Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Bachelorarbeiten

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

David Baumgartner

Hagenberg, Juli 2017

software engineering
IKM Fakultät Hagenberg

Inhalt

Eidesstattliche Erklärung

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

Datum, Unterschrift

Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Machine Learning und tiefe neuronale Netze mit TensorFlow

Theoretische
Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

David Baumgartner

Begutachtet von Prof.(FH) Priv.-Doz. DI Dr. Stephan Dreiseitl

Hagenberg, Mai 2017

software engineering
IKM Fakultät Hagenberg

Inhaltsverzeichnis

Kurzfassung

Neuronale Netzwerke werden seit Jahrzehnten erforscht und auch seit geraumer Zeit eingesetzt. Solche Netzwerke ermöglichen es, komplexe Systeme für logisch sehr komplexe Aufgaben zu entwickeln. Zum Beispiel werden bei der Übersetzung von Texten in Bildern neuronale Netzwerke eingesetzt. Diese Ausführung geschieht dabei zum Teil auf den Smartphones lokal. Eine lokale Ausführung war dabei nicht immer möglich, da die dafür benötigten Ressourcen nicht vorhanden waren. Zu Beginn der Erforschung von neuronalen Netzwerken konnten nur Modelle erstellt werden. Modelle wurden deshalb herangezogen, da die technischen Voraussetzungen noch nicht gegeben waren. Mit der Entwicklung von integrierten Schaltkreisen und immer leistungsfähigeren Recheneinheiten konnte diese Grundlage geschaffen werden. Dabei existieren noch Probleme, denn ein maschinell lernendes System muss trainiert werden. Ein solches Training kann zum Beispiel mit einem Schwimmer verglichen werden, der einen neuen Technikablauf integrieren möchte. So ein Vorgang benötigt viel Zeit bis der Ablauf adaptiert wird und im Anschluss fast vollautomatisch abläuft. Ähnlich geht es den neuronalen Netzwerken, welche anhand des Ergebnisses angepasst werden müssen. Der Einzug von maschinell lernenden Systemen in die Zivilgesellschaft ist dank ihres Erfolgs praktisch unumgänglich.
Im Rahmen dieser Bachelorarbeit wurde ein Beispiel anhand realer Daten umgesetzt und implementiert. An diesem Beispiel werden einige Eigenheiten solcher Systeme erklärt und beschrieben. Zusätzlich beinhaltet diese Arbeit auch eine Einführung in das Gebiet der neuronalen Netzwerke. Diese Grundlagen werden mit Hilfe der Bibliothek *TensorFlow* vervollständigt. *TensorFlow* bietet eine guten Einstieg, damit die Grundlagen besser und praktischer verstanden werden. Technisch nicht so versierte Personen sollten nach dem Erlernen der Grundlagen deshalb eine Abstraktion verwenden, wie zum Beispiel *Keras*.

Abstract

Neuronal networks have been reasearched for decades and are growing in commercial usage. They make it possibility to create systems that can solve very complex problems, e.g. the translation of pictures to text. [1] Such systems need computational power which was not available in former times. So at the start there were only models and today we can do that nearly on our smartphones. The development of integrated circuits enabled more research and a more efficient development in that research area. But these systems need more computational power for training than our smartphones currently have. For the training there is time needed to adapt and recognize patterns, like a swimmer how wants to adapt a new technique to be more efficient. In general all these systems get more and more evolved and are reaching a level where they are simultaneously integrated in our daily life.
The present thesis includes an example with real data which was developed to visualize a problem and how it could be solved. Additionally the work contains an introduction to the topic of neuronal networks, it's basics and the basic math behind it. By applying the basics to the framework *TensorFlow*, it will get more practical and understandable. For people who are technically not experienced, frameworks like *Keras* exist.

[1] Google Translate app: https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html

Kapitel 1

Einleitung

1.1 Motivation

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

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

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

1.2 Problemstellung

Die grundlegende Problemstellung lässt sich mit folgender Frage beschreiben:

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

In dieser Frage stecken mehrere nichttriviale Punkte.
Ein Punkt stellt das Gebiet des maschinellen Lernens generell dar, sowie die praktische Umsetzung von Problemstellungen. Im Speziellen bieten neurona-

le Netzwerke neue Möglichkeiten, Probleme in der Informationstechnologie zu lösen, sowie Vorgänge in der Natur besser zu verstehen.

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

1.3 Zielsetzung

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

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

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

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

Kapitel 2

Begriffe im Maschinellen Lernen

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

2.1 Data Science

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

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

2.2 Machine Intelligence

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

2.3 Machine Learning

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

2.4 Neuronale Netzwerke

Die Theorie und die ersten Grundlagen wurden im Jahre 1943 von Warren McCulloch und Walter Pitts geschaffen, die ein Modell entwickelten, jedoch nicht die technischen Möglichkeiten hatten dieses umzusetzen. Dieses führte zur 'Threshold Logik', welche bestimmt, ab wann und wie stark ausgeprägt etwas weitergegeben wird [12]. Durch die Entwicklung des 'Backpropagation'-Algorithmus (siehe 2.8) ist es möglich, Netzwerke mit mehr als drei Ebenen zu trainieren. Neuronale Netzwerke bestehen aus Neuronen, die miteinander verbunden sind und gemeinsam ein Netzwerk ergeben [6].

2.4.1 Neuron

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

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

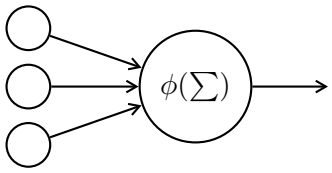

Abbildung 2.1: Neuron mit Eingang, Kernfunktion, Aktivierungsfunktion

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

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

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

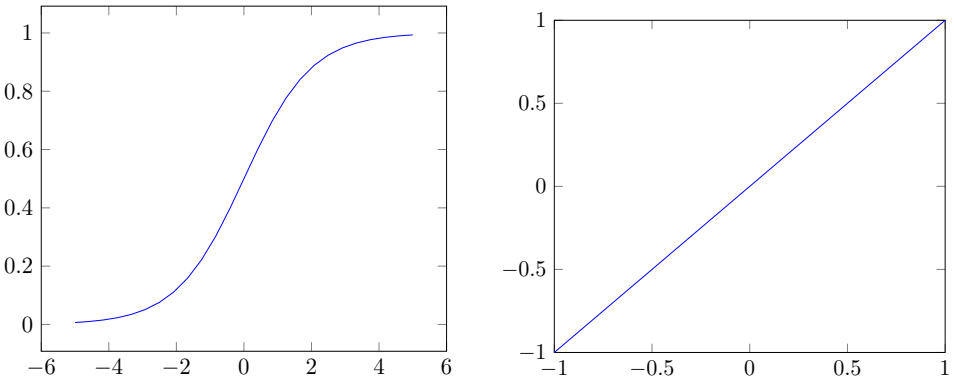

Abbildung 2.2: Basisaktivierungsfunktionen: (l) eine Sigmoid-Funktion, (r) eine lineare Funktion

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

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

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

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

2.5 Ebenen/Layer

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

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

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

In der Abbildung 2.3 ist ein einfachen FeedFordward Netzwerk abgebildet. Dieses beinhaltet alle Typen mit Input-, Outputlayer und voll vernetzte Layer mit Neuronen und Bias Neuron.

2.6 Informationen Merken und Wiedererkennung

Durch das Anpassen der Gewichtungen bei jedem Dateneingangsstrom mit Hilfe des Backpropagation-Algorithmus ist es möglich, Zustände zu spei-

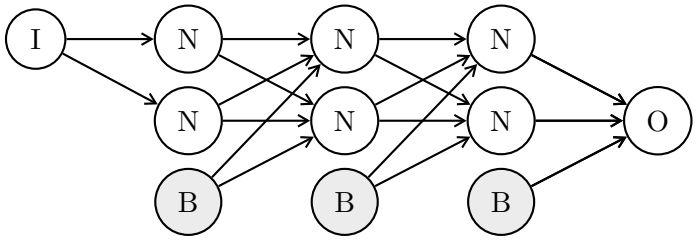

Abbildung 2.3: Einfaches Neuronales FeedFordward Netzwerk

chern und diese auch zu merken. Sollte ein ähnlicher Dateneingang stattfinden, wo zuvor schon einer vorhanden war, dann sollte dieser ähnlich behandelt werden. Dieser kann möglicherweise zu derselben Kategorie gehören, wie der zuvor schon bekannt gemachte und gelernte Dateneingang.

2.7 Konvergieren im Maschinellen Lernen

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

2.8 Backpropagation

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

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

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

2.9 Trainieren

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

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

2.10 Allgemeine Probleme

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

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

Overfitting bezeichnet ein Problem, welches nicht nur maschinelles Lernen betrifft, sondern auch Menschen und andere Lebewesen. Ein Student lernt zum Beispiel für eine Prüfung und ist im Besitz einer Klausur aus einem Vorjahr. Nach öfterem Durchspielen der Fragen und laufenden Selbsttests,

befindet er sich in der Lage diese Klausur mit einer sehr hohen Wahrscheinlichkeit zu bestehen. Dabei hat sich die Klausur in seinem Gehirn eingeprägt, aber nicht das Stoffgebiet zu welchem er eine Klausur schreiben muss. Das Problem wird als Overfitting bezeichnet und beschreibt, dass etwas gemerkt wurde, aber nicht gelernt worden ist und somit eine Abwandlung von Informationen nicht wiedererkannt wird.

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

2.11 Domänenklassen

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

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

2.11.1 Clustering

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

2.11.2 Regression

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

2.11.3 Klassifikation

Bei der Klassifikation werden die Ergebnisse einer Regression, welche meist als Gleitkommazahl vorliegen, auf eine bestimmte Anzahl an Klassen übertragen. Der Unterschied liegt im Ergebnis, welches produziert wird. Hier werden Daten dem Netzwerk übergeben und dieses muss vorhersagen, zu welcher Klasse sie gehören. Dies wird in einer überwachten Umgebung durchgeführt. Die Klassen für die Vorhersage sind vorab schon bekannt und können mit den Daten aus dem Outputlayer des Netzwerks verglichen werden und infolge Justierungen durchgeführt werden [6].

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

2.11.4 Predict

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

2.11.5 Robotics

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

2.11.6 Computer Vision

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

muss zum Beispiel ein Verkehrszeichen von einem Passanten unterschieden werden können.

2.12 Neuronale Netzwerktypen

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

2.12.1 FeedForward

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

2.12.2 Self-Organizing Map

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

2.12.3 Hopfield Neuronal Network

Ein Hopfield Neuronal Network (HNN) [4] ist ein einfaches Netzwerk, welches aus einem Layer besteht. In diesem Layer sind alle Neuronen mit jedem anderen Neuron verbunden. Dieses Muster wurde von Hopfield (1982) erfunden. Im Gegensatz zu anderen Netzwerken können Hopfield Netzwerke in einer Matrix abgebildet werden, in welcher die Gewichtung zu den einzelnen Neuronen abgebildet werden. Das Problem bei diesem Typ ist, dass jedes Neuron auf dem Status des anderen aufbaut. Dies stellt ein Problem für die Reihenfolge der Berechnung dar, was zu einem nicht stabilen Zustand führt. Durch das Hinzugeben einer Energiefunktion kann festgestellt werden, in welchem Zustand sich das Netzwerk befindet. Hiermit kann ein Haltepunkt definiert werden, ab welchem keine Trainingsiteration mehr durchlaufen werden soll.

2.12.4 Boltzmann Machine

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

2.12.5 Deep FeedForward

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

2.12.6 NEAT

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

2.12.7 Convolutional Neural Network

Convolutional Neural Network werden selbst nicht als komplettes eigenes Netzwerk verwendet, sondern in FeedForward Netzwerken. Im Speziellen, wenn es um Bilderkennung geht. Dabei werden zwei Ebenen nicht voll vernetzt sondern nur teilweise und somit Gewichtungen eingespart. Außerdem können die Gewichtungen geteilt werden, sodass immer in dieselbe Richtung verlaufende Verbindungen dieselbe Gewichtung aufweisen. Dies ermöglicht es, komplexe Strukturen zu speichern und trotzdem die Speicherauslastung niedrig zu halten und die Effektivität aufrecht zu halten.

2.12.8 Recurrent Network

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

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

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

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

Abbildung 2.4: Domänen zu Typen Matrix [6]

2.13 Domänen und Typen Matrix

Wie in Abbildung 2.4 erkennbar ist, existiert kein Netzwerkgrundtyp, der für alle Problemdomänen geeignet ist. Dies führt zu der Schlussfolgerung, dass je nach Aufgabe und Ziel ein entsprechendes Grundgerüst gewählt werden muss. Auf Basis dieses Grundgerüsts können uneingeschränkt weitere Eigenheiten aus anderen Netzwerken eingebaut werden. In der Praxis findet man selten ein Netzwerk von nur einem Typ für eine größere Problemstellung. Das Problem wird auf mehrere kleinere Problemstellungen herabgebrochen, welche hintereinander und teilweise parallel gelöst werden. Dabei übernimmt jedes Teilnetzwerk eine kleine Aufgabe des Gesamten und zwar die eine, für die es entwickelt wurde. Aktuelle Netzwerke wie das 'Inception v3' Netzwerk von Google Research benötigen zwei Wochen mit acht Grafikkarten zum Trainieren. Ab diesem Zeitpunkt ist es im Stande, akkurate Resultate zu liefern. Dieses Netzwerk ist sehr komplex und besteht nicht nur aus 3 Ebenen, was zur Schlussfolgerung führt, dass je tiefer das Netzwerk ist, desto aufwändiger ist es zu trainieren.

2.14 Optimierung

Optimierungen beeinflussen das Lernverhalten und das Speicherverhalten eines Netzwerkes.

Lernrate skaliert die Lerngeschwindigkeit, wie im Punkt Backpropagation beschrieben.

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

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

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

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

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

2.15 Trainingsgeschwindigkeitssteigerung

GPU - GPGPU ist die Bezeichnung für die Verwendung des Grafikprozessors über seine ursprüngliche Auslegung darüber hinaus. In der aktuellen Zeit ist es nicht mehr möglich, eine Geschwindigkeitssteigerung zu erreichen,

indem die Taktrate des Prozessors erhöht wird. Deshalb wird mehr parallelisiert, da die Recheneinheiten kleiner werden und so mehrere auf derselben Fläche Platz finden. Eine Grafikkarte besitzt die Eigenschaft, gleichförmige Operationen in einem Schritt auf sehr viele Objekte gleichzeitig auszuführen. So werden viele Pixel auf einmal eingefärbt oder eine Multiplikation großer Matrizen durchgeführt. Der Geschwindigkeitsvorteil kommt dabei durch den hohen Grad an Parallelität, da die Grafikkarte hauptsächlich für solche Operationen ausgelegt worden ist.

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

Kapitel 3

TensorFlow

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

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

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

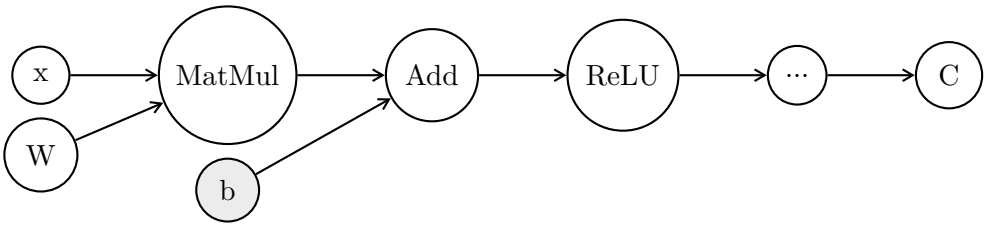

Abbildung 3.1: Der resultierende Teilgraph aus dem Codefragment 3.1 nach dem Beispiel in [1]

auf mobilen Plattformen eingesetzt werden.

3.0.1 Graphs/Dataflowgraph

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

Listing 3.1: TensorFlow Python-Codefragment zur Definition eines Teils des Graphen

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

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

3.0.2 Datenfluss Programmierung

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

3.0.3 Operation

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

3.0.4 Sessions

Die Session repräsentiert die Laufzeit eines Graphen. Dieser Session wird ein Graph übergeben, welcher zuvor initialisiert werden muss. Ohne die Initialisierung der Knoten und Verbindungen würde die weitere Ausführung nichts produzieren, da alle Werte 0 sind. Sie stellt eine weitere Funktion zur Verfügung, genannt *Run*. Der Run-Funktion wird eine Liste an Endknoten übergeben, welche berechnet werden sollen und die zu dem initialisierten Graphen gehören. Die Platzhalter-Tensoren werden mit Daten verknüpft und an den Graphen gereicht. In den meisten Fällen wird ein Graph einmal erstellt und mehrfach ausgeführt [1].

3.0.5 Tensor

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

3.0.6 Hyperparameter

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

3.1 Bibliotheksinhalt

3.1.1 Datentypen

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

3.1.2 Operationen

Konstanten und Zufallswerte

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

- *tf.zeros* erstellt einen Tensor mit den angegebenen Matrizendimensionen, bestehend aus 0 und von einem definierten Datentypen.

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

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

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

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

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

- *tf.random_crop* liefert einen zufälligen Teil eines Tensors mit derselben Anzahl an Dimensionen jedoch mit der spezifizierten Größe.

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

Variables

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

Transformationen

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

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

- *tf.shape* liefert eine genaue Aufschlüsselung des Tensors mit der Dimension und der Tiefe.
- *tf.size* repräsentiert die Anzahl an Elementen in einem Tensor. Diese Anzahl ergibt sich aus den konkreten Werten.
- *tf.rank* verhält sich ähnlich zu *tf.size* mit dem Unterschied, dass die Anzahl der Dimensionen gezählt werden.
- *tf.reshape* wird verwendet, um Tensoren in eine neue Struktur zu bringen. Dabei steht für das Einebnen der Struktur auf eine Ebene eine Kurzschreibweise zur Verfügung, mit -1 als Zieldefinition.
- *tf.squeeze* entfernt ganze Dimensionen aus dem gegebenen Tensor. Ohne Achsenangabe werden alle Dimensionen mit der Größe 1 entfernt oder es werden die spezifizierten Dimensionen herausgenommen.
- *tf.expand_dims* gliedert eine Dimension in einen Tensor wieder ein. Standardmäßig an der Indexstelle 0, außer es wurde spezifiziert.

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

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

Mathematik

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

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

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

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

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

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

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

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

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

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

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

Flusskontrolle

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

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

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

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

Images

Encodieren und Decodieren von Bilddateien wird in TensorFlow direkt unterstützt. Dabei können Bilder der Bildformate Gif, Jpeg und PNG

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

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

gelesen werden, sowie das Erstellen von Bildern in diese Formaten, ausgenommen Gif. In allen Fällen wird das Bild als Zeichenkette mit Pfad angegeben.

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

Beschneiden wird dann benötigt, wenn aus einem Bild ein Teil herausgenommen werden soll. Zum Herausnehmen stehen wiederum mehrere Operationen zur Verfügung, welche mit umschließenden Boxen arbeiten oder prozentual von der Bildmitte ausgehen.

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

Input und Readers

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

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

Konvertierungsoperationen ermöglichen es, Dateien, die mit TensorFlow Readers gelesen wurden weiter zu verarbeiten, so kann zum Beispiel eine CSV Datei decodiert verwendet werden.
Des Weiteren sind Protokoll Buffer sowie Queues implementiert, die zum Vorverarbeiten von Daten dienen. [6]

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

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

Neuronale Netzwerke

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

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

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

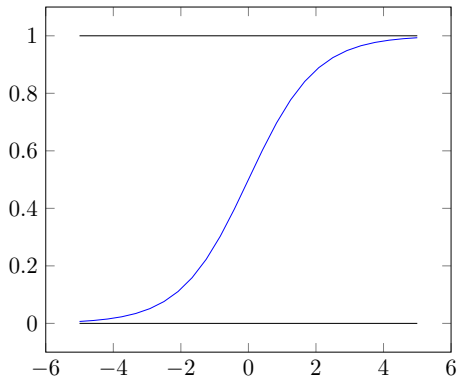

Abbildung 3.2: Sigmoide Aktivierungsfunktion mit den Grenzwerten an den Stellen 1 und 0

- *tf.relu* ersetzt mittlerweile immer mehr die sigmoiden Funktionen. Ein Grund dafür ist, dass die Berechnung eines Wertes in einer sigmoiden Funktion ressourcenintensiver ist und dass negative Werte meist nicht gewollt sind. Die rektifiziert lineare Funktion ist sehr viel einfacher, da Werte unter 0 als 0 weitergegeben werden und Werte darüber linear sind. Somit resultiert ein Eingangswert von -0.1 in einer 0 und ein Wert von 0.5 in 0.5. Wie in der Abbildung 3.3 ersichtlich ist, führt dies bei einem negativen Wert dazu, dass sich eine Multiplikation mit der Gewichtung in der nächsten Ebene ebenfalls in einer 0 repräsentiert und somit in der Addition ignoriert wird.
- *tf.tanh* genannt Hyperbolic Tangent, gehört ebenfalls zu den grundle-

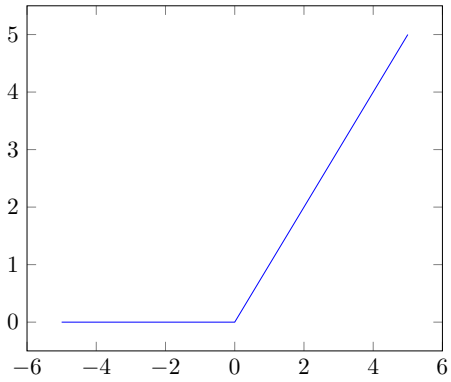

Abbildung 3.3: Rektifiziert lineare Aktivierungsfunktion

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

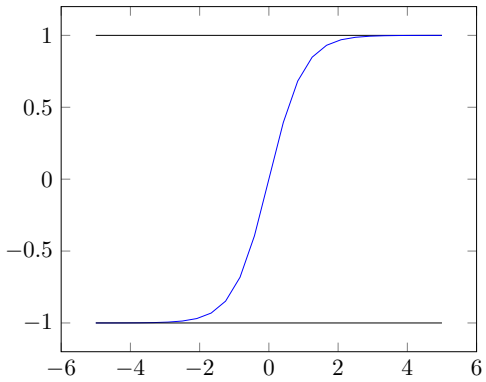

Abbildung 3.4: Hyperbolic Tangents Aktivierungsfunktion mit den Grenzwerten an den Stellen 1 und -1

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

Faltungs-Operationen werden bei Bilderkennungen unter anderem deshalb verwendet, da eine Operation mit unterschiedlichen Daten in einem Schritt auf mehrere Daten angewendet werden kann. Dabei wird ein Fenster über ein Bild geschoben und auf jedem Bild wird im selben Fenster die

Operation durchgeführt. Diese Operation generalisiert die darunterliegenden Daten so, als ob sie auf etwas reagiert hätten. Dies entspricht in etwa dem, als ob ein Auge auf etwas reagiert hätte.

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

Des Weiteren stehen noch andere spezialisierte Versionen implementiert zur Verfügung.

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

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

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

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

- *tf.nn.softmax* bildet alle Ergebnisse auf einen prozentualen Bereich ab. So ergeben alle möglichen Ausgänge in Summe 100%, was so viel bedeutet, dass ein Ergebnis eine gewisse Wahrscheinlichkeit besitzt.
- *tf.nn.softmax_cross_entropy_with_logits* bietet die Möglichkeit, den Fehlerwert für Diskret-Klassifikationen zu berechnen, wobei jedes Ergebnis genau einer Klasse zugeordnet werden muss. Diese Funktion kann zum Trainieren verwendet werden, benötigt die unskalierten Werte des Netzwerkes und liefert für jeden Eintrag im Batch einen Fehlerwert.

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

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

Running Graphs

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

Training

Optimizers stellen einen weiteren Kernteil des Systems dar. TensorFlow stellt eine Menge an implementierten Optimierungsalgorithmen zur Verfügung. Diese Operationen trainieren den Graphen mit der gewählten Technik des gewählten Algorithmus. Diese Implementierungen versuchen die gegebenen Kosten eines Graphen zu minimieren. Bei der Verwendung von *minimize* führt die Operation zwei Schritte in einem aus. In diesem wird der Gradient berechnet und dieser wird direkt auf die Variablen adaptiert. Diese Schritte können in einzelne zerlegt werden, wenn mit den berechneten Gradienten noch etwas zusätzlich durchgeführt werden soll. Die Berechnung wird dabei mit *opt.compute_gradients* ausgelöst, was eine Liste mit Paaren liefert. Diese Liste kann bearbeitet werden, aber auch zu Testzwecken mitprotokolliert werden. Die Gradienten werden im dritten Schritt mit *opt.apply_gradients*

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

auf die Variablen angewendet. Jeder Optimierungsalgorithmus verfügt über Eigenheiten und spezielle Verhalten, welche bei der Auswahl des Optimierers berücksichtigt werden sollten.

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

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

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

3.1.3 TensorBoard

TensorBoard stellt eine Erweiterung des TensorFlow-Systems dar, im Sinne einer Toolerweiterung. Jeder Graph kann in ein File serialisiert werden, welches als Event-File bezeichnet wird. Dies hat zur Folge, dass dieser auch wieder geladen werden kann. Bei dieser Serialisierung werden alle Informationen des Graphen inklusive der Gewichtungen in die definierte Datei

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

gespeichert. TensorBoard bietet nun die Möglichkeit, diesen Graphen zu laden und zu visualisieren. Zu den Graph-Informationen kann jeder Tensor mitgespeichert werden und als Diagramm visualisiert werden - inklusive einer zeitlichen Komponente. Dies ermöglicht es, den Verlauf des Trainings zu analysieren. Aus einem Graphen können mehrere dieser Event-Files erzeugt werden sowie Zustände festgehalten werden. Beim Laden eines Graphen in die TensorFlow- sowie TensorBoard-Umgebung kann spezifiziert werden, welcher Zeitpunkt geladen werden soll. Damit wird ermöglicht, viele Trainingsdurchläufe zu durchlaufen und bei einer Verschlechterung der Präzision zu einem früheren Zustand zurückzuspringen. Dieses Tool ermöglicht es in das Verhalten eines Graphen ein wenig Einsicht zu nehmen und so die sogenannte Black Box zu durchleuchten.

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

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

Listing 3.2: TensorFlow Codefragment zur Namescope-Verwendung in Graphen

Wie im Codefragment 3.2 beschrieben, werden die Tensoren und Operatoren *b, W, x, relu* in einem Block zusammengefasst. In diesem Beispiel gibt es keinen Tensor, welcher in den Block übergeben wird, da die Daten in der Ausführung von außerhalb des Systems in dieses gelangen. Die Operation *relu* und der daraus resultierende Tensor bilden den Ausgang des Blocks. Diese Technik der Namensbereiche ermöglicht es, den Graphen zu strukturieren, da nicht wie in *tf.zeros([100])* viele einzelne Knoten dargestellt werden, sondern abstrahiert werden, aber weiterhin einsehbar sind.

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

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

Die Informationen für die Lernrate sowie des Verlustes werden in diesem Fall am besten festgehalten. Dies wird ermöglicht, indem die Tensoren, welche die Lernrate sowie den Verlust beinhalten, in die Methode *tf.summary.scalar* jeweils eingebracht werden. Bei jedem Schreibzyklus in das Event-File werden diese Informationen übernommen und stehen daraufhin in TensorBoard zur Verfügung.

Distributions stellt eine weitere Funktionalität von TensorBoard dar. Diese Funktionalität war in den Versionen vor 'r1.0' noch unter dem Punkt Histogramm zu finden. Im Allgemeinen werden Tensoren mit der Metho-

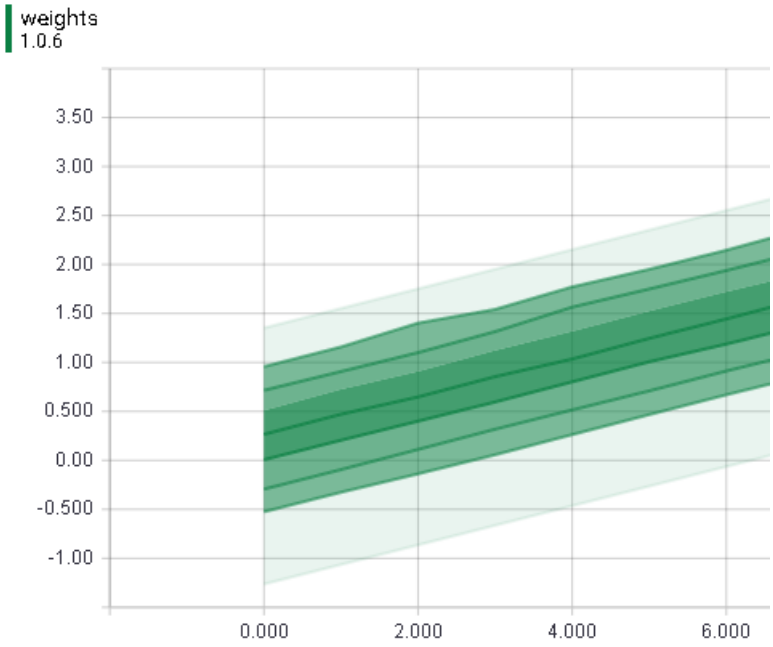

Abbildung 3.5: Verteilung der Werte in einem Tensor über die Zeit

de *tf.summary.histogram* wieder in das Event-File serialisiert. Das Ergebnis stellt eine Verteilung für die Werte dar, die im Tensor vorkommen. Das Diagramm repräsentiert auf der X-Achse die Anzahl der Schritte, die durchgeführt worden sind. Die Y-Achse gibt die konkreten Werte wieder, welche sich im Tensor über die Zeit befinden. Im Diagramm 3.5 wird ein Tensor mit 100 Werten dargestellt. Dieser Tensor wurde mit einer Normalverteilung initialisiert, wobei die Standardabweichung bei 0.5 liegt und der Median zu Beginn bei 0.2, mit einer geringen Abweichung. Die Linien in diesem Diagramm und ihre Einfärbungen präsentieren die Verteilung der Werte im beobachteten Tensor. Die Verteilung muss von unten nach oben gelesen werden, dabei ergibt die unterste Linie den Minimalwert, der vorgekommen ist. Die nächste Linie besagt, dass sich 7 % der Werte im Bereich zwischen dem geringsten und der zweiten Linie befinden, was inklusive des geringsten Wertes ist. Der nächste Bereich definiert, wie in einer gaußschen Normalverteilung, dass sich bis zum Ende dieses Bereiches 16 % darin befinden. Im Gesamten sind dies 9 Markierungen mit 8 Bereichen, welche zusammen alle Werte im Tensor widerspiegeln. Diese Folge an prozentualen Anteilen lautet wie folgt: $min, 7\,\%, 16\,\%, 31\,\%, 50\,\%, 69\,\%, 84\,\%, 93\,\%, max$. Im Diagramm 3.6 ist diese Verteilung besser ersichtlich, zusätzlich befinden sich im Listing 3.3 die Rohdaten der Diagramme.

```
1  -1.2645409,  -0.92252451, -0.68004417,  -0.63273954,  -0.57005012,
2  -0.5477351,  -0.54554129, -0.51146084,  -0.50482351,  -0.4872272,
```

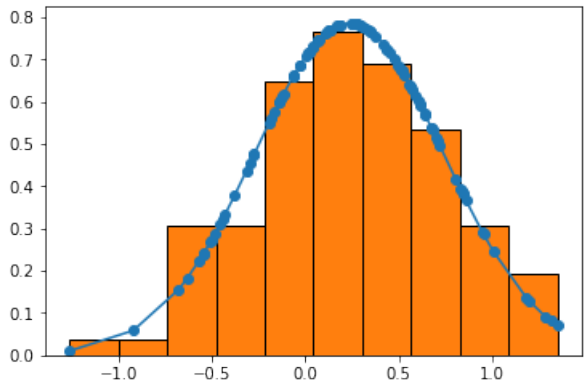

Abbildung 3.6: Verteilung der Werte in dem Tensor zu Diagramm 3.5

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

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

Histogram basiert auf denselben Daten wie die Verteilungsansicht. Im Grunde präsentiert diese Ansicht die Daten nur auf eine andere Art und Weise. Wie auch in der anderen Darstellung werden die Schritte direkt auf einer Achse dargestellt. Im Falle des Histogramms 3.7 ist dies die Achse, welche sich dreidimensional aus dem Hintergrund des Bildes in den Vordergrund zieht. Die horizontalen Achsen, welche zu jedem Schritt gezeichnet werden, projizieren ihre Wertebereiche immer auf die vorderste Achse. Auf dieser wird der Wertebereich abgebildet, in welchem sich die Werte im Tensor befinden. Die Erhebungen und die sich darunter bildenden Flächen geben die Verteilung der Werte wieder. So wird beim Überfahren eines Schrittes mit

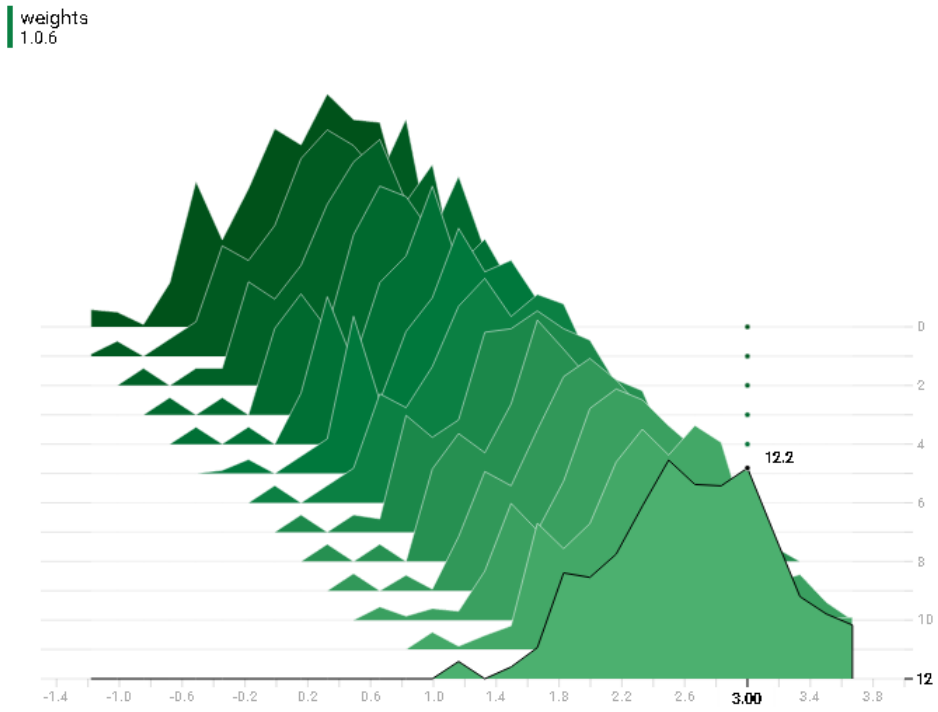

Abbildung 3.7: Verteilung der Werte in dem Tensor in einem Histogramm

der Maus dieser aktiviert, wie im Histogramm 3.7 ersichtlich ist. Im Falle dieses Diagramms und dieser Stelle bedeutet das, dass sich in der Nähe des Wertes 3.00 ungefähr 12.2 Einträge im Tensor befinden. Anders ausgedrückt haben 12.2 konkrete Werte im Tensor den Wert 3.00 oder einen naheliegenden. Durch die Verteilung der Werte entsteht nun der Fall, dass ein Teil der Einträge auch zu einem Wertebereich vorher oder nachher gehören kann. Dieses Diagramm stellt die Verteilung in Wertebereiche dar, wobei alle vertikalen Werte in einem Schritt die Anzahl der Werte im Tensor ergeben müssen. Im Falle dieses Beispiels ergeben sie aufsummiert einen Wert von 100.044, was gerundet die 100 Einträge im Tensor bestätigt. Die Aufteilung der Werte in Wertebereiche mit Teilzuweisungen erklärt auch den Schrittverlauf im Histogramm 3.7. Hier ist ersichtlich, dass die Verteilungen und Zugehörigkeiten immer ein wenig vom vorhergehenden abweichen, obwohl in diesem Beispiel in jedem Schritt konstant 0.2 zu jedem Wert hinzu addiert worden ist. Dies lässt sich bei einer geringen Anzahl an Werten, wie hier mit 100, leichter beobachten, als bei einer sehr viel höheren.
TensorBoard bietet noch weitere Möglichkeiten, wie Bilder- oder Soundinhalte mit in das Event-File zu geben, um diese dann in Tensorboard weiter zu verwenden. So können diese Inhalte durch den Graphen gesendet werden und dabei beobachtet werden. Die letzte Erweiterung in Tensorboard ist

der Punkt mit 'Embedding' [10], wo gelernte Informationen, so wie sie vom Graphen gruppiert worden sind, dargestellt werden können.
Dieses Kapitel repräsentiert die grundsätzliche Funktionalität des TensorFlow-Systems. Dabei wurde auf die Grundlagen und die am meisten benötigten Methoden eingegangen. Ihr Verständnis stellt die Grundlage für das nächste Kapitel dar. In diesem wird ein praktisches Beispiel mit TensorFlow erläutert.

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

Kapitel 4

Facial Keypoints Detection

4.1 Ausgangssituation

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

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

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

4.2 Vorbereitung

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

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

Listing 4.1: 15 Schlüsselpunkte im Gesicht eines Menschen

Jeder diese Schlüsselpunkte besteht aus einer X und einer Y Koordinate. Diese Datei besitzt zusätzlich in der 31. Spalte das Bild mit dem dazugehörigen Gesicht. Das Bild ist encodiert abgelegt und besteht aus $96 * 96$ Werten. In diesem Sinne stehen nur Bilder in Graustufen zur Verfügung mit einer Auflösung von $96 * 96$ Pixel und einer Farbtiefe von 2^8, wie in der Abbildung 4.1 zu erkennen ist. Für diese Aufgabe werden im Grunde auch nur die Konturen benötigt, was somit den einen Farbkanal erklärt, aber auch die Aufgabe schwieriger macht, denn mehrere Farbkanäle könnten mit Filtern bearbeitet werden und so noch weitere Konturen hervorheben.

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

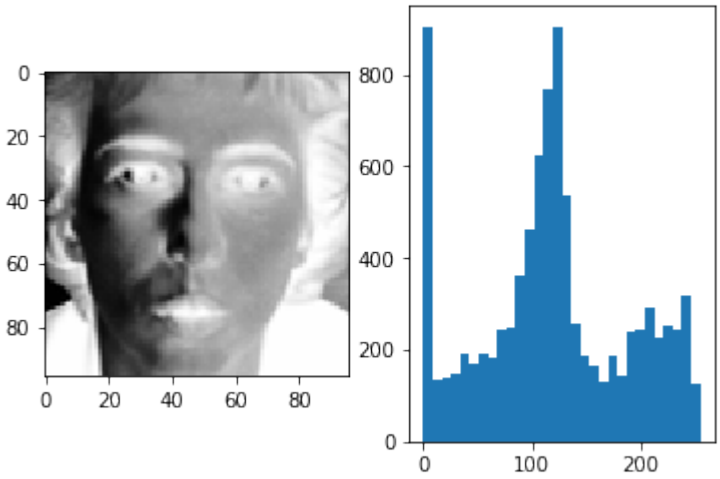

Abbildung 4.1: Ein Gesicht aus dem Datenbestand mit der Verteilung der Graustufenwerte

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

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

4.2.1 Daten vorbereiten und normalisieren

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

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

```
1 import pandas as pd
2 import numpy as np
```

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

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

Listing 4.3: Daten einlesen und einschränken

4.2.2 Evaluations- und Errorfunktion

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

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

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

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

Listing 4.4: Verlustberechnung, konkreter Opitimierungsalgorithmus, Genauigkeitsberechnung

4.3 Neuronale Ebenen vorbereiten

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

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

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

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

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

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

Listing 4.5: Definition der Convolutional- und Vollvernetzten-Ebenen

4.4 Neuronale Ebenen verknüpfen

Diese Definitionen der Ebenen aus dem Listing 4.5 müssen im nächsten Schritt zu einem Netzwerk zusammengesetzt werden. Ein neuronales Netzwerk, welches relativ leichtgewichtig ist und diese Problematik relativ brauchbar lösen kann, ist im Grunde ein sehr vereinfachtes 'VGG16' Netzwerk[4]. Für diesen Fall besteht dieses aus 3 Convolutional-Ebenen und 3 Vollvernetzten-Ebenen, wie im Listing 4.6 zu sehen ist. Das originale VGG16 Netzwerk besitzt im Gegensatz zum aktuell verwendeten, 5 Convolutional-Ebenen mit je 2 integrierten Convolutional-Ebenen mit denselben Dimensionen. Die

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

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

```
1 def model(data):
2     net = conv_layer(data, 1, 32, "conv1")
3     net = conv_layer(net, 32, 64, "conv2")
4     net = conv_layer(net, 64, 128, "conv3")
5
6     # Transformieren in eine flache Struktur
7     dims = net.get_shape()[1:]
8     k = dims.num_elements()
9     with tf.name_scope('flatten'):
10        net = tf.reshape(net, [-1, k])
11
12    net = fc_layer(net, 256, "fc1", tf.nn.relu)
13    net = fc_layer(net, 256, "fc2", tf.nn.relu)
14    net = fc_layer(net, 4, "fc3")
15
16    return net
17
18 # Definition der Platzhalter für die Datenübergabe
19 train_data_node = tf.placeholder(tf.float32,
20             shape=(BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, 1))
21 train_labels_node = tf.placeholder(tf.float32,
22             shape=(BATCH_SIZE, 4))
23
24 # erstellen eines Graphen mit dem definierten Model
25 graph = model(train_data_node)
```

Listing 4.6: Modelldefinition des Graphen

4.5 Trainieren

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

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

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

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

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

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

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

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

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

Listing 4.7: Initialisierung des Graphen und Durchführen einer Epoche

4.6 Validierungsergebnisse

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

berücksichtigt.

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

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

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

4.7 Ergebnis

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

Abbildung 4.2: Ist-Visualisierung des Ergebnisses

ist und möglicherweise ein Rauschen in den Daten nicht ignoriert wurde, sondern darauf reagiert wird. In der Abbildung 4.3 lässt sich erkennen, dass die Sprünge nicht regelmäßig sind, das auf ein Mischen der Datensätze am Ende jeder Epoche zurückgeführt werden kann.

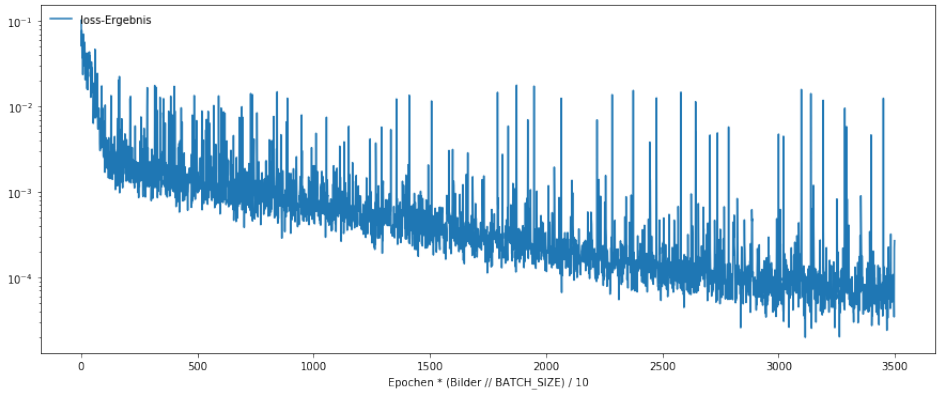

Abbildung 4.3: Trainings-Verlustergebnisse in 100 Epochen mit 350 Tests pro Epoche

Durch einen Test aller Datensätze und Herausfinden des besten und des schlechtesten Ergebnisses, lässt sich feststellen warum das Netzwerk diverse Probleme mit Bildern besitzt. Im Konkreten beinhalten die Daten einige

wenige dubiose Bilder. Ein Beispiel dazu befindet sich in der Abbildung 4.4. Hier ist erkennbar, dass in den zwei oberen Bildern mehrere Bilder auf einem WhiteBoard vorhanden sind. Zum einen ist hier das Problem, dass sich hier mehr als ein Gesicht im Bild befindet und das Netzwerk nicht für so einen Fall ausgelegt und trainiert worden ist. Zum anderen lässt sich erkennen, dass das Netzwerk hier ein anderes Bild bevorzugt, auf welchem ebenfalls Menschen abgebildet sind. Dies führt zu der Annahme, dass das Netzwerk sich hier nicht die Positionen durch das Trainieren gemerkt hat und dies als Rauschen wahrnimmt, sowie sich an den sonst zentralisierten Gesichtern orientiert und trotzdem versucht hier zwei Augen zu finden, welche mehr oder weniger nebeneinander liegen sollten. In den zwei unteren Bildern kann festgestellt werden, dass das Netzwerk akkurate Ergebnisse liefert.

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

Im besten Fall erzielt das Netzwerk eine Abweichung von $5.1624589e-05$, was in der 150. Epoche als sehr positiv gewertet werden kann. Das beste Ergebnis der 150. Epoche ist in der Abbildung 4.5 ersichtlich. An diesem Ergebnis lässt sich erkennen, dass das Netzwerk mehr oder weniger keine Probleme mit diversen Hautfarben und Kontrasten hat. Weiters werden Brillen ignoriert und auch leichte Neigungen nach links und rechts stellen keine Schwierigkeit dar.

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

Im Langzeittest bis zu der 800. Epoche lässt sich feststellen, dass das Netzwerk noch genauer wird, wobei hier der zeitliche Aufwand für das Trainieren sehr viel Zeit beansprucht und zusätzlich das Problem des Overfitting akut werden könnte. In Tabelle 4.1 sind zwei Systeme angeführt, auf welchen das Netzwerk trainiert wurde. Der Berechnungsvorteil der GPU ist klar ersichtlich.

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

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

Zur genaueren Feststellung der Funktionalität des Netzwerkes befindet sich

in der Abbildung 4.6 ein Test mit einem noch komplett unbekannten Gesicht. Dieses wurde für diesen Test manuell aufgenommen und konvertiert sowie im Weiteren in das Netzwerk als Testdatensatz eingefügt. Wie zu erkennen ist, befindet sich das Netzwerk in der 150. Epoche bereits in einem Zustand, welcher als verwendbar einstuft werden kann.

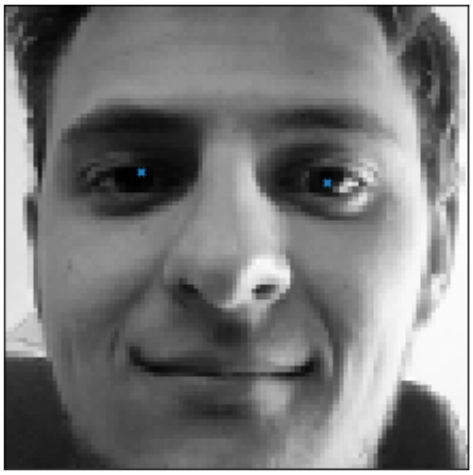

Abbildung 4.6: Testaufnahme mit Ergebnis des Netzwerkes

4.8 Graphenvisualisierung

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

4.9 Verbesserungen

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

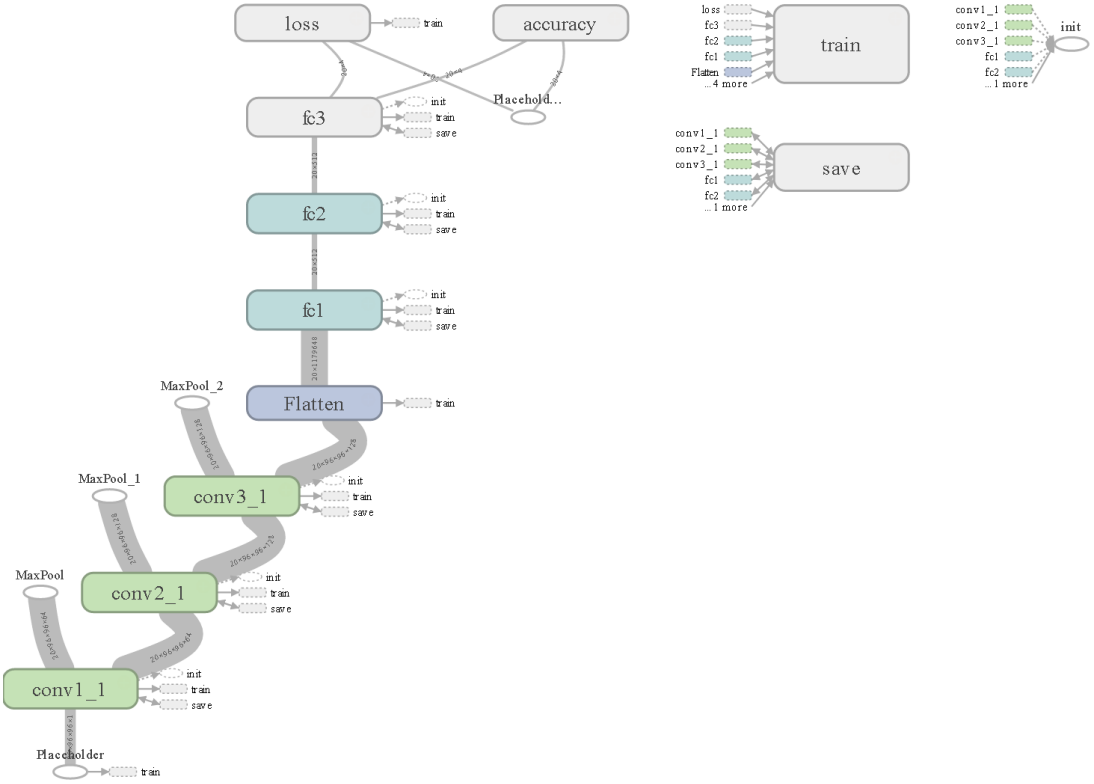

Abbildung 4.7: Aktueller Graphen des Beispiels

Möglichkeiten führen aber dazu, dass mehr Ressourcen benötigt werden.

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

Kapitel 5

Zusammenfassung & Ausblick

5.1 Zusammenfassung

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

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

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

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

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

5.2 Ausblick

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

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

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

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

Quellenverzeichnis

Literatur

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

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

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

[4] Howard B Demuth u. a. *Neural network design*. Martin Hagan, 2014 (siehe S. 12).

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

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

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

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

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

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

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

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

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

Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Routing in der Logistik

Praktische
Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

David Baumgartner

Begutachtet von Prof.(FH) Priv.-Doz. DI Dr. Stephan Dreiseitl

Hagenberg, Juni 2017

Inhaltsverzeichnis

Kurzfassung

Routing ist allgegenwärtig in Form von Navigationssystemen wie zum Beispiel Google Maps. Diese Navigationslösungen besitzen nur die Fähigkeit einfache *point-to-point* Routen zu finden. Dabei können Zwischenstopps vorhanden sein, aber mit einer fixen Reihenfolge. Routing und im speziellen Transportlogistik beinhalten mehr als nur Navigationsoptimierung. So müssen im Falle von UPS Paketauslieferungen optimiert werden, um Einsparungen zu ermöglichen.
Im Rahmen dieser Arbeit wurden die Problemstellungen wie *Traveling Salesman Problem* und weitere erarbeitet und erklärt. Des Weiteren wurde auf mögliche Lösungsansätze für solche Probleme eingegangen. Die Abläufe solcher Lösungsmöglichkeiten stellen einen weiteren Inhalt dar. Im Speziellen wird genauer auf den *Savings-Algorithmus* eingegangen und eine Beispielimplementierung mit Zeitfenster durchgeführt.

Abstract

Routing is a daily demand and is used within our navigation-systems like *Google Maps*. The navigation solution contains the functionality to find *point-to-point* routes with fixed sequence stops. The field of transport systems/logistics requires a more sophisticated stop optimiziation. As an example the package delivery service UPS uses stop und routing optimiziation to save millions of costs.
This thesis contains the basics about the *Traveling Salesman Problem* and its offsprings. Additionally it includes the basics to the solving approaches for these problems and how they work. There is an implementation of the *Savings-Algorithm* with the time window extension.

Kapitel 1

Einleitung

1.1 Motivation

Der Gütertransport existiert seit Waren transportiert werden. Nicht nur der Personenverkehr nimmt immer mehr zu, sondern auch der Gütertransport. Es werden immer mehr Güter weltweit erzeugt, dafür müssen zum Abnehmer meist weite Strecken zurückgelegt werden. Dies führt dazu, dass Straßen ausgebaut und Hauptverkehrsrouten adaptiert werden müssen. Vom Jahr 2014 auf 2015 nahm zum Beispiel das Güteraufkommen in Deutschland um $1,9\,\%$ zu.[1] Damit die Kosten konstanter bleiben, müssen Transportrouten optimierter erledigt werden. Dadurch werden Kosteneinsparungen erzielt. So wird auch der Umwelt etwas Gutes getan.

1.2 Problemstellung

Ein Problem in der Logistik stellt die Optimierung von Routen dar. Optimierungen von Routen bieten eine Möglichkeit, um effektiver Kunden zu erreichen. Dabei kann nicht nur Zeit gespart werden, sondern auch Fahrzeuge besser ausgenützt werden. Zum Beispiel erspart sich UPS durch intelligente Optimierung Millionen an Kilometern. Der Paketdienstleister UPS entwickelte sich unter anderem auch zu einem Technikunternehmen mit tausenden Servern. Diese lösen und analysieren ununterbrochen Routen, damit am Morgen sofort optimierte Touren zur Verfügung stehen. Das dafür entwickelte System nennt sich *ORION* und analysiert in Realzeit anhand der Verkehrslage. Dafür wurde seit 2003 an diesem System gearbeitet und erst im Jahr 2012 mit dem Betatesting begonnen.[2] Solche Probleme betreffen nicht nur Auslieferungsdienste wie UPS, sondern auch lokale Produzenten mit eigener Auslieferung. Diese sind meistens zusätzlich auf Zeitfenster bei

[1] Bundesverband Güterkraftverkehr; http://bgl-ev.de

[2] UPS ORION; https://www.pressroom.ups.com

den Kunden angewiesen, um einen Mehrwert für die Kunden zu bieten. Mit diesen Zeitfenstern wird die Problemstellung um einiges komplexer und schwerer lösbar.

1.3 Zielsetzung

Diese Arbeit soll die Grundlagen für Optimierungen in der Logistik näher bringen und einen Einstieg darstellen. Hierbei wird auch auf mögliche Kostenberechnungen und Kostenmatrizen eingegangen und Probleme aufgezeigt. Am Ende wird das Gebiet der Zeitfenster anhand eines Beispiels aufgearbeitet und eine mögliche Lösung präsentiert.

Kapitel 2

IT|PRO

2.1 Allgemein

Das Unternehmen IT|PRO wurde 1999 mit der Vision *Lösungen für Menschen* gegründet. Im Unternehmen werden aktuelle Technologien eingesetzt, um bestmögliche und wirtschaftliche Lösungen zu kreieren. Diese Technologien reichen vom *.NET Framework* bis hin zum *Angular 2/4 Framework* und zusätzlicher Hardwareentwicklung. Das Unternehmen bietet dabei Gesamtlösungen sowie spezielle Kundenlösungen und Standardprodukte. Es wird ein Komplettservice angeboten mit Beratung, Analyse der bestehenden Geschäftsprozesse mit Projektentwicklung und Inbetriebnahme bis hin zum Support. Im Hintergrund stehen engagierte Entwickler und Partnerbetriebe, damit eine Lösung aus einer Hand geliefert werden kann. Aktuell besteht das Team aus den beiden Geschäftsführern und rund 20 Mitarbeitern/Innen.

2.2 Arbeitsgebiet

Ein Teil der eigenen Softwarelandschaft im Unternehmen IT|PRO widmet sich der Routenplanung und Optimierung. Diese Applikation wurde in einem Softwareprojekt mit der FH-Hagenberg auf eine neue Version gebracht und dabei weitgehend neu implementiert. Im Rahmen des Praktikums wurde eine weitere Hauptumstellung durchgeführt. So wurde das Karten-Framework *Bing Maps* durch das freie Framework *Leaflet* ersetzt. Des Weiteren wurde das letzte fehlende Feature, ein Disponierungstool, aus der vorhergehenden Version neu implementiert. Eine weitere Aufgabe im Praktikum war die Überarbeitung und teilweise Neuimplementierung des Routing-Algorithmus. Dieser beruht auf einem Savings-Algorithmus mit der Erweiterung um Zeitfenster. In Kapitel 3 werden die Grundlagen im Bereich der Logistik erklärt und erläutert.

Kapitel 3

Logistik

3.1 Einsatzgebiet & Problemstellung

Logistik gehört wie die Datenanalyse im Bereich von BigData zu den Problemgebieten, welche sich nicht so einfach lösen lassen. Grundsätzlich sind sie exakt mit genügend Zeit lösbar, aber nicht in der benötigten oder vorhandenen Zeit. Dieses Problem wirkt sich im 21. Jahrhundert stark aus, da immer mehr Güter transponiert werden und diese über die gesamte Welt. Am Beispiel des Paketdienstes UPS lässt sich dies erkennen. Im Jahr 2006 war das tägliche Zustellvolumen von Paketen und Dokumenten bei 14.1 Millionen, im Jahr 2014 bereits bei 18.0 Millionen[1]. Dies entspricht einer Steigerung von $\sim 27\,\%$, bei einer gleichzeitigen Erweiterung der Fahrzeugflotte um $\sim 13\,\%$. Dies betrifft nicht nur Logistikunternehmen wie UPS, sondern auch zum Beispiel Brauereien, welche die Auslieferung ihrer Produkte nicht komplett ausgelagert haben.
Logistik beinhaltet mehr als nur Auslieferungen:

- Planung: Ein Disponent erstellt Listen an Aufträgen, die erledigt werden müssen;
- Organisation: Die Auftragslisten müssen einem Fahrzeug zugeteilt werden, wobei überprüft werden muss, ob die maximale Kapazität des Fahrzeugs nicht überschritten wird;
- Steuerung: Die geplante Route kann vom Disponenten manuell oder durch ein Informationssystem automatisiert optimiert werden;
- Abwicklung: Erfolgt durch den Fahrer mit dem zugeteilten Fahrzeug;
- Kontrolle: Durch Analyse der Touren und Fahrten kann ein Informationsrückfluss erfolgen, um den gesamten Prozess zu verbessern;

Ein weiteres Problem betrifft die Anzahl der Stopps. Wie in Tabelle 3.1 ersichtlich, steigt die Anzahl der möglichen Routen mit der Anzahl der Stopps.

[1] UPS - Weltweit, https://www.ups.com/content/at/de/about/facts/worldwide.html

Anzahl der Haltepunkte	Anzahl der möglichen Routen	Berechnungszeit in Sekunden bei 30 Routen/Sekunde
1	1	0.030
2	2	0.067
3	6	0.200
4	24	0.800
5	120	4.000
6	720	24.000
7	5.040	168.000
8	40.320	1344.000
9	362.880	12096.000
10	3.628.800	120960.000
11	39.916.800	1330560.000
12	479.001.600	15966720.000
13	6.227.020.800	207567360.000

Tabelle 3.1: Aufschlüsselung der Anzahl an möglichen Routen, gegeben durch die Berechnung der Fakultät der Anzahl an Knoten

Eine Route besteht hierbei aus einem Startdepot am Anfang und einem Enddepot am Ende. Zwischen diesen Punkten befinden sich mindestens ein Haltepunkt bis n viele. Diese Anzahl lässt sich durch die Zahl der Stopps und dessen Fakultät berechnen. Daraus lässt sich ableiten, dass bis zu einem Grad alle Möglichkeiten testbar sind, abhängig von der Rechenleistung. Angenommen ein Rechner berechnet 30 Möglichkeiten pro Sekunde, so würde man bei 8 Stopps eine Zeit von 1344 Sekunden bzw. $22,4$ Minuten benötigen.

3.1.1 Graph

Graphen im Sinne der Logistik sind anders definiert als im Bereich der neuronalen Netzwerke. In der Graphentheorie ist ein Graph ein System an Knoten und Kanten. Dabei kann dieser als Paar $G := (V, E)$ interpretiert werden, welches aus zwei Mengen besteht. Eine Menge mit den Knoten V und die zweite mit den Kanten E zwischen den Knoten. Im Sinne der Graphentheorie kann entlang der Kanten durch den Graph navigiert werden.
Eine Spezialisierung bilden gerichtete Graphen, bei welchen die Kanten zusätzlich eine Richtung beinhalten. Des Weiteren existieren noch gewichtete Graphen mit Gewichtungen an den Kanten. Sie beschreiben, wie groß die Kosten sind, wenn die Kante zum Navigieren verwendet wird. Ein Beispiel dazu befindet sich in der Abbildung 3.1, mit 4 Knoten und einem voll vernetzten System zwischen diesen. Die Spezialisierungen der einzelnen Graphen-Typen können kombiniert werden, sowie um weitere Bedürfnisse

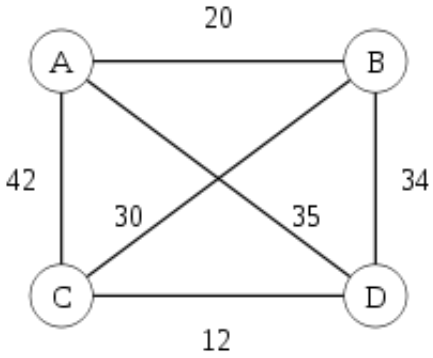

Abbildung 3.1: Beispiel für einen Graphen mit gewichteten Kanten und 4 Knoten als Kunden; Die Kosten sind dabei als ganze Zahlen an den Kanten angegeben; Bild: www.wikipedia.org

abgeändert werden [10].
Um durch einen Graphen zu navigieren existieren Algorithmen, welche zum Beispiel die kürzeste Route von einem Punkt zu allen anderen im Graphen finden. Hierzu zählt zum Beispiel der Dijkstra-Algorithmus, der als Standardalgorithmus im Gebiet der kürzesten Routenfindung eingesetzt wird.

3.1.2 Traveling Salesman Problem (TSP)

Das TSP oder auch unter dem Begriff *Vehicel Routing Problem (VRP)* bekannt, stellt ein Grundproblem der Logistik dar. Bei dieser Problemstellung sollen geografische Positionen einmal angefahren werden und dies mit dem geringsten Aufwand. Dies lässt sich in der Abbildung 3.2 erkennen. Dabei werden die größten Städte Deutschlands angefahren, der Start- und Zielpunkt sind dabei identisch. Dieses Problem kann mit einer Laufzeitkomplexität von $O((n-1)!/2)$ exakt gelöst werden, wobei mit zunehmender Größe des Faktors n, die Berechnungszeit wesentlich größer wird. Aus diesem Grund wurden heuristische Algorithmen entwickelt, welche eine mögliche Lösung finden, aber sehr wahrscheinlich nicht die effizienteste Route. Heuristische Lösungen werden eingesetzt, da sie zeit- und ressourcensparender sind und dabei verwendbare Lösungen liefern [6, 10].
Weiters werden einige spezialisierte Formen des VRP kurz erklärt und beschrieben.

Capacity Vehicle Routing Problem (CVRP): In dieser Erweiterung von VRP spielt besonders die Kapazität der Fahrzeuge eine Rolle, denn diese sollte so effizient wie möglich ausgenützt werden. Hierbei starten alle Fahrzeuge vom selben Depot, alle Kunden sind vorab bekannt. In diesem

Abbildung 3.2: Beispiel für eine Route durch Deutschland; Bild: www.wikipedia.org

Fall besitzen alle Fahrzeuge dieselbe Kapazität, welche in der Route nicht überschritten werden darf [6].
Lösungsalgorithmen für CVRP:

- Savings-Algorithmus: wird in Sektion 3.2.2 erklärt
- Der Sweep-Algorithmus verwendet ein *Cluster-first* Vorgehen. Hierbei werden Stopps einem Fahrzeug solange hinzugefügt, bis dessen Kapazität oder die maximale Routenlänge erreicht wird. Die Zuteilung zu einem Fahrzeug wird durch Polarkoordinaten festgestellt und dessen Winkel zum aktuell ausgewählten Punkt. Im Anschluss wird eine TSP Optimierung auf den erstellten Cluster an Haltepunkten ausgeführt und so eine optimierte Route generiert.
- Ein *Route-first* Vorgehen, wie von Beasley 1983 beschrieben, beginnt mit einer optimierten Route ohne Berücksichtigung der Nebenbedingungen. Im Anschluss wird diese Route auf Fahrzeuge aufgeteilt, sodass die Nebenbedingungen eingehalten werden.

Periodic Vehicle Routing Problem (PVRP): PVRP stellt wie CVRP eine Erweiterung des Basisproblems dar. In diesem Fall sollen Kunden nicht einmal, sondern mehrmals angefahren werden. Dieses Problem tritt zum Beispiel bei einem Hausbau mit mehreren Betonlieferungen auf, über eine längere Zeit abhängig von der Nachfrage [6].

Vehicle Routing Problem with Backhauls (VRPB): Das VRPB erweitert das CVRP um die Möglichkeit, Kunden nicht nur zu beliefern, sondern auch etwas entgegenzunehmen. Dabei entstehen zwei Teilmengen. Kunden die beliefert werden und Kunden von welchen abgeholt werden soll. Im Grunde erweitert sich das Basismodell um eine Vorrangregel, bei welcher zu beliefernde Kunden bevorzugt werden. Ansonsten könnte der Fall eintreten, dass ein Fahrzeug keine Zusatzladung mehr aufnehmen kann [10].

Vehicle Routing Problem with Time Windows (VRPTW): Das VRPTW erweitert ebenfalls das CVRP, sodass Kunden nicht zu jeder Zeit angefahren werden können. Dieses Problem ist aus der Praxis heraus gewachsen, denn manche Kunden und Lager sind nur zu bestimmten Zeiten anzutreffen. Zusätzlich könnten Straßensperren berücksichtigt werden, wie zum Beispiel eine Innenstadt mit Zustellungserlaubnis am Vormittag. Das Basismodell muss hierbei um Zeitintervalle erweitert werden, weiters müssen die Abfahrtszeiten vom Depot, Fahrzeiten für alle Teilstrecken und Servicezeit bei jedem Kunden vorhanden sein. Im Gesamten muss festgelegt werden wie optimiert werden soll, denn es besteht die Möglichkeit, dass Fahrzeuge warten müssen, bis der Kunde anzutreffen ist. Im Falle einer Wartezeit könnte zuerst ein anderer Kunde angefahren werden. Dies bedeutet, dass wahrscheinlich eine längere Strecke zurückgelegt werden muss [10].
Lösungsalgorithmen für VRPTW:

- Savings-Algorithmus
- Route-first, Cluster-second (Solomen 1986)
- Sequentielle Einfügeheuristik (Solomon 1987)
- Zeitorientierte Nearest-Neighbor Heuristik
- Tabu-Search
- Evolutionäre Algorithmen

Pick-up-and-delivery Problem (PDP): Das PDP stellt eine zusätzliche Spezialisierung des VRP dar. In diesem Fall müssen Abholungen und Lieferungen in einer Route erledigt werden. Dabei ist die Kapazität der Fahrzeuge sehr zu berücksichtigen, da diese erst wieder zusätzliche Ladungen aufnehmen können, wenn genügend Kapazität dafür vorhanden ist. Außerdem muss berücksichtigt werden, dass eine Ladungsaufnahme in dieser

Route später auch zugestellt wird. Diese Ladungen dürfen nicht als Auslieferung für eine weitere Route übernommen werden.

Diese Beschreibungen stellen nur eine Teilmenge der Problemdomänen dar, weshalb für genauere Informationen Fachlektüre hinzugezogen werden sollte. Des Weiteren existieren Meta-Heuristische Algorithmen, welche eine geringe Laufzeit besitzen, dafür kein exaktes Ergebnis liefern, sondern nur eine Annäherung.

3.2 Lösungsverfahren

3.2.1 Exakte Verfahren

Exakte Verfahren liefern die beste Lösung, aber nicht ohne Nachteile. Einer dieser Nachteile ist, dass die benötigte Zeit mit jedem weiteren Knoten im System stark ansteigt, da die zu überprüfenden Möglichkeiten steigen.

Branch-and-Cut: *Branch-and-Cut* vereint zwei Verfahren, welche ebenfalls im Gebiet der Logistik eingesetzt werden: Das Schnittebenenverfahren und das *Branch-and-Bound* Verfahren. Ziel dieser ganzzahligen linearen Optimierung stellt eine lineare Zielfunktion dar, welche gesucht wird. Diese Funktion wird mit Hilfe von linearen Ungleichungen gesucht, welche eine Menge an Punkten umschließen. Durch das Schnittebenenverfahren werden weitere Ungleichungen in das System eingefügt, welche von allen zulässigen Punkten erfüllt werden. Durch ein weiteres Lösen des Systems muss eine neue Lösung entstehen, welche näher am gesuchten Optimum liegt. Das *Branch-and-Bound* wird im Anschluss ausgeführt, wenn keine Schnittebenen mehr gefunden werden. Die genauere Beschreibung dieses Algorithmus befindet sich in der Sektion 3.2.2, da dieses Verfahren in der Fachliteratur den Übergruppen *exakte Verfahren* und *heuristische Verfahren* zugeordnet wird.

3.2.2 Heuristik

Heuristik bedeutet, mit wenig Wissen sowie mit begrenzter Zeit eine wahrscheinliche Lösung zu liefern. Heuristiken werden oft im Gebiet der Optimierungen eingesetzt. Zum exakten Lösen eines Logistik-Problems wird sehr viel Zeit und Speicher benötigt. Deshalb werden hier häufig Heuristiken eingesetzt. Ein Vorteil der Heuristik ist es, dass in kurzer Zeit gute Lösungen für schwierige Probleme gefunden können. Die Nachteile dieser Methoden sind aber nicht zu vernachlässigen, sie geben nämlich:

- keine Garantie für die optimale Lösung;
- keine Garantie für eine Lösung und für die Güte einer Lösung;

Branch-and-Bound: Im Grunde wird bei diesem Algorithmus ein Baum im Sinne der Informationstechnologie aufgebaut, in welchem anhand der Möglichkeiten gesucht und verglichen wird. Dabei werden Schwellenwerte für die Kosten festgelegt, welche bei einer Route nicht über- und unterschritten werden dürfen.

- *Branchen* (Verzweigung) ist der erste Schritt in diesem Algorithmus. Hierbei wird das Problem in mehrere Teilprobleme aufgeteilt und in einer Baumstruktur abgelegt.
- *Bounden* (Beschränkung) repräsentiert den zweiten Schritt, in welchem die Lösungsmenge beschränkt wird. Dabei wird überprüft, welche Zweige suboptimal sind und als mögliche Lösungen verworfen werden.

Eine Lösung wird verworfen, wenn das Gewicht eines *1-tree* größer ist als das Gewicht einer zuvor gefundenen möglichen Lösung. Ein *1-tree* beschreibt dabei einen Weg von der Hauptwurzel bis zu einem Endblatt. Diese Art der Routenfindung liefert rasch brauchbare Ergebnisse [9].

Nearest-Neighbor (NN): Die NN-Heuristik stellt ein mögliches Eröffnungsverfahren dar. Diese liefert zwar eine Approximation der Lösungen, aber meist nicht sehr gute Resultate. Der Algorithmus kann dazu verwendet werden, um eine Beispiellösung zu generieren. Diese Lösung kann als Ausgangszustand für Verbesserungsheuristiken eingesetzt werden, welche selbst keine Grundlösung finden würden. Der Grundalgorithmus wählt zufällig einen Knoten aus dem System und sucht sich anhand diverser Kriterien einen weiteren Knoten, welcher als *Nearest-Neigbor* bezeichnet wird. Von diesem ausgehend wird ein weiterer Knoten zur Route hinzugefügt, bis diese vollständig ist. Ein Nachteil dieses Algorithmus lässt sich am Ende erkennen, wenn die Route vollständig ist. Start und Endpunkt liegen oft sehr weit auseinander. Dieses Verhalten ist meist nicht gewünscht, da eine Rundreise erwartet wird und nicht eine Route im Sinne einer Linie und einem langen Rückweg [8].
Von diesem Algorithmus aus existieren noch einige weitere Spezialisierungen, wie zum Beispiel der *Variable Neighbourhood Search* Algorithmus. Dieser nützt den Vergleich von lokalen Optima zu globalen Optima, um effizientere Routen zu finden.

Savings-Algorithmus (SA): Der SA von Clarke und Wright gehört nicht zur Kategorie der Eröffnungsverfahren, sondern zu den Optimierungsalgorithmen. Bei diesem wird hauptsächlich mit einem Saving gearbeitet. Dieses Saving repräsentiert, wie viel gespart werden kann, wenn der Kunde B nach dem Kunden A angefahren wird und somit zwischen diesen Kunden nicht zum Depot zurückgekehrt werden muss. Dabei spielt die Berechnung des

Savings aus den Kosten eine wichtige Rolle, auf welche später noch genauer eingegangen wird.
Der Algorithmus beruht darauf, in jedem Schritt eine Route zu vergrößern und dabei einige andere zu entfernen. Der grundsätzliche Ablauf sieht wie folgt aus:

1. Zu Beginn müssen Werte zur Berechnung der Ersparnisse initialisiert werden, wie eine Zeit- oder Distanzmatrix. In diesem Schritt wird aus jedem Knoten/Kunde eine triviale Route erstellt, bestehend aus *Depot - Knoten - Depot*.
2. Im nächsten Schritt wird jede Route mit jeder anderen Route gepaart und der Savings-Wert berechnet. In diesem Schritt muss auch überprüft werden, ob die Kombinationen die Nebenbedingungen einhalten. Sollte dies nicht der Fall sein, muss diese Möglichkeit verworfen werden.
3. Die Kombination von zwei Routen mit dem höchsten Saving, wird im Anschluss zu einer neuen Route vereint.
4. Nach der Kombination müssen alle Möglichkeiten entfernt werden, welche die zweite Teilroute beinhalten. Zusätzlich müssen alle Teilrouten mit dem ersten Teil durch die neue Route ersetzt werden.
5. Infolgedessen wird bei Punkt 2 fortgesetzt, wenn sich noch Kombinationen an Routen in der Savingsliste befinden. Sollte dies nicht der Fall sein, so wurde eine optimierte Route gefunden oder mehrere Teilrouten, welche nicht mehr kombiniert werden können.

Sweep-Algorithmus: Der Sweep-Algorithmus verwendet wie zuvor schon erwähnt ein *Cluster-first* Vorgehen. Dieser verwendet ein geometrisches Verfahren zum Erstellen des Clusters, weshalb dieser nur in planaren Instanzen angewendet werden kann. Jeder Kunde im System wird in einem Polarkoordinatensystem repräsentiert. Jeder besitzt einen Winkel und einen direkten Abstand zum zentralen Depot.

- Zu Beginn wird ein beliebiger Knoten/Kunde als erster Haltepunkt in der Route ausgewählt.
- Im Anschluss wird zu diesem Ausgangspunkt ein weiterer Kunde hinzugefügt und zwar dieser mit dem geringsten Winkelabstand zum letzten Punkt in der Route.
- Dieser Prozess wird solange durchgeführt, bis eine Route eine maximale Anzahl an Punkten erreicht hat oder ein Fahrzeug keine Kapazität mehr zur Verfügung hat.
- Damit wurde ein Cluster an Stopps für ein Fahrzeug erstellt, welcher im einem zweiten Schritt zu einer optimierten Route entwickelt werden muss.

Somit gehört auch dieser Algorithmus zur Kategorie der Eröffnungsverfahren [8].

3.2.3 Meta-Heuristiken

Meta-Heuristik wird in der Informatik als Bezeichnung verwendet, wenn eine Lösung näherungsweise erzeugt wird. Diese sind nicht stark problemspezifisch orientiert, im Gegensatz zu Heuristiken. Diese Art von Algorithmen wird im Bereich der Logistik deshalb eingesetzt, weil sie lokalen Optima in schwereren und größeren Problemen entwischen können und eher zu einem globalen Optima tendieren oder konvergieren.

Tabu-Search (TS): Der *Tabu-Search* baut auf keiner bisher beschriebenen Methode auf. Im Unterschied zu anderen Algorithmen sind beim TS auch ungültige Lösungen zugelassen, welche die Nebenbedingungen verletzen. Es muss dafür eine Kostenfunktion existieren, mit welcher die Kosten einer Lösung festgestellt werden können. Für jede gefundene Lösung wird evaluiert, ob dies eine verwendbare Lösung ist oder nicht. Hierbei fließen die einzelnen Nebenbedingungen, mit einem hinzumultiplizierten Faktor, eine Rolle. So kann festgestellt werden, ob es sich um eine bessere Lösung handelt oder nicht. Eine Kernfunktionalität stellt die Tabu-Liste dar. In dieser werden Unmöglichkeiten gespeichert, welche nicht durchgeführt werden dürfen, solange sie sich in der Liste befinden. Zum Beispiel darf von einem Ort aus nicht nach Osten zum nächsten gefahren werden, da dieser Zug in der Iteration vorher durchgeführt worden ist.
Der Hauptalgorithmus besteht aus einer Hauptschleife, welche an ein Abbruchkriterium geknüpft ist. Zu Beginn wird dabei eine mögliche Lösung initialisiert und als aktuell beste Lösung angenommen. Im Anschluss werden weitere mögliche Lösungen erzeugt und die Beste daraus ausgewählt. Diese alternative Lösung wird mit der aktuell besten Lösung verglichen und als neue Beste definiert, wenn dies der Fall ist. Daraufhin wird die Tabu-Liste aktualisiert. Dieser Ablauf wird solange durchgeführt, bis das Abbruchkriterium eintritt.
Bei diesem Algorithmus wird nicht garantiert, dass eine gültige Lösung gefunden wird. So kann aber eine Lösung entstehen, welche weit geringere Kosten mit sich bringt als eine Lösung ohne Verletzung der Nebenbedingungen. Im Falle einer Applikation muss durch die Evaluierungsfunktion definiert sein, in wie weit eine Nichteinhaltung toleriert wird oder nicht [8].

Simulated Annealing (SA): SA ist für sich im Detail kein Algorithmus, sondern bezeichnet eine Klasse von Algorithmen, welche versuchen global zu optimieren. Die Idee und die Bezeichnung stammen hierbei aus der Härtung von Stahl. In der Stahlproduktion wird dieser *erwärmt* und wieder *abgekühlt*, was als *härten* bezeichnet wird. Dieser Prozess des Abkühlens führt

dazu, dass sich die Teilchen wie in einer Gitterstruktur anordnen. Zu Beginn kann von einer stochastischen Suche gesprochen werden und zum Ende hin von einer deterministischen Suche. Zum Ende des Prozesses werden die Schritte immer kleiner, wie bei der Härtung kühlen dabei die letzten Temperaturen sehr langsam ab. Im Gesamten wird eine global optimale Konfiguration gefunden, in dem dieser Prozess simuliert wird. Am Anfang wird deshalb von einer stochastischen Suche gesprochen, da das System Möglichkeiten annimmt, welche zu einer Verschlechterung des Ergebnisses führen. Wird die sogenannte Temperatur geringer, minimiert sich die Akzeptanz für Möglichkeiten, was wiederum deterministisch ist. Der stochastische Anteil ermöglicht dem lokalen Minima zu entkommen und dem globalen Minima sich anzunähern. Der Grundalgorithmus ähnelt dabei sehr dem *Tabu-Search* mit der Hauptschleife. Beim SA wird solange optimiert, bis das System einen erstarrten Zustand erreicht [5].

Ant Colony Optimierung (ACO): ACO repräsentiert eine Simulation von sozialen Interaktionen in der Natur. So wurde für diesen Algorithmus das Verhalten und die Interaktionen von Ameisen, Bienen und weiteren Insekten analysiert. Jedes Individuum dieser Insekten erledigt eine eigene Aufgabe, unabhängig von anderen in der Kolonie. Ein funktionierendes System kommt dadurch zustande, dass die Arbeiten jedes einzelnen Individuums auf den Arbeiten eines anderen aufbauen oder mit anderen verbunden sind. Durch die Kooperation in der Kolonie, entsteht die Möglichkeit, komplexe Probleme zu lösen und Überlebensstrategien auszuführen [1, 7].
Damit dieser Algorithmus eingesetzt werden kann, müssen einige Grundvoraussetzungen erfüllt sein:

- Eine Strategie zum Überprüfen einer gültigen Lösung, damit nur solche akzeptiert werden;
- Eine heuristische Funktion η zum Messen der Qualitäten der Teile, welche sich noch nicht in der Teillösung befinden;
- Eine Funktion zum Aktualisieren der Pheromone τ, welche die Wertigkeit der Kanten repräsentiert;
- Eine Funktion ϕ, welche auf der heuristischen Funktion η und den Pheromonen aus τ basiert und iterativ mögliche Lösungen konstruiert;

Der grobe Ablauf des Grundalgorithmus:

- Zu Beginn wird die Ameise an einem Startpunkt ausgesetzt;
- Diese besitzt nun die Möglichkeit sich mit einer bestimmte Anzahl von Zügen durch das Netzwerk zu bewegen unter Berücksichtigung der Funktion ϕ; dabei wird eine Tabu-Liste für diese Ameise erstellt;
- Im Anschluss wird die Tour der Ameise bewertet und die Pheromone der Kanten berechnet; eine weitere Ameise wird ausgesetzt bis die maximal Anzahl der Ameisen erreicht ist;

- Nachdem alle Ameisen einmal im Netzwerk waren, werden alle Pheromone an den Kanten aktualisiert;
- Diese entstandene Lösung wird nun mit der letzten verglichen und ein weiterer Durchlauf wird unter Berücksichtigung eines Abbruchkriteriums mit den neuen Pheromonen eingeleitet;

Evolutionäre Algorithmen (EA): EA sind wie SA kein Algorithmus, sondern ein Überbegriff für eine Klasse von Algorithmen und Methoden. Das Prinzip dieser Algorithmen beruht auf dem Vorgang einer biologischen Evolution. Die Natur einer biologischen Evolution ist es, sich zu verbessern und zu überleben, was äquivalent zu einem Optimierungsprozess ist. In der Natur sind die Vorgänge der Optimierung bei einem Menschen sehr langsam und für ein einzelnes Individuum nicht wahrnehmbar. Dabei muss sich jede Generation aufs Neue behaupten und beweisen, dass sie mit der aktuellen Umgebung besser zurechtkommt. Diese Selektion geschieht ganz nach *Charles Darwins* Worten *survival of the fittest*, sodass der Angepassteste/Fitteste sich am stärksten verbreitet beziehungsweise vermehrt. Die Veränderungen entstehen dabei durch einfache Fortpflanzung im Sinne einer Rekombination und durch Mutationen. So wird bei der Mutation nur ein Individuum benötigt, welches verändert wird und bei der Rekombination zwei.
Ein EA besitzt genau solche Eigenschaften und versucht hier die Natur nachzuahmen. Im Sinne eines Algorithmus werden die Rekombination und Mutation schneller durchgeführt, sodass immer Mengen an potentiellen Lösungen zur Verfügung stehen. Alle Lösungen werden evaluiert und ihre Fitness bestimmt. Unbrauchbare Lösungen werden aussortiert, wenn sie nicht fit genug sind, bis eine geeignete Lösung gefunden wurde oder ein Abbruchkriterium wirksam wird [2, 4].
Der grundsätzliche Ablauf sieht wie folgt aus:

1. Zu Beginn muss eine Beispiellösung initialisiert und bewertet werden, um einen Ausgangszustand zu erlangen;
2. Im Anschluss wird eine Selektion durchgeführt;
3. Die Selektierten werden rekombiniert und so Nachkommen erzeugt;
4. Auf die gesamte Menge an Lösungen wird eine Mutation angewendet;
5. Zum Abschluss werden alle Lösungen bewertet und ihre Fitness wird festgestellt;
6. Sollte kein Abbruchkriterium erfüllt sein, so wird ab Punkt 2 wieder fortgesetzt;

In diesem Kapitel wurden grundlegende Eigenschaften in der Logistik erklärt und Problemgebiete aufgezeigt. Es wurden einige Lösungsansätze beschrieben, um deren Grundalgorithmus zu verstehen. Im nächsten Kapitel wird auf Kostenfunktionen eingegangen und Probleme darin aufgezeigt.

Kapitel 4

Kostenberechnung

Die Kostenfunktion stellt in jedem VRP und dessen Ablegern eine Kernkomponente dar. Diese muss an das Problem angepasst werden und ist somit problem- und zielspezifisch. Mit dieser Funktion wird definiert wie und was optimiert werden soll. Zum Beispiel kann diese so ausgelegt sein, dass die Strecke oder Fahrzeit minimal sein soll oder beides gemeinsam. Im weiteren Verlauf dieses Kapitels wird auf einige Eigenschaften der Kostenfunktion für den Savings-Algorithmus eingegangen.

4.1 Distanz-Basierend

Rein auf Distanz-Basierende-Kostenfunktionen bringen meist nicht das gewünschte Ergebnis zu Tage. Dies lässt sich auf die Einschränkung der vorhandenen Daten zurückführen. Alle Ergebnisse basieren dabei rein auf den Distanzen, welche gefahren werden. In diesem Fall könnte ein Kunde einige wenige Kilometer entfernt sein, direkt aber nur über einen Feldweg. Dies würde zu einer kurzen Distanz führen ohne die Beschaffenheit der Strecke und der dabei benötigten Zeit zu berücksichtigen. Sollte trotzdem nur die Distanz zur Verfügung stehen, so ist dies besser als überhaupt keine Informationen zu haben.
Die Berechnung für die Ersparnis im SA ist in der Gleichung 4.1 zu finden. Dabei stellen die Punkte A und B zwei Kunden dar und DE das Depot, von welchem aus losgefahren wird. Die Variable *dis* repräsentiert die Distanzmatrix, welche alle Distanzen zwischen den Punkten beinhaltet. Die Ersparnis ist hierbei definiert als das, was eingespart wird, wenn zwischen Kunde A und B nicht zum Depot zurückgekehrt, sondern direkt von A nach B gefahren wird [3].

$$sav_{a,b} := dis_{a,z} + dis_{z,b} - dis_{a,b} \tag{4.1}$$

In der Abbildung 4.1 ist ein einfaches Netzwerk mit Kunden und einem Depot abgebildet. Die Abkürzung AB steht in diesem Beispiel für Autobahn,

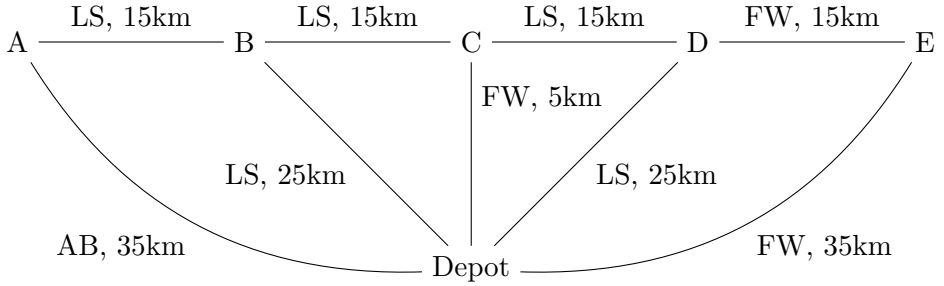

Abbildung 4.1: Ein einfaches Netzwerk mit 4 Kunden und einem Depot

LS für Landstraße und *FW* für Feldweg. Wie in dieser Abbildung festgestellt werden kann, existieren zwei Kanten mit einem Feldweg und eine mit einer Autobahn. An diesem Beispiel lässt sich erkennen, dass die Beschaffenheit des Fahrwegs ignoriert und vernachlässigt wird. In Tabelle 4.1 befinden sich die Ergebnisse der Ersparnisse zu Beginn des Savings-Algorithmus nach der Initialisierung. Im Gesamten lässt sich hier erkennen, dass lange Strecken bevorzugt werden. In diesem Fall würden zuerst die Routen *A* - *B* oder *D* - *E* zusammengefügt werden, abhängig von der Sortierung der Savings-Liste.

Route	Ersparnis
A $\leftrightarrow$ B	$35 + 25 - 15 = 45$
B $\leftrightarrow$ C	$25 + 5 - 15 = 15$
C $\leftrightarrow$ D	$5 + 25 - 15 = 15$
D $\leftrightarrow$ E	$25 + 35 - 15 = 45$

Tabelle 4.1: Aufschlüsselung der Ersparnisse anhand der Teilrouten zu Beginn der Optimierung

An diesem Beispiel lässt sich erkennen, dass die Beschaffenheit einer Straße ignoriert wird. Für den Algorithmus sieht dies so aus, als ob dies eine gute Kombination wäre. Im Grund stimmt dies, zumal die Ersparnis größer ist.

4.2 Zeit-Basierend

Zeit-Basierende-Kostenfunktionen beinhalten einen Vorteil gegenüber der Distanz-Basierten-Version, aber bergen auch Probleme. Der Vorteil der Zeitinformation liegt darin, dass die Beschaffenheit der Strecke versteckt mit einfließt. So wird auf einer Strecke mit Autobahn weniger Zeit benötigt, als auf derselben Distanz, aber mit Feldweg als Beschaffenheit.
Die Berechnung für die Ersparnis im Savings-Algorithmus ist in der Gleichung 4.2 zu finden. Dabei ist diese identisch mit der Gleichung aus 4.1 mit

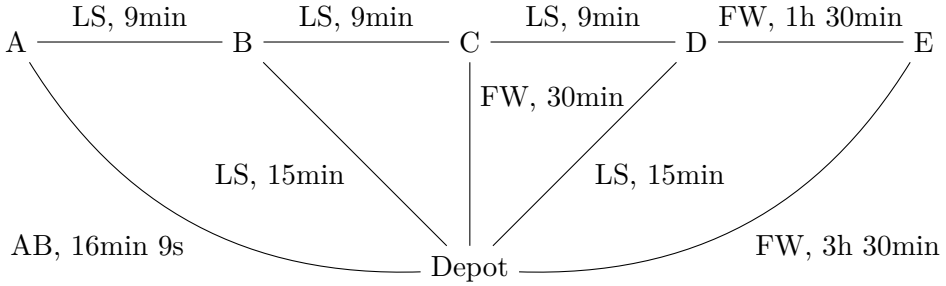

Abbildung 4.2: Derselbe Graph aus 4.1 mit 4 Kunden und einem Depot mit den benötigten Zeiten

dem Unterschied, dass hier eine Zeitmatrix *time* verwendet wird.

$$sav_{a,b} := time_{a,z} + time_{z,b} - time_{a,b} \tag{4.2}$$

In Abbildung 4.2 befindet sich das gleiche Netzwerk aus 4.1, aber mit Zeiten an den Kanten. Die Abkürzung AB steht für Autobahn mit $130\,km/h$, LS für Landstraße mit $100\,km/h$ und FW für Feldweg mit $10\,km/h$. In der Tabelle 4.2 befinden sich die Ergebnisse der Ersparnisse zu Beginn des Savings-Algorithmus nach der Initialisierung. Erkennbar ist hier, dass nicht die Strecke mit der Autobahn bevorzugt wird, sondern der langsame Feldweg. Dieses Verhalten führt unweigerlich dazu, dass der Feldweg als erste Kombination verwendet wird.

Route	Ersparnis
A $\leftrightarrow$ B	$16,15 + 15 - 9 = 22,15$
B $\leftrightarrow$ C	$15 + 30 - 9 = 36$
C $\leftrightarrow$ D	$30 + 15 - 9 = 36$
D $\leftrightarrow$ E	$15 + 210 - 90 = 135$

Tabelle 4.2: Aufschlüsselung der Ersparnisse anhand der Teilrouten zu Beginn der Optimierung mit den benötigten Zeiten als Basis

4.3 Distanz-Zeit-Basierend

Eine Kostenberechnung mit Distanz- und Zeitwerten bietet die Möglichkeit, die Beschaffenheit der Strecke zu berücksichtigen und so zum Beispiel Feldwege zu ignorieren. Diese Daten erzeugen ein Problem. Es wirft die Frage auf, wie die Distanzen und die Zeiten in eine Relation gesetzt werden sollen. Einfaches Addieren würde dazu führen, dass eine Strecke von $45\,km$ mit einer Geschwindigkeit von $10\,km/h$ zu einer ungewollt hohen Ersparnis

führen würde. Damit dieses Verhalten nicht auftritt, muss ein Wert durch den anderen dividiert werden.
Eine für diesen Fall akzeptable Verhältnisrechnung wäre eine Division aus der Strecke in Metern und der Zeit in Minuten. In Tabelle 4.3 sind hierzu die Wertigkeiten der Kanten aus den Abbildungen 4.1 und 4.2 zu finden. An diesem Beispiel lässt sich erkennen, dass die Kante vom *Depot* zum Knoten *A* mit $2786,37$ besser bewertet wird, im Gegensatz zu der Kante vom *Depot* zum Knoten *E*. Im Grunde ist so ein Verhalten das Gewünschte, denn eine Strecke mit einer schlechten Streckenbeschaffenheit wird schwächer bewertet. Für die weitere Savingsberechnung kann die Basisberechnung verwendet

Kante	Wertigkeit
A $\leftrightarrow$ B, B $\leftrightarrow$ C, C $\leftrightarrow$ D	$15000m/9min = 1666,67$
D $\leftrightarrow$ E	$15000m/90min = 166,67$
Depot $\leftrightarrow$ A	$45000m/16,15min = 2786,37$
Depot $\leftrightarrow$ B, Depot $\leftrightarrow$ D	$25000m/15min = 1666,67$
Depot $\leftrightarrow$ C	$5000m/30min = 166,679$
Depot $\leftrightarrow$ E	$45000m/210min = 214,29$

Tabelle 4.3: Aufschlüsselung der Kantenwertigkeit anhand der Strecke und der dafür benötigten Zeit

werden, mit der berechneten Distanz-Zeit-Matrix. Diese Matrix beinhaltet dabei die Werte wie zuvor beschrieben.
Basierend auf den in der Tabelle 4.3 berechneten Kantenwertigkeiten beinhaltet die Tabelle 4.4 die daraus resultierenden Ersparnisse. Dafür wird derselbe Graph aus 4.1 mit den neuen Kantenwertigkeiten verwendet. An diesem Beispiel lässt sich erkennen, dass die Autobahn vom *Depot* zum Knoten *A* bevorzugt wird.

Route	Ersparnis
A $\leftrightarrow$ B	$2786,37 + 1666,67 - 1666,67 = 2786,37$
B $\leftrightarrow$ C	$1666,67 + 166,67 - 1666,67 = 166,67$
C $\leftrightarrow$ D	$166,67 + 1666,67 - 1666,67 = 166,67$
D $\leftrightarrow$ E	$1666,67 + 214,29 - 166,67 = 1724,29$

Tabelle 4.4: Ersparnisse für die Kombinationen mit den berechneten Werten aus Distanz und Zeit

Die Findung der passenden Kostenberechnung stellt den Entwickler des Algorithmus vor ein größeres Problem, als die Implementierung des grundsätzlichen Algorithmus. So sollten mehrere Überlegungen festgehalten und getestet werden. So kann das Verhalten festgestellt werden und auch in welche Richtung optimiert wird. Das Überprüfen und Testen kann einfach in

einem Script mit einem Graph aus der Praxis erfolgen, bei welchem die optimale Route bekannt ist. Auf diese Art kann sichtbar gemacht werden, wie gut der Algorithmus und dessen Kostenfunktion optimieren und Lösungen finden.

Kapitel 5

Problemgebiete

5.1 Kostenberechnung

Die Kostenberechnung und somit im Falle des SA die Savings-Funktion stellt den Entwickler dieser Funktion vor eine größere Herausforderung. Dabei können Fehler entstehen, die meist nur sehr schwer zu finden sind und zwar nur, wenn ein gesamter Prozess durchgesteppt wird. Hier empfiehlt es sich eine Liste der zeitlichen Werte zu notieren, um im Nachhinein die Historie des Ablaufes noch einmal aufarbeiten zu können.
Wie in Sektion 4.3 erkennbar ist, führt eine Verhältnismatrix dazu, dass unterschiedliche lange Strecken vom selben Typ zum selben Wert führen. Das Ergebnis führt dazu, dass die Knoten, welche an eine Autobahn angebunden sind, zu Beginn verknüpft werden. Im Anschluss folgt die nächsten Klasse. Das Ergebnis wirkt hier auf den ersten Blick zufriedenstellend, aber nicht, wenn dies weiter verfolgt wird. Eine Abhilfe schafft dabei das Einbeziehen der Distanz in die Berechnung. Wie in der Gleichung 5.1 festgehalten, kann auf diese Weise eine Differenzierung erreicht werden.

$$kost_{a,b} := dis_{a,b} * (\frac{dis_{a,b}}{time_{a,b}}) \tag{5.1}$$

Diese Gleichung ermöglicht es Kanten mit großen Werten und unbrauchbaren Eigenschaften diese zu neutralisieren.

5.2 Nebenbedingung

Nebenbedingungen bieten die Möglichkeit, die konkrete Problemstellung in den Algorithmus einfließen zu lassen. Bei diesen muss beachtet werden, dass diese immer überprüft werden. Sollte eine Verletzung eintreten, muss zusätzlich entschieden werden, ob diese Möglichkeit verworfen werden soll oder doch toleriert wird. Im Falle einer Tolerierung kann zusätzlich mit Strafen gearbeitet werden, sodass eine solche Möglichkeit an Gewichtung verliert.

Time Windows (Zeitfenster): Mit der Verwendung von Zeitfenstern besteht die Erweiterung der Knoten um Zeitbereiche, in welchen ein Kunde anzutreffen ist. Für diesen Fall müssen alle Knoten mit dieser Information ausgestattet werden oder eine Standardzeit angenommen werden. Bei der Verwendung in einem Savings-Algorithmus muss bei jeder Berechnung der neuen Ersparnisse überprüft werden, ob diese Nebenbedingung eingehalten wird. Der Grundalgorithmus geht anlässlich einer Verletzung von einer scharfen Nichteinhaltung aus. Dies führt zur Entfernung einiger Möglichkeiten. Eine weitere Möglichkeit bietet das Herabstufen einer solchen Möglichkeit, was zu einer sogenannten weichen Nebenbedingung führt.
Es kann definitiv die Aussage getroffen werden, dass Routing-Optimierungen kein triviales Problem darstellen. In diesem Sinne gehören sie auch zu der Kategorie *NP-vollständig*. Dies bedeutet, dass sie sich nichtdeterministisch in Polynomialzeit lösen lassen. So gibt es viele Möglichkeiten, um das Problem zu lösen, aber nicht ohne entsprechenden zeitlichen Aufwand. Die Polynomialzeit bildet dabei eine Grenze zwischen dem Bereich des *praktisch lösbaren* sowie *praktisch nicht lösbaren*.

Im folgenden Kapitel wird ein Beispiel mit dem Savings-Algorithmus aufgearbeitet. Anhand dessen wird die Auswirkung der letzten Kosten/Savings-Funktion mit der Gleichung 5.1 genauer überprüft.

Kapitel 6

Logistik mit Zeitfenster

6.1 Saving mit Zeitfenster in Python

Dieses Beispiel stellt eine vereinfachte Welt dar, welche mit 5 Knoten/Kunden und 1 Depot ausgestattet ist. Das gesamte Netzwerk wurde nicht voll vernetzt, wie in Abbildung 6.1 ersichtlich ist. Aus diesem Grund muss der Knoten D von A oder B aus über E angefahren werden. Dies gilt auch für die Strecke vom Depot aus. Die Zeit- und Distanzmatrix sind in diesem Beispiel vollständig. So beinhalten sie auch die kombinierte Strecke von $A \rightarrow E \rightarrow D$ mit den aufsummierten Kosten. In Tabelle 6.1 sind alle benötigten Werte des Graphen 6.1 aufgeschlüsselt. Hierbei muss beachtet werden, dass die Zeit in Minuten und Sekunden angegeben ist. Das Beispiel beinhaltet den gesamten Algorithmus 6.1, welcher mit der Skriptsprache Python implementiert ist. Wie in einer Sektion zuvor schon beschrieben muss mit der Initialisierung begonnen werden. Hierzu müssen die Kostenmatrix sowie die trivialen Routen erstellt werden. Bei der Distanzmatrix ist zu beachten, dass diese in Kilometern beschrieben ist und intern auf Meter umgerechnet wird. Des Weiteren beinhaltet die Zeitmatrix die Zeiten bereits in Sekunden.

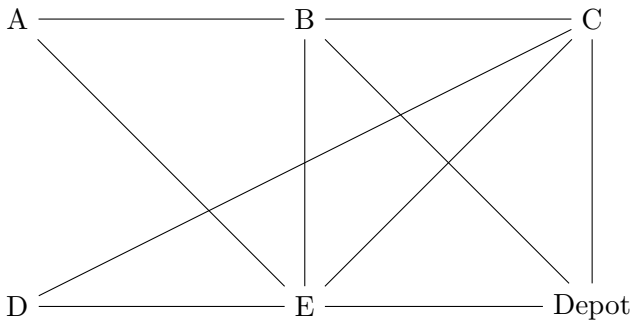

Abbildung 6.1: Beispiel Graph mit 5 Knoten/Kunden und 1 Depot

Kante	km/h	km	Zeit (mm:ss)
A ↔ B	80	10	07:30
A ↔ E	100	14,14	08:29
B ↔ C	90	10	06:40
B ↔ E	110	10	05:27
B ↔ DE	100	14,14	08:29
C ↔ D	130	22,36	10:19
C ↔ E	70	14,14	12:07
C ↔ DE	100	10	06:00
D ↔ E	90	10	06:40
E ↔ DE	100	10	06:00

Tabelle 6.1: Aufschlüsselung der Distanzen, Durchschnittsgeschwindigkeiten und der benötigten Zeiten

In dieser Implementierung des Algorithmus werden alle Zeiten in Sekunden gerechnet inklusive der Wartezeiten.
Eine triviale Route beinhaltet nur einen Knoten, sodass die Strecke *Depot → Knoten/Kunde → Depot* abgebildet wird. Weiters werden die möglichen Kombinationen erstellt und deren Ersparnis berechnet. Bei der Berechnung des Savings-Wertes muss die Nebenbedingung der Zeitfenster überprüft werden. Dieser erste Schritt erstreckt sich von Zeile 23 bis Zeile 70.
Im zweiten Schritt werden Routen kombiniert, bis keine Kombinationen mehr durchführbar sind. Dabei müssen nach der Kombination zweier Routen alle Einträge aus der Saving-Liste aktualisiert werden. Zu diesen Aktualisierungsaufgaben zählen:

- Entfernen aller Kombinationen, welche die zweite Route als Teil beinhalten;
- Ersetzen aller Routen in Kombinationen, welche die erste Route als Teil beinhalten;

Im Abschluss des zweiten Teils müssen die Ersparnisse mit überarbeiteten Routen neu berechnet und dabei die Zeitfenster geprüft werden.

Listing 6.1: Basis Implementierung des Saving-Algorithmus mit Zeitfenster

```
1  #!/usr/bin/env python
2  """
3  Saving-Algorithmus
4  Routing in Pyhton
5
6  Step 1:
7      - Init costmatrix
8      - Init base routes
9      - Init savings-list + check time windows
10     - Order savings-list desc
11 Step 2:
```

```
12      - Select top saving
13      - Merge Routes
14      - Check time windows
15      - Clear no more available savings
16      - Replace route a with new route a in savings
17      - Calc savings-list + order desc
18  """
19  import copy
20  from route import *
21  from util import *
22
23  node_de = node(0, 'DE', time_window(7, 19)) # -> Depot
24  node_a = node(1, 'A', time_window(10, 13)) # -> A
25  node_b = node(2, 'B', time_window(7, 19)) # -> B; Depot time window
26  node_c = node(3, 'C', time_window(8, 12)) # -> C
27  node_d = node(4, 'D', time_window(7, 19)) # -> D; Depot time window
28  node_e = node(5, 'E', time_window(8, 11)) # -> E
29
30
31  """      DE,     A,       B,       C,       D,       E"""
32  d_mx = [[0,      24.14,   14.14,   10,      20,      10      ], # DE
33          [24.14,  0,       10,      20,      24.14,   14.14   ], # A
34          [14.14,  10,      0,       10,      20,      10      ], # B
35          [10,     20,      10,      0,       22.36,   14.14   ], # C
36          [20,     24.14,   20,      22.36,   0,       10      ], # D
37          [10,     14.14,   10,      14.14,   10,      0       ]] # E
38
39  """      DE,     A,       B,       C,       D,       E"""
40  t_mx = [[0,      869,     509,     360,     760,     360     ], # DE
41          [869,    0,       450,     850,     909,     509     ], # A
42          [509,    450,     0,       400,     727,     327     ], # B
43          [350,    850,     400,     0,       619,     727     ], # C
44          [760,    909,     727,     619,     0,       400     ], # D
45          [360,    509,     327,     727,     400,     0       ]] # E
46
47  matrix = cost_matrix(d_mx, t_mx, 6)
48
49  print(matrix)
50
51  routes = []
52  routes.append(route([node_a], 0, node_de, t_mx))
53  routes.append(route([node_b], 1, node_de, t_mx))
54  routes.append(route([node_c], 2, node_de, t_mx))
55  routes.append(route([node_d], 3, node_de, t_mx))
56  routes.append(route([node_e], 4, node_de, t_mx))
57
58  savings = []
59  for item_a in routes:
60      for item_b in routes:
61          if item_a is not item_b:
62              s = saving(
63                      copy.deepcopy(item_a),
64                      copy.deepcopy(item_b),
65                      matrix)
```

```
66              if s is not -1:
67                  savings.append(s)
68
69 savings = util.sort_savings(savings)
70 util.print_savings(savings)
71
72 while len(savings) > 0:
73     saving = savings.pop(0)
74     print('\nSelected Saving')
75     util.print_savings([saving])
76
77     route_b_id = saving.r_b.r_id
78     route = saving.r_a
79     route.add_stops(saving.r_b)
80
81     print('\nMerged Routes')
82     print(route)
83
84     savings = [x for x in savings if x.r_b.r_id != route_b_id]
85     savings = [x for x in savings if x.r_a.r_id != route_b_id]
86
87     routes = [r for r in routes if r.r_id != route_b_id]
88
89     for i in range(len(routes)):
90         if routes[i].r_id == route.r_id:
91             routes[i] = route
92
93     for saving in savings:
94         if saving.r_a.r_id == route.r_id:
95             saving.r_a = route
96         elif saving.r_b.r_id == route.r_id:
97             saving.r_b = route
98
99     savings = [x for x in savings if x.calc(matrix) is not -1]
100
101    savings = util.sort_savings(savings)
102
103    print('\nCurrentSaving & CurrentRoutes')
104    util.print_savings(savings)
105
106 print('\n\t ---- Result ---- ')
107 print(routes)
```

6.2 Ergebnis

Gesamtheitlich kann festgestellt werden, dass eine grundsätzlich optimierte Route gefunden wurde. Trotzdem sieht es danach aus, als ob es noch bessere Routen geben müsste, wie im Ergebnis 6.2 ersichtlich. Dabei muss wiederholt werden, dass es sich bei diesem Algorithmus um eine heuristische Lösung handelt.

Listing 6.2: Ergebnis des Beispiels

```
1    ---- Result ----
2 [id: 4
3 5: E [8.0 : 11.0] | 3240
4 4: D [7.0 : 19.0] | 0
5 1: A [10.0 : 13.0] | 3771
6 2: B [7.0 : 19.0] | 0
7 3: C [8.0 : 12.0] | 0
8 ]
```

Die Route $Depot \rightarrow E \rightarrow D \rightarrow A \rightarrow B \rightarrow C \rightarrow Depot$ spiegelt zusätzlich eine Eigenheit des Savings-Algorithmus wieder. Dazu muss die Historie genauer betrachtet werden. Wie im Listing 6.3 zu erkennen ist, werden in der ersten Vereinigung die Routen mit den Knoten A und B kombiniert. Der Grund liegt in der Ersparnis, da Knoten, die weiter vom Depot entfernt sind und nahe beieinander liegen, einen hohen Wert erlangen.

Listing 6.3: Erste Kombination von zwei Teilrouten

```
1 Selected Saving
2 SavingsCount: 1
3 841172.84
4    r_a: 1
5    r_b: 2
6
7 Merged Routes
8 id: 0
9 1: A [10.0 : 13.0] | 9931
10 2: B [7.0 : 19.0] | 0
```

Im Vergleich dazu bekommt die Vereinigung $C \leftrightarrow D$ nur auf einen sehr niedrigen Wert. Dies kommt mit der großen Entfernung zwischen den Punkten zustande und der geringeren Entfernung zum Depot.
In einem direkten Vergleich der generierten Tour und einer manuell zusammengestellten Tour lässt sich feststellen, dass es eine bessere Route gibt. In der Tabelle 6.2 befinden sich die Zeit- und Distanzkosten der generierten Tour. Die Tabelle 6.3 beinhaltet die Zeit- und Distanzkosten der manuellen

	E	D	A	B	C	DE	Summe
Zeit	360	400	400 + 509	450	400	360	2879 s
Distanz	10	10	10 + 14,14	10	10	10	74,14 km

Tabelle 6.2: Benötigte Zeit und Strecke der generierten Tour

erstellten Tour. Im Vergleich lässt sich erkennen, dass die Kante $C \leftrightarrow D$ mehr Zeit und Distanz mit sich bringt. Dafür wird die Kante $D \leftrightarrow E$ nicht mehrfach benötigt. Der zeitliche Unterschied liegt bei 32 Sekunden ohne Berücksichtigung der Wartezeiten. In Bezug auf den realen Straßenverkehr, kann dieser Wert vernachlässigt werden. Die berechnete Kostenmatrix führt

	C	D	E	A	B	DE	Summe
Zeit	360	619	400	509	450	509	2847 s
Distanz	10	22,36	10	14,14	10	14,14	60,64 km

Tabelle 6.3: Benötigte Zeit und Strecke der manuellen Tour

in diesem Fall dazu, dass die Strecke und die Zeit optimiert werden. Sollte aber die Distanz minimiert werden, so müsste eine Distanzmatrix als Kostenmatrix verwendet werden. Eine andere Möglichkeit wäre die Berechnung der Kosten abzuändern, sodass sich die Distanzen stärker widerspiegeln.

Kapitel 7

Zusammenfassung

Routing in der Logistik bietet die Möglichkeit effizienter Auslieferungen zu erledigen. Dabei existieren allgegenwärtig Navigationssysteme wie Google Maps. Diese Lösungen liefern meist nur die Möglichkeit einfache *point-to-point* Optimierungen durchzuführen. Diese Konsumentenlösungen verwenden ohne Internetzugang nur Offline-Daten ohne aktuelle Verkehrslagen zu berücksichtigen. Im Falle von Belieferungen wird eine höhere Abstraktion benötigt. So muss nicht nur die Strecke zwischen den Haltepunkten optimiert werden, sondern auch die Reihenfolge dieser. Dies führt somit von *point-to-point* Optimierungen, wie der Dijkstra-Algorithmus, zu Knoten/-Kunden Optimierungen, wie dem *Tabu-Search*.
Im Rahmen der Arbeit wurden einige Problemstellungen der Logistik aufgezeigt. Es wurden auch mögliche Lösungsansätze für solche Probleme gezeigt und beschrieben. Der Savings-Algorithmus mit seinen Eigenheiten wurde zudem genauer beschrieben. Des Weiteren stellen Kostenfunktionen eine zentrale Komponente in Routing-Algorithmen dar. Durch diese wird definiert, wie und nach was optimiert wird.
Der Einsatz von Optimierungen in der Logistik benötigt spezielle Algorithmen. Die Komplexität solcher Routing-Probleme stellt eine besondere Herausforderung dar. So kann in begrenzter Zeit nicht einfach das Optimum gefunden werden. Eine Ausnahme besteht, wenn die Anzahl an Stopps so gering ist, dass alle Kombinationen getestet werden können. Aus dieser zeitlichen Einschränkung heraus werden meistens heuristische und metaheuristische Lösungen verwendet. Somit werden Routing-Probleme, wie das *Traveling Salesman Problem*, die Menschheit noch weiter beschäftigen.

Quellenverzeichnis

Literatur

[1] Ajith Abraham, He Guo und Hongbo Liu. „Swarm intelligence: foundations, perspectives and applications“. In: *Swarm Intelligent Systems*. Springer, 2006, S. 3–25 (siehe S. 67).

[2] Hans-Georg Beyer u. a. *Evolution are Algorithmen Begriffe und Definitionen*. Techn. Ber. (siehe S. 68).

[3] Geoff Clarke und John W Wright. „Scheduling of vehicles from a central depot to a number of delivery points“. *Operations research* 12.4 (1964), S. 568–581 (siehe S. 69).

[4] Christian Fischer. *Intelligente Systeme in der Logistik*. 2008. URL: http://www.informatik.uni-ulm.de (siehe S. 68).

[5] Scott Kirkpatrick, Mario P Vecchi u. a. „Optimization by simulated annealing“. *science* 220.4598 (1983), S. 671–680 (siehe S. 67).

[6] Gilbert Laporte. „The vehicle routing problem: An overview of exact and approximate algorithms“. *European journal of operational research* 59.3 (1992), S. 345–358 (siehe S. 60–62).

[7] Antonio Mucherino, Stefka Fidanova und Maria Ganzha. „Ant colony optimization with environment changes: an application to GPS surveying“. In: *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*. IEEE. 2015, S. 495–500 (siehe S. 67).

[8] Jakob Puchinger. *Optimierungsverfahren in der Transportlogistik*. 2010. URL: https://www.ads.tuwien.ac.at/teaching/ss10/Transportlogistik/OTL_10.pdf (siehe S. 64, 66).

[9] Richard Wiener. „Branch and Bound Implementations of the Traveling Salesperson Problem - Part 4: Distributed processing solution using RMI.“ *Journal of Object Technology* 2.6 (2003), S. 51–65 (siehe S. 64).

[10] Bernhard Wurzer. „Fallbeispiele zur Tourenplanung mit Hilfe spezieller Tourenplanungssoftware am Beispiel eines Grazer Kleinunternehmens“. Diss. uniwien, 2010 (siehe S. 60, 62).