mehrere Lösungen korrekt, die sich in ihrer subjektiven Güte unterscheiden.
Weit verbreitet ist auch der Ansatz, RNNs darauf zu trainieren Texte zu generieren, die im Stil von beispielsweise Shakespeare geschrieben sind, dem Aufbau eines Wikipedia Artikels oder einer wissenschaftlichen Veröffentlichung ähneln[1]. Aufgaben wie diese sind jedoch noch schwerer beziehungsweise gar nicht objektiv anhand von Qualitätskriterien zu bewerten.

Im Forschungsgebiet der *Verarbeitung von natürlicher Sprache* (engl. natural language processing (NLP)) gibt es ein Teilgebiet, das sich mit dem Modellieren von Sprache (engl. language modeling) auseinandersetzt. Hierbei wird unter anderem versucht, die Wahrscheinlichkeit von Worten oder Sätzen in einem Kontext vorherzusagen. Bei dieser Aufgabe ist im Vergleich zu den vorherigen die beste Möglichkeit geboten das Training qualitativ zu bewerten, weshalb diese für die weiter Verwendung ausgewählt wurde.

Das zur Evaluation verwendete RNN wurde 2014 von Zaremba et al. in [34] vorgestellt. Es besteht aus insgesamt zwei Schichten, die jeweils aus 200 rekurrenten Neuronen bestehen und über 30 Zeitschritte ausgerollt werden. Die Ausgangsschicht enthält so viele Neuronen, wie der Trainingsdatensatz unterschiedliche Worte besitzt, hier also 10.000 (vgl. Abschnitt 4.2.2.2).

Die Güte eines zur Sprachmodellierung eingesetzten Recurrent Neural Networks wird mithilfe der Perplexity gemessen. Diese wird im Abschnitt 4.2.2.2 näher erläutert.

4.2 Verwendete Metriken zur Evaluierung der Deep Learning Frameworks

In diesem Abschnitt werden unterschiedliche Metriken vorgestellt, die im Verlauf der Arbeit für diverse Bewertungen verwendet werden. Der Fokus dieser Arbeit liegt auf der Skalierbarkeit der Frameworks, weshalb zuerst die Metriken zu deren Bewertung und anschließend jene, die zum Bewerten der Güte der Netze nutzbar sind, beschrieben werden.

4.2.1 Metriken zur Bewertung der horizontalen Skalierbarkeit

In diesem Abschnitt sind zwei Metriken beschrieben, die im Folgenden eingesetzt werden, um die Geschwindigkeit und vor allem die Skalierbarkeit zu bewerten.

[1] Blog-Artikel über verschiedene Anwendungsfälle von RNNs: `https://goo.gl/FodLp5`

Zuvor müssen jedoch zwei Begriffe eingeführt werden, die beim Trainieren von Neuronalen Netzen eine entscheidende Rolle spielen:

Batch:
Ein Batch besteht aus einer definierten Anzahl an Trainingsbeispiele, der Batchgröße, die zufällig aus dem Trainingsdatensatz entnommen werden. In jedem Trainingsdurchlauf (vgl. Abschnitt 2.1.3.1) wird nicht nur ein Beispiel des Trainingsdatensatzes, sondern der ganze Batch verwendet um die Anpassungen der Trainigsparameter zu berechnen. Das generiert den Vorteil, dass die Gradienten genauere sind, da mithilfe von Vektorisierung zeitgleich die Aktivierungswerte aller Daten des Batches berechnet werden.

Epoch:
Ein Epoch bedeutet, dass jedes Beispiel des Trainingsdatensatzes einmal in einem Batch verwendet wurde. Bei jedem Trainingsdurchlauf werden dabei so viele zufällig Trainingsdaten entnommen, wie die Batchgröße definiert. Die Trainingsdaten werden innerhalb eines Epochs nicht doppelt verwendet, somit ist der letzte Batch eines Epochs tendenziell kleiner als die Batchgröße definiert.

4.2.1.1 Batches pro Sekunde

Um die Geschwindigkeit des Training eines Frameworks bestimmen zu können, gibt es unterschiedliche Möglichkeiten. Die in dieser Arbeit verwendete nennt sich *Batches pro Sekunde* und hat den Vorteil, dass sie auch eine gültige Aussage über die Geschwindigkeit erlaubt ohne das resultierende Ergebnis, also die Güte des Netzes, zu betrachten. Denn Trainingszeiten bis zu einer bestimmten Güte des Netzwerks setzen voraus, dass die Implementierung in Abhängigkeit des verwendeten Frameworks optimiert wurde, dies ist jedoch nicht Teil dieser Arbeit.

Um die Batches pro Sekunde berechnen zu können, genügt es, die Anzahl aller Batches durch die gesamte Trainingszeit zu teilen. Die Anzahl der Batches hängt dabei von der Anzahl Trainingsbeispiele *daten*, der Anzahl Epochs *epochs*, der Anzahl verwendeter Worker-Maschinen *worker* und der Größe der einzelnen Batches *batch_groese* ab. Gleichung 4.1 zeigt Berechnung der Metrik *batches_pro_sekunde*.

$$batches\_pro\_sekunde = \frac{\frac{daten}{batch\_groese} * epochs * worker}{dauer} = \frac{daten * epochs * worker}{batch\_groese * dauer} \quad (4.1)$$

Da die Metrik Batches pro Sekunde keine Aussage über die Skalierbarkeit trifft, wird eine weitere benötigt, die im folgenden Abschnitt beschrieben ist.

4.2.1.2 Speedup

Der Speedup (dt. Beschleunigung) beschreibt die Verbesserung der Leistung zwischen zwei Systemen, die dieselbe Aufgabe lösen, aber über unterschiedliche Ressourcen verfügen.

In dieser Arbeit wird die Leistung des System in Form des Durchsatzes gemessen, also die Anzahl Trainingsbeispiele, die pro Zeiteinheit berechnet werden können. Die Ressourcen,

die erhöht werden, ist die Anzahl der Worker-Maschinen. Der optimale Speedup entspricht dem Faktor um den die Anzahl der Worker-Maschinen erhöht wurde, bezogen auf einen Referenzwert, der in dieser Arbeit immer der Durchsatz mit einer Worker-Maschine ist. Die Berechnung des Speedups ist in Gleichung 4.2 zu sehen.

$$speedup = \frac{batches\_pro\_sekunde_{workeranzahl}}{batches\_pro\_sekunde_1} \tag{4.2}$$

4.2.2 Metriken zur Bewertung der Güte der Neuronalen Netze

Wie im Abschnitt 4.1 beschrieben ist, wird der Trainingsvorgang qualitativ bewertet, hierzu werden Metriken benötigt, die in den folgenden Abschnitten erläutert sind.

4.2.2.1 Gütemessung der Bildklassifizierung: Accuracy

Die Metrik *Accuracy* (dt. Genauigkeit oder Richtigkeit) beschreibt wie viel Prozent der Ergebnisse in einer Ergebnismenge korrekt sind. Mathematisch wird die Genauigkeit wie in Gleichung 4.3 berechnet.

$$genauigkeit = \left(\frac{anzahl\_korrekter\_ergebnisse}{anzahl\_aller\_ergebnisse} \right) * 100 \tag{4.3}$$

In vielen praktischen Anwendungsfällen genügt das alleinige Betrachten der Genauigkeit nicht, denn in einigen Domänen gibt es den wichtigen Unterschied zwischen dem fehlerhaften Nichterkennen und dem fehlerhaften Erkennen. Zum Beispiel macht es bei einem automatischen Filter, der die Bildern auf jugendfreie Inhalte prüft, einen wichtigen Unterschied, ob fälschlicherweise ein Bild als *jugendfrei* erkannt wird, oder ob es fälschlich als *nicht jugendfrei* erkannt wird. Die Priorität, wird klar darauf liegen, alle nicht jugendfreien Bilder herauszufiltern und in Kauf zu nehmen, dass fälschlicherweise als jugendfrei erkannte Bilder ebenfalls entfernt werden.[2]

Da es in dieser Arbeit nur interessant ist, ob eine bestimmte Klasse falsch ist, ist die Accuracy gut dafür geeignet, als Metrik zur Bewertung der Güte verwendet zu werden.

4.2.2.2 Gütemessung der Sprachmodellierung: Perplexity

Bei Neuronalen Netzen, oder allgemeiner bei Modellen des maschinellen Lernens, die für das Verarbeiten von natürlicher Sprache verwendet werden, kommen sehr häufig die beiden nahverwandten Metriken *Entropy* und *Perplexity* zum Einsatz.

Entropy

Die Informationsentropy der Informationstheorie ist ein Maß für den durchschnittlichen Informationswert, den eine Nachricht hält, die aus einer Datenquelle mit zugehöriger Wahrscheinlichkeitsverteilung kommt[3]. Um die Entropy eines Neuronalen Netzes zu berechnen,

[2] Blog-Artikel über die Metrik Accuracy: `https://goo.gl/A6mFr4`

[3] Informationen zur Entropy: `https://goo.gl/VKxgFU`

dessen Wahrscheinlichkeitsverteilung nicht bekannt ist, wird mithilfe einer Testdatenmenge der Größe N die durchschnittliche Entropy mit der Formel, die in Gleichung 4.4 zusehen ist, berechnet. [15]

$$H = -\frac{1}{N}\sum_{i=1}^{N} ln(p_i) \tag{4.4}$$

Die Perplexity hingegen sagt umgangssprachlich aus, wie überrascht das Neuronale Netzwerk von den erzeugten Ausgangswerten ist und wird verwendet, um eine intuitivere Metrik zu geben. Diese berechnet sich nach der in Gleichung 4.5 dargestellten Formel.

$$perplexity = e^{H} \tag{4.5}$$

Anders ausgedrückt beschreibt die Perplexity aus wie viel Möglichkeiten zufällig gewählt werden kann, um dasselbe Ergebnis zu erhalten. [15]

Im Gegensatz zur Accuracy sollte die Perplexity möglichst gering sein, damit mit einer möglichst hohen Wahrscheinlichkeit das folgende Wort vorhersagbar ist.

4.3 Verwendete Datensätze zur Evaluierung

Nach der Diskussion der Aufgaben, die für die Evaluation herangezogen werden, und der Erläuterung der hierzu verwendeten Metriken, folgt in diesem Abschnitt eine Vorstellung der eingesetzten Datensätze.

Der Fokus lag hierbei auch auf der Verwendung von gängigen Datensätzen, um die Implementierung des Testaufbaus zu vereinfachen.

4.3.1 Datensatz für die Bildklassifizierung: Fashion-MNIST

Der Modified National Institute of Standards and Technology (MNIST) Datensatz ist sehr weit verbreitet und wird häufig für Benchmarks im Bereich der Bildklassifizierung verwendet. Er besteht aus insgesamt 60.000 Trainings und 10.000 Testbildern mit einer Größe von 28×28 Pixeln, die mit Graustufen dargestellt sind. Darauf zu sehen sind, wie Abbildung 4.1a zeigt, unterschiedliche handgeschriebene Zahlen im Bereich von 0 bis 9. Der original Datensatz ist auf der Internetseite von *Yann LeCun* verfügbar[4].

Wie LeCun auf seiner Internetseite zeigt, ist es für moderne CNNs ohne größere Schwierigkeiten möglich, eine Fehlerrate von unter 1% beim Testdatensatz zu erreichen. Häufig wurde die Einfachheit des MNIST Datensatzes kritisiert, als Antwort hierauf veröffentlichte die Firma Zalando im August 2017 den Datensatz *Fashion-MNIST*. Dieser entspricht den selben Rahmenbedingungen und kann entsprechend einfach ausgetauscht werden. Enthalten sind ebenfalls 60.000 Graustufen 28×28 Pixel große Trainings und 10.000 Testbilder, die unterschiedliche Kleidungsstücke aus ebenfalls 10 verschiedenen Klassen zeigen. Beispiele sind in der Abbildung 4.1b zu sehen.
Der Komplexitätsunterschied der Bilder zeigt sich durch einen Genauigkeitsunterschied der Klassifikation von etwa 10%. [33]

[4]Informationen zu und der Datensatz MNIST selbst: `http://yann.lecun.com/exdb/mnist/`

(a) Neun zufällige Beispielbilder des MNIST Datensatzes.

(b) Neun zufällige Beispielbilder des Fashion-MNIST Datensatzes.

Abbildung 4.1: Gegenüberstellung der beiden Datensätze MNIST und Fashion-MNIST.

Wie im Abschnitt 4.1 beschrieben ist, soll eine qualitative Bewertung des Trainings möglich sein. Da Unterschiede der Genauigkeit zwischen MXNet und TensorFlow bei sehr hohen Genauigkeiten schwer auszumachen sind, fiel die Entscheidung zur Evaluation den Fashion-MNIST Datensatz heranzuziehen.

4.3.2 Datensatz zur Sprachmodellierung: Penn Tree Bank

Für das Testen von Modellen oder Neuronalen Netzen im Bereich der Sprachverarbeitung wird häufig der Datensatz *Penn Tree Bank (PTB)* verwendet. Da dieser aus nur circa 1.000.000 Worte besteht[5], ist er vergleichsweise klein und Modelle können entsprechend schnell trainiert werden.

In dieser Arbeit wurde die von Tomas Mikolov leicht überarbeitete Version verwendet[6]. Die Unterschiede zeigen sich darin, dass Mikolov sowohl sehr seltene Worte durch die spezielle Zeichenfolge *<unk>*, als auch Zahlen durch den Buchstaben *N* ersetzt hat. Diese Anpassungen reduzieren die Komplexität, die jedoch für die Evaluierung in dieser Arbeit genügt. Im weiteren Verlauf ist mit PTB der von Mikolov überarbeitete Datensatz gemeint.

PTB besteht aus Auszügen von 2,499 unterschiedlicher Artikel des *Wall Street Journal*[5]. Das Vokabular beträgt insgesamt 10.000 englische Worte. Für das Training wurde ein Menge von 880.000 und zum Testen 78.000 Worte verwendet.

4.4 Umgebungsbedingungen des Testaufbaus

In diesem Abschnitt werden die Umgebungsbedingungen, die mit freundlicher Unterstützung der Firma inovex GmbH in Karlsruhe[7] zur Verfügung gestellt wurden, beschrieben.

[5] Informationen über die Penn Treebank: `https://catalog.ldc.upenn.edu/ldc99t42`

[6] Tomas Mikolovs Internetseite: `http://www.fit.vutbr.cz/~imikolov/rnnlm/`

[7] Internetauftritt: `https://www.inovex.de/de/`

Die Evaluierung der Frameworks findet auf Basis der *Google Cloud Platform (GCP)* statt[8]. Hierzu wird der Service *Kubernetes Engine* verwendet[9]. Dieser bietet mithilfe des Orchestrierungssystem Kubernetes[10] die Möglichkeit, Anwendungen auf einem Computer Cluster auszuführen und dieses zu verwalten.

Konkret wurde ein Cluster in der GCP Region *europe-west1-c* verwendet, das mit Maschinen des Typs *n1-standard-1* bestückt ist. Ein Knoten dieses Typs entspricht einer der folgenden virtualisierten CPUs: 2.0 GHz Intel Skylake, 2.2 GHz Intel Xeon E5 v4, 2.3 GHz Intel Xeon E5 v3, 2.5 GHz Intel Xeon E5 v2 oder 2.6 GHz Intel Xeon E5[11]. Für die Messungen wurde darauf geachtet, dass alle Knoten dem Typ *2.5 GHz Intel Xeon E5 v2* entsprechen.

Obwohl das Trainieren von Neuronalen Netzen auf CPUs deutlich ineffizienter und langsamer ist als auf GPUs (vgl. Kapitel 2.4) wird die Evaluation auf Maschinen ausgeführt, die keinen Zugriff auf GPU Ressourcen besitzen. Ausschlaggebend hierfür ist, dass Kubernetes zum Zeitpunkt dieser Arbeit lediglich eine experimentelle API zur Verfügung stellt, um auch auf die Rechenleistung von GPUs zugreifen zu können.

Da der Fokus dieser Arbeit auf der Evaluation der horizontalen Skalierbarkeit von MXNet und TensorFlow liegt, und nicht auf der Messung der Performanz des Gesamtsystems, verspricht dieser Testaufbau Ergebnisse, die in Relation zueinander betrachtet werden können.

[8] Internetauftritt: `https://cloud.google.com`

[9] Internetauftritt: `https://cloud.google.com/kubernetes-engine/`

[10] Internetauftritt: `https://kubernetes.io`

[11] Informationen über diverse Maschinen Typen: `https://goo.gl/CcmFwF`

5 Aufbau und Implementierung der Versuchsumgebung

Nach der Beschreibung der verfügbaren Umgebung und Erläuterung des Aufbaus der Versuche im vergangenen Kapitel, wird in diesem die Implementierung des Versuchsaufbaus beschrieben.
Im ersten Teil sind die verwendeten Technologien, die zum Aufbau der Infrastrukturbasis eingesetzt wurden, erklärt. Anschließend wird die konkrete Umsetzung beschrieben.
Im zweiten Teil wird auf die Implementierung der Neuronalen Netze und deren Training eingegangen und erläutert, wie diese sowohl mit TensorFlow, als auch mit MXNet umgesetzt werden.

5.1 Eingesetzte Technologien zum Aufbau der Infrastrukturbasis

In den folgenden Abschnitten werden die drei zum Aufbau der Infrastruktur verwendeten Technologien Docker, Kubernetes und Helm vorgestellt. Dabei werden nur die für diese Arbeit interessanten Teile der Technologien beschrieben.

5.1.1 Kubernetes

Googles Kubernetes Engine[1], die als Grundlage für den Versuchsaufbau zur Verfügung steht, basiert auf dem Open Source Projekt *Kubernetes*[2], das ebenfalls von Google entwickelt wurde. Kubernetes ist ein Orchestrierungssystem, das Docker Container in Computer Clustern ausführt, und diese mithilfe der Konzepte von *Pods* und *Labels* diese logisch zu Applikationen gruppieren kann, um ein einfaches Verwalten zu ermöglichen. [16] Im Folgenden sind die für diese Arbeit wichtigen Bereiche kurz beschrieben.

Pod

Ein *Pod* ist die kleinste Einheit, die bei Kubernetes verwendet wird. Er kann unterschiedliche Aspekte des Cluster-Betriebs kapseln und verbindet diese mit Labels. Der in dieser Arbeit verwendete Anwendungsfall beschränkt sich auf das zur Verfügung stellen einer Docker Umgebung, die das Ausführen von Docker Containern erlaubt. Jeder Pod erhält eine Cluster interne IP- Adresse und ist ein flüchtiges Objekt. Das bedeutet, dass es

[1] Internetauftritt: `https://cloud.google.com/kubernetes-engine/`
[2] Internetauftritt: `https://kubernetes.io`

nach dem Beenden, unabhängig davon, ob wegen einem Absturz oder einem geplanten Herunterfahren, nicht wiederhergestellt werden. [16]

Label

Ein *Label* ist ein Schlüssel/Werte Paar, das Objekten zugewiesen werden kann, wobei die Schlüssel der Labels pro Objekt eindeutig sein müssen. Mit Hilfe der Labels ist es möglich, genau eins oder eine Menge von mehreren Objekten anzusprechen. [16]

Service

Da Pods flüchtig sind und somit nicht garantiert werden kann, dass ein Pod immer unter der selben IP- Adresse ansprechbar ist, wird mit *Services* ein Mechanismus geboten, der diese Problematik löst. Ein Service besteht aus einem Namen, einem sogenannten *Label Selector*, der dafür sorgt, dass sich der Service mit der Menge Pods verbindet, die entsprechende Labels tragen, und enthält eine Strategie wie eintreffende Anfragen auf die Menge der Pods verteilt werden. Kubernetes bietet die Möglichkeit, einen Cluster internen Domain Name System (DNS) Server zu verwenden, der es ermöglicht, dass die von den Containern ausgeführten Anwendungen, den Namen des Services zu einer IP- Adresse eines Pods auflösen können. [16]

ConfigMap

Durch *ConfigMap*s wird die Möglichkeit geboten, einen generischen Container oder Pod beim Starten mit Konfigurationen zu versehen. Diese können als Umgebungsvariablen, als Schlüssel/Werte Paare oder als Dateien verwendet werden. [16]

Mehrfach wurde bisher erwähnt, dass Docker Container benötigt werden, um Software mithilfe von Kubernetes zu orchestrieren. Was Docker ist und wie Docker Container erstellt und verwendet werden, wird im folgenden Abschnitt beschrieben.

5.1.2 Docker

Docker[3] ist eine Software zum Isolieren von Anwendungen mithilfe von Containervirtualisierung. Das bedeutet, dass es die Möglichkeit bietet, Software, zusammen mit allen benötigten Abhängigkeiten zum Beispiel Bibliotheken, Laufzeitumgebung und Konfiguration, in ein ausführbares Paket, dem Container, zu packen. Auf einem Rechner können gleichzeitig mehrere Container ausgeführt werden, diese sind komplett voneinander und dem Hostsystem isoliert und können sich nicht direkt beeinflussen. [8] Im Folgenden werden die wichtigsten Begriffe beschrieben und erläutert, wie ein Image erstellt wird, das Starten und Ausführen wird hier nicht behandelt, da dies vollständig von Kubernetes übernommen wird.

Dockerfile

Ein *Dockerfile* ist ein Textdokument, das schrittweise beschreibt, welche Änderungen, ausgehend von einem bereits existierendem Image, dem *parent Image*, vorgenommen

[3] Docker Internetseite: `https://www.docker.com`

werden sollen. Docker bietet die Möglichkeit, ein solches einzulesen und sequentiell ab zuarbeiten. War dies erfolgreich wird ein neues Image erstellt. [8]

Image

Ein *Image* ist eine Sammlung von Änderungen, die ausgeführt wurden, um den aktuellen Zustand herzustellen und enthält die Laufzeitparameter, die zum Ausführen des Containers benötigt werden. Ein erstelltes Image kann nicht geändert werden, jedoch als parent Image als Basis eines weiteren dienen. [8]

Container

Ein *Container* entspricht einer Instanz eines Images zur Laufzeit. Das bedeutet, dass die Umgebung innerhalb des Containers genau so zur Verfügung steht, wie sie mithilfe des Dockerfiles definiert wurde und im Image gespeichert ist. Alle Änderungen, die zur Laufzeit an der Containerumgebung vorgenommen werden, sind nach einem Neustart verloren. [8]

Registry

Eine *Registry* ist eine Anwendungen, die für das Verwalten von Docker Images konzipiert ist. Sie bietet die Möglichkeit, diverse Images zu versionieren und an einer zentralen Stelle für andere über das Internet oder in einem privaten Netzwerk bereitzustellen. [8]

Da zur Evaluation der beiden Frameworks die Anzahl der am Training der Neuronalen Netze beteiligten Maschinen variiert werden soll, wird eine Software benötigt, die dynamisch die entsprechenden Kubernetes Ressourcen erstellen kann. Diese Möglichkeit bietet die Anwendung Helm, die im folgenden Abschnitt beschrieben ist.

5.1.3 Helm

Die Software *Helm*[4] ist ein Paketmanager für Anwendungen, die in einem Kubernetes Cluster ausgeführt werden. Es bietet die Möglichkeit, Softwarekomponenten, Werkzeuge oder auch ganze Anwendungen als *Charts* zu verpacken. Ein Chart enthält alle notwendigen Kubernetes Ressourcen, die benötigt werden, um dieses in einem Kubernetes Cluster auszuführen. Bedingt durch Kubernetes sind die Dateien eines Charts vom Typ YAML. YAML ist ein Programmiersprachenübergreifender Serialisierungsstandard[5]. Ein Chart kann beliebig oft gestartet werden und wird zur Laufzeit *Release* genannt. [13]

Helm ist eine zweigeteilte Software, die aus einer serverseitgen Komponente, *Tiller* genannt, und einer clienseitigen Komponente besteht. Im Normalfall wird Tiller als Pod im Ziel Kubernetes Cluster ausgeführt und wartet auf Befehle des Clients. Dieser ist eine Kommandozeilen-Schnittstelle (engl. Command-Line Interface (CLI)), die das Chart an Tiller sendet. Dieser erzeugt und startet anschließend die Kubernetes Ressourcen.

In dieser Arbeit ist vor allem interessant, dass Helm Charts dynamisch mithilfe von Templates erstellt werden können. Hierzu nutzt Helm das Paket `template`[6] der Program-

[4] Internetauftritt: `https://helm.sh`

[5] Informationen zu YAML: `http://yaml.org`

[6] `template` Paket für die Programmiersprache Go: `https://godoc.org/text/template`

miersprache Go[7] [13]. Dies in Verbindung mit der Möglichkeit, *Values* zu verwenden, die beim Starten eines Helm Charts definiert werden, ergibt ein mächtiges Instrument, um Releases sehr flexibel zu gestalten.

Im folgenden Abschnitt wird beschrieben, wie mit den, in diesem Abschnitt vorgestellten Technologien, die notwendige Infrastrukturbasis für die Evaluation implementiert wurde.

5.2 Implementierung der Infrastrukturbasis zur experimentellen Evaluation

Im vergangenen Kapitel sind alle nötigen Aspekte der verwendeten Technologien beschrieben, um die folgenden Erläuterungen des Infrastrukturaufbaus nachvollziehen zu können. Die Reihenfolge des Vorgehens unterscheidet sich trotz der unterschiedlichen Frameworks nicht.

Zuerst muss für jedes der Frameworks ein Docker Image erstellt werden, das zukünftig verwendet wird, um den Container für das Training der Neuronalen Netze zu starten.
In einem weiteren Schritt muss das Docker Image in einer Registry veröffentlicht werden, auf die das Kubernetes Cluster Zugriff hat. Hierzu wird die *GitLab Container Registry* des GitLab Servers verwendet, auf dem ebenfalls der Quellcode dieser Arbeit verwaltet wird.

GitLab[8] ist ein auf Git basiertes Versionierungswerkzeug, das unter anderem auch die Möglichkeit bietet, Docker Images zu verwalten[9].

Anschließend wird das Helm Chart erstellt, das verwendet wird, um auf dynamische Art und Weise die nötigen Kubernetes Ressourcen zu erzeugen.

5.2.1 Beschreibung der Komponenten für TensorFlow

Wie aus den vorherigen Abschnitten hervorgeht, werden diverse Komponenten benötigt, um eine funktionsfähige Infrastruktur zur Verfügung zu stellen. In den folgenden Abschnitten werden diese beschrieben.

Docker Image

Da TensorFlow keine besondere Infrastruktur benötigt, um das Training verteilen zu können (vgl. Abschnitt 3.2.2), ist das hierfür benötigte Dockerfile entsprechend einfach.

Wie der Installationsanleitung von TensorFlow zu entnehmen ist[10], genügt es, einen entsprechenden Python Interpreter zu installieren. Dieser wird zum Ausführen des Trainingsskripts benötigt. Mithilfe des Python Paketmanagers *pip* kann TensorFlow anschließend komfortabel in der Implementierung installiert werden, die das Trainieren auf der CPU ermöglicht. Listing 5.1 zeigt das resultierende Dockerfile.

```
1 FROM ubuntu:16.04
```

[7] Go- Lang Internetseite: `https://golang.org`
[8] GitLab Internetseite: `https://gitlab.com/`
[9] Blog-Artikel über GitLab: `https://goo.gl/a2EOY4`
[10] TensorFlows Installationsanleitung: `https://goo.gl/8MbNWZ`

```
2
3 RUN apt-get update && apt-get install -y python3 python3-pip
4 RUN pip3 install --upgrade pip
5 RUN pip3 install --upgrade tensorflow
6
7 WORKDIR /opt/tensorflow
```

Listing 5.1: Das Dockerfile zum Erstellen des TensorFlow Images.

TensorFlows Anleitung zum verteilten Trainieren beschreibt, dass es möglich und gängig ist, ein Traingsskript zu implementieren, das sowohl den Quellcode für PS und Worker enthält[11]. Somit ist es möglich, dasselbe Skript generisch einzusetzen und abhängig von Argumenten den Ablauf entsprechend zu steuern. Die Implementierung des Trainingsskripts ist im Abschnitt 5.3 näher erläutert.
Hier ist wichtig, dass das Trainingsskript sowohl eine Liste aller beteiligten Worker als auch aller beteiligten PS in Form einer kommaseparierte Liste von IP- Adressen oder Domainnamen und einem zugehörigen Port benötigt. Weiter muss übergeben werden, welcher Job (PS oder Worker) auszuführen ist und welche Nummer dieser besitzt. Weitere Informationen hierzu im Abschnitt 5.3.

Helm Chart

Aus der obigen Beschreibung wird klar, dass Helm abhängig von den beiden definierten Values, die die Anzahl zu startender Maschinen enthalten, `tfCluster.settings.jobs.worker` und `tfCluster.settings.jobs.ps` entsprechend viele Pods starten muss, die sich hauptsächlich in den, an die Container übergebenen Argumenten, unterscheiden. Listing 5.2 zeigt die wichtigsten Ausschnitte, die für das Erstellen von mehreren Pods verantwortlich sind. In Zeile 4 und 5 bis Zeile 35 und 36 sind zwei Schleifen implementiert, die über diese Values iterieren und unterschiedliche Pod Ressourcen erzeugen.

```
1  {{ $num_ps := .Values.tfCluster.settings.jobs.ps }}
2  {{ $num_worker := .Values.tfCluster.settings.jobs.worker }}
3
4  {{ range $job, $nb := .Values.tfCluster.settings.jobs }}
5  {{ range $i, $e := until (int $nb) }}
6  apiVersion: v1
7  kind: Pod
8  metadata:
9    name: {{ $job }}-{{ $i }}
10   labels:
11     job: {{ $job }}
12     task: {{ $i }}
13 spec:
14   containers:
15 # weitere Konfigurationen ...
16     args:
```

[11] Anleitung für verteiltes Training: `https://www.tensorflow.org/deploy/distributed`

```
17        - python3
18        - train.py
19        - {{ range $k := until (int $num_ps) }}tensorflow-ps-{{ $k }}:5000
20          {{ if lt (int $k | add 1) (int $num_ps) }}{{print "," }}
21          {{ end }}{{ end }}
22        - {{ range $k := until (int $num_worker) }}tensorflow-worker-{{ $k }}:5000
23          {{ if lt (int $k | add 1) (int $num_worker) }}{{print "," }}
24          {{ end }}{{ end }}
25        - {{ $job }}
26        - {{ $i }}
27  # weitere Konfigurationen ...
28        volumeMounts:
29          - name: script
30            mountPath: /opt/tensorflow
31      volumes:
32        - name: script
33          configMap:
34            name: tensorflow-config-map
35  {{ end }}
36  {{ end }}
```

Listing 5.2: Ausschnitt des Helm Pod Template des TensorFlow Helm Charts.

Die Argumente, die dem Container zum Starten des Skripts übergeben werden, sind zwischen Zeile 17 und 24 zu sehen. Auch hier sind zwei Schleifen in den Zeilen 19/20 und Zeilen 22/23 implementiert, die eine kommaseparierte Liste aller PS, beziehungsweise der Worker erstellen.

Wenn das TensorFlow Chart, das die Pod Konfiguration aus Listing 5.2 enthält, mit dem Befehl, der in Listing 5.3 zu sehen ist, gestartet wird, dann werden drei unterschiedlich Pods generiert. Einer wird die Rolle des PS übernehmen und die anderen beiden die der Worker. Die dabei generierten Listen enthalten zum einen *tensorflow-ps-0:5000* und zum anderen *tensorflow-worker-0:5000,tensorflow-worker-1:5000*.

```
1  helm install --set tfCluster.settings.jobs.ps=1\
2    --set tfCluster.settings.jobs.worker=2\
3    tensorflow
```

Listing 5.3: Helm *install* Befehl, zum Starten des Trainings.

Damit die Pods jedoch über die generierten Domainnamen kommunizieren können, wird zusätzlich für jeden Pod ein Service benötigt. Wie beim Template der Pods, werden die Services mithilfe von Schleifen generiert, zu sehen ist dies in Listing 5.4.

```
1  {{ range $job, $nb := .Values.tfCluster.settings.jobs }}
2  {{ range $i, $e := until (int $nb) }}
3  kind: Service
4  apiVersion: v1
5  metadata:
6    name: tensorflow-{{ $job }}-{{ $i }}
```

```
7  spec:
8    ports:
9    - port: 5000
10     targetPort: 5000
11   selector:
12     job: {{ $job }}
13     task: {{ $i }}
14 {{ end }}
15 {{ end }}
```

Listing 5.4: Ausschnitt des Helm Service Template des TensorFlow Helm Charts.

Zu erkennen ist, dass der Name des Services den Domainnamen entspricht, die als Argumente dem Traingsskript übergeben werden. Weiter wird deutlich, dass sich jeder Service mit dem Pod verbindet, dessen Labels den Werten der beiden Selektoren `job` und `task` entspricht.

Um Änderungen am Trainingsskript einfach übernehmen zu können, werden diese mithilfe einer ConfigMap in den Container kopiert. Die hierfür verwendete ConfigMap ist im Listing 5.5 zu sehen.

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: tensorflow-config-map
5 data:
6   {{ (.Files.Glob "scripts/*").AsConfig }}
```

Listing 5.5: Ausschnitt des Helm ConfigMap Template des Tensorflow Helm Charts.

Helm bietet hierfür die Template Funktion `Files.Glob`, die es ermöglicht, den Inhalt eines ganzen Verzeichnisses, hier `scripts/*`, als Daten der ConfigMap hinzuzufügen[12]. Um den Inhalt im Container der Pods verwenden zu können, muss, wie in Zeile `28` bis `34` des Listings 5.2 zu sehen ist, ein Verzeichnis eingebunden werden, das anschließend mit den Daten der ConfigMap befüllt wird.

5.2.2 Beschreibung der Komponenten für MXNet

Nach der sehr detailreichen Erläuterung der Vorgehensweise im vorherigen Abschnitt, werden in diesem Kapitel lediglich die Unterschiede beleuchtet.

Ein großer Unterschied findet sich direkt bei der Art und Weise, wie das verteilte Training gestartet wird. Das Framework MXNet bietet unterschiedliche Mechanismen, wobei hier aus Gründen der Einfachheit die Möglichkeit gewählt wurde, das Training mithilfe einer SSH Verbindung zu starten (vgl. Abschnitt 3.2.3).

Docker Image

Auch bei MXNet wurde dessen Installationsanleitung[13] verwendet um, die nötigen Schritte

[12] Dokumentation zum Verwenden von Dateien mithilfe von Helm: `https://goo.gl/EZG7pk`

[13] MXNets Installationsanleitung: `http://mxnet.incubator.apache.org/install/index.html`

in das Dockerfile zu übernehmen. Der Unterschied zu TensorFlow ist jedoch die Tatsache, dass ein Docker Container, der als Worker am Training teilnehmen soll, per SSH erreichbar sein muss. Das bedeutet, jeder Container muss zusätzlich zur Installation von MXNet ebenfalls eine SSH Server Installation enthalten und diesen beim Start des Containers ausführen, um eingehende Verbindungen annehmen zu können. Hierzu wurde, wie im Listing 5.6 zu sehen ist, als Standardkommando, das beim Starten des Containers, soweit nicht anders angegeben, automatisch ausgeführt wird, das Starten des SSH Servers eingetragen.

```
1 # weitere Konfigurationen ...
2
3 RUN apt-get update && apt-get install -y --no-install-recommends ssh
4 CMD ["/usr/sbin/sshd", "-D", "-e"]
```

Listing 5.6: Ausschnitt des Dockerfiles für MXNet.

Aus einem Tutorial[14], das unter anderem beschreibt, wie das Training auf mehrere Rechner verteilt wird, geht hervor, dass MXNet zum Starten des Trainings ein dediziertes Startskript zur Verfügung stellt. Das Skript `launch.py` unterstützt nicht nur SSH, sondern auch alle anderen derzeit angebotenen Möglichkeiten (vgl. Abschnitt 3.2.3). Mithilfe des Arguments `--launcher` kann zwischen diesen unterschieden werden, im Fall von SSH sind drei weitere Argumente nötig. `--num-workers` beschreibt, wie viele Worker-Maschinen zum Training verwendet werden. Dem Argument `--hostfile` wird eine Datei übergeben, die entweder die IP- Adressen oder Domainnamen, der Worker-Maschinen enthält. Das letzte Argument `command` kann aus beliebig vielen Werten bestehen und entspricht dem Kommando, das auf den Worker-Maschinen mithilfe der SSH Verbindung ausgeführt wird.

Wegen dieses konzeptionellen Unterschiedes ergeben sich auch Unterschiede beim Implementieren des Helm Charts.

Helm Chart

Einer dieser Unterschiede ist, dass MXNet keinen dedizierten PS nutzt, sondern diese als eigener Prozess auf jeder der Maschine ausgeführt werden. Auch das Starten dieser wird von dem zur Verfügung gestellten Startskript übernommen. Daher vereinfacht sich das Template für die Pod Ressourcen der Worker-Maschinen leicht. Zum einen genügt eine Schleife, die entsprechend dem Value `Cluster.worker` viele Pods erzeugt und zum anderen fällt die Angabe des Befehls weg, den der Container nach dem Start ausführen soll, denn dieser ist bereits im Dockerfile kodiert. Ein Ausschnitt der vereinfachten Schleife und der Container Konfiguration ist in Listing 5.7 zu sehen, dieses zeigt zusätzlich, dass durch die Verwendung von Values die komfortable Möglichkeit geboten ist, Docker Images aus anderen Registries zu beziehen.

```
1 {{ $num_worker := .Values.Cluster.worker }}
2 {{ $repo := .Values.image.registry_path }}
3 {{ $name := .Values.image.name }}
4 {{ $tag := .Values.image.tag }}
5
```

[14] MXNet Tutorial, zum Starten des verteilten Trainings: `https://goo.gl/UZ2byD`

```
{{ range $i := until (int $num_worker) }}
# weitere Konfigurationen ...
spec:
  containers:
    - name: mxnet-worker-{{ $i }}
      image: {{ $repo }}/{{ $name }}:{{ $tag }}
      ports:
        - containerPort: 22
      volumeMounts:
        - name: script
          mountPath: /code
    volumes:
      - name: script
        configMap:
          name: mxnet-config-map
# weitere Konfigurationen ...
{{ end }}
```

Listing 5.7: Ausschnitt des Worker Template des MXNet Helm Charts.

Zusätzlich wird ein weiterer Pod benötigt, der das Startskript `launch.py` ausführt und somit als Scheduler fungiert (vgl. Abschnitt 3.2.3). Damit der Scheduler nicht das Training startet, bevor alle der Worker-Maschinen bereit sind, wird ein *Init Container* verwendet, beziehungsweise für jeden der Worker wird ein dedizierter Init Container verwendet.

Wie der Dokumentation der Init Container von Kubernetes[15] entnommen wurde, werden diese dazu verwendet, um zu testen, ob der Scheduler eine SSH Verbindung zu den Workern aufbauen kann. Denn erst, wenn jeder der Init Container sich ohne Fehler beendet hat, wird der Haupt- Container, in diesem Fall der Scheduler, der den verteilten Trainingsprozess startet, hochgefahren. Im Listing 5.8 ist der Ausschnitt des Templates zu sehen, der die Init Container konfiguriert.

```
{{ $num_worker := .Values.Cluster.worker }}
{{ $repo := .Values.image.registry_path }}
{{ $name := .Values.image.name }}
{{ $tag := .Values.image.tag }}

# weitere Konfigurationen ...
spec:
  initContainers:
{{ range $i := until (int $num_worker) }}
    - name: wait-for-mxnet-worker{{ $i }}
    image: {{ $repo }}/{{ $name }}:{{ $tag }}
    command: ["sh","-c", "until ssh mxnet-worker{{ $i }};"
            "do echo waiting for mxnet-worker{{ $i }}; done"]
{{ end }}
  containers:
    - name: {{ $name }}
```

[15] Dokumentation der Init Container: `https://goo.gl/Dg00zC`

```
17      args:
18        - python
19        - /mxnet/tools/launch.py
20        - --num-workers
21        - {{ .Values.Cluster.worker }}
22        - --launcher
23        - ssh
24        - -H
25        - /code/hosts
26        - python
27        - /code/train.py
28  # weitere Konfigurationen ...
```

Listing 5.8: Ausschnitt des Scheduler Template des MXNet Helm Charts.

Da auch hier die Pods miteinander kommunizieren müssen, werden Services benötigt, um dies zu gewährleisten. Diese wurden analog zu den Services für TensorFlow umgesetzt und werden daher hier nicht noch einmal beschrieben.

Wie Listing 5.8 zeigt und eingangs erwähnt wurde, muss dem Skript `launch.py` eine Datei übergeben werden, die alle Domainnamen der Worker enthält. Da der Inhalt dieser Datei vom Wert des Values `Cluster.worker` abhängt, wurde das Template der ConfigMap, wie in Listing 5.9 zu sehen, im Gegensatz zu TensorFlows, erweitert.

```
1   {{ $num_worker := .Values.Cluster.worker }}
2
3   # weitere Konfigurationen ...
4   data:
5     hosts: |-
6   {{- range $i := until (int $num_worker) }}
7     mxnet-worker{{ $i }}
8   {{- end }}
9
10  {{ (.Files.Glob "scripts/*").AsConfig }}
```

Listing 5.9: Ausschnitt des ConfigMap Template des MXNet Helm Charts

Nach der Erläuterung, wie mithilfe von Docker, Kubernetes und Helm die Grundlagen für die experimentelle Evaluation dieser Arbeit gelegt wurden, folgt jetzt ein Überblick, wie Neuronale Netzwerke mit den beiden Frameworks implementiert werden. Dabei wird vor allem gezeigt, wie das Training auf mehrere Rechner verteilt wird.

5.3 Implementierung von Neuronalen Netze mit TensorFlow

Um Neuronale Netze mithilfe von TensorFlow zu implementieren werden prinzipiell zwei Schritte benötigt. Im ersten geht es darum, den Berechnungsgraphen (vgl. Abschnitt 3.2.2) zu erstellen, also lediglich die Berechnungen, die zukünftig ausgeführt werden sollen, in eine entsprechende Repräsentation zu bringen. Im zweiten Schritt kann der Berechnungsgraph, oder Teile davon, durch die Unterstützung einer Session ausgeführt werden. [10]

In diesem Abschnitt wird die Implementierung dieser Schritte, mit Blick auf das verteilte Trainieren, näher beleuchtet.

5.3.1 Erstellen und Ausführen eines Berechnungsgraphen

TensorFlow bietet die Möglichkeit, einen Berechnungsgraphen in Abhängigkeit eines bereits trainierten Neuronalen Netzwerks zu erstellen. Dabei muss es sich nicht zwangsläufig um ein optimal trainiertes Netz handeln, sondern es ist auch möglich während des Trainings, in regelmäßigen Abständen Checkpoints zu schreiben, die zum Wiederherstellen eines Trainingszwischenschritts dienen können. In diesem Abschnitt wird jedoch behandelt, wie Neuronale Netze von Grund auf implementiert und trainiert werden. Ausführliche Informationen zum Wiederverwenden von Netzen finden sich in Aurélien Gérons Buch [10] oder in der TensorFlow Dokumentation[16].

In Listing 5.10 ist zu sehen, wie ein sehr einfacher Berechnungsgraph erstellt wird. Wichtig ist, dass `print(berechnungsgraph)` nicht wie erwartet den Wert 42 ausgibt, sondern: `Tensor("add:0", shape=(), dtype=int32)`.
Tensor ist der zentrale Datentyp bei TensorFlow, in diesem Fall entspricht er einem einfachen Wert, was dargestellt wird durch: `shape=()`. In den meisten Fällen stellt ein Tensor jedoch eine multidimensionale Matrix dar.

```
1 import tensorflow as tf
2
3 a = tf.Variable(initial_value=5)
4 b = tf.Variable(initial_value=6)
5 c = tf.Variable(initial_value=12)
6
7 berechnungsgraph = a * b + c
8 print(berechnungsgraph)
```

Listing 5.10: Erstellen eines einfachen Berechnungsgraphen mit TensorFlow.

Der Grund für die Tatsache, dass nicht der erwartete Wert ausgegeben wird, ist der, dass hier lediglich der Berechnungsgraph erstellt wurde. Wie bereits angesprochen, wird eine Session benötigt, um die von einem Berechnungsgraphen dargestellte Berechnungen auszuführen.

Im Gegensatz zum Berechnungsgraphen, der Tensoren und Operationen verbindet, repräsentiert eine *Session* die unterschiedlichen zur Verfügung stehenden Ressourcen[17]. Um den Berechnungsgraphen aus Listing 5.10 auszuführen, muss zuerst eine Instanz der Session Klasse erstellt werden, die anschließend das Ausführen übernimmt. Mit einem `with`- Block der Programmiersprache Python kann komfortabel das Erstellen und Löschen des Session Objekts und das damit in Verbindung stehende Reservieren und Freigeben der Ressourcen implementiert werden. In Listing 5.11 ist das Ausführen des Graphen zu sehen.

```
1 with tf.Session() as session:
```

[16] Wiederverwenden von Neuronalen Netzen: `https://goo.gl/jGBe8x`
[17] Informationen über `Graph` und `Session` Klasse: `https://goo.gl/xaxCZX`

```
2   init = tf.global_variables_initializer()
3   init.run()
4   ergebnis = session.run(fetches=berechnungsgraph)
5
6 print(ergebnis)
```

Listing 5.11: Erstellen einer TensorFlow Session und Ausführen des Graphen.

Wie bei vielen Programmiersprachen muss eine Variable nach dem Deklarieren, das in Listing 5.10 zu sehen ist, vor dem ersten Zugriff initialisiert werden, TensorFlow bietet hierfür eine Methode, die es ermöglicht, eine weitere Operation im Graphen zu erstellen, die beim Ausführen alle Variablen initialisiert. Das hier ausgegebene Ergebnis entspricht dem erwarteten Wert: 42.

5.3.2 Implementierung des CNN und RNN mit TensorFlow

Für Berechnungsgraphen, die das Trainieren von Convolutional oder Recurrent Neural Networks darstellen, bietet TensorFlow Methoden an, die größere Teile des gesamten Graphen erstellen und für die Berechnung von ganzen Schichten der Netze stehen.

5.3.2.1 Implementieren des CNN

Im Modul `layers` werden unterschiedliche Methoden angeboten, die Entwickler beim Erstellen eines Neuronalen Netzes unterstützen. Für das Implementieren eines CNNs sind vor allem Convolutional und Pooling Schichten interessant (vgl. Abschnitt 2.2).
Die Parameter der Methoden `conv2d()` beziehungsweise `max_pooling2d()` spiegeln hierbei die wichtigen Eigenschaften der Schichten wider.

```
1  conv1 = tf.layers.conv2d(inputs=input_matrix, filters=20, kernel_size=(5, 5),
2              activation=tf.nn.tanh)
3  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=(2, 2), strides=2)
4
5  conv2 = tf.layers.conv2d(inputs=pool1, filters=50, kernel_size=(5, 5),
6              activation=tf.nn.tanh)
7  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=(2, 2), strides=2)
8
9  flatten = tf.contrib.layers.flatten(inputs=pool2)
10 fully = tf.layers.dense(inputs=flatten, units=500, activation=tf.nn.tanh)
```

Listing 5.12: Ausschnitt der LeNet-5 Implementierung mit TensorFlow.

In Listing 5.12 ist deutlich zu erkennen, dass TensorFlow die Komplexität der einzelnen Schichten weitestgehend abstrahiert und Entwicklern eine einfache API bietet.

5.3.2.2 Implementieren des RNN

Ein weiteres, das `contrib` Modul, enthält Code, der teilweise stark in der Entwicklung ist. Funktionen oder ganze Unterpakete können sich von Version zu Version unterscheiden,

oder nicht mehr vorhanden sein[18].
Für das Entwickeln von RNNs wird das Paket `contrib.rnn` zur Verfügung gestellt. In Listing 5.13 ist zu sehen, dass TensorFlow auch bei RNNs viele der Eigenheiten, wie zum Beispiel das Stapeln und rekurrente Verbinden der Schichten, abstrahiert.

```
1  def lstm_zelle():
2    return tf.contrib.rnn.BasicLSTMCell(num_units=anzahl_neuronen)
3
4  zellen = [lstm_zelle() for _ in range(anzahl_schichten)]
5  lstm_gestapelt = tf.contrib.rnn.MultiRNNCell(cells=zellen)
6
7  for zeitschritt in range(anzahl_zeitschritte):
8    ausgabe, zustand = lstm_gestapelt(inputs=daten[:, zeitschritt], state=zustand)
9
10 ergebnis = zustand
```

Listing 5.13: Ausschnitt der RNN Implementierung mit TensorFlow.

Weiter ist auch zu erkennen, dass manuell implementiert werden muss, wie weit das Netzwerk zum Trainieren ausgerollt wird (Zeile 7 und 8).

Unabhängig davon, welche Netzart implementiert wurde, muss eine weitere Operation dem Berechnungsgraph hinzugefügt werden, die die Fehlerfunktion berechnet. Für sehr viele gängige Problemstellungen, beziehungsweise Netzarten, bietet TensorFlow Fehlerfunktionen. Die beiden in dieser Arbeit verwendeten sind in Listing 5.14 dargestellt.

```
1 xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
2                     labels=labels)
3 cnn_fehlerfkt = tf.reduce_mean(input_tensor=xentropy)
4
5 rnn_fehlerfkt = tf.contrib.seq2seq.sequence_loss(logits=logits, targets=labels,
6                     tf.ones(shape=[batch_groese, anzahl_zeitschritte]))
```

Listing 5.14: Fehlerfunktionen der Neuronalen Netze.

Wie im Abschnitt 2.1.3.1 beschrieben ist, wird beim Trainieren des Netzes versucht, den Fehler zu minimieren. TensorFlow bietet für den letzten Schritt, der die Gradienten und die daraus resultierenden neuen Trainingsparameter berechnet, diverse sogenannte Optimierer an. Mithilfe dieser, kann die letzte Operation des Berechnungsgraphen, wie in Listing 5.15 zu sehen, erstellt werden.

```
1 optimierer = tf.train.GradientDescentOptimizer(learning_rate=lernrate)
2 trainings_operation = optimierer.minimize(loss=fehlerfkt)
```

Listing 5.15: Erstellen eines Optimierer Objekts und der Trainingsoperation.

Das Ausführen der `trainings_operation` führt dazu, dass genau ein Trainingsschritt mit den übergebenen Trainingsdaten berechnet wird und die neuen Trainingsparameter gespeichert werden. Um das Training verteilt auszuführen, müssen Änderungen des Codes vorgenommen werden, die im Folgenden beschrieben sind.

[18]Informationen des `contrib` Moduls von: `https://www.tensorflow.org/api_docs/python/tf/contrib`

5.3.3 Codeanpassungen um Neuronalen Netze verteilt zu Trainieren

Um das Training über mehrere Maschinen zu verteilen, müssen an unterschiedlichen Stellen des Code Veränderungen vorgenommen werden.
Zuerst muss die Umgebung, also die Anzahl der Worker und PS, spezifiziert werden, hierzu wird eine Instanz der `ClusterSpec` Klasse verwendet. Wie in Listing 5.16 zu sehen ist, werden hierzu die Listen aller Worker und PS verwendet, die mithilfe des Helm Charts generiert und als Argument übergeben werden (vgl. Abschnitt 5.2.1).

```
ps_hosts = args.ps_hosts.split(',')
worker_hosts = args.worker_hosts.split(',')
cluster_spec = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})

with tf.device(tf.train.replica_device_setter(cluster=cluster_spec,
    worker_device="/job:worker/task:{}/cpu:0".format(args.task_index))
):
  # Erstellen des Neuronalen Netzes
```

Listing 5.16: Initialisieren und Verwenden der `ClusterSpec` Instanz für die Datenparallelität.

Weiter ist zu erkennen, dass die Implementierung des Neuronalen Netzes in einen `with` Block gekapselt wird. Die Funktion `device()` ermöglicht es, Operationen des Berechnungsgraphen an eine bestimmte Recheneinheit zu binden. Im Fall der Datenparallelität, wie sie in dieser Arbeit zum Einsatz kommt, wird eine weitere Methode, nämlich `replica_device_setter()`, dazu verwendet, um den gesamten Graphen an die lokal vorhandene CPU zu binden. Beim Verwenden von mehreren PS sorgt diese Methode ebenfalls dafür, dass sich jeder Worker per Round Robin mit genau einem verfügbaren PS verbindet.

Eine weitere Änderung ist, dass eine verteilte Trainingssession erstellt werden muss, mit der sich jeder Worker verbindet. Hierfür kann die Klasse `MonitoredTrainingSession` verwendet werden, die sowohl das korrekte Initialisieren und Verwenden übernimmt, also nur ein Worker die Session erstellt und alle anderen nur beitreten. Aber auch weitere Besonderheiten wie beispielsweise das Schreiben und Wiederherstellen von Checkpoints kann diese übernehmen. In Listing 5.17 ist zu erkennen, dass zuvor eine Instanz der Klasse `Server` erstellt werden muss, die genau einen Teilprozess des verteilten Trainings repräsentiert.

```
server = tf.train.Server(server_or_cluster_def=cluster_spec,
                         job_name=job_name, task_index=args.task_index)

with tf.train.MonitoredTrainingSession(master=server.target,
  is_chief=(args.task_index== 0)
) as session:
  # Trainingsschleife ...
```

Listing 5.17: Server Objekt Erstellung und Verwenden der *MonitoredTrainingSession*.

Im folgenden und abschließenden Teil wird die Verwendung des Werkzeugs TensorBoard beschrieben.

5.3.4 Verwendung von TensorBoard

Das Werkzeug TensorBoard bietet die Möglichkeit, Werte zu loggen und diese anschließend über eine zeitliche Achse aufzutragen. Die Verwendung basiert ebenfalls auf dem Berechnungsgraphen, denn es wird eine Operation erstellt, die mithilfe der Session ausgeführt wird. Zuvor ist es jedoch notwendig, eine Instanz der Klasse `FileWriter`, wie in Listing 5.18 gezeigt, zu erstellen.

```
1 writer = tf.summary.FileWriter(logdir="/tensorflow/logs")
2
3 tf.summary.scalar(name="accuracy", tensor=genauigkeit)
4 summary_operation = tf.summary.merge_all()
5
6 _, summary_wert = session.run([trainings_operation, summary_operation])
7 writer.add_summary(summary=summary_wert, global_step=globaler_schritt)
```

Listing 5.18: Erstellen einer `FileWriter` Instanz.

Weiter ist in Zeile 3 zu erkennen, dass es lediglich genügt, mit der Methode `scalar()` zu definieren, um welche Datumsart es sich bei der zu schreibenden Metrik handelt. Alternativen für einen skalaren Wert sind: Audio, Bilder, Histogramme und Text. Um eine Operationen dem Berechnungsgraphen hinzuzufügen kann mit `merge_all()` dafür gesorgt werden, dass alle *summaries* zusammengefasst werden. Zum Berechnen genügt das Ausführen der Operation mithilfe einer Session, diese Ergebnisse werden im letzten Schritt der `add_summary()` Methode der `FileWriter` Instanz übergeben. Diese sorgt durch das Verwenden eines Akkumulators dafür, dass die Werte effizient auf die Festplatte geschrieben werden.

5.4 Implementierung von Neuronalen Netze mit MXNet

Auch MXNet nutzt einen Berechnungsgraphen, um die Operationen symbolisch darzustellen (vgl. Abschnitt 3.2.3). Hierzu wird das Modul `symbol` angeboten, dieses bietet die Möglichkeit, einzelne Symbole, also Knoten des Graphen, zu erstellen und mithilfe von verschiedenen Operationen zu verbinden. Ähnlich wie bei TensorFlow wird auch in Listing 5.19 ein unerwarteter Wert ausgegeben, nämlich: `<Symbol _plus1>`.

```
1 import mxnet as mx
2
3 a = mx.sym.Variable("a")
4 b = mx.sym.Variable("b")
5 c = mx.sym.Variable("c")
6
7 berechnungsgraph = a * b + c
8 print(berechnungsgraph)
```

Listing 5.19: Erstellen eines einfachen Berechnungsgraphen mit MXNet.

Dies zeigt, dass auch MXNet das Erstellen und Ausführen des Berechnungsgraphen voneinander trennt.

Die von MXNet verwendete Datenstruktur nennt sich `NDArray` und beschreibt mit ihrem Namen sehr genau, wie sie aufgebaut ist, nämlich als multidimensionales Array. In der Regel werden 32 Bit lange float Zahlen verwendet, dieser Datentyp kann jedoch angepasst werden.

5.4.1 Implementierung des CNN und RNN mit MXNet

Wie bereits angesprochen bietet das Modul `symbol` unterschiedliche Methoden, die zum Erstellen von diversen Knoten des Berechnungsgraphen dienen.

5.4.1.1 Implementieren des CNN

In Listing 5.20 ist ein Ausschnitt der Implementierung des LeNet-5 CNN zu sehen. Auffällig ist der sehr kleine Unterschied zu TensorFlows Funktionen, auch hier wird deutlich, dass alle im Abschnitt 2.2 erläuterten Besonderheiten der CNNs in Form von Parametern wiederzufinden sind.

```
1  conv1 = mx.symbol.Convolution(data=data, kernel=(5, 5), num_filter=20)
2  tanh1 = mx.symbol.Activation(data=conv1, act_type="tanh")
3  pool1 = mx.symbol.Pooling(data=tanh1, pool_type="max", kernel=(2, 2),
4                            stride=(2, 2))
5
6  conv2 = mx.symbol.Convolution(data=pool1, kernel=(5, 5), num_filter=50)
7  tanh2 = mx.symbol.Activation(data=conv2, act_type="tanh")
8  pool2 = mx.symbol.Pooling(data=tanh2, pool_type="max", kernel=(2, 2),
9                            stride=(2, 2))
10
11 flatten = mx.symbol.Flatten(data=pool2)
12 fc1 = mx.symbol.FullyConnected(data=flatten, num_hidden=500)
13 tanh3 = mx.symbol.Activation(data=fc1, act_type="tanh")
```

Listing 5.20: Ausschnitt der LeNet-5 Implementierung mit MXNet.

Anders ist es bei der Implementierung von RNNs. Der von MXNet implementierte Ansatz unterscheidet sich von TensorFlows, das zeigt sich auch in der Form des Quellcodes, der im Folgenden erläutert wird.

5.4.1.2 Implementieren des RNN

Um RNNs die Möglichkeit offen zu lassen, Eingangdaten zu verwenden, die unterschiedliche Längen aufweisen, kann MXNet zur Laufzeit des Trainings das Modell anpassen. Hierzu muss eine Methode implementiert werden, die als einzigen Parameter die Sequenzlänge entgegen nimmt und ein Tripel zurückgibt, das aus dem Symbol, sowie den Namen der Symbole der Eingangsdaten und der Labels besteht. Listing 5.21 zeigt den relevanten Ausschnitt.

```
stapel = mx.rnn.SequentialRNNCell()

for _ in range(anzahl_schichten):
  stapel.add(cell=mx.rnn.LSTMCell(num_hidden=anzahl_neuronen))

def sym_gen_fn(laenge_sequenz):
  # ...
  stapel.reset()
  ausgabe, zustaende = stapel.unroll(length=laenge_sequenz, inputs=eingang)

  # ...

  return ergebnis, "daten", "label"
```

Listing 5.21: Ausschnitt der RNN Implementierung mit MXNet.

Im folgenden Abschnitt wird beschrieben, wie ein erstellter Berechnungsgraph anschließend zum verteilten Trainieren des zu Grunde liegenden Neuronalen Netzes verwendet wird.

5.4.2 Verteiltes Trainieren eines Neuronalen Netzes mit MXNet

In den bisherigen Abschnitten wurde lediglich beschrieben, wie der Berechnungsgraph mithilfe des Moduls `symbol` erstellt werden kann. In diesem Abschnitt wird erläutert wie es möglich ist, den Berechnungsgraphen zum Trainieren der Neuronalen Netze zu verwenden.

Hierzu bietet MXNet das Paket `module`, das das Symbol, das den Berechnungsgraphen repräsentiert, und einen Kontext verbindet. Der Kontext definiert dabei, ob die Berechnungen auf einer GPU oder CPU ausgeführt werden. Im Falle des CNNs kann das Module, wie in Listing 5.22 zu sehen, initialisiert werden.

```
lenet_module = mx.mod.Module(symbol=lenet, context=mx.cpu())
```

Listing 5.22: Initialisieren des CNN Modules.

Bedingt durch das Verwenden eines RNNs, dessen Eingangsdaten von dynamischer Größe sein können, unterscheidet sich das hier eingesetzte und in Listing 5.23 zu sehende Module leicht.

```
rnn_module = mx.mod.BucketingModule(sym_gen=sym_gen_fn, context=mx.cpu())
```

Listing 5.23: Initialisieren des RNN Modules.

Da sowohl `Module`, als auch `BucketingModule` von der Oberklasse `BaseModule` erben, können beide dessen `fit()` Methode verwenden. Diese übernimmt das gesamte Trainieren des zu Grunde liegenden Neuronalen Netzes. Mithilfe der Parameter dieser Methode kann komfortabel, wie in Listing 5.24 zu sehen ist, sowohl die zu verwendende Metrik, als auch der Optimierer und vieles mehr konfiguriert werden.

```
module.fit(
  train_data=trainings_daten,
```

```
3   eval_data=test_daten,
4   eval_metric=mx.metric.Perplexity(0),
5   kvstore=mx.kvstore.create("dist_async"),
6   optimizer="sgd",
7   optimizer_params={"learning_rate": lernrate},
8   num_epoch=anzahl_epochs
9 )
```

Listing 5.24: Starten des verteilten Trainings mit MXNet.

Alleinig der Parameter `kvstore` ist ausschlaggebend dafür, ob und wie das Training verteilt ausgeführt wird. Alternativen für `"dist_async"` wären die Werte: `"dist_sync"` und `"local"`. Hierbei wird entweder das verteilte Training mit synchroner Datenparallelität ausgeführt, oder gar nicht verteilt und nur mit der lokalen Maschine gearbeitet.

Im nächsten und abschließenden Abschnitt dieses Kapitels wird kurz beschrieben wie mithilfe von TensorBoard die verwendeten Metriken der Neuronalen Netze gesammelt werden können.

5.4.3 Verwendung von TensorBoard mit MXNet

Es gibt ein Python Paket, das es ermöglicht, Daten im TensorBoard Format zu sammeln. Nach dem Importieren dieses Pakets ist es ähnlich wie bei TensorFlow möglich, eine `FileWriter` Instanz zu initialisieren. Die `fit()` Methode bietet mit dem Parameter `batch_end_callback` die Möglichkeit, eine oder mehrere Callback Funktionen zu registrieren, die nach dem Beenden eines Trainingsdurchlaufs aufgerufen wird. In Listing 5.25 Zeile 7 ist zu erkennen, dass aus dem Parameter der Callback Funktion die aktuelle Metrik des Trainingsdurchlaufs ausgelesen werden kann. Anschließend wird diese an die FileWriter Instanz übergeben, der das Schreiben der Daten auf die Festplatte übernimmt.

```
1 from tensorboard.writer import FileWriter
2 from tensorboard.summary import scalar
3
4 writer = FileWriter(logdir="/mxnet/logs")
5
6 def writer_callback(params):
7   wert = params.eval_metric.get()[1]
8   writer.add_summary(summary=scalar(name="accuracy", data=wert),
9                      global_step=globaler_schritt)
```

Listing 5.25: Verwenden von TensorBoard in Verbindung mit MXNet.

6 Durchführung und Auswertung experimenteller Versuche

Nach dem Aufbau der Infrastruktur und der Implementierung der Neuronalen Netze, die im vergangenen Kapiteln vorgestellt wurden, gilt es nun, Versuche durchzuführen und Daten zu sammeln.

In unterschiedlichen Experimente wurden separat für RNNs und CNNs diverse Versuche durchgeführt. Die Ergebnisse sind in den folgenden Abschnitten beschrieben.

6.1 Versuchsdurchführung: Skalieren des Trainings von CNNs

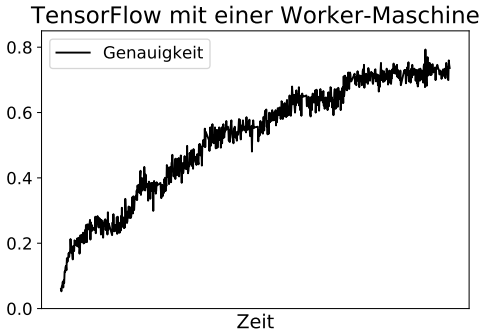
Beim verwendeten CNN LeNet-5, gibt es weder für MXNet, noch für TensorFlow Informationen, welche Konfigurationen verwendet werden sollen, um ein möglichst gutes und schnelles Ergebnis zu erhalten. Dies liegt vor allem an der Tatsachen, dass dieses Netzwerk nicht aus einem Tutorial entnommen wurde, oder davon inspiriert ist. Aus diesem Grund beginnt das erste Experiment der folgenden Reihe damit, eine Konfiguration zu finden, die für möglichst beide Frameworks ein gutes Ergebnis erzielt.

Wenn keine weiteren Informationen vorhanden sind, wurden die Versuche mit einer Abstufung von 1, 2, 5, 10, 25 und 50 Worker-Maschinen und im Falle von TensorFlow mit 1 Parameter Server durchgeführt.

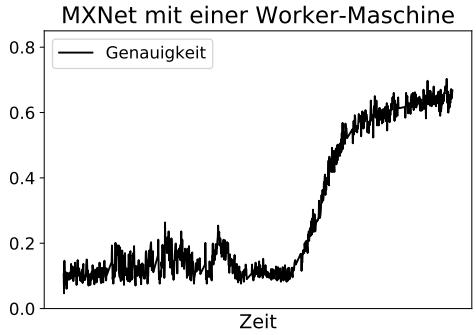
6.1.1 Experiment 1: Finden der passenden Konfiguration

Beginnend mit einer Batchgröße von 512 und einer Lernrate von 0.1 wurden die ersten Versuche über insgesamt 8 Epochs durchgeführt. Dabei wurde deutlich, dass MXNet bei einer geringen Anzahl an Workern von 1 oder 2 Maschinen Lernfortschritte machte bei 5 oder mehr Maschinen blieb dieser jedoch aus. Eine Halbierung der Lernrate auf 0.05 half, diesen Effekt zu verschieben, somit konnte das CNN auch mit 10 Worker-Maschinen trainiert werden. Nach einer weiteren Halbierung der Lernrate auf 0.025 war es möglich, 25 Maschinen zu verwenden. TensorFlow hingegen zeigte lediglich in der Lerngeschwindigkeit und der maximal erreichten Genauigkeit Unterschiede.

Typische Lernkurven der Frameworks sind in Abbildung 6.1 zu sehen. Dabei ist gut zu erkennen, dass TensorFlow in Abbildung 6.1a direkt nach Beginn und über das gesamte Training hinweg eine steigende Genauigkeit aufweist. MXNet hingegen, wie es in Abbildung 6.1b dargestellt ist, startet mit einem Plateau, das sich bis über die Hälfte der Trainingsdauer zieht. Trotzdem ist das Endergebnis von MXNet nicht deutlich schlechter als das von TensorFlow, denn wie ebenfalls zu erkennen ist, ist die Lernkurve von MXNet steiler.



(a) Genauigkeitsverlauf des Trainings mit TensorFlow.



(b) Genauigkeitsverlauf des Trainings mit MXNet.

Abbildung 6.1: Training des CNN über 8 Epochs mit Batchgröße 512 und Lernrate 0.025.

Da es mit den Konfigurationen, die in Abbildung 6.1 verwendet wurden, möglich war, mit beiden Frameworks positive Ergebnisse zu erzielen, werden diese für weitere Versuche verwendet; sprich eine Batchgröße von 512, eine Lernrate von 0.025 und insgesamt 8 Epochs Training.

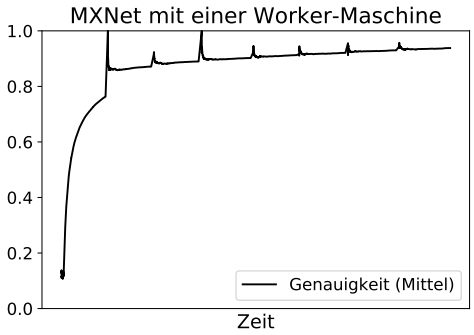
6.1.2 Experiment 2: Trainingsverlauf mit sehr kleiner Batchgröße

Nach den Graphen in Abbildung 6.1 stellt sich die Frage, warum ein solches Plateau in der Lernkurve von MXNet entsteht. Weiter ist aufgefallen, dass beim Erhöhen der Worker-Maschinen Anzahl schneller begonnen wird zu lernen. Dies liegt daran, dass sich durch die Verwendung von mehreren Maschinen der Durchsatz an Trainingsdaten erhöht, was die Dauer verkürzt, bis ein Lerneffekt eintritt.

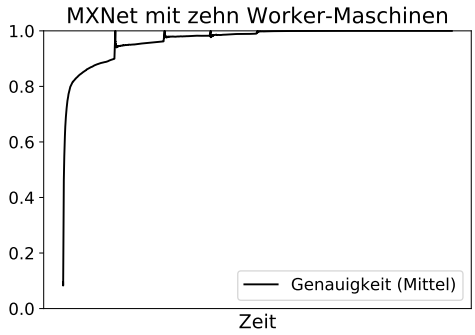
Eine Menge von 512 Beispielen pro Batch ist im Verhältnis zu den insgesamt 60.000 Trainingsbeispielen, die der MNIST Datensatz enthält, groß. Daher ist die Batchgröße in diesem Experiment mit 5 Bilder sehr klein gewählt. Das Ergebnis der Messungen mit 1 und 10 Worker-Maschinen sind in Abbildung 6.2 dargestellt. Kurven sind deutlich glatter, weil hier die mittlere Genauigkeit innerhalb eines Batches dargestellt ist. Die hohe Dynamik der Kurven aus dem vergangenen Abschnitt würde bei einer sehr kleinen Batchgröße dazuführen, dass das Wesentliche nicht mehr erkennbar ist.

Es fällt auf, dass das Plateau deutlich kleiner ausfällt und der Lernprozess somit um einiges schneller zu einer hohen Genauigkeit kommt. Weiter wird deutlich, dass bei der Verwendung von 10 Worker-Maschinen das Plateau quasi nicht mehr vorhanden ist.
Diese Versuche zeigen, dass bei der Verwendung von MXNet, das Finden der Richtigen Konfigurationen, auch *Hyperparameter tuning*[1] genannt, für ein effizientes Training, sehr wichtig ist. Ohne gute Konfigurationswahl kann es im Gegensatz zu TensorFlow sogar passieren, dass beim Trainieren keine Fortschritte erzielt werden.

[1]Hyperparameter sind Parameter, die vom Programmierer geändert werden müssen und nicht durch das Training geändert werden (bspw. Batchgröße, Lernrate, ...) [10].



(a) Genauigkeitsverlauf des Trainings mit einer Worker-Maschine.



(b) Genauigkeitsverlauf des Trainings mit zehn Worker-Maschine.

Abbildung 6.2: Training des CNN mit MXNet über 8 Epochs mit Batchgröße 5 und Lernrate 0.025.

Weiter ist zu erkennen, dass MXNet bei einer geringeren Batchgröße ein deutlich besseres Ergebnis erreicht.

6.1.3 Experiment 3: Skalierbarkeit der Frameworks mit Batchgröße 512

Nach dem Finden der Konfigurationen: Batchgröße 512, Lernrate 0.025 und Epochs 8, wurde mit jedem Framework Messungen durchgeführt. Die aus den Versuchen resultierende

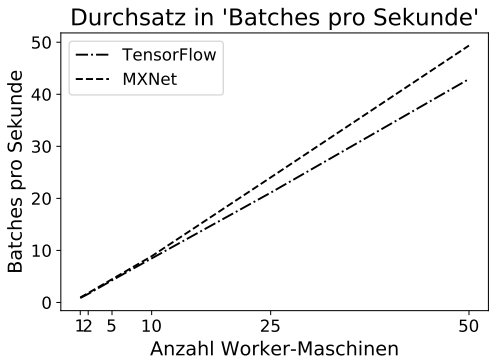


Abbildung 6.3: Messung des Durchsatzes mit unterschiedlicher Worker-Maschinen Anzahl.

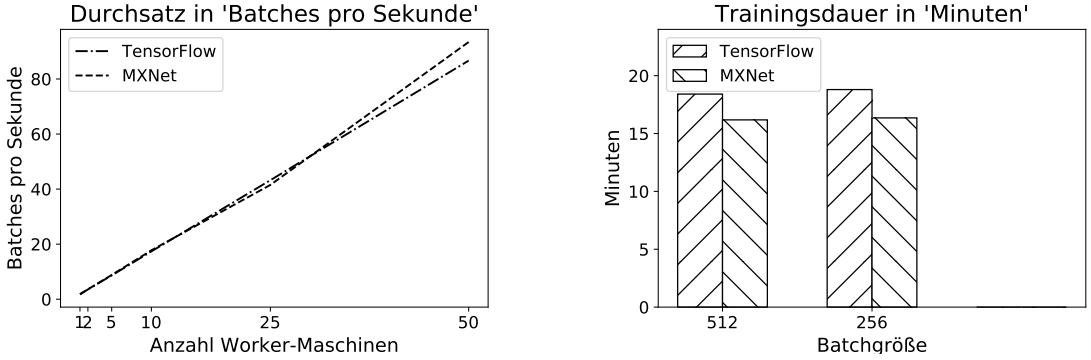
Messwerte sind in der Abbildung 6.3 dargestellt. Es ist zu erkennen, dass beide Frameworks bis zu einer Worker-Maschinen Anzahl von 50 Stück linear skalieren, wenngleich der Durchsatz von MXNet höher als der von TensorFlow ist. Der Grund dafür, dass beide Frameworks linear skalieren, liegt daran, dass das Verhältnis zwischen der Rechenzeit, die benötigt wird, um ein Trainingsdurchlauf mit 512 Beispielen zu berechnen, und dem Kommunikationsaufwand, der entsteht, wenn die lokalen Ergebnisse der Worker ausgetauscht und aggregiert werden müssen, sehr hoch ist.

Typischerweise würden die Graphen einen nicht linearen Bereich aufweisen, wenn dieses Verhältnis kleiner wäre. Hierzu könnte zum einen die Anzahl der Worker-Maschinen

weiter erhöht werden, oder, was in dem folgenden Experiment getestet wird, die Batchgröße verringert. Dadurch, dass weniger Trainingsdaten für einen Trainingsdurchlauf verwendet werden, sinkt die Berechnungszeit und mit ihr das Verhältnis zwischen Rechenzeit und Kommunikationsaufwand. Getrieben von dieser Vermutung wurde im nächsten Experiment die Batchgröße auf 256 halbiert.

6.1.4 Experiment 4: Skalierbarkeit der Frameworks mit Batchgröße 256

Nach dem Halbieren der Batchgröße ist zu erkennen, dass sich die beiden Kurven, die in Abbildung 6.4a zu sehen sind, im Vergleich zum vorherigen Experiment einander deutlich genähert haben. Weiterhin skalieren jedoch beide Frameworks linear. Aus diesem Grund wurde im fünften Experiment die Batchgröße erneut auf 128 Beispiele halbiert.



(a) Messung des Durchsatzes mit unterschiedlicher Worker-Maschinen Anzahl.

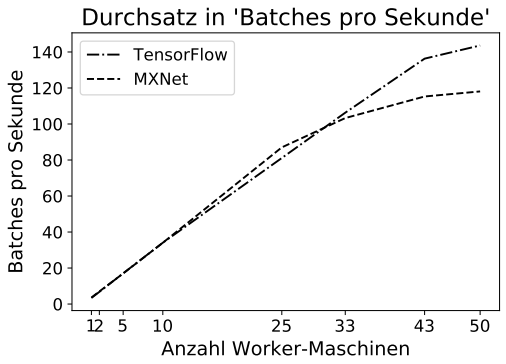
(b) Grundtrainingsdauer der Frameworks mit unterschiedlicher Batchgröße.

Abbildung 6.4: Gegenüberstellung des Durchsatzes und der Grundtrainingsdauer bei einer Batchgröße von 256 Bilden.

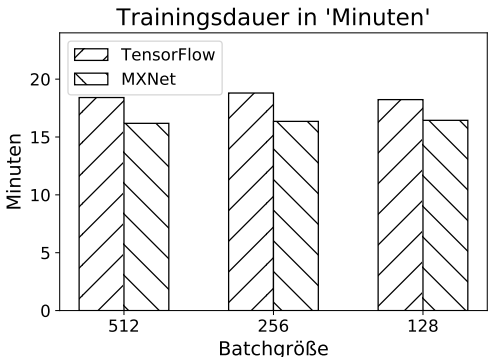
Die Abbildung 6.4b zeigt, dass es für keines der Frameworks, bezogen auf die Trainingsdauer mit einer Worker-Maschine, einen Unterschied macht, ob die Batchgröße 512 oder 256 beträgt. Auch ist zu erkennen, dass MXNet schneller als TensorFlow ist. Die Berechnungszeiten sind zum vorherigen Experiment konstant geblieben und MXNets Durchsatz ist fast identisch zu TensorFlows, was darauf hindeutet, dass die Skalierungseigenschaften von TensorFlow besser sind als die von MXNet.

6.1.5 Experiment 5: Skalierbarkeit der Frameworks mit Batchgröße 128

Mit einer Batchgröße von 128 Beispielen zeigt sich, dass die Kurve nach dem Bereich der linearen Skalierung flacher wird, wie in Abbildung 6.5a zu sehen ist. Wie die Abbildung 6.5b zeigt, ist die Trainingszeit mit 128 Beispielen pro Batch auf etwa demselben Niveau wie mit 256 oder 512. Hier zeigen sich die besseren Skalierungseigenschaften von TensorFlow deutlich. Bis zu einer Anzahl von 25 Worker-Maschinen ist der Durchsatz beider Frameworks annähernd gleich gut, anschließend gelingt es Tensorflow mit steigender Anzahl Worker-Maschinen den Durchsatz stärker zu erhöhen als MXNet.



(a) Messung des Durchsatzes mit unterschiedlicher Worker-Maschinen Anzahl.



(b) Grundtrainingsdauer der Frameworks mit unterschiedlicher Batchgröße.

Abbildung 6.5: Gegenüberstellung des Durchsatzes und der Grundtrainingsdauer bei einer Batchgröße von 128 Bilden.

Da die Kurven bisher lediglich gezeigt haben, wie sich die Frameworks bezüglich der Metrik Batches pro Sekunde verhalten und diese wenig über die möglichen Beschleunigungen aussagt, ist in Abbildung 6.6 der Speedup der Frameworks abgebildet.

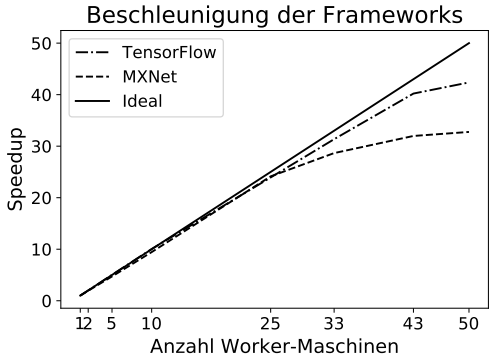


Abbildung 6.6: Speedup der Frameworks beim Trainieren von CNNs mit Batchgröße 128.

Abbildung 6.6 zeigt, dass bis zu einer Anzahl von 25 Worker-Maschinen beide Frameworks etwa linear und fast optimal skalieren. Anschließend nutzt TensorFlow die Erhöhung der Worker-Maschinen Anzahl deutlich besser aus.

6.2 Versuchsdurchführung: Skalieren des Trainings von RNNs

Sowohl MXNet[2], als auch TensorFlow[3] bieteten Tutorials, die dasselbe RNN verwenden, wie jenes, das in dieser Arbeit ebenfalls implementiert wurde. Daher fällt das erste Experiment, das zum Finden der passenden Konfigurationen diente, weg.

[2]Bucketing Tutorial: `https://mxnet.incubator.apache.org/how_to/bucketing.html`
[3]RNN Tutorial: `https://www.tensorflow.org/tutorials/recurrent`

Aus den beiden Tutorials, die unterschiedliche Konfigurationen vorstellen, wurde eine in der Mitte liegende gewählt, um die Versuche durchzuführen. Hierbei wurde die Batchgröße mit 20 Wörtern, die Lernrate mit 0.05 und 4 Epochs gewählt.

6.2.1 Experiment 1: Skalierbarkeit der Frameworks mit Batchgröße 20

Nach dem ersten Experiment und dem Visualisieren der Messergebnisse, die in Abbildung 6.7 zu sehen sind, ist aufgefallen, dass der Unterschied zwischen den beiden Frameworks deutlich größer als bei der CNN Implementierung ist. TensorFlow zeigt sich hinsichtlich

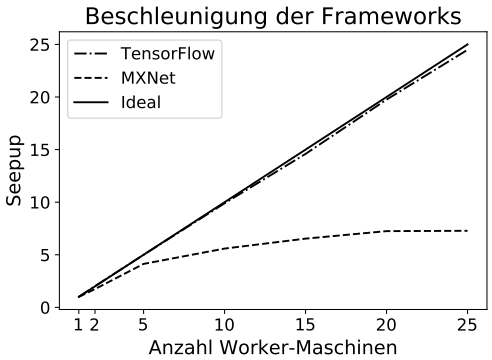
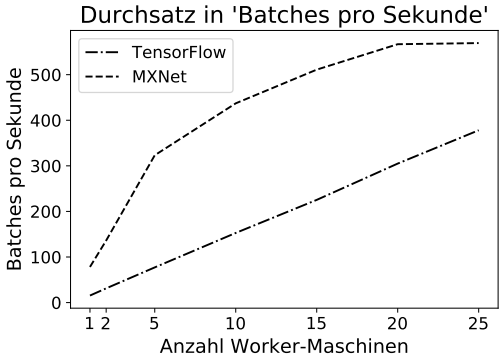


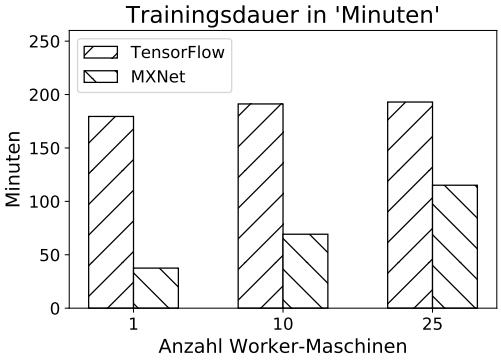
Abbildung 6.7: Speedup der Frameworks beim Trainieren von RNNs mit Batchgröße 20.

des Speedups um einiges besser und skaliert bis zu einer Anzahl von 25 Worker-Maschinen etwa linear und fast ideal. MXNet hingegen skaliert bis zu 5 Worker-Maschinen annähernd linear, anschließend wird die Skalierungseigenschaft, gemessen am Speedup, zunehmend schlechter, bis ab etwa 20 Worker-Maschinen keine Steigerung mehr möglich ist.

Betrachtet man hingegen jedoch die Kurven, die die Messwerte der Batches pro Sekunde visualisieren, zeigt sich ein anderes Bild, siege Abbildung 6.8a zeigt. In dem betrachteten



(a) Messung des Durchsatzes mit unterschiedlicher Worker-Maschinen Anzahl.



(b) Trainingsdauer der Frameworks mit unterschiedlicher Worker-Maschinen Anzahl.

Abbildung 6.8: Gegenüberstellung des Durchsatzes und der Trainingsdauer bei einer Batchgröße von 20 Wörtern.

Bereich der Worker-Maschinen Anzahl liegt der Durchsatz von TensorFlow immer unter

dem von MXNet. Abbildung 6.8b stellt weiter dar, dass auch hier die Differenz an Batches pro Sekunde durch die größere Berechnungsdauer von TensorFlow bedingt ist.

Um zu testen, wie sich die Frameworks verhalten, wenn das Verhältnis zwischen Rechenzeit und Kommunikation geringer ist, wird im nächsten Experiment die Batchgröße auf 5 Wörter reduziert.

6.2.2 Experiment 2: Skalierbarkeit der Frameworks mit Batchgröße 5

Nach dem Visualisieren der Messergebnisse, die in Abbildung 6.9 zu sehen sind, zeigt sich, dass sich sowohl die Beschleunigungen von TensorFlow als auch von MXNet verschlechterten. Beiden Frameworks ist es möglich, bis zu einer Worker-Maschinen Anzahl

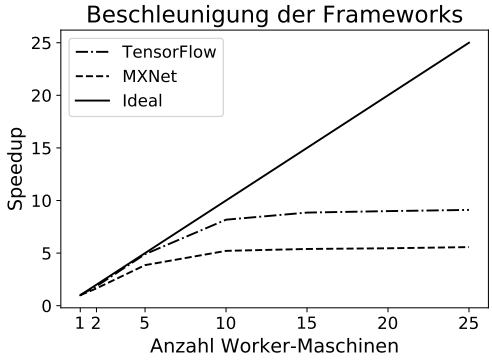
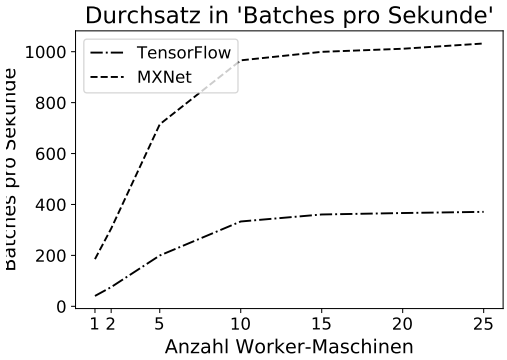


Abbildung 6.9: Speedup der Frameworks beim Trainieren von RNNs mit Batchgröße 5.

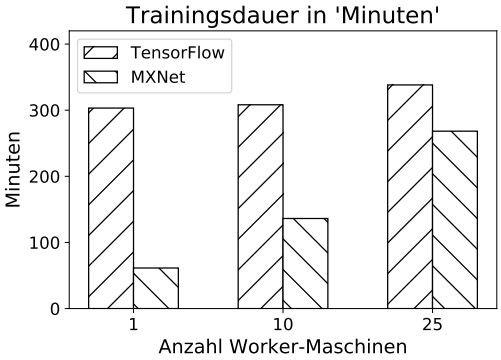
von 5 Stück etwa linear zu skalieren. TensorFlow, das bis dahin sogar fast ideal skalierte, kann, wenn auch deutlich langsamer, den Durchsatz bis zu 15 Worker-Maschinen weiter erhöhen. Die MXNet Kurve liegt immer unter der von TensorFlow und ist überall flacher, was dazu führt, dass es MXNet nur bis zu einer Worker-Maschinen Anzahl von 10 Stück möglich ist, zu skalieren. Im Vergleich zum vorherigen Experiment fällt ebenfalls auf, dass die Verschlechterungen der Beschleunigungen von MXNet deutlich geringer sind, als die von TensorFlow.

Obwohl die Beschleunigungen TensorFlows erneut in jedem der betrachteten Fälle besser ist als die von MXNet, zeigt die Abbildung 6.10a, dass MXNets Durchsatz deutlich über dem von TensorFlow liegt. Der Abstand zwischen diesen beiden Kurven hat sich sogar vergrößert, was vor allem an, im Vergleich zum vorherigen Experiment, TensorFlows schlechteren Skalierbarkeit liegt. Denn ab etwa 10 Worker-Maschinen ist der Zuwachs des Durchsatzes bei beiden Frameworks gering.

Die Abbildung 6.10b zeigt, dass, im Vergleich zu den vorherigen Experimenten, die Grundrechenzeit beider Frameworks höher ist. Ähnlich ist jedoch, dass sich bei TensorFlow die Rechenzeit mit steigender Anzahl Worker-Maschinen kaum erhöht, wohingegen MXNet ein deutlicher Zuwachs der Rechenzeit aufweist.
Auch hier ist die bessere Skalierungseigenschaft von TensorFlow zu erkennen. Fügt man bei 10 oder mehr Worker-Maschinen noch weitere hinzu, dann steigt sowohl der Durchsatz, als auch die Berechnungsdauer nur marginal.
Bei MXNet hingegen ist ein Zuwachs des Durchsatzes jenseits der 10 Worker-Maschinen



(a) Messung des Durchsatzes mit unterschiedlicher Worker-Maschinen Anzahl.



(b) Trainingsdauer der Frameworks mit unterschiedlicher Worker-Maschinen Anzahl.

Abbildung 6.10: Gegenüberstellung des Durchsatzes und der Trainingsdauer bei einer Batchgröße von 20 Wörtern.

ebenso kaum möglich. Das Hinzufügen weiterer Worker-Maschinen führt lediglich dazu, dass sich die Berechnungszeit deutlich erhöht.

6.3 Zusammenfassung der Experimente und Bewertung der Frameworks

Die Ergebnisse der Versuche, die in den vergangenen Abschnitten beschrieben wurden, lassen sich wie folgt zusammenfassen: Die Implementierung der Neuronalen Netze in Bezug auf den Durchsatz beim Training ist bei MXNet effizienter als bei TensorFlow. Die Skalierungseigenschaften sind bei TensorFlow jedoch besser ausgeprägt.
Sowohl bei CNNs, als auch bei RNNs konnte beobachtet werden, dass MXNets Durchsatz höher ist, als der von TensorFlow. Besonders bei komplexeren Netzarten zeigt sich die höhere Effizienz der Implementierung durch einen deutlich höheren Durchsatz. Beim Verwenden von einfacheren Netzen ist der Unterschied des Durchsatzes zwischen TensorFlow und MXNet geringer. Bei den durchgeführten Experimenten war es TensorFlow nur beim Trainieren des implementierten CNN mit einer Batchgröße von 128 Bildern und mehr als 30 Worker-Maschinen möglich, einen höheren Durchsatz als MXNet zu erreichen.
Beim Betrachten der Skalierbarkeit der Frameworks zeigt sich jedoch, dass TensorFlow in jedem der Fälle besser skaliert. Abgesehen vom zweiten Experiment der RNN Implementierung, gelingt es TensorFlow sogar, eine lineare Skalierung des Durchsatzes beizubehalten.

Aus den Messergebnissen lässt sich bewertend ableiten, dass, besonders bei komplexeren Netzarchitekturen, oder beim Arbeiten mit vielen Daten in einem Batch, MXNet das bessere Framework ist. TensorFlow empfiehlt sich nur bei kleinen Neuronalen Netzen, die eine möglichst einfach zu berechnende Struktur aufweisen und mit ebenso kleinen Batchgrößen trainiert werden. Nur dann ist es möglich, dass die gezeigten besseren Skalierungseigenschaften TensorFlows die bessere Implementierung MXNets schlagen und eine höhere Performanz hinsichtlich des Durchsatzes beim verteilten Trainieren erreicht wird.

7 Zusammenfassung und weiterführende Arbeiten

In dieser Arbeit wurde die horizontale Skalierung von generischen Deep Learning Frameworks für die Programmiersprache Python experimentell evaluiert. Dabei kam der Service Kubernetes Engine der Google Cloud Platform zum Einsatz, bei dem jeder Knoten mit genau einer CPU des Typs *2.5 GHz Intel Xeon E5 v2* bestückt ist.

Hierzu wurde zuerst ein Überblick über die verfügbaren Deep Learning Frameworks geschaffen und anhand von ausgearbeiteten Aspekten eine Auswahl getroffen. Nach der anschließenden Diskussion diverser Aufgabenstellungen, sind die Datensätze und Metriken der Aufgaben erläutert worden, die eine umfangreiche Evaluation versprechen. Nach der Herleitung und Erläuterung wurden sowohl die infrastrukturelle Basis, als auch die notwendigen Neuronalen Netze implementiert.

Bei den darauffolgenden Experimenten konnte gezeigt werden, dass MXNet eine deutlich effizientere Implementierung der Neuronalen Netze aufweisen kann. Besonders bei komplexen Netzarten, wie den RNNs, zeigte sich dies durch den deutlich höheren Durchsatz und eine geringere Trainingszeit. Beim Implementieren von CNNs ist der Unterschied der Trainingsdauer deutlich geringer, was sich ebenso durch einen geringeren Unterschied bezüglich der Durchsätze von MXNet und TensorFlow zeigte.
In den betrachteten Fällen war es TensorFlow lediglich beim Trainieren des implementierten CNN mit einer kleinen Batchgröße von 128 Bildern und mit mehr als 30 Worker-Maschinen möglich, einen höheren Durchsatz als MXNet zu erreichen.

Zusammenfassend lässt sich sagen, dass MXNet immer dann einen Vorteil hat, wenn die Berechnungszeit in großem Verhältnis zum Kommunikationsaufwand steht. Dies ist insbesondere bei komplexen Netzarten, aber auch bei größeren Batches der Fall. Der Einsatz von TensorFlow ist nur bei einfachen Netzen mit kleiner Batchgröße und vielen Worker-Maschinen interessant.

Weiterführende Arbeiten

Im Verlauf dieser Arbeit sind diverse Ideen entstanden, die in weiterführenden Arbeiten verwertet werden könnten, folgend eine kurze Beschreibung:
Ein weiteres Framework, das auf Basis dieses Versuchsaufbaus evaluiert werden kann, ist CNTK. Dieses besitzt ebenfalls alle nötigen Eigenschaften (vgl. Abschnitt 3.3) und wurde in dieser Thesis aus zeitlichen Gründen nicht weiter betrachtet.

TensorFlow und MXNet nutzen beide das Konzept des Parameter Servers, jedoch setzt MXNet diesen Ansatz leicht anders um und implementiert einen sogenannten zweistufigen KVStore (vgl. Abschnitt 3.2.3). Bei der Verwendung von Worker-Maschinen, die mehrere Recheneinheiten besitzen, werden diese zuerst lokal aggregiert. Zur Aggregation

der Trainingsparameter über alle Worker-Maschinen hinweg, werden lediglich die Zwischenwerte verwendet. Ein Vergleich zwischen den beiden Implementierungen in Form einer experimentellen Evaluation verspricht hierbei ebenfalls interessante Ergebnisse.

Zuletzt sollte erwähnt werden, dass die Durchführungen dieser Versuchsreihe auf Basis von GPUs, Ergebnisse erzielen sollten, die einen direkten Vergleich für die Praxis liefern. Wie im Abschnitt 4.4 beschrieben ist, war diese Möglichkeit zum Beginn dieser Arbeit mithilfe von Kubernetes nur in einem experimentellen Zustand gegeben und wurde deshalb nicht weiter verfolgt.

Literatur

[1] Nipun Agarwala, Yuki Inoue und Axel Sly. “Music Composition using Recurrent Neural Networks”. In: (2018). https://web.stanford.edu/class/cs224n/reports/2762076.pdf.

[2] Yoshua Bengio. “Deep Learning of Representations for Unsupervised and Transfer Learning”. In: *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*. UTLW’11. Washington, USA, 2011, S. 17–37.

[3] N. Buduma und N. Locascio. *Fundamentals of Deep Learning: Designing Next- Generation Machine Intelligence Algorithms*. O’Reilly Media, 2017.

[4] Heng-Tze Cheng u. a. “Wide & Deep Learning for Recommender Systems”. In: *CoRR* abs/1606.07792 (2016). arXiv: 1606.07792.

[5] Paul Covington, Jay Adams und Emre Sargin. “Deep Neural Networks for YouTube Recommendations”. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. New York, NY, USA, 2016.

[6] Jeffrey Dean u. a. “Large Scale Distributed Deep Networks”. In: *NIPS*. 2012.

[7] Li Deng und Dong Yu. *Deep Learning: Methods and Applications*. Techn. Ber. Mai 2014. URL: https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/.

[8] *Docker Dokumentation*. https://docs.docker.com. [Online; abgerufen am 03-Jan-2018]. 2018.

[9] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36 (1980), S. 193–202.

[10] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media, 2017.

[11] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[12] Karol Gregor u. a. “DRAW: A Recurrent Neural Network For Image Generation”. In: *CoRR* abs/1502.04623 (2015).

[13] *Helm Dokumentation*. https://docs.helm.sh. [Online; abgerufen am 03-Jan-2018]. 2018.

[14] J. J. Hopfield. “Neural networks and physical systems with emergent collective computational abilities”. In: *Proceedings of the National Academy of Sciences* 79.8 (1982), S. 2554–2558.

[15] Daniel Jurafsky und James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0130950696.

[16] *Kubernetes Dokumentation*. `https://kubernetes.io/docs/`. [Online; abgerufen am 03-Jan-2018]. 2018.

[17] Yann Lecun u. a. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, S. 2278–2324.

[18] Honglak Lee u. a. "Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks". In: *Commun. ACM* 54.10 (2011), S. 95–103.

[19] Mu Li u. a. "Scaling Distributed Machine Learning with the Parameter Server". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. USENIX Association, 2014, S. 583–598.

[20] Martin Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[21] Warren S. McCulloch und Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), S. 115–133.

[22] Josh Patterson. *Deep learning : a practitioners approach*. OReilly, 2017.

[23] C. Ramya, G. Kavitha und K. S. Shreedhara. "Recalling of Images using Hopfield Neural Network Model". In: *CoRR* abs/1105.0332 (2011).

[24] Microsoft Research. *Microsoft Cognitive Toolkit - CNTK - Documentation*. URL: `https://docs.microsoft.com/en-us/cognitive-toolkit/index` (besucht am 30. 10. 2017).

[25] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), S. 65–386.

[26] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Hrsg. von David E. Rumelhart und James L. Mcclelland. MIT Press, 1986, S. 318–362.

[27] Yaniv Taigman u. a. "DeepFace: Closing the Gap to Human-Level Performance in Face Verification". In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, S. 1701–1708. ISBN: 978-1-4799-5118-5.

[28] Deeplearning4j Development Team. *Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0*. URL: `http://deeplearning4j.org`.

[29] ND4J Development Team. *ND4J: N-dimensional arrays and scientific computing for the JVM, Apache Software Foundation License 2.0*. URL: `http://nd4j.org`.

[30] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *CoRR* abs/1605.02688 (2016), S. 19.

[31] Tianqi Chen u. a. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. In: *CoRR* abs/1512.01274 (2015), S. 6.

[32] Yonghui Wu u. a. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[33] Han Xiao, Kashif Rasul und Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 28. Aug. 2017.

[34] Wojciech Zaremba, Ilya Sutskever und Oriol Vinyals. “Recurrent Neural Network Regularization”. In: *CoRR* abs/1409.2329 (2014).