Deep Learning in der Bilderkennung

Steuerung autonomer mobiler Roboter mit Hilfe neuronaler Netze

Bachelorarbeit

im Fachbereich 2: Informatik und Ingenieurwissenschaften

vorgelegt von:	Arno Fuhrmann
Matrikelnummer:	974589
Abgabedatum:	31.05.2016
Erstgutachter:	Prof. Dr. Thomas Gabel
Zweitgutachter:	Prof. Dr. Christian Baun

Zusammenfassung

Die vorliegende Bachelorarbeit zeigt anhand einer praktischen Anwendung mit einem Roboterfahrzeug, wie Computer selbständig lernen sich durch visuelle Wahrnehmung in ihrem Umfeld zu orientieren und zu navigieren. Die Basis der Bachelorarbeit sind Konzepte des Deep Learning in künstlichen neuronalen Netzen im Bereich der Bilderkennung sowie der theoretische und praktische Vergleich von Konzepten und Methoden zum Training dieser Netze. Zudem wird mit einer Evaluierung festgestellt, welche der Methoden sich für die autonome Steuerung und Navigation des eigens für diese Aufgabe entwickelten Roboterfahrzeugs eignet. Mit dem praktischen Einsatz des Fahrzeugs wird die Umsetzung der theoretischen Konzepte und der Modelle des Deep Learning in künstlichen neuronalen Netzen demonstriert.

Abstract

This present bachelor thesis deals with the problem of machine learning for orientation and navigation by visual perception, based on a practical example using a robot vehicle. The basis of this thesis are deep learning concepts in artificial neural networks in the field of image recognition and the theoretical and practical comparison of training models for this networks. The usability of those methods in the context of robot navigation in dynamic environments is evaluated, using a specially for this work developed robot vehicle. Finally, the implementation of the theoretical concepts of these deep learning models is demonstrated with the practical use of this robot vehicle.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

CNN	Convolutional Neural Network
CVPR	Conference on Computer Vision and Pattern Recognition
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
KNN	Künstliches neuronales Netz
MLP	Multilayer-Perceptron

1 Einleitung

Die außerordentlichen Fähigkeiten und Leistungen des menschlichen Gehirns sind für die Forschung der künstlichen Intelligenz stetige Inspiration, die Funktionsweise des Gehirns zu verstehen und in Analogie auf Computern mittels künstlicher neuronaler Netze zu simulieren [11]. Seit den Arbeiten zum ersten Modell für Neuronen, der von Warren McCulloch und Walter Pitts im Jahr 1943 vorgestellten McCulloch-Pitts-Zelle [21], haben weiterentwickelte, mathematische und programmiertechnische Konzepte für künstliche neuronale Netze sowie stetige Leistungssteigerungen der Computerhardware Produktinnovationen ermöglicht, die zuvor als utopisch oder gar unmöglich galten [19]. Trotz vieler nachweisbarer Erfolge bei der Entwicklung praktischer Anwendungen für viele Lebensbereiche wurde es nach anfänglicher Euphorie zunächst merklich still um das Thema künstliche Intelligenz und künstliche neuronale Netze, da die anfänglichen viel zu hohen Erwartung nicht erfüllt werden konnten. Insbesondere durch die seit der Einführung der Convolutional Neural Networks durch LeCun et al. [19] enorm gesteigerte Leistungsfähigkeit von Computern und den Einsatz von Deep Learning in sogenannten Deep Convolutional Neural Networks konnten bei der Klassifizerung von Bildern herausragende Ergebnisse erzielt werden [16][31][32]. Computer mit Softwareanwendungen die diese künstlichen neuronalen Netze einsetzen, werden damit in die Lage versetzt, sich in ihrem Umfeld, analog zu Menschen, durch visuelle Wahrnehmung zu orientieren [3]. Spätestens seit den Diskussionen um die mit Hilfe von künstlichen neuronalen Netzen autonom fahrenden Autos im öffentlichen Straßenverkehr, die bereits in der Umsetzung erprobt werden, sind die enormen Möglichkeiten, die der Einsatz von künstlichen neuronalen Netzen bietet, wieder in das Bewusstsein der Öffentlichkeit zurückgekehrt [2].

1.1 Ziel der Arbeit

Entwickler von mobilen autonomen Systemen, die die Navigation auf Grundlage von Bilddaten durchführen, wurden in der Vergangenheit aufgrund von begrenzter Leistungsfähigkeit und Flexibilität der vorhandenen Systeme vor große Herausforderungen gestellt. Ziel der vorliegenden Bachelorarbeit ist der theoretische und praktische Vergleich von aktuellen Methoden des maschinellen Lernens im Kontext der Navigation von autonomen mobilen Robotern mit Hilfe des Deep Learning in der Bilderkennung. Es wird der Ansatz verfolgt, bereits veröffentlichte Modelle mit guten Erkennungsleistungen bezüglich ihrer Fähigkeit zur Bilderkennung in dynamischen Systemen zu prüfen und nach einer Auswahl eine Lösung zur Steuerung des im Rahmen dieser Arbeit entwickelten Roboterfahrzeugs zu implementieren. Dazu werden die in ihrem Aufbau unveränderten Modelle mit den vorgesehenen Datensätzen sowie angepasste Modelle mit eigenen Trainingsdaten hinsichtlich ihrer Lauffähigkeit und Performanz auf Standard-Hardware geprüft. Abschließend erfolgt die Umsetzung der theoretischen Konzepte und der Modelle des Deep Learning in künstlichen neuronalen Netzen mit dem praktischen Einsatz eines Roboterfahrzeugs.

1.2 Methodologie

Es existieren verschiedene Frameworks für neuronales Lernen. Angefangen mit frühen Programmbibliotheken wie OpenCV aus dem Jahre 2000 existieren heute einige weitere Frameworks wie Torch, Theano, Caffe, Neon und TensorFlow. Während Theano, Neon und TensorFlow auf Python basieren, wird für Caffe C++ und für Torch die weniger verbreitete Scriptsprache Lua verwendet. Bis auf Theano sind alle Frameworks dafür vorbereitet mehrere Grafikprozessoren zur Berechnung nutzen zu können. Neben den aufgeführten Frameworks befindet sich der IBM Machine Learning Stack noch in der Entwicklung.

Der thematische Einstieg erfolgte über einen Udacity-Kurs zu Deep Learning mit TensorFlow. In diesem Kurs erfolgte der Einstieg in die Thematik mit der Programmiersprache Python, mit der die Funktionsweise von Deep Learning einprägsam dargestellt werden konnte. Aufgrund der guten Dokumentation und des zur Verfügung gestelltem Sourcecodes, auch zu den behandelten

Modellen, wurden TensorFlow und Python für die praktische Umsetzung der Konzepte des Deep Learning in dieser Arbeit ausgewählt.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in 6 Kapitel. Kapitel 2 gibt zunächst einen Einblick in die Grundlagen der Bilderkennung, beschreibt die unterschiedlichen künstlichen neuronalen Netze sowie ihre Lernmethoden und schließt mit einem Machbarkeitsnachweis ab.

In Kapitel 3 werden aktuelle Modelle der Deep Convolutional Neural Networks mit Top-Ergebnissen in internationalen Wettbewerben beschrieben und für eine weitere Verwendung im Verlauf dieser Arbeit ausgewählt.

Kapitel 4 beschreibt die praktische Umsetzung des Trainings der ausgewählten Modelle, zunächst ohne Anpassungen unter Verwendung von Standard-Datensätzen und danach mit Anpassungen sowie eigenen Trainingsdaten.

Nachdem die Modelle hinsichtlich ihrer Fähigkeit zur Bilderkennung in dynamischen Systemen evaluiert wurden, befasst sich Kapitel 5 mit dem praktischen Einsatz eines dieser Modelle zur Navigation eines im Rahmen dieser Arbeit entwickelten Roboterfahrzeugs.

Kapitel 6 schließt die Arbeit mit einem Fazit ab.

2 Grundlagen

Die Simulation der beeindruckenden Leistungen biologischer Nervensysteme mit Hilfe technischer Lösungen, z.B. durch die Nutzung schneller Computersysteme, ist Teil des Forschungsbereichs der Informatik und der künstlichen Intelligenz. Die bisherigen Lösungsansätze zu dieser herausragenden Aufgabenstellung orientieren sich wesentlich an den Ergebnissen der Forschung zu Nervensystemen und insbesondere zum Zentralnervensystem [11]. Die biologischen Nervensysteme liefern mit Neuronen und Synapsen sowie den darin ablaufenden neuronalen Prozessen die strukturellen und funktionellen Vorlagen für den Aufbau künstlicher neuronaler Netze. Einer der Anwendungsbereiche, die in besonderem Maße von den Innovationen der Entwicklung von künstlichen neuronalen Netzen (KNN) profitieren, ist die Bilderkennung [6].

Das vorliegende Kapitel beschäftigt sich zunächst mit den Grundlagen der Bildverarbeitung, von der Auswertung einzelner Pixel über einfache und folgend komplexere Strukturen. Es zeigt beispielhaft die Arbeitsweise von Filtern in der Bildverarbeitung anhand sogenannter Faltungen (Convolutions) [18][29] und leitet über zur Einführung in den Aufbau neuronaler Netze, den Einsatz verschiedener Lernmethoden für diese Netze sowie zu den Elementen des Deep Learning in KNN. Den Abschluss dieses Kapitels bildet ein einfacher Machbarkeitsnachweis, mit dem die Leistungen der Bildverarbeitung in neuronalen Netzen exemplarisch beleuchtet werden.

2.1 Bildverarbeitung

Die Bildverarbeitung ist eine viel beachtete wissenschaftliche Disziplin mit vielen Verzweigungen in unterschiedlichste Forschungs- und Technikbereiche, in denen Bildverarbeitungstechniken angewendet werden. Besonders durch die rasante Entwicklung in der Bildsensorik und der Computertechnologie, mit stetig

steigender Speicherkapazität und Rechenleistung, bei gleichzeitiger Miniaturisierung, sowie durch die Bereitstellung kostengünstiger Computerleistungen im Internet und die weite Verbreitung von Kamerasystemen z.B. auf Smartphones, entsteht ein zunehmendes Interesse nach weiteren kreativen Möglichkeiten in der Bildverarbeitung. Bereits heute werden in weiten Bereichen Computer und andere Maschinen zur Auswertung und Darstellung von Bildern und Bildinhalten eingesetzt. Verfahren wie die Mustererkennung befähigen diese Maschinen neben exakten Eingaben auch dynamische und weniger konkrete Signale zu verarbeiten, die gewonnenen Informationen zu segmentieren, zu klassifizieren und über die weitere Verarbeitung in einem gewissen Umfang selbst zu entscheiden. Um die Leistung der Maschinen bei der Bildverarbeitung weiter zu verbessern und neue Anwendungsmöglichkeiten zu erschließen, wird unermüdlich versucht, die herausragenden Orientierungsfähigkeiten, die Menschen und Tiere durch Aufnahme und Verarbeitung visueller Signale erreichen, zu simulieren und nachzubilden. Für die Orientierung und Navigation im Raum dienen Menschen und Tieren neben weiteren Sinnen vor allem Signale in Form von Lichtreizen, über Photorezeptoren im Auge als einzelne Bildpunkte, die im visuellen Cortex weiterverarbeitet werden [5]. Als Entsprechung der Bildpunkte des Sehens von Menschen und Tieren dienen in der Informatik Rastergrafiken, die den Bildverarbeitungseinheiten zugeführt werden. Diese Rastergrafiken bieten Informationen zu Farbraum und Farbtiefe, die in der Form von Pixeln als Raster angeordnet sind.

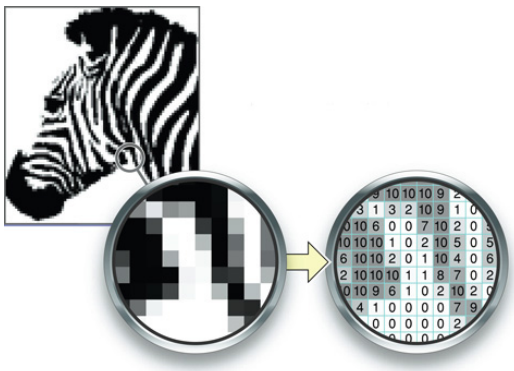

Abbildung 2.1: Bilder liegen digital als zweidimensionales Raster von Pixeln gemäß derer Intensitäten vor. Quelle: Entnommen aus *Performing Convolution Operations*, https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html, Abruf: April 2016.

Diese Pixelinformationen können dann ähnlich wie die Lichtreize, die im visuellen Cortex des menschlichen Gehirns verarbeitet werden, als Eingabe für KNN genutzt werden, mit dem Ziel, die Fähigkeiten des Gehirns nachzuahmen. Die neueren Entwicklungen in der Informatik und im Bereich der künstlichen Intelligenz schließen ein, dass nicht nur vorgegebene Muster erkannt werden, sondern dass Computer die Fähigkeit erhalten, aus den zugeführten Informationen selbständig Muster zu erlernen und bei der Objekterkennung anzuwenden. Grundlagen für den Erwerb dieser neuen Fähigkeiten, wie z.B. die Faltung, werden im Folgenden ausführlich behandelt.

2.1.1 Faltung

Eine der wichtigsten Operationen in der Signal- und Bildverarbeitung ist die sogenannte Faltung (engl. „Convolution“) [29]. Eine Faltung ist eine Anordnung von Pixeln, wie z. B. eine Kante, also der Wechsel der Intensitäten von benachbarten Pixeln entsprechend einem Muster.

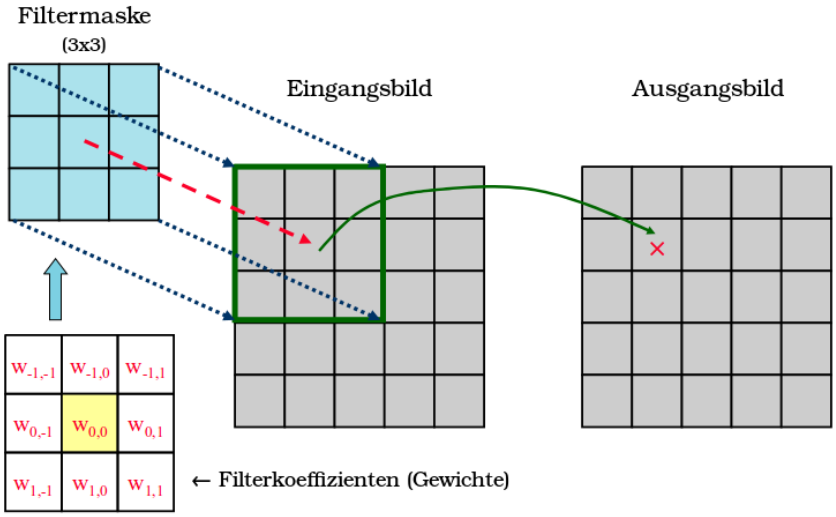

Abbildung 2.2: Das Prinzip der Convolution, bei der eine Filtermatrix über das Eingabebild gelegt wird, um die Werte im Ausgabebild zu berechnen. Quelle: In Anlehnung an *Vorlesung Bildverarbeitung und Algorithmen*, Prof. Dr. Wolfgang Konen [15].

Wie Abbildung 2.2 zeigt, wird mit Hilfe einer Filtermaske, hier mit einer Größe von 3x3 Pixeln, die gewichtete Summe der Pixel einer Bildnachbarschaft

berechnet und somit die Pixel des Ausgangsbildes gebildet [15]. Die Gewichte sind die Koeffizienten der Filtermaske. Zur Berechnung der Convolution werden die Ein- und Ausgangsbilder sowie die Filtermaske als Matrizen definiert. Um einen eindeutigen Mittelpunkt festlegen zu können, sind für Faltungsmatrizen nur ungerade Dimensionen (z. B. 3x3, 5x5 etc.) zulässig.

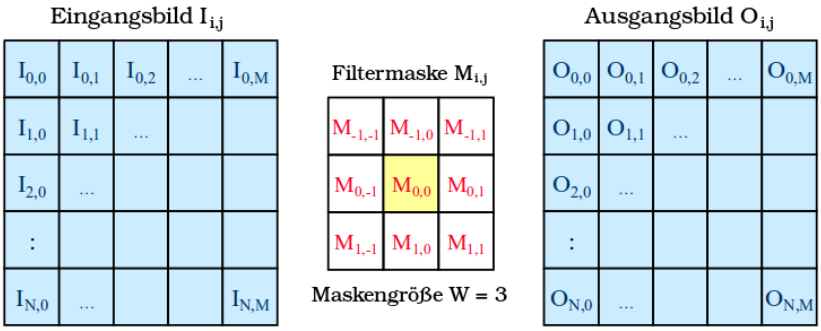

Abbildung 2.3: Die Berechnung der Elemente des Ausgangsbildes werden mit Hilfe der Formel 2.1 berechnet, indem die gewichtete Summe der Elemente des Eingangsbildes gebildet wird, über denen die Filtermaske liegt. Quelle: In Anlehnung an *Vorlesung Bildverarbeitung und Algorithmen*, Prof. Dr. Wolfgang Konen [15].

Die Berechnung der Convolution erfolgt bei gegebenen Matrizen wie in Abbildung 2.3 mit folgender Gleichung:

$$O[i,j] = \sum_{p=-W/2}^{W/2} \sum_{q=-W/2}^{W/2} I[i-p, j-q] * M[p,q] \tag{2.1}$$

Gesucht ist jeweils ein Pixel im Ausgangsbild $O[i,j]$. Die 3x3 Filtermaske wird auf das Eingangsbild I gelegt und die gewichtete Summe der benachbarten Pixel gebildet. Vereinfacht ergibt sich die Formel:

$$O = I * M \text{ (Faltung)} \tag{2.2}$$

Die Filtermaske wird jeweils um ein Pixel versetzt und so schrittweise über alle Bildpositionen gelegt. Dabei liefert die Faltung an einer Position im Eingangsbild den Bildwert der entsprechenden Position im Ausgangsbild. Da der Filter je nach Größe mindestens die äußerste Pixelreihe nicht erfassen kann,

entsteht dort ein Bildrand, dessen Pixel nicht wie in der Faltungsgleichung 2.1 angegeben, berechnet werden können. Es existieren verschiedene Methoden zur Randbehandlung wie Zero-/Grauwert-Padding, Last-Value oder einfach der Verzicht auf einen Filter im Randbereich [29]. Da die Randbereiche bei der Verwendung von Convolutions zur Bilderkennung meist keine relevanten Informationen enthalten, kann dort auf eine Randbehandlung verzichtet werden.

2.2 Neuronale Netze und Lernalgorithmen

Das maschinelle Lernen hat das Ziel, den funktionalen Zusammenhang zwischen Ein- und Ausgabedaten zu schätzen [29]. Da unser Gehirn die effizienteste Maschine des Lernens und Erkennens ist, liegt es nahe, die Strukturen des Gehirns nachzubilden. Die verschiedenen Ansätze, KNN zu bilden und lernen zu lassen, werden in diesem Kapitel vorgestellt.

2.2.1 Künstliche neuronale Netze

Künstliche neuronale Netze bestehen aus künstlichen Neuronen, die, dem biologischen Vorbild der Nervenzelle folgend, Eingaben gewichten und über eine Aktivierungsfunktion eine Ausgabe erzeugen [33]. Wie Abbildung 2.4 zeigt, bestehen KNN aus einer Eingabeschicht, ggf. weiteren Zwischenschichten und einer Ausgabeschicht.

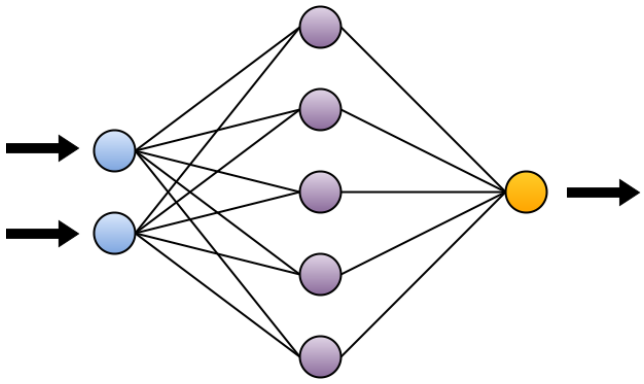

Abbildung 2.4: Vereinfachte Darstellung eines KNN mit der Eingabeschicht links, einer Zwischenschicht („Hidden Layers“) in der Mitte und der Ausgabeschicht rechts.

Es gibt unterschiedliche Strukturen von KNN. Sogenannte einschichtige feedforward-Netze, mehrschichtige feedforward-Netze und rekurrente Netze [26]. Die einschichtigen Netze, zu sehen in Abbildung 2.4, bestehen aus einer Ein- und einer Ausgabeschicht und leiten die Ausgaben nur in Verarbeitungsrichtung weiter („feedforward“). Mehrschichtige Netze, sogenannte Multilayer Perceptrons (MLP), bestehen neben Ein- und Ausgabeschicht aus weiteren Schichten, den sogennanten „Hidden Layers“ und können sowohl feedforward-gesteuert als auch rekurrent sein. Rekurrente Netze besitzen im Gegensatz zu den feedforward-gesteuerten Netzen auch rückgerichtete Kanten, sodass die Ausgaben eines Schrittes erneut als Eingaben für Neuronen in davor liegenden Schichten oder innerhalb der eigenen Schicht dienen können und somit die Dynamik des Netzes erhöhen [17].

2.2.2 Lernmethoden

Es existieren verschiedene Methoden für das Lernen in neuronalen Netzen, z. B. das überwachte Lernen (engl. „supervised learning“), das bestärkende Lernen (engl. „reinforcement learning“) und das unüberwachte Lernen (engl. „unsupervised learning“) [26]. Grundlage zur Klassifizierung in dieser Arbeit ist das Supervised Learning.

Supervised Learning

Supervised Learning basiert darauf, „gelabelte“ Datensätze zu trainieren, also Datensätze, bei denen die korrekten Ergebnisse bereits zugeordnet sind. Techniken des Supervised Learnings werden in feedforward und MLP Modellen angewendet. Das Lernen in überwachten Modellen wird auch Fehlerrückführungsalgorithmus (engl. „error backpropagation algorithm“) genannt. Das Netz wird trainiert, indem der Algorithmus auf Basis des Fehlersignals, also der Unterschied zwischen errechneten und gewünschten Ausgabewerten, die synaptischen Gewichte der Neuronen anpasst. Das synaptische Gewicht ist proportional zum Produkt des Fehlersignals und der Eingabeinstanz des synaptischen Gewichts. Auf Basis dieses Prinzips besteht jede Lernepoche aus zwei Schritten, einem Vorwärtsschritt (engl. „forward pass“) und einem Rückwärtsschritt (engl. „backward pass“) [26]:

1. Beim **Forward Pass** erhält das Netz eine Eingabe in Form eines Vektors, der das Netz Neuron für Neuron durchläuft und als Ausgabesignal in der Ausgabeschicht erscheint. Das Ausgabesignal hat die Form $y(n) = \varphi(v(n))$ mit $v(n)$ als lokales Feld eines Neurons mit $v(n) = \sum w(n)y(n)$. Der Ausgabewert $o(n)$ wird mit dem tatsächlichen Wert $d(n)$ verglichen und der Fehler $e(n)$ berechnet. Die synaptischen Gewichte des Netzes werden in diesem Schritt nicht verändert.

2. Im **Backward Pass** wird der Fehler, also der Ausgabewert des zu korrigierenden Layers, an die davor liegenden Layer zurückgeführt. Damit wird der lokale Gradient für jedes Neuron in jedem Layer berechnet, sodass die synaptischen Gewichte des Netzes gemäß der Delta-Regel verändert werden:

$$\Delta w(n) = \eta * \delta(n) * y(n)$$

Dieser rekursive Vorgang wird so lange wiederholt, bis das Netz konvergiert ist.

Reinforcement Learning

Das Reinforcement Learning ist eine Lernmethode, die auf einen Agenten in einer dynamischen Umgebung angewendet wird, der verschiede Aktionen zum Erreichen seines Ziel durchführen kann. So werden nicht wie bei anderen Algorithmen eine Problemstellung und eine Reihe durchzuführender Aktionen definiert. Stattdessen muss das Netz diejenigen Aktionen bestimmen, die den besten Erfolg versprechen [30]. Um den Erfolg für den Agenten messbar zu machen, erhält er für ausgeführte Aktionen eine Belohnung (engl. „Reward“). Die Belohnung erfolgt einerseits bei einem Wechsel in einen anderen Zustand, andererseits wird ein gesamt zu erwartender Gewinn errechnet. Der Agent versucht dann durch jede Aktion eine hohe Belohnung zu erhalten und den gesamten Gewinn zu maximieren. Die Vorteile von Reinforcement Learning in dynamischen Systemen machen es besonders für den Einsatz in Spielen interessant. So ist eine Kombination aus Supervised und Reinforcement Learning Grundlage für das bekannte AlphaGo, ein Agent, der das Brettspiel Go beherrscht und sogar internationale Top-Spieler besiegen konnte [27].

Unsupervised Learning

Der Erfolg von Systemen des maschinellen Lernens erfordert häufig sehr große Datensätze, die bereits gelabelt sind und deren Gewinnung einen großen Aufwand darstellt. Unsupervised Learning ist eine Lernmethode für Eingaben, die Muster enthalten, welche nicht oder größtenteils nicht klassifiziert sind, was den Lernprozess sehr erschwert [13]. Es ist möglich, auf gänzlich unbewertete Datensätze, z. B. große Mengen an Daten aus dem Internet, zurückzugreifen. Im Unsupervised Learning können dann beispielsweise Strukturen wie Ecken und Kanten oder auch Strukturen aus bestimmten Objektklassen (z. B. Autoreifen oder Teile eines Gesichts) unter Zuhilfenahme eines kleinen gelabelten Datensatzes, hilfreich für die Objekterkennung sein [20].

2.3 Deep Learning

Deep Learning ist ein Ansatz im maschinellen Lernen, um Wissen aus Erfahrung zu gewinnen und das zu lösende Problem mit Hilfe einer Hierarchie von Lösungskonzepten zu verstehen, die jeweils durch einfachere Teillösungen definiert sind [10]. Durch diesen Ansatz kann die Notwendigkeit der Definition jeglichen für den Computer zur Problemlösung notwendigen Wissens durch den Programmierer vermieden werden. Die Hierarchie von Lösungskonzepten macht es dem Computer möglich, komplexe Lösungsansätze aus einfacheren Ansätzen zu bilden. Ein Graph, der diese Konzepte aufeinander abbildet, besteht aus vielen Schichten und ist somit „tief“. Deshalb spricht man bei diesem Ansatz der künstlichen Intelligenz von „Deep Learning“. Im Folgenden werden Techniken des Deep Learning und die Vorgehensweise der Bilderkennung in tiefen Netzen erläutert. Es werden mögliche Schwachstellen sowie deren Lösungsmöglichkeiten aufgezeigt und ein Machbarkeitsnachweis für die Bilderkennung durchgeführt.

2.3.1 Deep Convolutional Neural Networks

Seit ihrer Einführung durch LeCun et al. [19] in den frühen 90er Jahren des letzten Jahrhunderts zeigen Multilayer-Perceptrons hohe Leistungsfähigkeit in

Aufgaben wie der Klassifizierung handgeschriebener Ziffern und der Gesichtserkennung [34]. Erst durch die Entwicklung von Convolutional Neural Networks (CNN) und den Einsatz von Grafikprozessoren (GPGPU - General purpose computing on graphics processing units) zur Parallelisierung der Berechnungen im maschinellen Lernen können auch in deutlich komplexeren Aufgabenfeldern herausragende Ergebnisse erzielt werden [16]. CNN sind an den Aufbau des visuellen Cortex von Tieren angelehnt, in dem u. a. komplexe Anordnungen von Neuronen auf Kanten-ähnliche Muster reagieren. Während MLP aus vollvernetzten Schichten („Fully Connected Layers“) bestehen, ist die Anordnung der Kanten in CNN von den definierten Convolutions (Kapitel 2.1.1) abhängig. Die Kombination aus mehreren Convolutional und Fully Connected Layern zu einem Deep Convolutional Neural Network hat sich in der Erkennung und Klassifizierung von Bildern bewährt [16].

2.3.2 Bilderkennung mit Convolutional Neural Networks

Die Filter für die Convolutions werden auf die Eingabe angewendet und berechnen die Ausgabe, sodass lokale Verbindungen geschaffen werden, in denen jede erfasste Region der Eingabe mit einem Neuron in der Ausgabe verbunden ist. Diese Verbindungen und der Ablauf der Bilderkennung mit CNN werden anhand Abbildung 2.5 gezeigt.

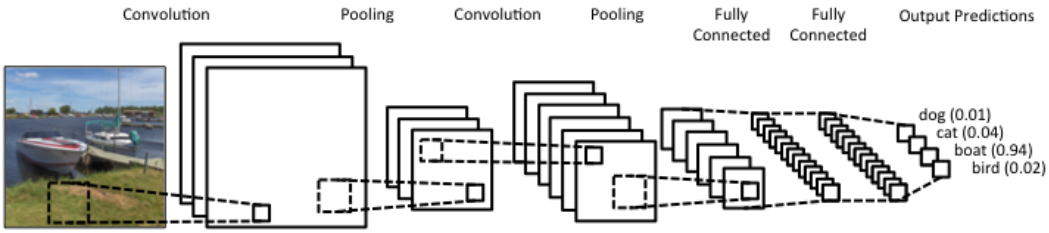

Abbildung 2.5: CNN Modelle lernen über eine hirarchische Struktur von Layern und Transformationen komplexe Bildmuster aus großen Mengen von Daten. Quelle: Entnommen aus *Convolutional Neural Networks in practice*, https://www.clarifai.com/technology, Abruf: Mai 2016.

Jeder Layer wendet auf seine Eingabedaten verschiedene, meist hunderte oder tausende Filter an und kombiniert deren Ergebnisse. Die Werte der Filter werden während des Trainings von Layer zu Layer mathematisch transformiert

und ergeben als Endprodukt der Ausgabeschicht ein näherungsweises Muster der Eingabedaten. Auf diese Weise lernt ein Netz für die Klassifizierung von Bildern beispielsweise einfache Kanten aus den Pixeln der Eingabeschicht zu erkennen. Im nächsten Layer werden mehrere Kanten als einfache Formen erkannt und diese Formen dann wiederum dazu verwendet, übergeordnete Features zu erkennen. Werden die Formen aus den einzelnen Schichten extrahiert, erhält man Bilder wie in Abbildung 2.6.

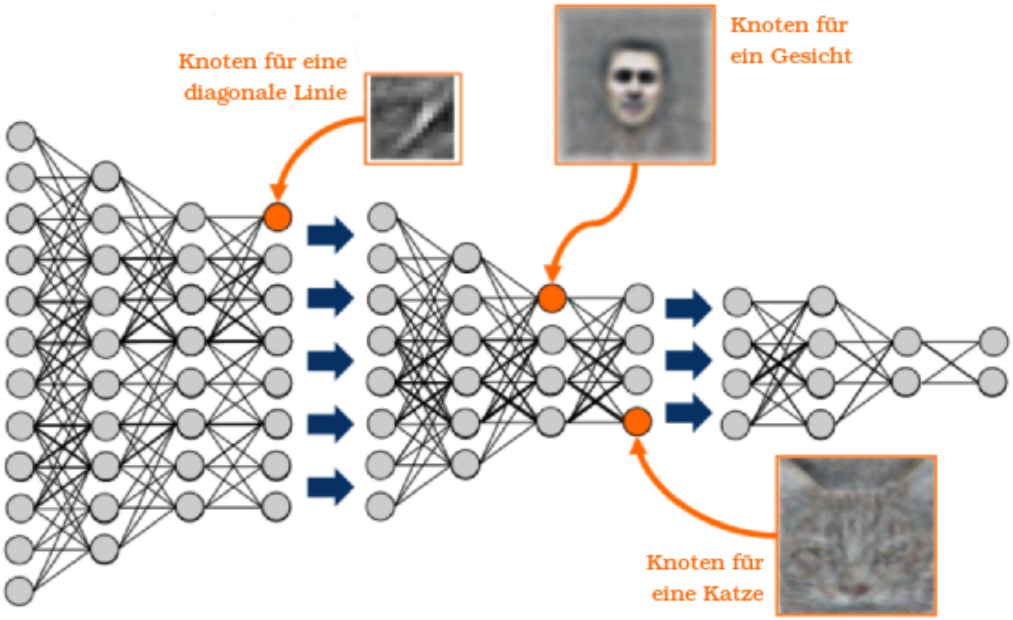

Abbildung 2.6: Die Abbildung zeigt, dass verschiedene Neuronen bei Erkennung verschiedener erlernter Features, von einfachen Formen bis hin zu komplexen Strukturen, ausgelöst werden. Quelle: Angelehnt an *Deep Convolutional Neural Networks for Smile Recognition* [9].

2.3.3 Overfitting

Beim Trainieren von neuronalen Netzen mit einer sehr großen Zahl an Parametern und damit einhergehender hoher Komplexität der Netze, kann leicht das Problem von Über- oder Unteranpassung (engl. „Over- / Underfitting“) auftreten, welche sowohl durch einen hohen Bias, als auch durch eine hohe Varianz entstehen können [14][28]. Ein hoher Bias löst Underfitting, eine hohe Varianz bei niedrigem Bias Overfitting aus.

Bei der aus der Statistik bekannten Regression, soll z. B. bei gegebenen Datenpaaren $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ eine Funktion $f(x)$ gefunden werden, die sich gut für die Abbildung auf die zugrunde liegenden Daten eignet, also der Fehler $|f(x_i) - y_i|$ für ein Beispielpaar (x_i, y_i) sehr klein ist. Auch wenn sich die Regression von der Klassifizierung unterscheidet, eignet sie sich gut um Over- und Underfitting zu beschreiben.

Wird $f(x)$ als Polynom n-ten Grades gewählt, erhält man die folgende Funktion:

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1} + a_n x^n \tag{2.3}$$

Mit zunehmendem Grad steigt die Komplexität des Polynoms und man erhält für die Abbildung auf den Daten beispielsweise folgende Graphen:

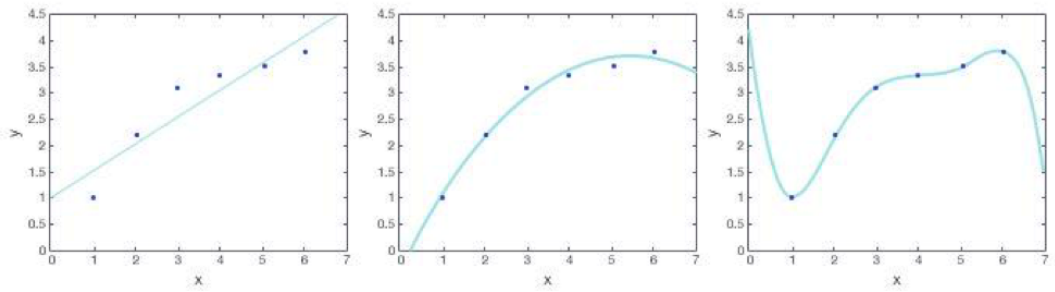

Abbildung 2.7: Die Graphen zeigen ein Polynom 1. Grades [links], ein Polynom 2. Grades [Mitte] und ein Polynom 5. Grades [rechts]. Quelle: Entnommen aus http://blog.fliptop.com/blog/2015/03/02/bias-variance-and-overfitting-machine-learning-overview/, Abruf: Mai 2016.

Der Graph rechts wird perfekt auf den Werten abgebildet, da jedes Polynom mit einem Grad von mindestens $n - 1$ auf einen Datensatz der Größe n abgebildet werden kann, verläuft allerdings sehr sprunghaft und scheint auf der linken Seite nicht dem Verlauf des Musters der Werte zu folgen. Man spricht hier von Overfitting, da das Netz also zu stark trainiert hat, um jeden Wert zu treffen. Diese Funktion hat einen niedrigen Bias aber eine hohe Varianz. Wird in Zukunft eine Voraussage hinsichtlich des Zusammenhangs von x und y getroffen, ist es unwahrscheinlich, dass diese Funktion ein akkurates Ergebnis liefert, besonders wenn $x < 1$ ist. Das Polynom 1. Grades auf der linken

Seite hat einen hohen Bias, da dass Modell unflexibel ist. Unabhängig davon, wie stark das Muster der Daten gekrümmt ist, kann immer nur die Form einer Geraden angenommen werden, sodass man hier von Underfitting spricht. Das mittlere Polynom repräsentiert eine gute Anpassung auf die Daten und es ist kein Over- oder Underfitting erkennbar.

Overfitting ist üblicherweise ein Hauptproblem beim Training neuronaler Netze und kommt deutlich häufiger vor als Underfitting. Durch Overfitting reagiert das Modell zu stark auf Rauschen, wie z.B. weitere Objekte, die sich neben dem zu erkennenden Objekt auf dem Bild befinden, statt vorgesehene Muster zu trainieren. Je komplexer das ausgewählte Modell, desto niedriger wird die Fehlerrate des Trainingsets sein und bei hoher Komplexität null erreichen.

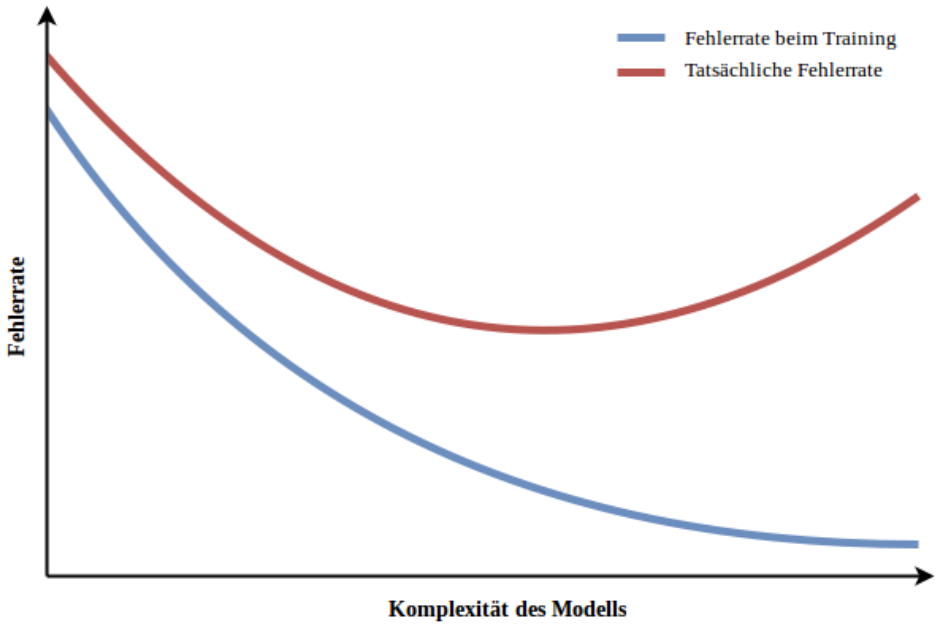

Abbildung 2.8: Die Abbildung zeigt den Verlauf der Fehlerrate beim Training eines Netzes in blau und die tatsächliche Fehlerrate in rot bei Overfitting.

Um Overfitting zu verhindern, können verschiedene Methoden angewendet werden. Die einfachste Methode ist die Bereitstellung zusätzlich verfügbarer Trainingsdaten. Liegen keine weiteren Daten vor, kann die Komplexität der Netze verringert werden, wie z.B. durch die Reduzierung der Anzahl an Hidden Layer oder durch die Anwendung von Dropout [28].

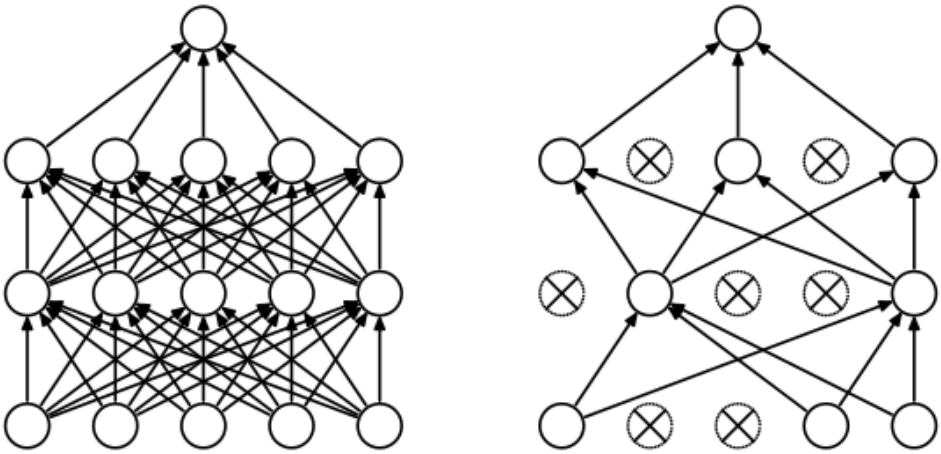

Abbildung 2.9: Abgebildet ist ein KNN ohne [links] und mit Dropout [rechts]. Quelle: Entnommen aus *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* [28].

Dropout beruht, wie in Abbildung 2.9 dargestellt, auf einem stochastischen „Ausschalten“ von Neuronen und ihren Verbindungen während der Trainingsphase, sodass sie nicht mehr mit dem Netz interagieren können. Es erfolgt eine Verkleinerung der Netze und somit eine Reduzierung der Komplexität und eine gegenseitige Anpassung der Teilnetze wird verhindert.

2.3.4 Machbarkeitsnachweis künstlicher neuronaler Netze

Vor einer im Rahmen dieser Arbeit geplanten praktischen Umsetzung einer der zuvor beschriebenen Lösungsstrategien, soll mit Hilfe eines ausgewählten, aktuellen Frameworks für Softwareentwicklungen in KNN und anhand eines einfachen Prototypen ein Machbarkeitsnachweis durchgeführt werden. Dieser Prototyp verwendet als grundlegende Daten den MNIST-Datensatz.[1] Er besteht aus einer Sammlung von 60.000 Graustufenbildern mit handgeschriebenen Ziffern, die von Mitarbeitern des Statistischen Bundesamts der USA und von amerikanischen Oberschülern stammen [22]. Sie wurden auf eine einheitliche Größe von 28x28 Pixeln aus größeren Schwarzweißbildern gebracht und zentriert. Dieser Datensatz wird zur Evaluierung vieler neuer Modelle für das Training von KNN verwendet, da er sich aufgrund seiner Aufteilung, Größe und Normalisierung der Bilder ideal eignet, um die Korrektheit eines Modells zu

[1] Mixed National Institute of Standards and Technology Database, siehe: http://yann.lecun.com/exdb/mnist/, Abruf: April 2016.

überprüfen sowie erforderliche Feinjustierungen für eine bessere Erkennungsleistung und eine geringere Fehlerrate vorzunehmen, bevor die eigentlich interessierenden Bilddaten mit neuen Modellen trainiert werden.

Die Entwicklung des hier verwendeten Prototyps zur Erkennung von handgeschriebenen Ziffern erfolgt in der Programmiersprache Python, die neben C++ TensorFlow nutzen kann. TensorFlow ist eine Open-Source Bibliothek für numerische Berechnungen mit Hilfe von Datenflussgraphen [1]. Die Knoten im Graphen repräsentieren dabei mathematische Operationen, wobei die Kanten des Graphen als multidimensionale Daten-Arrays, sogenannte Tensoren, zur Kommunikation zwischen den Knoten fungieren.

Um effiziente Berechnungen in Python zu implementieren, werden üblicherweise - wie auch in der vorliegenden Arbeit - weitere Bibliotheken wie NumPy verwendet, das rechenintensive Operationen wie Matrizenmultiplikation unter der Verwendung anderer, performanterer Programmiersprachen außerhalb von Python durchführen kann. Beim häufigen Wechsel zurück zu Python entsteht unglücklicherweise eine größere Menge Overhead, was besonders bei der Berechnung auf GPUs (Graphics Processing Unit) Geschwindigkeitsprobleme beim Datentransfer verursacht. Zur Behebung dieses Problems bietet TensorFlow die Möglichkeit, einen Graphen mit interagierenden Operationen zu beschreiben, der außerhalb von Python ausgeführt wird, statt einzelne rechenintensive Operationen unabhängig von Python zu berechnen.

Zur Berechnung der Wahrscheinlichkeiten für die zu bewertenden Ziffern eignet sich u. a. die sogenannte multinomiale logistische Regression (engl. „Softmax“), die zur Schätzung von Gruppenzugehörigkeiten bzw. einer entsprechenden Wahrscheinlichkeit hierfür dient [25]. Zunächst werden die Klassenzugehörigkeitsfaktoren der Eingaben (in den Klassen 0-9 für jede Ziffer) aufsummiert. In einem zweiten Schritt werden diese Zugehörigkeitsfaktoren in Wahrscheinlichkeiten umgewandelt. Um die Zugehörigkeit eines Bildes zu einer Klasse zu ermitteln, wird die gewichtete Summe der Pixelintensitäten gebildet. Ist die Gewichtung bei einer hohen Pixelintensität negativ, führt dies dazu, dass das Bild nicht in die entsprechende Klasse eingeteilt wird. Eine positive Gewichtung spricht dafür, dass das Bild der entsprechenden Klasse zugehörig ist.

Zur Steuerung von Einflüssen, die unabhängiger vom Input sind, sollte zusätzlich ein Bias eingeführt werden. Für die Klassenzugehörigkeit ϑ einer Klasse i bei einem Input x ergibt sich so folgende Gleichung:

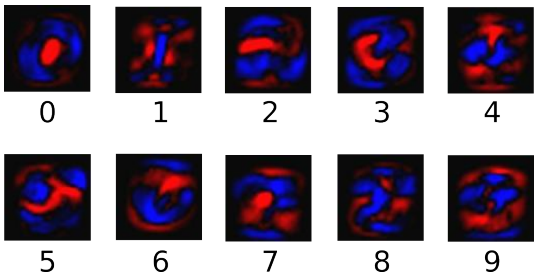

Abbildung 2.10: Die Abbildung zeigt die erlernten Gewichtungen für jede Klasse von 0 bis 9. Negative Gewichtungen werden rot, positive blau dargestellt. Quelle: Entnommen aus *TensorFlow* [1].

$$\vartheta_i = \sum_j W_{i,j} x_j + b_i \tag{2.4}$$

Hierbei steht W_i für die Gewichte, b_i für den Bias der Klasse i und j für den Index der Summe der Pixel des Input-Bildes x.

Unter Verwendung von Softmax werden dann die Klassenzugehörigkeiten ϑ in Wahrscheinlichkeiten y umgewandelt:

$$y = \text{softmax}(\vartheta) \tag{2.5}$$

Softmax liefert eine Aktivierungsfunktion, die den Output in eine Wahrscheinlichkeitsverteilung über 10 Klassen formt. Die Softmax-Funktion wird wie folgt definiert:

$$\text{softmax}(x) = \text{normalize}(\exp(x)) \tag{2.6}$$

Im Detail erhält man die folgende Funktion:

$$\text{softmax}(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{2.7}$$

Die Exponentierung bedeutet in diesem Zusammenhang, dass eine zusätzliche Einheit der Klassifizierung die Gewichtung einer Hypothese multiplikativ erhöht. Eine Klassifizierung weniger führt andererseits dazu, dass eine Hypothese einen Anteil seines ursprünglichen Gewichts zurückerhält. Eine Hypothese

kann nie null oder eine negative Gewichtung erhalten. Danach werden die Gewichtungen durch Softmax normalisiert, sodass sie in Summe eins und somit eine gültige Wahrscheinlichkeitsverteilung bilden.

Es ergibt sich also kompakt die folgende Gleichung:

$$y = \text{softmax}(Wx + b) \tag{2.8}$$

Das Modell zur Klassifizierung des MNIST-Datensatzes kann jetzt unter Verwendung von Softmax trainiert werden. Zur Bewertung des Modells wird häufig Kreuzentropie als Kostenfunktion angewendet. Diese Kostenfunktion wird wie folgt beschrieben:

$$H_{y'}(y) = -\sum_i y'_i \log(y_i) \tag{2.9}$$

In dieser Funktion ist y die vorausgesagte Wahrscheinlichkeitsverteilung und y' ist die tatsächliche Verteilung, entsprechend der Ziffer auf dem jeweiligen Bild. Die Kreuzentropie führt also eine Messung aus, wie ineffizient die Voraussagungen des Modells sind.

Zur Evaluierung der Lernergebnisse steht neben der textuellen Ausgabe der Genauigkeit und der Kosten (engl. „cost“ oder „loss“) nach jedem n-ten Trainingsschritt das in TensorFlow enthaltene TensorBoard zur Verfügung. Dieses visualisiert Daten, die zuvor während der Ausführung aufgezeichnet wurden und darüber hinaus den Graphen des neuronalen Netzes.

Die Durchführung des Trainings und die Evaluierung der Ergebnisse werden in Kapitel 4 beschrieben.

3 Auswahl der Convolutional Neural Networks

Der Machbarkeitsnachweis in Kapitel 2.3.4 dient zur Einführung in Deep Convolutional Neural Networks unter Einbeziehung von Softmax und Kreuzentropie ohne die Verwendung spezieller Modelle. Im vorliegenden Kapitel soll nun auf bekannte Modelle und ihre Besonderheiten eingegangen und diese bezüglich ihrer Eignung für die Verwendung zur Bilderkennung in dynamischen Systemen geprüft werden. Alle Modelle haben das Ziel die Erkennung von Objekten, beispielsweise mit Hilfe größerer Datensätze, Methoden zur Vermeidung von Overfitting sowie effizienterer Algorithmen zu verbessern. Ein wichtiger Ansatz der die Erkennung deutlich verbessern kann, ist die Verwendung von CNN, die in Kapitel 2.3.1 beschrieben sind.

Es existieren derzeit eine Reihe von besonders erfolgreichen Modellen von CNN, die sehr gute Ergebnisse in den verschiedenen internationalen Wettbewerben für Bilderkennung und -klassifizierung erreichen. Zwei der bestplatzierten Architekturen der letzten Zeit wurden für diese Arbeit evaluiert und werden im Folgenden beschrieben.

Die folgende Tabelle zeigt die Erstplatzierten des ImageNet LSVRC in den Jahren 2012-2014 in der Aufgabe „Object detection with additional training data“ [31].

Tabelle 3.1: Gewinner des ILSVRC 2012-2014

Team	Jahr	Top-5-Fehlerrate
SuperVision	2012	15.3%
Clarifai	2013	11.2%
GoogLeNet	2014	6.67%

Das Team SuperVision um Alex Krizhevsky verwendet ein Modell namens AlexNet [16]. Das Gewinner-Team Clarifai 2013 ist ein Unternehmen im Bereich künstliche Intelligenz, die eine eigene API und Code in verschiedenen Programmiersprachen und für verschiedene Plattformen (wie Android, iOS u.a.) für Deep Learning zur Verfügung stellen.[2] Das Team GoogLeNet um Yann LuCun setzte in 2014 das ebenfalls oben beschriebene Modell Inception ein, das die Top-5-Fehlerrate im Vergleich zum Vorjahr nahezu halbierte. Da die Modelle AlexNet und Inception gut dokumentiert sind, der Sourcecode offen ist und jeweils TensorFlow als API genutzt wird, wurden diese beiden Modelle auf der in Kapitel 4.1 beschriebenen Hardware implementiert und getestet. Das Modell von Clarifai fand keine Berücksichtigung, da wie oben erwähnt, ein kommerzieller Hintergrund besteht und eine eigenentwickelte, teilweise lizenzpflichtige API verwendet wird.

3.1 SuperVision AlexNet

Die von Krizhevsky et al. vorgestellte Architektur AlexNet war die erfolgreichste Architektur der ImageNet Large-Scale Visual Recognition Challenge 2010 (ILSVRC10) und erzielte eine Top-5-Fehlerrate von 17.0% und eine Top-1-Fehlerrate von 37.5%. Auch zwei Jahre später konnte eine verbesserte Architektur von Krizhevsky et al. mit einer Top-5-Fehlerrate von 15.3%, verglichen mit dem zweitplatzierten Modell mit 26.2%, den ersten Platz belegen [16]. Aufgrund der guten Ergebnisse und der freien Verfügbarkeit an Dokumentation und Sourcecode, wurde dieses Modell zur Überprüfung einer Eignung für die Bilderkennung und Navigation des Roboterfahrzeugs ausgewählt.

Von den Autoren wurde nicht die übliche Vorgehensweise zahlreicher früherer Modelle gewählt, die für den Output der Neuronen die Funktionen $f(x) = tanh(x)$ oder $f(x) = (1 + e^{-x})^{-1}$ verwenden. Stattdessen wurde hinsichtlich der Trainingszeit das deutlich schnellere $f(x) = max(0, x)$ gewählt. Dieser Ansatz wird von den Autoren, wie auch schon zuvor von Nair und Hinton [23], Rectified Linear Units (ReLUs) genannt und beschleunigt CNNs um ein Vielfaches im Vergleich zu der Verwendung von $tanh$.

[2] Clarifai API, siehe: https://github.com/Clarifai, Abruf: April 2016.

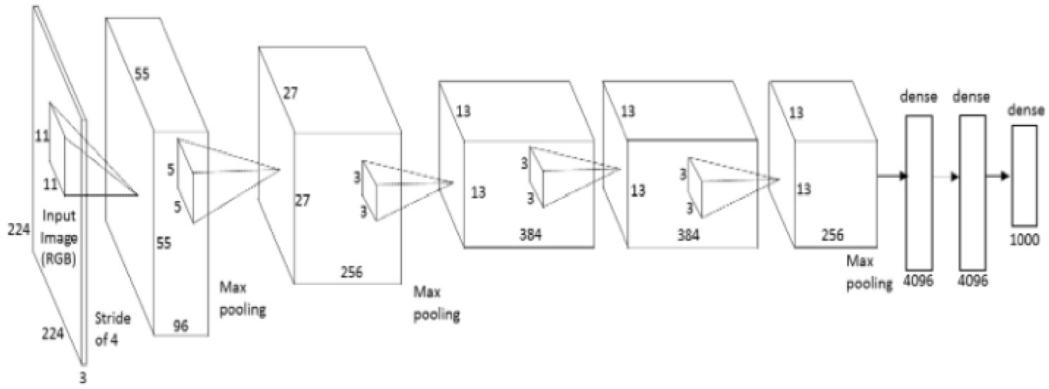

Abbildung 3.1: Die Abbildung zeigt die einzelnen Layer des AlexNet-Modells. Quelle: Entnommen aus *ImageNet Classification with Deep Convolutional Neural Networks* [16].

Wie Krizhevsky et al. in ihrer Arbeit beschreiben, besteht AlexNet aus 5 Convolutional Layern und 3 Fully Connected Layern [16]. Wie Abbildung 3.1 zeigt, sind die 50176 (224*224) Neuronen der Eingabeschicht mit 2 Convolutional Layern verbunden, auf die jeweils „Max Pooling“ angewendet wird. Beim Max Pooling werden die Eingaben in nicht-überlappende Bereiche segmentiert und jeweils das Maximum dieser ausgegeben. Darauf folgen weitere 3 Convolutional Layer, erneutes Max Pooling und 3 voll verbundene Schichten. Tabelle 3.2 zeigt die Layer mit ihren Daten.

Tabelle 3.2: Aufbau AlexNet

Typ	Filter	Größe
Conv	11x11	96
Pool	3x3	256
Conv	5x5	256
Pool	3x3	384
Conv	3x3	384
Conv	3x3	384
Conv	3x3	256
Pool	3x3	4096
Vollverb.	-	4096
Vollverb.	-	4096
Vollverb.	-	1000

3.2 GoogLeNet Inception

Ein weiteres Modell, das in der ILSVRC14 herausragende Ergebnisse erzielt hat, ist das sogenannte Inception, vorgestellt von Szegedy et al. [31]. Die Architektur zeichnet sich insbesondere dadurch aus, dass bei einer deutlich gesteigerten Tiefe und Breite des neuronalen Netzes der Berechnungsaufwand konstant bleibt.

Die architektonischen Entscheidungen zur Optimierung des Netzes basieren u. a. auf der Hebb'schen Lernregel, die besagt, dass je häufiger ein Neuron A gleichzeitig mit einem Neuron B aktiv ist, desto eher werden diese beiden Neurone im Netz miteinander interagieren [12].

Inzwischen wurde das Modell weiter verfeinert und liegt in Version 3 als Inception-V3 vor [32]. Die Autoren Szegedy et al. versprechen durch die weiteren Anpassungen eine Top-5-Fehlerrate von 3,5%, was eine Verbesserung von 25% gegenüber den besten veröffentlichten Ergebnissen darstellt und die beim ILSVRC2014 erzielte Fehlerrate nahezu halbiert. Abbildung 3.2 zeigt die verbesserte Architektur von Inception-V3.

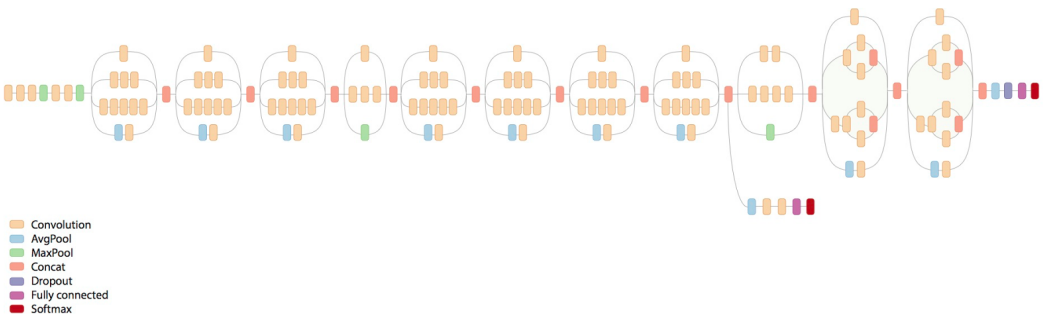

Abbildung 3.2: Die Abbildung zeigt das von Szegedy et al. weiter verbesserte Modell Inception-V3. Quelle: Entnommen aus *Rethinking the Inception Architecture for Computer Vision* [32].

Der Hauptgedanke von Szegedy et al. bei Inception ist die Reduzierung der Größe der Convolutional Layer, indem diese durch kleine CNN, sogenannte „Inception Modules“, ersetzt werden.

Convolutional Layer mit größer dimensionierten Filtern (wie 5x5 oder 7x7) neigen dazu, unverhältnismäßig rechenintensiv zu sein. So ist beispielsweise ein 5x5 Convolutional Layer $25/9 = 2{,}78$ mal so rechenintensiv wie ein 3x3 Convolutional Layer mit der gleichen Anzahl an Filtern. Eine einfache Verringerung der Größe der Convolutinal Layer würde eine verminderte Erkennungsleistung mit sich führen. Statt also die Größe der Convolutional Layer zu verringern, werden sie durch Netze mit eigenen kleineren Convolutional Layern ersetzt. Die Ein- und Ausgabeschichten dieser Netze haben die gleiche Größe wie die der ursprünglichen Convolutional Layer. Da diese Netze wiederum parallelisiert werden können und so Rechenzeit gespart werden kann, ergibt sich trotz der größeren Tiefe dieser Netze eine gesteigerte Gesamtperformanz.

Tabelle 3.3: Aufbau Inception

Typ	**Filter**	**Größe**
Conv	3x3	32
Conv	3x3	32
Conv	3x3	32
Pool	3x3	64
Conv	3x3	64
Conv	3x3	80
Conv	3x3	192
3x Inception	-	288
5x Inception	-	768
2x Inception	-	1280
Pool	8x8	2048
Vollverb.	-	2048
Vollverb.	-	1000

Das Inception-V3-Modell hat insgesamt eine Tiefe von 42 Layern, bestehend aus 6 Convolutional Layern, 10 Inception Modulen und 2 vollverbundenen Schichten (siehe Tabelle 3.3). Die ersten drei Inception Module bestehen aus 7, die nächsten fünf Inception Module aus 10 und die letzen zwei Inception Module aus 9 Convolutional Layern. Wie bei AlexNet hat die letzte Schicht 1000 Ausgaben für die 1000 Klassen von ImageNet.

4 Training und Optimierung

Nachdem die Auswahl des für das Training bevorzugten CNN-Modells erfolgte, befasst sich das folgende Kapitel mit dem Lernprozess, der aus Gründen der Performanz nicht auf dem Raspberry Pi durchgeführt werden kann, sondern auf einer leistungsstärkeren Plattform stattfinden muss. Vor dem eigentlichen Training werden die Trainungsumgebung konfiguriert sowie die Trainings- und Validierungsdaten vorbereitet. Nach dem Ende eines jeden Testdurchlaufs werden die Ergebnisse analysiert und dokumentiert.

4.1 Systemkonfiguration

Der Aufbau und das Training von tiefen neuronalen Netzen ist durch die große Zahl an Berechnungen sehr ressourcenlastig. Daher ist es erforderlich, entsprechend performante Hardware einzusetzen. Diese Berechnungen, die in den Forward- und Backwardsteps für jeden Knoten im Graphen ablaufen, sind hoch-parallelisierbar, weshalb sich der Einsatz einer leistungsfähigen GPU (Graphical Processing Unit) in besonderem Maße eignet. Eine GPU mit ihren mehreren hundert Rechenkernen hat bei der Verarbeitung von parallelen Rechenaufgaben einen deutlichen Vorteil gegenüber einer aktuellen CPU mit ihren 4 - 8 Kernen [7]. Das Training der neuronalen Netze wurde auf einem AMD FX-6100 Prozessor mit 6 logischen Kernen und einem Prozessortakt von 3.30 Ghz durchgeführt. Der Rechner verfügt über 8 GB Arbeitsspeicher und war zu Beginn mit einer AMD Radeon HD 6570 Grafikkarte ausgestattet. Die anfängliche Recherche zu den Hardwarevoraussetzungen für Deep Learning ergab, dass die vorhandene Grafikkarte für den Einsatz von CNN, die vergleichsweise hohen Grafik-Arbeitsspeicher benötigen, nicht geeignet ist. Außerdem ergab die Prüfung der Voraussetzungen, dass vorrangig Grafikkarten von NVIDIA für Deep Learning mit CNN verwendet werden, da hierfür ein gut entwickeltes Framework vom Hersteller zur Verfügung gestellt wird. Da selbst beim Einsatz

einer leistungsfähigen GPU mit langen Trainingslaufzeiten für ein neuronales Netz zu rechnen ist, wurde die Anschaffung einer NVIDIA GeForce GTX 750 TI Grafikkarte mit 2GB DDR4-RAM zum Preis von 140 Euro entschieden. Erste Tests bestätigten, dass die neue GPU die Laufzeit der Tests um das 4-fache gegenüber der CPU senken konnte.

4.2 Beschaffung und Aufbereitung der Trainingsdaten

Für das Training des Machbarkeitsnachweises sowie der zu testenden Modelle, wurden verschiedene Datensätze aus unterschiedlichen Quellen geladen. Beim Machbarkeitsnachweis handelt es sich, wie zuvor beschrieben, um den MNIST-Datensatz, zu beziehen von der Internetseite von Yann LeCun.[3] Dieser Datensatz muss vor den Trainingsläufen nicht erst bereinigt werden und ist sofort einsetzbar.

Neben dem MNIST-Datensatz steht ein weiterer Standard-Datensatz zur Verfügung, der sehr häufig zum Testen von neuen Modellen herangezogen wird. Dieser Datensatz wird von der University of Oxford als „17 Category Flower Dataset“[4] bereitgestellt. Wie der Name bereits vermuten läßt, handelt es sich dabei um Bilder 17 verschiedener Blumenarten. Zudem stehen im Internet jeweils Beispiel-Sourcecodes für die Modelle AlexNet und Inception zur Verfügung[5], mit denen dieser Standard-Datensatz geladen und trainiert werden kann. Die University of Oxford stellt neben dem Blumen-Datensatz, der auch mit 102 Klassen vorliegt, noch 28 weitere gelabelte Datensätze zur Verfügung.[4] Die umfangreichste Quelle für gelabelte Datensätze ist jedoch ImageNet[6], dessen Datensätze auch in der in Kapitel 3 beschriebenen jährlich ausgetragenen ImageNet Large-Scale Visual Recognition Challenge als Datenbasis dienen.

Im Rahmen dieser Arbeit wurde, inspiriert durch Roboterfußball, ein klassischer, schwarz-weißer Fußball als zu bewertendes Objekt gewählt. Für einen ersten Test wurde deshalb ein Datensatz zum Thema Fußball von ImageNet geladen. Aus diesem Datensatz wurden zunächst jene Bilder entfernt, die

[3] MNIST-Datensatz, siehe: http://yann.lecun.com/exdb/mnist/, Abruf: April 2016.
[4] Oxford Bilddatensätze, siehe: http://robots.ox.ac.uk/~vgg/data/, Abruf: April 2016.
[5] TensorFlow, siehe: https://github.com/tensorflow/, Abruf: April 2016.
[6] Bild-Datenbank ImageNet, siehe: http://www.image-net.org, Abruf: April 2016.

Fußball-Szenen mit Spielern oder optisch zu unterschiedliche Bälle zeigen. Als Vergleichsdatensatz wurden von ImageNet Bilder ausgewählt, die keinen Fußball oder dem Fußball sehr ähnliche Objekte enthalten, damit das neuronale Netz effektiv lernt zu unterscheiden, ob sich in einem Bild ein Fußball befindet oder nicht. Zudem wurde mit Hilfe der Kamera des Raspberry Pi ein eigener Datensatz erstellt, indem 2000 Bilder (800 Bilder mit und 1200 Bilder ohne Fußball) mit einer Auflösung von 640x480 in gleichen Räumlichkeiten aufgenommen wurden. Die Trainingsergebnisse mit diesen unterschiedlichen Datensätzen werden im nächsten Kapitel anhand von unterschiedlichen Experimenten dargestellt.

Abbildung 4.1: Diese beiden Beispielbilder der Trainingsdaten wurden in den Räumlichkeiten der Frankfurt University of Applied Sciences aufgenommen und enthalten teilweise Stühle, Tische und weitere Gegenstände. Das linke Bild zeigt den Fußball als Trainingsobjekt in einer mittleren Entfernung, auf dem rechten Bild ist kein Fußball enthalten.

4.3 Training der Convolutional Neural Networks

Nachdem die Modelle und Datensätze für das Training ausgewählt und vorbereitet waren, konnten diese Modelle nun auf der beschriebenen Hardware trainiert und die Trainingsergebnisse ausgewertet werden. Das Training der Netze erfolgte in der Reihenfolge des Umfangs der Modelle. Zunächst wurde das Modell des Machbarkeitsnachweises (siehe Kapitel 2.3.4), durchgeführt und ausgewertet. Danach folgten Training und Ergebnisevaluierung der Modelle von Alex Krizhevsky und GoogLeNet's Inception (siehe Kapitel 3). Diese bei-

den Modelle wurden zunächst mit den zuvor genannten Standard-Datensätzen und folgend mit eigenen Bildern trainiert.

Machbarkeitsnachweis

Im Machbarkeitsnachweis wird der MNIST-Datensatz zum Training verwendet, um handgeschriebene Ziffern von 0-9 zu klassifizieren. Ein erster Versuch erfolgte mit einem Modell, das aus zwei vollverbundenen Schichten besteht. Zur Steigerung der Genauigkeit wurden in einem weiteren Versuch zwei Convolutional Layer eingefügt, auf deren Ausgaben jeweils Max Pooling angewendet wurde. Die 60.000 Bilder des MNIST-Datensatzes wurden in 50.000 Bilder für das Training und 10.000 Bilder für die Evaluierung aufgeteilt. Tabelle 4.1 zeigt Daten zu diesen beiden Versuchen.

Tabelle 4.1: Daten zum Machbarkeitsnachweis

Layer	Datensatz	Menge der Bilder	Klassen	Gen.
2x Fully Connected	MNIST	50.000 / 10.000	10	91%
2x Convolutional 2x Max Pooling 2x Fully Connected	MNIST	50.000 / 10.000	10	99,2%

Der Machbarkeitsnachweis erzielte auf der in Kapitel 4.1 aufgeführten Hardware im ersten Testlauf eine Genauigkeit von 91%. Durch den Einsatz von mehreren Convolutional Layern konnte dieses Ergebnis deutlich verbessert und eine Genauigkeit von 99,2% erzielt werden.

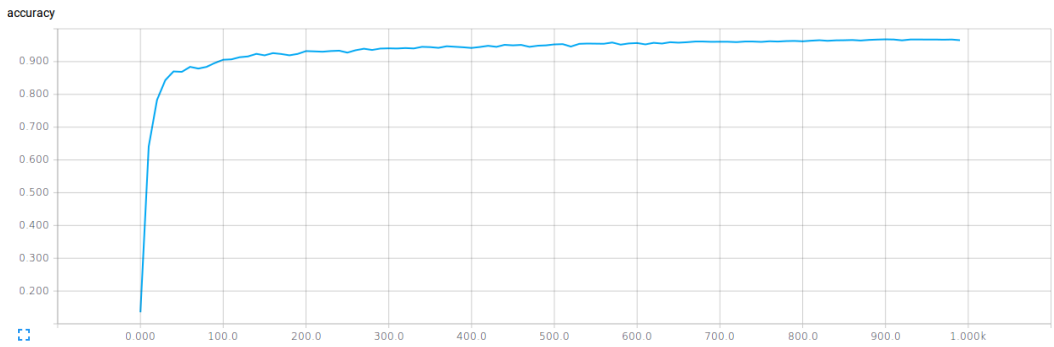

Abbildung 4.2: Entwicklung der Genauigkeit über 1000 Schritte

Abbildung 4.2 zeigt die Entwicklung der Genauigkeit über 1000 Trainingsschritte. Nach einem sehr steilen Anstieg der Genauigkeit innerhalb weniger Trainingsschritte folgt ein deutliches Abflachen gegen 90%. Dieser Kurvenverlauf ist ein typisches Bild für ein erfolgreich trainierendes Netz. Insgesamt wurden 20.000 Trainingsschritte ausgeführt. Nach Evaluierung der Lernergebnisse ergab sich die in der vorstehenden Tabelle gezeigte Genauigkeit von 99,2%. Mit einigen Anpassungen und entsprechend langer Trainingszeit können hier über 99,9% Genauigkeit erzielt werden [4]. Aufgrund solch hervorragender Resultate werden die besten kommerziellen neuronalen Netze, trainiert mit handgeschriebenen Ziffern und Buchstaben, von Unternehmen zur automatischen Verarbeitung von Überweisungen, Schecks und Briefen eingesetzt [24].

AlexNet

Die Funktionsfähigkeit und fehlerfreie Ausführung sowie das reibungslose Zusammenwirken der eingesetzten technischen Komponenten des Machbarkeitsnachweises wie Bestriebssystem, Frameworks, Anwendungsmodule und Hardware konnten in einem erfolgreichen Test- und Trainingslauf unter Beweis gestellt werden. Daraufhin konnte in einem nächsten Schritt das in Kapitel 3 beschriebene, komplexere AlexNet in Angriff genommen werden. Für einen ersten Trainingslauf wurde der Beispieldatensatz „17 Category Flower Dataset“ eingesetzt. Die in 17 Kategorien eingeteilten Blumenbilder des Test-Datensatzes liegen in unterschiedlicher Auflösung zu je 80 Bildern pro Kategorie vor und sind auf einzelne Ordner 0-16 aufgeteilt, die als Label der Bilder genutzt werden. Dieses erste Training des AlexNet erfolgte mit den im Beispiel-Sourcecode voreingestellten Parametern. Die wichtigsten Einstellungen sind hier die Lernrate als Faktor für die Gewichtsanpassung nach einem Trainingsschritt und die Batchsize, also die Menge der pro Epoche trainierten Bilder. Eine Epoche besteht jeweils aus einem Forward-Pass und einem Backward-Pass, die in Kapitel 2.2.2 beschrieben sind. Die Lernrate beträgt hier 0,001, die Batchsize 64, es werden 10% der geladenen Daten zur Evaluierung herangezogen und es soll 1000 Epochen lang trainiert werden. Bereits bei der Initialisierung des gestarteten Netzes kam es mit den voreingestellten Parametern zu einem Speicherüberlauf der Grafikkarte, der nach einer Wiederholung des Trainingslaufs mit einer reduzierten Batchsize von 32 Bildern pro Epoche nicht mehr auftrat.

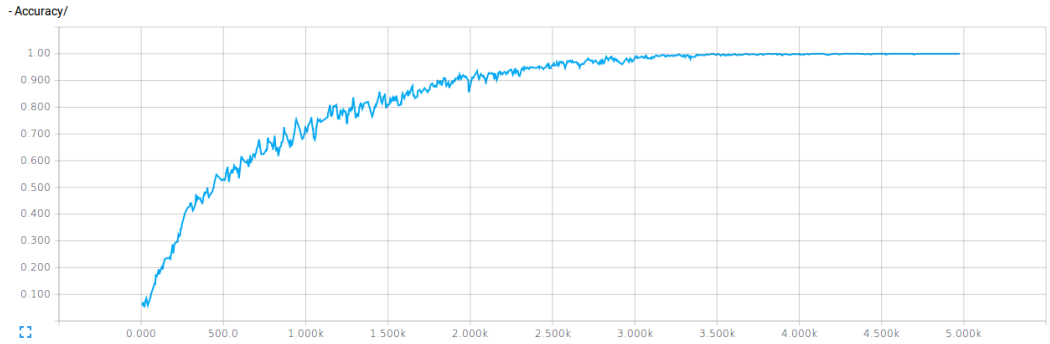

Abbildung 4.3: Der Verlauf der Genauigkeit beim Training zeigt zunächst einen Anstieg mit zunehmender Abflachung gegen 90%.

Die Genauigkeit, die während des Trainings validiert wurde, stieg zu Beginn entgegen der Erwartungen weniger stark an und flachte dann gegen 90% ab. Zudem erreichte die Genauigkeit 100%, was auf Overfitting hindeutet. Für eine Analyse des Problems wurde ein weiterer Trainingslauf mit einem selbst zusammengestellten Datensatz gestartet. Hierzu wurde nach der Registrierung und entsprechender Freigabe auf ImageNet ein gelabelter Datensatz von 1345 Fußball-Bildern geladen und bereinigt, wie im Kapitel 4.2 beschrieben. Zusätzlich wurden zur Unterscheidung in eine zweite Klasse weitere Bilder von ImageNet geladen, die keine Fußbälle enthalten. Der Datensatz bestand somit nach Bereinigung aus ca. 200 Bildern mit und 1000 Bildern ohne Fußball.

Das Training wurde auch in diesem Durchlauf mit den im Beispiel-Sourcecode voreingestellten Parametern gestartet. Der Loss sank hier sehr schnell ab und die Genauigkeit stieg fälschlicherweise innerhalb kurzer Zeit auf 100%. Eine Veränderung der Lernrate auf 0,001, 0,005, 0,01 und 0,02 ergab keine abweichenden Ergebnisse. Bei einer Lernrate von $\geq$ 0,03 hingegen bewegte sich die Genauigkeit auch über längere Trainingszeiten durchgehend um 50%. Da sich durch die Anpassung der Lernrate keine Besserung einstellte, wurde als nächstes der Datensatz vergrößert. Mit den selbsterstellten Bildern, die in den Räumen der Hochschule aufgenommen wurden, betrug die Anzahl der Bilder in der Fußball-Klasse nun ca. 800 Bilder und die Klasse ohne Fußbälle verfügte über ca. 2000 Bilder. Das Training mit diesem Datensatz wurde mit einer Lernrate von 0,03, sowie allen anderen Parametern des Beispiel-Sourcecodes unverändert, durchgeführt. Die Entwicklung der Genauigkeit, zu sehen in Abbildung 4.4, zeigt ein anderes Verhalten als die Versuche, die nur mit dem ImageNet-Datensatz durchgeführt wurden. Die Genauigkeit konnte während

der Trainingszeit insgesamt gesteigert werden und erreichte über 80%, jedoch erfolgte ein Einbruch der Genauigkeit im frühen Verlauf.

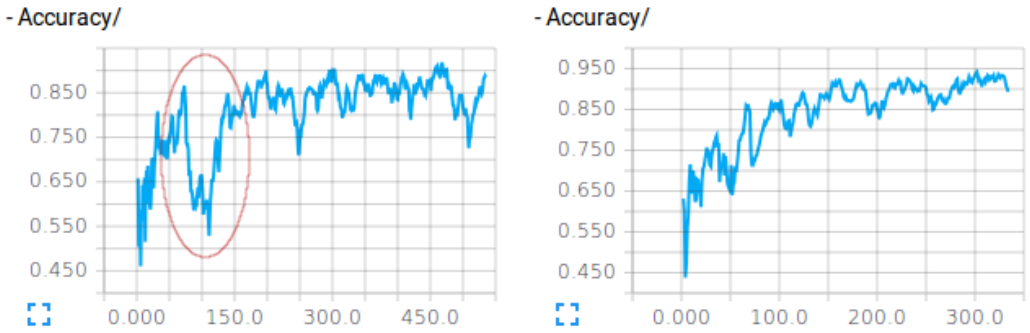

Abbildung 4.4: Die Abbildung zeigt den Einbruch der Genauigkeit beim Training des AlexNet-Modells mit Bildern von ImageNet und selbsterstellten Bildern [links] im Vergleich zum Training ohne Bilder von ImageNet [rechts].

Der starke Einbruch der Lernkurve nach dem ersten Viertel auf dem in Abbildung 4.4 gezeigten Abschnitt, deutet auf den Wechsel der Bilder von ImageNet auf die selbsterstellten Bilder hin, da diese sich in Auflösung, Qualität und der Umgebung bei der Aufnahme unterscheiden. Um diese Irritation des Trainings zu vermeiden, wurden die Bilder von ImageNet aus beiden Klassen des Trainings-Datensatzes entfernt. Zum Vergleich zeigt die Abbildung ein Testtraining ohne Bilder von ImageNet. Ein weiteres längeres Training mit selbsterstellten Bildern zeigte bis auf einzelne Ausschläge eine Lernkurve, die dem gewünschten Verlauf nahe kommt (siehe Abbildung 4.5) und bewertete nach Abschluss des Trainings einige Testbilder korrekt.

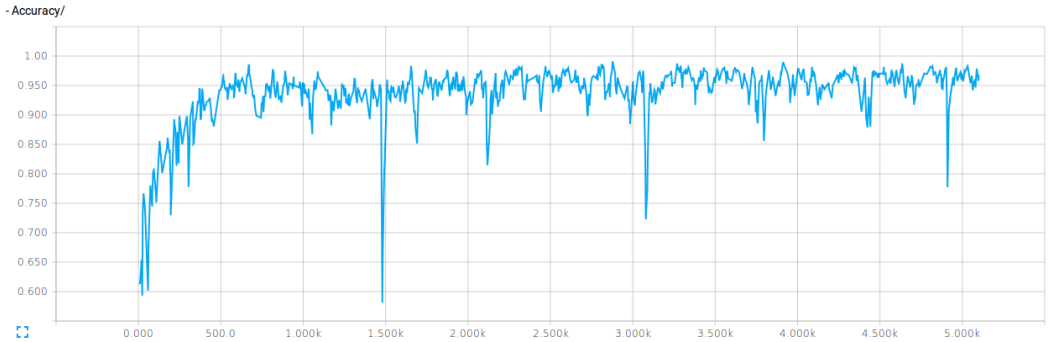

Abbildung 4.5: Die Abbildung zeigt die Genauigkeit des Trainings: Modell AlexNet mit selbserstellten Bildern.

Nach diesem Lernerfolg sollte vor einem weiteren Training geprüft werden, ob das trainierte Modell geladen werden kann. Die API TFLearn, in der auch der Beispielcode für das Training vorliegt, stellt zum Speichern und Laden die Funktionen `model.save(name)` und `model.load(name)` zur Verfügung. Mit diesen Funktionen können die während des Trainings erlernten Gewichte gespeichert und wieder geladen werden, nicht jedoch das Netz selbst. Um ein Modell zu laden, muss also zuerst das Netz erneut definiert und aufgebaut werden. Danach können die Gewichte geladen und das Training fortgesetzt oder Bilder ausgewertet werden. TFLearn stellt zur Auswertung von Bildern bisher noch keine direkte Methode zur Verfügung, wie sich bei der Kommunikation mit dem Entwickler herausstellte. Es konnte jedoch mit Unterstützung des Entwicklers ein Workaround erarbeitet werden. Der Entwickler versprach, eine geeignete Funktion zu einem späteren Zeitpunkt zu implementieren. Bei der Umsetzung dieser Umgehungslösung stellte sich heraus, dass der Aufbau des Modells mit den geladenen Gewichten den Speicher der Grafikkarte überfordert. Nach erneutem Kontakt mit dem Entwickler von TFLearn wurde versuchsweise die Größe des Netzes angepasst. Die letzten beiden vollverbundenen Schichten wurden zunächst von 4096 auf 3072 Neuronen vermindert, trainiert und danach geladen. Auch dieses mal erfolgte ein Abbruch aufgrund eines Speicherüberlaufs. Nach weiterer Verminderung der Schichten auf 2048 Neuronen war es möglich, das Modell zu laden und anschließend die Bilder zu bewerten. Nachdem eine funktionsfähige Struktur für das Laden der Gewichte gefunden war, musste das Netz nun erneut länger trainiert werden, da für die Tests zum Laden der Gewichte nur sehr kurz trainiert wurde. Ein erneutes Training zeigte, dass ähnliche Ergebnisse erzielt wurden, wie in dem vorangegangenen Versuch mit 4096 Neuronen in den vollverbundenen Schichten (siehe Abbildung 4.6).

Nach diesen Lernergebnissen wurde die Auswertung von segmentierten Einzelbildern getestet. Hier ergab sich, dass die Bewertung von 12 einzelnen Segmenten knapp über 2 Sekunden dauert, was für eine flüssige Navigation des Roboterfahrzeugs nicht ausreicht. Da bisher keine Funktion zur Bewertung von Einzelbildern in TFLearn zur Verfügung steht und die Auswertungsdauer kurzfristig nicht verringert werden konnte, wurde aufgrund der begrenzten Zeit und der Notwendigkeit zur Verwendung der für das Training zur Verfügung stehenden Hardware, auf weiteres Training des Modells AlexNet verzichtet und zur Evaluierung des Modells Inception übergegangen.

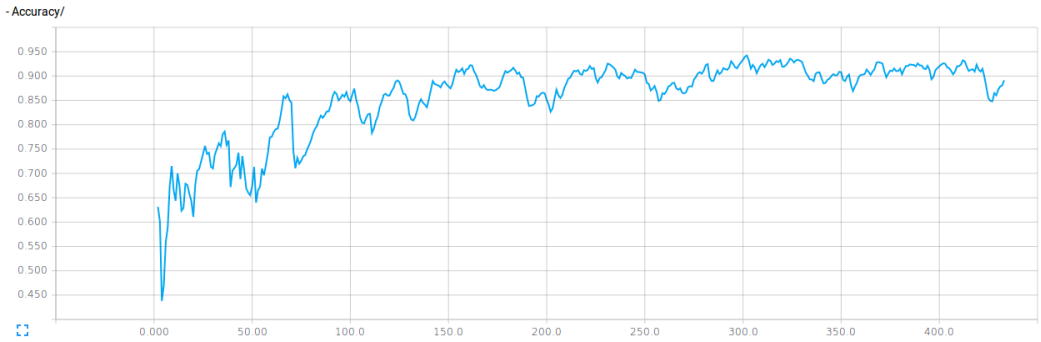

Abbildung 4.6: Eine Verringerung der Neuronen von 4096 auf 2048 in den vollverbundenen Schichten zeigt vergleichbare Trainingsergebnisse (siehe Schritte 0-500 in Abbildung 4.5).

Inception

Das Training des Inception-V3-Netzes wurde zunächst mit dem ImageNet-Datensatz auf der in Kapitel 4.1 beschriebenen Hardware durchgeführt. Der Download der Bilder für 1000 Klassen dauerte ca. 42 Stunden und nimmt insgesamt ca. 500 GB Speicherplatz in Anspruch.

Wie von Szegedy et al. beschrieben, wurde das Training von Inception für die Teilnahme an der ILSVRC auf einem Desktop-PC mit 128GB CPU RAM und 8 NVIDIA Tesla K40 Grafikkarten (mit jeweils 12GB RAM) trainiert, konnte aber ebenfalls auf einem Desktop-PC mit 32GB RAM und einer NVIDIA Tesla K40 getestet werden. Diese Angaben ließen vermuten, dass bei dem für die vorliegende Arbeit eingesetzten Desktop-PC mit 8GB RAM und einer NVIDIA GTX 750 TI Grafikkarte mit 2GB RAM, ein Training nicht ohne Probleme möglich sein würde. Um Inception auch auf weniger performanter Hardware trainieren zu können, geben Szegedy et al. Hinweise, wie z.B. die Reduzierung des Speicherfaktors der Eingabe-Queue, in der die für das Training vorbereiteten Bilder bis zur Verwendung gehalten werden, sowie die Verringerung der Batchsize durch Ausprobieren auf die für die Grafikkarte maximal mögliche Größe.

So konnte Inception mit einer maximalen Batchsize von 8, verglichen mit der von den Autoren verwendeten Batchsize von 64, gestartet werden. Andernfalls kam es bereits beim Aufbau des Graphen zum Absturz, da der Speicher der Grafikkarte überlastet war. Mit einer Batchsize von 8 konnte der Loss, bei einem initialen Wert von ca. 14, auf 10 reduziert werden. Sobald der Loss-Wert

gegen 10 lief, brach das Training allerdings immer ab, da die Traingsergebnisse in NaN (engl. „Not a Number“ - „Keine Zahl“) resultierten. Hierzu geben Szegedy et al. eine zu hohe Lernrate als Auslöser an und versprechen Abhilfe durch deren Reduzierung. Durch diese Reduzierung würde jedoch die Trainingsdauer, die bereits durch die stark verringerte Batchsize deutlich erhöht war, noch zusätzlich verlängert werden.

Da das Erreichen guter Trainingsergebnisse offenbar nur durch weiteren Einsatz finanzieller Mittel zur Beschaffung einer oder mehrerer leistungsfähigerer Grafikkarten gelöst werden kann, wurde auf weitere Versuche mit dem Training des Inception-Modells verzichtet. Obwohl aufgrund der wiederholten Abbrüche keine eigenen verwendbaren Trainingsergebnisse mit dem Inception-Modell erzielt werden konnten, fand eine Evaluierung der Erkennungsleistung des Modells statt, indem das bereits von GoogLeNet trainierte Modell, das zum Download zur Verfügung gestellt wird, getestet wurde.

Das Laden des trainierten Modells konnte erfolgreich durchgeführt werden und lieferte die von Szegedy et al. beschriebenen Ergebnisse [31] bei der Auswertung von Einzelbildern die mit dem Raspberry Pi aufgenommen wurden. Es wurden verschiedenste Gegenstände fotografiert und durch das Modell größtenteils korrekt bewertet. Die Auswertung von Bildern, die einen Fußball enthielten, erfolgte mit sehr guten Ergebnissen.

Die Ergebnisse der Versuche mit beiden Modellen und unterschiedlichen Datensätzen führten zu der Entscheidung, Inception mit dem vortrainierten Modell für einen praktischen Einsatz mit dem Roboterfahrzeug zu implementieren. Die praktische Vorgehensweise wird in Kapitel 4.4 und Kapitel 5 beschrieben.

4.4 Positionsbestimmung durch Segmentierung

Nachdem auf Basis der Ergebnisse aus den Trainingsläufen auf die Verwendung des trainierten Modells von Inception zur Bildauswertung zurückgegriffen wurde, musste festgelegt werden, wie die Auswertung zur Positionsbestimmung im Bild verwendet werden kann.

Um die Position des Fußballs auf dem Bild zu bestimmen und anhand dieser die Navigation des Roboters zu steuern, wird das Bild in einzelne Segmente

unterteilt. Jedes dieser Segmente wird anschließend zur Bewertung herangezogen. Das Segment mit der höchsten Erkennungswahrscheinlichkeit enthält den Fußball. Liegt das Segment, oder, falls in mehreren Segmenten eine hohe Wahrscheinlichkeit vorliegt, die Segmente, im rechten oder linken Bereich, navigiert das Roboterfahrzeug in diese Richtung. Gruppieren sich die Segmente mit relativ hoher Erkennungswahrscheinlichkeit in der Mitte, wird keine Richtungsänderung veranlasst, sondern weiter in diese Richtung gefahren. Eine gute Erkennungsrate wurde in den Tests mit einer Segmentierung in 12 Bildteile erreicht. Diese Bildteile sind entsprechend 160x160 Pixel groß, bei einer Kameraauflösung von 640x480 des Raspberry Pi. Für die Teilung eines Bildes in einzelne Segmente wird die Klasse `TileReader` implementiert, die im folgenden Listing zu sehen ist.

```
1  class TileReader:
2
3    def __init__(self, img, tileSize):
4      self.tileSheet = img
5      self.tileSize = tileSize
6      self.margin = 1
7
8    def getTile(self, tileX, tileY):
9      posX = (self.tileSize * tileX) + (self.margin * (tileX + 1))
10     posY = (self.tileSize * tileY) + (self.margin * (tileY + 1))
11     box = (posX, posY, posX + self.tileSize, posY + self.
       tileSize)
12     return self.tileSheet.crop(box)
```

Listing 4.1: Klasse für die Segmentierung von Bildern

Zunächst wird mit `reader = TileReader(image, size)` ein Objekt der Klasse angelegt. Als Parameter `image` wird das zu segmentierende Bild und als Parameter `size` die gewünschte Größe der quadratischen Segmente übergeben. Auf dem Server wird dann in zwei verschachtelten Schleifen, für die bei einer Einteilung in 12 Segmente mit je 160x160 Pixeln entstandenen 3 Zeilen und 4 Spalten, iteriert und in jedem Schritt mit der Funktion `getTile(col, row)` ein Segment abgerufen und bewertet.

5 Entwicklung des Roboterfahrzeugs

Für die Objekterkennung und -lokalisierung mit Hilfe eines KNN ist eine System- und Rechenleistung erforderlich, die für im Markt erhältlichen, mobil einsetzbaren Plattformen eine große Herausforderung darstellt. Da es bisher lediglich für den Raspberry Pi 3 Model B, einen der leistungsstärksten Vertreter von Single Board Computern (SBC), eine Portierung von TensorFlow gibt, die von Abrahams et al. auf GitHub zur Verfügung gestellt wird[7], kommt dieser in dem hier durchgeführten Projekt zur Verwendung.

Der Raspberry Pi 3 Model B verfügt über alle Komponenten die nötig sind, um das geplante Projekt umzusetzen. Er verfügt über ein Camera Serial Interface (CSI) zum einfachen Anschluss einer für den Pi erhältlichen Kamera zur Aufnahme der auszuwertenden Bilder, über 40 GPIO-Ports, z. B. zur Ansteuerung von Motoren und Ultraschall-Sensoren und über WLAN für die Kommunikation mit einem Server. Abbildung 5.1 zeigt die Platine mit den gekennzeichneten Komponenten.

Der Raspberry Pi ist die zentrale Einheit der aufgebauten Hardware, deren gesamte Komponenten hier mittels Python-Programmen angesteuert werden. Auf dem Raspberry Pi kommt das darauf zugeschnittene Betriebssystem Raspbian zum Einsatz, welches auf Debian „Jessie“ basiert. Dies erleichtert durch die vorinstallierte Software eine Konfiguration der hardwarespezifischen Komponenten. Für die Verbindung zur Kamera wurde die API `python-picamera` und für Training und Auswertung von KNN die oben beschriebene Raspberry Pi-Portierung von TensorFlow installiert.

[7] Abrahams, J.: Raspberry Pi-Portierung TensorFlow, siehe: https://github.com/samjabrahams/tensorflow-on-raspberry-pi, Abruf: Mai 2016.

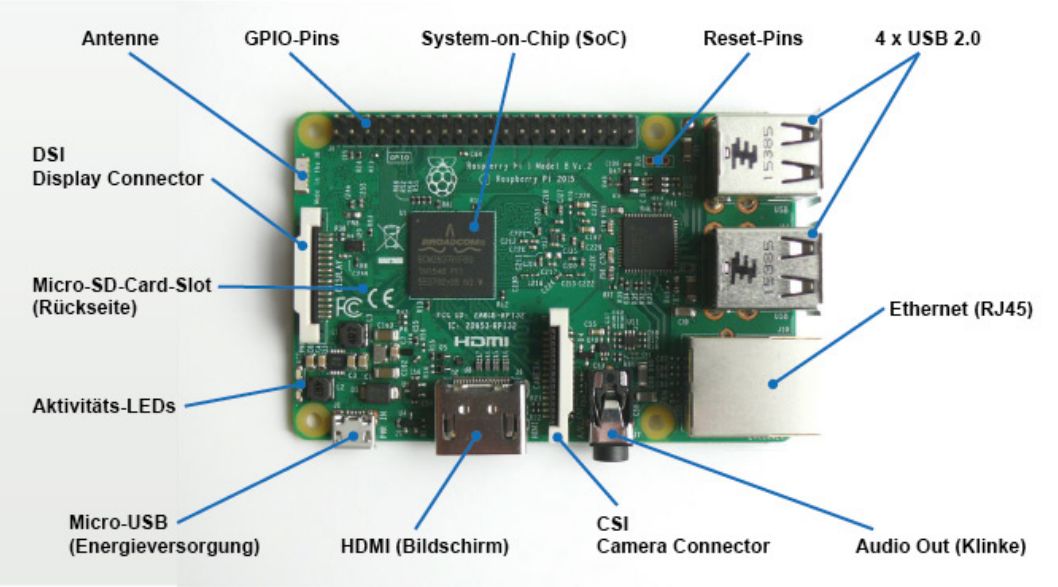

Abbildung 5.1: Der Single Board Computer enthält alle wichtigen Komponenten, die auch in Desktop-PCs zu finden sind. Neben 4 USB 2.0 Anschlüssen, RJ45-Netzwerk-, Audio- und HDMI-Anschluss, verfügt der Raspberry Pi 3 Model B außerdem über einen 40-Pin GPIO-Port, einen Display-Connector und einen Kamera-Anschluss. Quelle: Entnommen aus *Elektronik-Kompendium*, https://www.elektronik-kompendium.de/sites/raspberry-pi/1905251.htm, Abruf: Mai 2016.

5.1 Entwicklung der Hardware

Da es sich bei dem Roboterfahrzeug um kein vorkonfektioniertes System handelt, muss vor der eigentlichen Entwicklung der Hardware eine detaillierte Planung mit Definition und Entwurf für die grundsätzliche Eignung aller Komponenten und deren Zusammenwirken für das Vorhaben des mobilen Einsatzes mit KNN erstellt werden. Bestimmend hierbei sind die geforderten Funktionen und die tatsächlich erreichbare Leistungfähigkeit des Gesamtsystems mit den ausgewählten Komponenten. Wichtig ist auch die Planung der Tests für die mechanische und elektrische Abstimmung, um Fehler im Zusammenbau nicht mit der Zerstörung einzelner oder aller Komponenten bezahlen zu müssen. Bei der Planung nicht zu vernachlässigen sind insbesondere auch die Kosten für den Bau des Gesamtsystems Roboterfahrzeug. Wie üblich gibt es hierbei nach oben keine Grenzen. Die Kosten für die vorliegende praktische Ausführung des Hardwaresystems mit Raspbery PI belaufen sich auf ca. 200 Euro.

Die wesentlichen hardwareseitigen Erfordernisse für ein funktionsfähiges selbststeuerndes Fahrzeug mit Objekterkennung sind die Stromversorgung, Motorsteuerung, Abstandsmessung, Kameraanschluss, Rechnereinheit und Datenkommunikation.

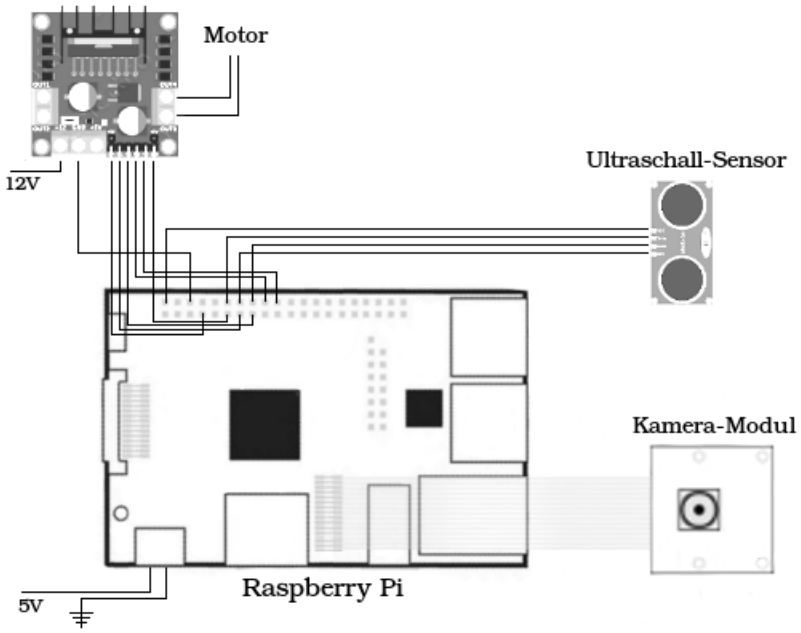

Abbildung 5.2: Abgebildet ist der Raspberry Pi, mit dem die H-Brücke zur Steuerung der Motoren, der Ultraschall-Sensor zur Kollisionsvermeidung und das Kameramodul verbunden sind.

Stromversorgung

Die Stromversorgung des Roboterfahrzeugs erfolgt über 8 Mignonzellen mit jeweils 1,2 V Spannung und einer Kapazität von 2500 mAh. Die Mignonzellen erreichen in Reihe geschaltet so eine Gesamtspannung von 8 * 1,2 V = 9,6 V. Zur Versorgung des Raspberry Pi 3 ist eine Eingangsspannung von 5 V vorgeschrieben, die mittels eines Abwärtswandlers aus den 9,6 V gewonnen werden. Durch die Mignonzellen erfolgt außerdem die Energieversorgung der L298 H-Brücke. Eine Versorgung des Raspberry Pi ist prinzipiell ebenfalls über die H-Brücke möglich, die über einen 5 V-Ausgang verfügt. Erfahrungen an-

derer Verwender besagen aber, dass diese Ausgangsspannung nicht zuverlässig gewährleistet werden kann.[8]

Motorsteuerung

Die vier Räder des Roboterfahrzeugs werden jeweils mit einem Getriebemotor angetrieben, der mit 3-6 V Gleichstrom arbeitet. Die Stromversorgung der Motoren und ihre Ansteuerung erfolgt über die L298 H-Brücke. Es werden jeweils zwei Motoren für eine Seite des Roboterfahrzeugs in Reihe geschaltet, somit können beide rechten bzw. beide linken Motoren nur gleichzeitig und nicht getrennt angesteuert werden. Die Motoren haben bei 5 V einen Leerlaufstrom von 150 mA und 850 mA unter Volllast.

Da es sich bei den Motoren nicht um Schrittmotoren handelt, müssen sie, um die Geschwindigkeit regelbar zu gestalten, über ein PWM-Signal (Pulsweitenmodulation) angesteuert werden, das softwareseitig auf dem Raspberry Pi erzeugt wird. Die Motoren werden also pro Zeitspanne mehrfach an- und ausgeschaltet, allerdings in so kurzen Intervallen, dass sie sich noch in Drehbewegung befinden. Je häufiger sie an- und ausgeschaltet werden, desto schneller drehen sie sich.

Um das Roboterfahrzeug lenken zu können, werden die Motoren unterschiedlich schnell gedreht. Eine linkskurve erfolgt, indem die Motoren der rechten Seite schneller als die auf der linken Seite gedreht werden und umgekehrt. Eine Drehung auf der Stelle kann durch das entgegengesetzte Drehen der Motoren der linken und der rechten Seite durchgeführt werden.

Abstandsmessung

Um die Kollision des Roboterfahrzeugs mit Gegenständen oder Wänden zu verhindern, ist eine Abstandsmessung erforderlich. Diese wird mittels eines HC-SR04 Ultraschall-Moduls umgesetzt. Dieses Ultraschall-Modul kann, je nach Reflektionsrichtung und -intensität, bei unterschiedlichen Materialien und Reflektionswinkeln Entfernungen in einem Bereich von 2 cm bis 4 m messen.

[8] Stapel, I.: Raspberry Pi Cardboard Car, siehe: http://www.cardboard-car.com/top-story/raspberry-pi-roboter-stromversorgung/6499, Abruf: Mai 2016.

Der Sensor wird über einen Trigger-Pin angesteuert, um das Aussenden des Ultraschall-Signals einzuleiten. Sobald das reflektierte Signal wieder vom Ultraschall-Sensor aufgenommen wird, kann dies am Echo-Pin ermittelt werden. Softwareseitig wird dann die verstrichene Zeit und so die Entfernung zum Objekt errechnet, wie im Kapitel 5.2 beschrieben.

Kameraanschluss

Der Anschluss der Kamera erfolgt wie vorgesehen über ein Flachbandkabel am Camera Serial Interface (CSI) des Raspberry Pi. Die Kamera wird dann etwas erhöht mit einer starren Verbindung auf der Grundplatte des Roboterfahrzeugs angebracht, sodass der Blickwinkel der Kamera nur durch die Lenkbewegungen des Fahrzeugs gesteuert werden kann. Das Kameramodul verfügt über ein Fixfokusobjektiv mit 5 Megapixel und hat somit eine maximale Auflösung von 1920x1080. Für das Projekt wurde eine Auflösung von 640x480 gewählt, welche als Qualität für die Objekterkennung ausreicht und die Auswertung beschleunigt. Die Einstellungen der Kamera wie Helligkeit, Auflösung etc. sind variabel und können softwareseitig eingestellt werden.

Rechneranschlüsse

Der Raspberry Pi 3 verfügt über 40 Pins, also 14 Pins mehr als seine Vorgänger. Diese hohe Anzahl muss jedoch für das Anwendungsbeispiel nicht ausgereizt werden. Die folgende Abbildung 5.3 zeigt die Pin-Belegung für die H-Brücke zur Ansteuerung der Motoren und des Ultraschall-Moduls.

Pin 2 wird für die Stromversorgung und Pin 14 für die Masse des Ultraschall-Moduls verwendet. Die Auslösung und das Verarbeiten des darauffolgenden Echo des Moduls werden über die Pins 12 und 16 geregelt. Mit den Pins 7 und 11 werden jeweils die Motoren einer Seite aktiviert und an den Pins 13, 15, 18 und 22 wird jeweils das PWM-Signal erzeugt, mit dem die Geschwindigkeit der Motoren geregelt wird.

Bei der Verwendung der GPIO-Pins ist eine genaue Planung und Durchführung wichtig. Da die Pins nicht gegen Überlastung abgesichert sind, kann durch falsche Verkabelung die gesamte Platine des Raspberry Pi unwiederbringlich zerstört werden.

Pin#	Verwendung		Verwendung	Pin#
01	-		5V Ultraschallsensor	02
03	-		-	04
05	-		Ground H-Brücke	06
07	ENA H-Brücke		-	08
09	-		-	10
11	ENB H-Brücke		Trigger Ultraschallsensor	12
13	IN1 H-Brücke		Ground Ultraschallsensor	14
15	IN2 H-Brücke		Echo Ultraschallsensor	16
17	-		IN3 H-Brücke	18
19	-		-	20
21	-		IN4 H-Brücke	22

Abbildung 5.3: Die Abbildung zeigt die oberen 22 Pins des GPIO-Ports auf dem Raspberry Pi. Die weiteren 18 Pins sind nicht enthalten, da sie nicht verwendet wurden.

Datenkommunikation

Zur Kommunikation zwischen dem Raspberry Pi und dem Server, der zur Auswertung der Bilder zur Verfügung gestellt wird, wird das integrierte WLAN-Modul eingesetzt. Mit dem WLAN-Standard 802.11n können sehr hohe Geschwindigkeiten erreicht werden, was die Übertragung der Bilder und die Antwort vom Server in sehr kurzer Zeit möglich macht. Die Zeit, die die Datenübertragung in Anspruch nimmt, kann gegenüber der um ein Vielfaches höheren Dauer der Bildauswertung vernachlässigt werden.

5.2 Programmierung

Abbildung 5.4 zeigt ein Grobkonzept der auf dem Raspberry Pi und dem Server eingesetzten Softwarekomponenten. Natürlich muss auch hier bei der Entwicklung der Software und den Tests, Rücksicht auf die Besonderheiten der verwendeten Hardwarekomponenten genommen werden, um Ausfall oder Schaden an der Hardware durch Fehler in der Programmierung zu vermeiden. Für die eingesetzte Software fallen keine Kosten an, da bei eingesetzter Fremdsoftware nur Open-Source-Komponenten Verwendung finden.

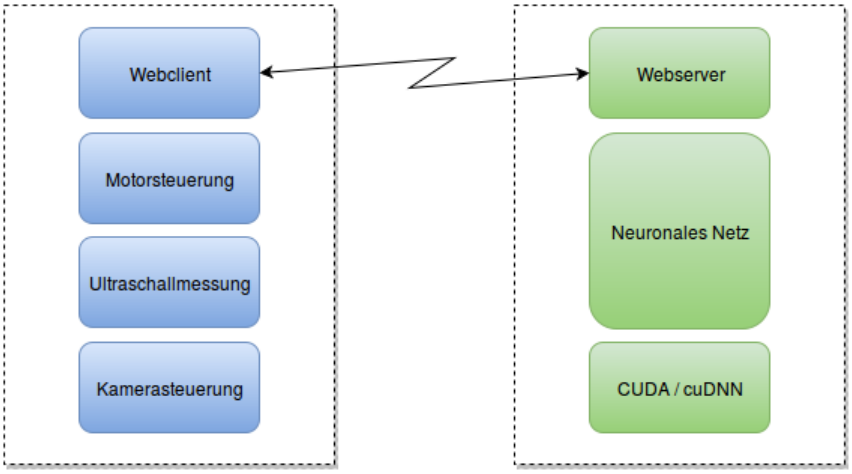

Abbildung 5.4: Grobkonzept der eingesetzten Softwarekomponenten

5.2.1 Bewegungssteuerung

Für die Ansteuerung der Motoren wird eine von Ingmar Stapel auf seiner Internetseite[9] zur Verfügung gestellte freie API eingesetzt, die eine vereinfachte Ansteuerung der Motoren ermöglicht. Die API stellt dafür folgende Befehle in Python zur Verfügung:

```
1 # Den Motor-Modus setzen
2 setMotorMode(motor, mode)
3
4 # Die Leistung des Motors regeln -1 (min) bis +1 (max)
5 setMotorLeft(power)
6 setMotorRight(power)
```

Listing 5.1: Befehle für die Motoransteuerung

Durch die Funktion `setMotorMode()`, die als Parameter `motor` die Werte „leftmotor“ bzw. „rightmotor“ und als Parameter `mode` „forward“, „reverse“ und „stop“ erwartet, kann der Modus der Motoren eingestellt werden und mit `setMotorLeft()` und `setMotorRight()` die Motoren jeweils mit der als Parameter `power` übergebenen Leistung gedreht werden. Als `power`-Wert kann die Leistung als `float` von -1.0 bis 1.0 übergeben werden und der Motor somit sowohl vorwärts als auch rückwärts in gewünschter Geschwindigkeit betrieben werden.

[9] Stapel, I.: API Motorsteuerung, siehe: http://www.cardboard-car.com/top-story/raspberry-pi-roboter-stromversorgung/6499, Abruf: Mai 2016.

Die Distanzwerte des Ultraschall-Sensors werden alle 20 ms ermittelt, um rechtzeitig vor einem Hinderniss abbremsen und lenken zu können. Um den Ultraschall-Sensor ansprechen zu können, müssen zunächst die entsprechenden GPIO-Pins am Raspberry Pi festgelegt und initialisiert werden.

```
# Trigger und Echo Pins des Ultraschallsensors festlegen
# (Harware-Pins 12 und 16: interne Bezeichnung GPIO18/GPIO23)
GPIO_TRIGGER = 18
GPIO_ECHO = 23

# Die Pins als Ein- und Ausgang definieren
# Senden eines Signals (GPIO.OUT=HIGH) / Empfangen eines Singals
    (GPIO.IN=LOW)
GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
GPIO.setup(GPIO_ECHO, GPIO.IN)
```

Listing 5.2: Initialisierung der GPIO-Pins für die Distanzmessung

Nach dieser Initialisierung kann die nachfolgende Klasse in einer Schleife alle 20 ms aufgerufen werden.

```
def measureDistance():
  # Trigger-Pin auf HIGH setzen, um Ultraschall-Signal
    auszusenden
  GPIO.output(GPIO_TRIGGER, True)

  # Trigger-Pin nach 0.01ms auf LOW setzen, um die Aussendung
    des Ultraschall-Signals zu stoppen
  time.sleep(0.00001)
  GPIO.output(GPIO_TRIGGER, False)

  # Start- und Stop-Zeit für die Zeitmessung definieren
  StartTime = time.time()
  StopTime = time.time()

  # Start-Zeit speichern (Echo-Pin auf LOW)
  while GPIO.input(GPIO_ECHO) == 0:
    StartTime = time.time()

  # Zeitpunkt des Signalempfangs speichern (Echo-Pin auf HIGH)
  while GPIO.input(GPIO_ECHO) == 1:
    StopTime = time.time()
```

```
20
21  # Differenz zwischen StartTime und StopTime berechnen
22  TimeElapsed = StopTime - StartTime
23  # Mit Schallgeschwindigkeit (34300cm/s) multiplizieren und
      durch 2 teilen (2 Wege)
24  distance = (TimeElapsed * 34300) / 2
25
26  # Entfernung zurückgeben
27  return distance
```

Listing 5.3: Klasse für die Ultraschall-Distanzmessung

In der Klasse `measureDistance()` wird ein kurzes Ultraschall-Signal gesendet, indem der entsprechende Trigger-Pin für 0,01 ms auf HIGH gesetzt wird. Danach wird die Zeit, die von der Aussendung des Signals bis zum Empfang des reflektierten Signals vergangen ist, durch die Schallgeschwindigkeit von 34.300 cm/s und dann durch zwei geteilt, da die Zeit für den Weg des Schalls bis zu und von dem reflektierenden Objekt gemessen wird. Die Rückgabe der Klasse ist die Distanz mit einer theoretischen Genauigkeit von ±3 mm. Die Berechnung basiert auf der Annahme einer Umgebungstemperatur von etwa 20°C, da die Schallgeschwindigkeit unter anderem temperaturabhängig ist und so Abweichungen von 0,60 m/s pro Grad Celsius auftreten können [8]. Eine Lösung dieses Problems ist mit Hilfe von Temperaturmessung und entsprechender Anpassung der Berechnung möglich, wird aber im Kontext dieser Arbeit als nicht erforderlich erachtet.

Der Ablauf der Navigation wird in Kapitel 4.4 beschrieben.

5.2.2 Kamerasteuerung

Für das Steuern der spezifischen Kamera des Raspberry Pi über den CSI-Port wird die API `python-picamera` verwendet. Mit dieser API können bei der Initialisierung eines Kamera-Objektes Einstellungen wie Auflösung, Helligkeit und Effekte vorgenommen werden und es stehen Methoden für die Aufnahme von Bildern, Videos und die Bereitstellung als Stream in verschiedenen Formaten zur Verfügung.

Der Raspberry Pi arbeitet mit der Auswertung von Einzelbildern und nicht mit einem Videostream. Die entsprechenden Einzelbilder werden segmentiert und die einzelnen Segmente mit Hilfe des zuvor trainierten CNN ausgewertet,

um auch die Position des gesuchten Objekts zu erfassen. Die genaue Vorgehensweise wird in Kapitel 4.4 beschrieben.

5.2.3 Client-Server-System

Wie bei der Auswertung erster Tests in Kapitel 5.2.5 beschrieben, dauert die Bewertung einzelner Bilder auf dem Raspberry Pi aufgrund der fehlenden Ressourcen und vor allem der fehlenden Möglichkeit die GPU des Raspberry Pi zu nutzen, zu lange (ein Bild ca. 3 Sekunden). Deshalb wird, statt die Bilder auf dem Raspberry Pi auszuwerten, eine Client-Server-Lösung implementiert, sodass die Bilder vom Raspberry Pi auf einen leistungsstärkeren Rechner übertragen werden, auf dem eine Auswertung von Bildern in einem Bruchteil der Zeit stattfinden kann. Die empfangenen Bilder werden auf diesem Rechner segmentiert, bewertet und die Bewertung zurück zum Raspberry Pi gesendet, sodass dieser auf deren Grundlage eine Navigation durchführen kann.

Die Client-Komponente ist ein Bestandteil der Steuerungssoftware auf dem Raspberry Pi. Es wird zunächst eine TCP-IP-Verbindung zum Server aufgebaut.

```
1 # Einen Client-Socket mit dem Server verbinden
2 client_socket = socket.socket()
3 client_socket.connect(HOST, PORT)
```

Listing 5.4: Aufbau einer Verbindung zum Server

Ist die Verbindung hergestellt, werden Bilder der Kamera des Raspberry Pi zur Auswertung an den Server gesendet. Nach der Übertragung eines Bildes wird auf die Antwort vom Server gewartet, ob ein Fußball auf dem Kamera-Bild entdeckt wurde und in welchem Bereich sich dieser befindet.

```
1 for _ in camera.capture_continuous(stream, 'jpeg'):
2   # Die Länge des Captures an den Stream senden und flushen um
3   # sicherzustellen, dass tatsächlich gesendet wird
4   connection.write(struct.pack('<L', stream.tell()))
5   connection.flush()
6
7   # Den Stream auf Anfang setzen und die Bilddaten senden
8   stream.seek(0)
```

```
9   connection.write(stream.read())
10
11  # Den Stream für den nächsten Capture zurücksetzen
12  stream.seek(0)
13  stream.truncate()
14
15  # Auf Antwort des Servers warten
16  reply = client_socket.recv(4096)
```

Listing 5.5: Aufnahme und Senden von Bildern in einer Schleife

Wenn der Server eine erfasste Position des Fußballs meldet, wird die Richtung an die Motorsteuerung weitergegeben, indem der laufende Navigationsvorgang unterbrochen wird, um das Fahrzeug entsprechend rechts oder links zu lenken oder auf den entdeckten Fußball zuzusteuern, wenn dieser sich in der Mitte des Bildes befindet. Der Vorgang der Motorsteuerung ist im entsprechenden Kapitel oben beschrieben.

5.2.4 Gesamtsteuerung

Mit der Einführung des Client-Server-Systems, aufgrund der in Kapitel 5.2.5 beschriebenen Geschwindigkeitsdefizite bei der Bildauswertung auf dem Raspberry Pi, erfolgte eine Umstellung des ursprünglichen Softwareentwurfs. Die Aufnahme von Bildern, die Übertragung zum Server und die Auswertung der Antwort findet nun in einem Client-Modul statt, wobei der ursprüngliche Sourcecode, der für die Navigation des Roboterfahrzeugs verantwortlich war, in eine Klasse ausgelagert wurde, die als Thread parallel zur Bildauswertung läuft.

Sobald eine Verbindung zum Server hergestellt wurde, wird ein Objekt der Klasse `Navigation()` als Thread gestartet. Das Roboterfahrzeug bewegt sich dann selbstständig durch den Raum und vermeidet Kollisionen, indem es sich dreht, wenn ein Gegenstand den Weg versperrt. Während dieser Navigation werden laufend Bilder zur Auswertung an den Server übertragen. Wurde ein Fußball im Bild erkannt, meldet der Server anhand der Erkennung in einzelnen Bildsegmenten (siehe Kapitel 4.4), in welcher Region des Bildes sich der Fußball befindet. Der Ablauf der Navigation teilt sich dann in drei Vorgehensweisen zu den möglichen Positionen des Fußballs:

1. **Links:** Der Thread `Navigation()` wird pausiert und es erfolgt eine Drehung nach links, indem die Funktion `navigateLeft()` ausgeführt wird.

2. **Rechts:** entsprechend Punkt 1 wird die Navigation pausiert und eine Drehung nach rechts durchgeführt.

3. **Mitte:** Die Entfernung wird mit Hilfe des Ultraschallmoduls mit der Funktion `measureDistance()` gemessen. Ist die Distanz < 30 cm, wird die Navigation pausiert, solange der Fußball im Bild ist $\rightarrow$ Der Fußball wurde erreicht.

Solange der Server keine Erkennung meldet, wird durchgehend der Thread `Navigation()` ausgeführt. Die Ergebnisse der Evaluation dieses Vorgehens werden im nächsten Kapitel beschrieben.

5.2.5 Navigationsergebnisse

Da die Verarbeitung der Bilder auf dem Raspberry Pi 3 mit ca. 3 Sekunden pro Bild erfolgt und pro Auswertung i.d.R. mehrere Bildsegmente ausgewertet werden müssen (siehe Kapitel 4.4), würde das die Navigation stark verlangsamen. Deshalb wurde eine Lösung implementiert, bei der die Bilder über eine Client-Server-Lösung auf einen leistungsfähigen Rechner übertragen werden (siehe Kapitel 5.2.3). Dort erfolgt eine Auswertung der 12 Segmente im Bild in < 70 ms pro Segment, sodass eine hinreichend schnelle Auswertung eines Bildes für eine flüssige Navigation möglich ist. Wie zuvor beschrieben, erfolgt die Bewertung durch das Inception-Netz, auch wenn nur Teile des Fußballs im Bild sichtbar sind, sehr zuverlässig. Die Tests haben ergeben, dass der kleine Testfußball in einer Distanz von ca. 10 cm bis zu 2 m zuverlässig erkannt werden kann.

Die Erkennungsreichweite kann stark erhöht werden, indem die Bilder mit maximaler Auflösung aufgenommen und durch eine Segmentierung in kleinere Elemente unterteilt werden. Dies führt allerdings wiederum dazu, dass der Ball mit abnehmender Distanz immer schlechter erkannt wird, da die Segmente bei zunehmender Objektgröße keine Teilmuster des Balls mehr enthalten. Um dem Abhilfe zu schaffen, kann die Segmentierung und Auswertung wiederholt mit zu- oder abnehmender Segmentgröße durchgeführt werden, um sowohl nahe als auch weit entfernte Objekte zu erkennen. Dieses Vorgehen ist allerdings deutlich zeitaufwendiger als das hier verwendete Verfahren und würde die Navigation des Roboterfahrzeugs bis zum Erreichen des Objekts sehr langwierig gestalten.

6 Fazit

Ziel der Arbeit war zu zeigen, dass mit den in der Bilderkennung eingesetzten Methoden des Deep Learning in Deep Convolutional Neural Networks einem Roboterfahrzeug die Fähigkeit verliehen werden kann, anhand selbständig erlernter Muster von Gegenständen Objekte in seiner Umgebung, analog zu Menschen, visuell wahrzunehmen, sich an den Objekten im Raum zu orientieren und autonom zu navigieren. Dieses angestrebte Ziel konnte trotz gering vorhandener Ressourcen erfolgreich umgesetzt werden. Die einfach gehaltene praktische Anwendung kann nicht verbergen, welche ungeahnten Möglichkeiten sich mit Hilfe der Methoden des selbständigen Lernens in künstlichen neuronalen Netzen damit für neue Anwendungen eröffnen. Zudem stehen alle in der vorliegenden Arbeit untersuchten Modelle der Deep Convolutional Neural Networks in einem jährlich stattfindenden, weltweiten, öffentlichen Wettbewerb und werden wie in den vorangegangenen Jahren mit Sicherheit Jahr für Jahr bessere Leistungen aufzeigen. Für den kommenden Wettbewerb wurden bereits im Vorfeld einschneidende Verbesserungen in der Lernleistung von einem der Wettbewerber angekündigt.

Auch wenn, wie in der Arbeit gezeigt, die Geschwindigkeit des Lernens in künstlichen neuronalen Netzen aufgrund der zu verarbeitenden sehr großen Bilddatenmengen noch immer stark mit der Rechenleistung und Speicherkapazität der im Computer eingesetzten Grafikkarten korreliert, ist die unbestrittene Wirksamkeit der Konzepte und Methoden sowie ihre tatsächliche Umsetzung nachgewiesen.

Aufgrund der gegenseitigen Abhängigkeiten von Rechenleistung, Speicherkapazität und Zeitbedarf für das Training der neuronalen Netze bestand die Schwierigkeit, dass das im Rahmen der Arbeit eingesetzte Computersystem aufgrund geringer Rechenleistung und Kapazität von den zu bewältigenden Datenmengen bzw. Laufzeiten oftmals überfordert und abgebrochen wurde. Dennoch ist es durch geeignete Maßnahmen gelungen, das für diese Arbeit gesteckte Ziel in der zur Verfügung stehenden Zeit umzusetzen.

Die Auseinandersetzung mit den dieser Arbeit zugrunde liegenden Konzepten und Modellen des Deep Learning in der Bilderkennung, deren Erfolg wesentlich auf der Verwendung komplexer mathematischer Konzepte der linearen Algebra und Methoden der Statistik beruhen, waren eine Herausforderung und aufgrund der überzeugenden Ideen tief beeindruckend. Die einmal geschaffene Anwendung mit dem Roboterfahrzeug lässt sich für künftige Anwendungen sicherlich weiter ausbauen, woran weiterhin großes Interesse besteht.

Literaturverzeichnis

[1] Abadi, Martın ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu u. a.: *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015 2.3.4, 2.10

[2] Beiker, Sven A.: Legal aspects of autonomous driving. In: *Santa Clara L. Rev.* 52 (2012), S. 1145 1

[3] Benavidez, Patrick ; Jamshidi, Mo: Mobile robot navigation and target tracking system. In: *System of Systems Engineering (SoSE), 2011 6th International Conference on* IEEE, 2011, S. 299–304 1

[4] Benenson, Rodrigo: *Classification Datasets Results*. Version: 2016. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html, Abruf: Mai 2016 4.3

[5] Davson, Hugh: *Physiology of the Eye*. Elsevier, 2012 2.1

[6] Egmont-Petersen, Michael ; Ridder, Dick de ; Handels, Heinz: Image processing with neural networks—a review. In: *Pattern recognition* 35 (2002), Nr. 10, S. 2279–2301 2

[7] Ghorpade, Jayshree ; Parande, Jitendra ; Kulkarni, Madhura ; Bawaskar, Amit: GPGPU processing in CUDA architecture, 2012 4.1

[8] Giancoli, Douglas C.: *Physik: Lehr-und Übungsbuch*. Pearson Deutschland GmbH, 2010 5.2.1

[9] Glauner, Patrick O.: Deep Convolutional Neural Networks for Smile Recognition. In: *arXiv preprint arXiv:1508.06535* (2015) 2.6

[10] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. http://www.deeplearningbook.org. Version: April 2016. – Book in preparation for MIT Press 2.3

[11] Haykin, Simon ; Network, Neural: A comprehensive foundation. In: *Neural Networks* 2 (2004), Nr. 2004 1, 2

[12] Hebb, Donald O.: *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005 3.2

[13] Hinton, Geoffrey E.: Connectionist learning procedures, Elsevier, 1989, S. 185–234 2.2.2

[14] Hinton, Geoffrey E. ; Srivastava, Nitish ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan R.: Improving neural networks by preventing co-adaptation of feature detectors. In: *arXiv preprint arXiv:1207.0580* (2012) 2.3.3

[15] Konen, Wolfgang: *Vorlesung Bildverarbeitung und Algorithmen*. Version: 2006. http://ffb.uni-lueneburg.de/ffb-files/file/fuenfterteil.pdf, Abruf: Mai 2016 2.2, 2.1.1, 2.3

[16] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: Imagenet classification with deep convolutional neural networks. In: *Advances in neural information processing systems*, 2012, S. 1097–1105 1, 2.3.1, 3, 3.1, 3.1

[17] Lazar, Andreea ; Pipa, Gordon ; Triesch, Jochen: SORN: a self-organizing recurrent neural network. In: *Frontiers in computational neuroscience* 3 (2009), S. 23 2.2.1

[18] LeCun, Yann ; Bengio, Yoshua ; Hinton, Geoffrey: Deep learning, Nature Publishing Group, 2015, S. 436–444 2

[19] LeCun, Yann ; Boser, Bernhard ; Denker, John S. ; Henderson, Donnie ; Howard, Richard E. ; Hubbard, Wayne ; Jackel, Lawrence D.: Backpropagation applied to handwritten zip code recognition, MIT Press, 1989, S. 541–551 1, 2.3.1

[20] Lee, Honglak ; Grosse, Roger ; Ranganath, Rajesh ; Ng, Andrew Y.: Unsupervised learning of hierarchical representations with convolutional deep belief networks, ACM, 2011, S. 95–103 2.2.2

[21] McCulloch, Warren S. ; Pitts, Walter: A logical calculus of the ideas immanent in nervous activity. In: *The bulletin of mathematical biophysics* 5 (1943), Nr. 4, S. 115–133 1

[22] Merkert, Johannes: Ein künstliches neuronales Netz selbst gebaut. In: *CT Programmieren*, Heise Verlag, April 2016, S. 116–121 2.3.4

[23] Nair, Vinod ; Hinton, Geoffrey E.: Rectified linear units improve restricted boltzmann machines. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, S. 807–814 3.1

[24] Nielsen, Michael A.: *Neural Networks and Deep Learning*. Version: 2015. http://neuralnetworksanddeeplearning.com, Abruf: Mai 2016 4.3

[25] Rese, Mario: Logistische Regression. In: *Multivariate Analysemethoden*. Springer, 2000, S. 104–144 2.3.4

[26] Sathya, R ; Abraham, Annamma: Comparison of supervised and unsupervised learning algorithms for pattern classification, Citeseer, 2013, S. 34–38 2.2.1, 2.2.2, 2.2.2

[27] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent ; Van Den Driessche, George ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Panneershelvam, Veda ; Lanctot, Marc u. a.: Mastering the game of Go with deep neural networks and tree search. In: *Nature* 529 (2016), Nr. 7587, S. 484–489 2.2.2

[28] Srivastava, Nitish ; Hinton, Geoffrey ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan: Dropout: A simple way to prevent neural networks from overfitting, JMLR. org, 2014, S. 1929–1958 2.3.3, 2.3.3, 2.9

[29] Süsse, Herbert ; Rodner, Erik: *Bildverarbeitung und Objekterkennung*. Springer Verlag, 2014 2, 2.1.1, 2.1.1, 2.2

[30] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement learning: An introduction*. MIT Press, 1998 2.2.2

[31] Szegedy, Christian ; Liu, Wei ; Jia, Yangqing ; Sermanet, Pierre ; Reed, Scott ; Anguelov, Dragomir ; Erhan, Dumitru ; Vanhoucke, Vincent ; Rabinovich, Andrew: Going Deeper With Convolutions. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015 1, 3, 3.2, 4.3

[32] Szegedy, Christian ; Vanhoucke, Vincent ; Ioffe, Sergey ; Shlens, Jonathon ; Wojna, Zbigniew: Rethinking the Inception Architecture for Computer Vision. In: *arXiv preprint arXiv:1512.00567* (2015) 1, 3.2, 3.2

[33] Yegnanarayana, B: *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009 2.2.1

[34] Zeiler, Matthew D. ; Fergus, Rob: Visualizing and understanding convolutional networks. In: *Computer vision–ECCV 2014*, Springer Verlag, 2014, S. 818–833 2.3.1

Eidesstattliche Erklärung

Ich, Arno Fuhrmann, Matrikel-Nr. 974589, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Deep Learning in der Bilderkennung Steuerung autonomer mobiler Roboter mit Hilfe neuronaler Netze*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Bachelorarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in zweifacher Ausfertigung und gebunden im Prüfungsamt der Frankfurt University of Applied Sciences abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Frankfurt, den 31. Mai 2016

Arno Fuhrmann

