Masterthesis

Nao You See Me:
Echtzeitfähige Objektdetektion für mobile Roboter mittels Deep Learning

Bernd Poppinga

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 13. November 2018

Matrikelnummer: 2787416
Studiengang: Informatik
Fachbereich 3: Mathematik und Informatik

Datum: 13. November 2018

Erstgutachter: Dr. Tim Laue
Zweitgutachter: Prof. Dr. Ron Kikinis

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Frage, ob es möglich ist, einen Deep Learning basierten Objektdetektor zu entwickeln, der auf mobilen Robotern in Echtzeit läuft. Um dies zu überprüfen, wird ein Roboterdetektor für die Standard Platform League entwickelt. Dazu werden zunächst aktuelle Objektdetektoren analysiert, um sich einen Überblick über das Thema der Objektdetektion mittels Deep Learning zu verschaffen. Anschließend werden aktuelle Entwicklungen aus der Standard Platform League und Methoden für eine effiziente Inferenz betrachtet. Aus diesem Wissen wird dann ein eigenes Konzept für einen Objektdetektor abgeleitet, welcher im Anschluss so effizient gestaltet wird, dass er echtzeitfähig wird. Zusätzlich wird dieser noch um eine Distanzschätzung erweitert. Die Evaluation am Ende zeigt, dass der entwickelte Roboterdetektor überzeugende Ergebnisse erzielen und dennoch echtzeitfähig bleiben kann. Somit kommt diese Arbeit zu dem Ergebnis, dass es möglich ist, einen echtzeitfähigen Deep Learning basierten Objektdetektor für mobile Roboter zu entwickeln.

Abstract

This thesis analyzes whether it is possible to create a Deep Learning based object detector that runs in real-time on mobile robots. For that purpose, a robot detector for the Standard Platform League will be developed. In order to implement such a detector, first state-of-the-art object detectors will be analyzed to get an overview of the various aspects of object detection using deep learning. Following, recent developments in the Standard Platform League and methods for an efficient inference will be reviewed. Based on the knowledge gained, an own concept for an object detector will be deduced, which will be made so efficient that it becomes real-time capable. Furthermore, a distance-estimation will be added. The evaluation shows convincing results for the robot detector, which is still able to keep its real-time capability. Therefore, this work comes to the conclusion that it is possible to build a real-time capable object detection for mobile robots using Deep Learning.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Deep Learning ist heutzutage oft das Mittel der Wahl, wenn es um Objekterkennung geht. Es können drei Arten von Erkennung unterschieden werden: Klassifikation, Detektion und Segmentierung. Klassifikation stellt dabei im Allgemeinen das einfachste Problem dar, weil für das gesamte Bild lediglich eine Klasse bestimmt werden muss. Aus diesem Grund beginnt die Erfolgsgeschichte von Deep Learning zur Objekterkennung auch 2011 mit der Entwicklung von AlexNet [Krizhevsky et al., 2017], welches damals einen neuen Bestwert bei der Klassifikation des ImageNet-Datensatzes [Russakovsky et al., 2015] erzielen konnte. Die Objektdetektion ist insofern ein schwierigeres Problem, als das zur Klasse nunmehr auch Koordinaten bestimmt werden müssen und das gegebenenfalls mehr als einmal pro Bild. Im Bereich der Objektdetektion sorgte 2014 die Entwicklung von DeepFace [Taigman et al., 2014] für Aufsehen. Eng verbunden mit der Objektdetektion steht die Objektsegmentierung, bei der die exakte Kontur der Objekte gefunden wird. Nur ein Jahr nach DeepFace, 2015, erschien mit Segnet [Badrinarayanan et al., 2015] die erste Variante einer Segmentierung mit Deep Learning. Diese Nähe spiegelt sich auch in der aktuellen Entwicklung wieder, da sich zum Beispiel der Objektdetektor Faster R-CNN [Ren et al., 2015] zum Objektsegmentator Masked R-CNN [He et al., 2017] erweitern lässt, indem ein weiterer Output hinzugefügt wird.

Ein spannendes Beispiel für die Nutzung von Objekterkennung auf mobilen Robotern, stellt die Standard Platform League (SPL) des RoboCups (siehe Kapitel 2.3) dar. Ähnlich, wie bei Menschen, spielt die visuelle Wahrnehmung auch in der SPL eine übergeordnete Rolle. Die dort verwendeten Roboter nehmen ihre Umwelt fast ausschließlich über ihre Kameras wahr. Diese werden beispielsweise dazu verwendet, den Ball, Linien oder Hindernisse zu erkennen, wobei es wichtig ist, dass dies in Echtzeit passiert, da sich die Spielsituation schnell ändern kann. Dies geschah in der Vergangenheit meistens über klassische Objekterkennungsalgorithmen, welche unter anderem Hough-Transformationen und Musterabgleiche verwendeten [Röfer, Laue, Hasselbring et al., 2018]. Aktuell hält aber die Nutzung von Deep Learning Einzug in die SPL. Somit ist es mittlerweile fast Standard, dass die Ballerkennung über einen Objektklassifikator erfolgt, welcher neuronale Netze verwendet [Röfer, Laue, Hasselbring et al., 2018; Htwk, 2018]. Außerdem werden vermehrt Ansätze veröffentlicht, welche die Nutzung von Deep Learning auch in weiteren Anwendungsszenarien der SPL erforschen. Da sich bisher allerdings lediglich Klassifikatoren etabliert haben, wäre es der nächste Schritt, Objektdetektion mittels Deep Learning in der SPL einzuführen. Jedoch ist dieses Problem wesentlich komplexer und die Rechenleistung der Roboter ist stark begrenzt, weshalb zunächst Mittel und Wege gefunden werden müssen, um dies echtzeitfähig berechenbar zu machen.

Die Entwicklung von echtzeitfähigen Objektdetektoren wurde in den letzten Jahren stark vorangetrieben. Durch die Nutzung von Deep Learning sind Überwachungsszenarien, wie sie oft in Filmen und Serien dargestellt werden, keine reine Fiktion mehr (siehe Abbildung 1.1). Die aktuelle Forschung im Bereich der echtzeitfähigen Objektdetektoren macht es möglich, auf robuste Art und Weise Personen auf Bildern von Überwachungskameras zu erkennen. Außerdem kann der Objektdetektor YOLO v2 [Redmon und Farhadi, 2017] beispielsweise, über 9000 verschiedene Objektklassen identifizieren. Diese beeindruckenden Ergebnisse können allerdings nicht mit jedem System erzielt werden. Um Objektdetektoren, die dem Stand der Technik entsprechen, mit Bildraten zwischen 30 und 60 Bildern pro Sekunde betreiben zu können, braucht es Computersysteme mit viel Rechenleistung.

Solche Systeme stellen zwar bei stationären Anwendungen kein Problem dar, mobile Roboter können diese Leistung jedoch nicht aufbringen, da Grafikkarten zum einen viel Platz einnehmen und zum anderen die extreme Rechenleistung auf Kosten eines hohen Stromverbrauchs geht. Aus diesem Grund erforscht diese Arbeit, inwiefern es möglich ist, aktuelle Objektdetektionssysteme auf mobile Robotern zu portieren und dabei die Echtzeitfähigkeit zu erhalten. Um das Szenario möglichst realistisch zu gestalten, soll das

Abbildung 1.1: **Objektdetektion zur Überwachung.** Die Abbildung zeigt links die Anwendung des Objektdetektors YOLO v2 auf Kamerabildern in einer Fußgängerzone, dabei werden sowohl Personen, als auch andere Objekte, wie Fahrräder erkannt. Das rechte Bild zeigt einen Ausschnitt aus dem Trailer der 5. Staffel der Serie *Person of Interest* [IGN, 2016]. Dabei kommen die Ergebnisse des echten Detektors der Fiktion erstaunlich nah.

zu entwickelnde System im Rahmen der SPL des RoboCups getestet werden. Dort soll es zum ersten Mal Objektdetektion mittels Deep Learning ermöglichen. Als Testszenario wird dazu die Erkennung von anderen Robotern gewählt, da dies auch mit klassischen Methoden der Objekterkennung heute noch oft Schwierigkeiten bereitet.

1.2 Zielsetzung

Dem in der SPL verwendeten Roboter mit der Bezeichnung Nao steht lediglich ein Intel Atom Prozessor zu Verfügung, welcher selbstverständlich nicht dazu in der Lage ist, die gleiche Leistung wie aktuelle Grafikkarten zu vollbringen und vor allem auch nicht auf die Berechnung von neuronalen Netzen optimiert ist. Allerdings sind die Objektdetektoren aus dem Stand der Technik sehr allgemein gehalten. Das soll bedeuten, dass es unterschiedliche Kameras, Kameraperspektiven und Umgebungen in den Daten gibt. Dies stellt eine große Menge an Variationen dar, welche mitgelernt werden muss. Um den Detektor dennoch auf dem Nao echtzeitfähig zu gestalten, soll dieser zunächst für das Erkennen von Naos in einem typischen Szenario der SPL optimiert werden. Dadurch können viele Annahmen getroffen werden, welche das zu lösende Problem drastisch reduzieren. Dieses Prinzip hat in der Vergangenheit schon bei der Entwicklung des Ballklassifikators zum Erfolg geführt. Aus diesem Grund soll in dieser Arbeit nun die Frage untersucht werden, ob dies auch für Objektdetektoren möglich ist. Sofern die Erkennung der Roboter positive Ergebnisse liefern sollte, soll auch die Verwendung von weiteren Klassen, wie z.B. Pfosten, getestet werden. Zudem sollen die folgenden Bedingungen erfüllt werden.

Allgemeinheit Obwohl der Detektor für das Erkennen des humanoiden Roboters Nao optimiert werden soll, sollen keine Eingenschaften des Roboters verwendet werden, welche andere Objekte nicht hätten. Der 2018 veröffentlichte Orientierungsdetektor von B-Human [Mühlenbrock und Laue, 2018] nutzt beispielsweise aus, dass die Füße des Roboters eine bestimmte Form haben, welche es ermöglicht die Orientierung zu bestimmten. Solche Annahmen sollen vermieden werden. Durch diese Verallgemeinerung wird gewährleistet, dass der Detektor auch auf andere mobile Robotersysteme übertragbar bleibt. Die einzigen Eigenschaften, welche somit verwendet werden dürfen, sind die Größe und das Format des zu erkennenden Objektes, was genau den Eigenschaften entspricht, die einer Bounding Box zu entnehmen sind.

Echtzeitfähigkeit „Unter Echtzeit versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu

vorherbestimmten Zeitpunkten anfallen.“(DIN 44300) [„Echtzeit, Echtzeitsysteme, Echtzeitbetriebssysteme“ 2005]. Das bedeutet, dass das die Inferenz des neuronalen Netzes beendet sein muss, sobald das nächste Bild bereit ist. Ausgehend von natürlichen 60Hz dürfen pro Bild demnach 16ms verbraucht werden. Da auf dem Robotersystem allerdings noch weitere Prozesse, wie zum Beispiel der Ballerkenner laufen, soll sich die Laufzeit des Detektors auf maximal 7ms beschränken.

Robustheit Die Fähigkeit sich bewegen zu können, ist ein größer Vorteil mobiler Roboter. Allerdings entsteht aus Sicht der Bildverarbeitung auch ein großes Problem: die Bilder verschwimmen. Die Aufgabe der Roboterdetektion wäre wesentlich einfacher, wenn alle Bilder gestochen scharf wären. Da dies allerdings nicht der Fall ist, muss der Detektor auch mit verwackelten Bildern umgehen können. Ein weiteres Problem, welches speziell die SPL betrifft, sind die Beleuchtungsverhältnisse, da in der SPL vermehrt auf natürliche Beleuchtung gesetzt wird, welche mitunter sehr extrem sein kann. Deshalb soll der Detektor auch mit schlecht ausgeleuchteten Bildern umgehen können. In Abbildung 1.2 sind zwei Beispiele für solche Bilder zu sehen.

Distanz Da die Schüsse der Roboter derzeit ca. 6m weit gehen, sollten Roboter auch über diese Distanz gut erkannt werden können (ca. 70%). Auf der Hälfte dieser Entfernung wird jedoch vorausgesetzt, dass ca. 90% der Roboter erkannt werden können. Generell soll sich der Detektor im Zweifel zunächst auf die nahen Objekte konzentrieren, da diese wichtiger sind.

Präzision Die Erkennung von Robotern ist wichtig, aber ebenso wichtig ist die Vermeidung von *False Positives*. Die Präzision sollte unabhängig von der Entfernung niemals unter 80% fallen.

Um dies zu erreichen, wird zunächst der aktuelle Stand der Technik von Objektdetektoren untersucht. Des Weiteren wird der Stand der Technik für die Nutzung von Deep Learning in der SPL und für effizientere Inferenz von neuronalen Netzen analysiert. Aus den Ergebnissen soll zunächst ein eigenes Konzept für einen guten Objektdetektor für mobile Roboter abgeleitet werden, bei dem die Echtzeitfähigkeit zweitrangig ist. Dazu wird der Detektor modelliert, ein passendes Training entworfen und letztendlich wichtige Hyperparameter gefunden. Im Anschluss wird der Objektdetektor echtzeitfähig gemacht, indem die Methoden für eine effizientere Inferenz evaluiert und angewendet werden. Danach wird der Objektdetektor um eine Distanzschätzung erweitert. Zum Ende dieser Arbeit wird dann sowohl die Qualität des Objektdetektors, um sich abschließend ein Urteil über die Nutzung von Deep Learning zur Objektdetektion auf mobilen Robotern bilden zu können.

Abbildung 1.2: **Schwierige Bilder.** Die Abbildung zeigt links ein stark verschwommenes Bild aus einem Testspiel von B-Human und rechts ein Bild mit schlechten Lichtverhältnissen aus einem Datensatz der Nao Devils.

1.3 Was ist Deep Learning

An dieser Stelle ist es ein besonderes Interesse des Autors, noch einmal auf den Begriff Deep Learning einzugehen. Selbstverständlich hat jeder schon einmal etwas davon gehört, fängt man jedoch an nachzufragen, was dieses Deep Learning denn sei, dann bekommt man viele unterschiedliche Antworten. Gemeinsam haben diese Antworten, dass Deep Learning offenbar eng mit neuronalen Netzen verbunden ist. Aber wo enden neuronale Netze und wo fängt Deep Learning an oder ist es gar dasselbe?

Der Titel dieser Arbeit enthält mit Absicht diesen komplizierten Ausdruck und auch nicht ohne Grund wurde darauf verzichtet Deep Learning ins Deutsche zu übersetzen, da dieser Ausdruck mittlerweile mehr repräsentiert, als nur die Worte „tief “ und „Lernen“. Aus diesen Gründen wird an dieser Stelle, vor Beginn der eigentlichen Arbeit, der Ausdruck Deep Learning erklärt und vor allem zu anderen Begriffen abgegrenzt.

Um die Bedeutung des Ausdruckes aufzuklären, soll zunächst eine abstraktere Sicht auf neuronale Netze und maschinelles Lernen im Allgemeinen gefunden werden. Dabei ist die Vorstellung, dass ein Computer tatsächlich etwas lernt zwar nett abern sehr irreführend. Maschinelles Lernen ist nichts anderes als das Ausführen von Regressionen mit verschiedenen Methoden. Es wird im Endeffekt versucht eine unbekannte Funktion möglichst gut nachzuahmen. Dabei hat jede Methode eine bestimmte Menge an Funktionen, welche sie besonders gut darstellen kann. Die Stärken und Schwächen ergeben sich dabei entweder aus der Struktur selbst, aus dem Lernalgorithmus oder beidem. Neuronale Netze sind eine mögliche Art der Regression, um eine Funktion zu lernen. Das besondere an neuronalen Netzen ist, dass diese ursprünglich als allgemeines Berechenbarkeitsmodell entwickelt worden sind und somit in der Lage sind jede berechenbare Funktion abzubilden. Das soll heißen, wenn man eine Funktion f kennt, dann kann man sie durch ein neuronales Netz darstellen. Allerdings heißt das nicht, dass man die Funktion f lernen kann. Und genau hier kommt der Begriff Deep Learning zum Einsatz.

In der Einleitung zu ihrem Buch *Deep Learning* [I. Goodfellow et al., 2016] beschreiben I. Goodfellow et al. Deep Learning als Hierarchie von Konzepten bzw. Features, welche als Graph betrachtet tief ist. Dabei scheint die hierarchische Featureextraktion, wie sie beim Deep Learning vorkommt, das Erlernen von Funktionen mittels Backpropagation einfacher zu gestalten. Durch geschicktes Gestalten der Struktur des Netzes wird also versucht, das Lernproblem so umzuformulieren, dass hierarchische Features gelernt werden. Dabei ist die Tiefe des Netzes kein hinreichendes Kriterium für Deep Learning, sondern ein notwendiges. Da neuronale Netze so vielseitig sind, können aber auch tiefe Netze eine unpassende Lösung lernen. Es handelt sich also um ein wackeliges Gebilde, welches behutsam aufgebaut werden muss. Dazu zählt zum einen die Definition des Problems und die Struktur des Netzes aber zum anderen auch der Optimierer und die Lossfunktion. Wenn alles zusammenpasst, werden unter Umständen hierarchische Features gefunden und somit Deep Learning betrieben. Deshalb ist die Frage „Was ist Deep Learning?“ weniger eine Frage, wie man lernt, sondern viel mehr was man gelernt hat. Das bedeutet anderseits auch, dass Deep Learning keinesfalls auf neuronale Netze beschränkt ist, sondern auch mit anderen Methoden ausgeführt werden kann, sofern diese hierarchisch angeordnet werden können.

2 Grundlagen

Im Folgenden werden die Grundlagen erklärt, welche für das Verständnis dieser Arbeit benötigt werden. Dabei wird zunächst auf die Begrifflichkeiten der Objekterkennung eingegangen. Anschließend werden wichtige Metriken erklärt und die Standard Platform League vorgestellt. Zum Schluss wird auf neuronale Netze eingegangen, wobei insbesondere auf die genutzten Layer erklärt werden. Im weiteren Verlauf der Arbeit werden weitere Grundlagen speziellerer Themen vermittelt, welche an dieser Stelle allerdings aus dem Kontext gegriffen wären und deshalb zu einem späteren Zeitpunkt vorgestellt werden.

2.1 Objekterkennung

Objekterkennung wird im Rahmen dieser Arbeit als Oberbegriff für die drei Verfahren Objektklassifikation, Objektdetektion und Objektsegmentierung verwendet. Nachfolgend werden diese drei Begriffe und deren Zusammenhänge erklärt. Abbildung 2.1 zeigt schematisch, welche Informationen das jeweilige Verfahren aus einem Bild extrahieren kann.

Objektklassifikation Die Objektklassifikation reduziert ein gesamtes Bild auf eine Klasse C_{image}. Dadurch erhält man zum Beispiel die Information, dass sich ein bestimmtes Objekt im Bild befindet, allerdings nicht wo genau. Aus diesem Grund nutzt beispielsweise der Ballerkenner von B-Human [Röfer, Laue, Hasselbring et al., 2018] eine Nachverarbeitung, welche die genaue Position des Balles im klassifizierten Patch bestimmt. Insgesamt gehen bei der Klassifizierung sehr viele Informationen verloren, jedoch macht gerade dies die Klassifikation sehr robust und einfach zu lernen, da die Klasse des Bildes nicht von einzelnen Pixeln abhängt.

Objektdetektion Die Objektdetektion muss pro Bild für jedes relevante Objekt eine Klasse bestimmen und zusätzlich noch die Ausmaße der Bounding Box bestimmen. Dadurch erhält man nun zusätzlich noch die Information, in welchem Bereich sich ein Objekt befindet und wie viele Objekte es tatsächlich gibt. Durch unendlich oft wiederholte Anwendung eines Objektklassifikators auf jeweils unterschiedlichen Ausschnitten des originalen

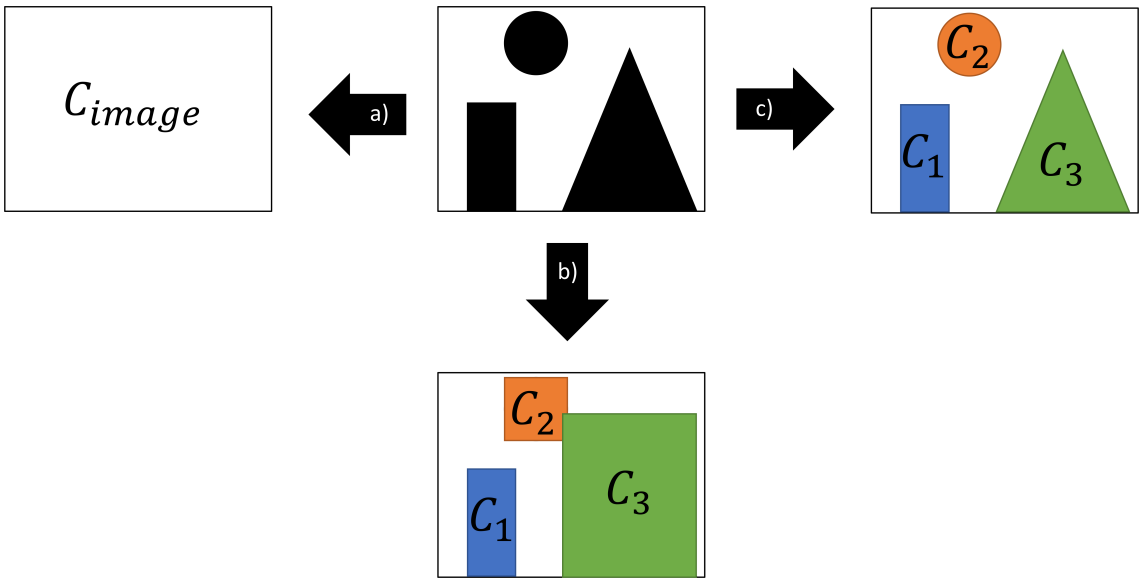


Abbildung 2.1: **Übersicht Objekterkennung.** C_{image} bezeichnet die Klasse des Bildes. C_i bezeichnet die Klasse von Objekt i wobei $i \in \{1, 2, 3\}$. a) zeigt die Anwendung eines Klassifikators. b) zeigt die Anwendung eines Detektors. c) zeigt die Anwendung eines Segmentators. Es ist zu sehen, dass von a bis c immer mehr Informationen des Bildes enthüllt werden, sodass das Ergebnis der Segmentierung aussieht wie das Original.

Bildes, kann ein Klassifikator als Detektor verwendet werden. In der Praxis hat es sich als nützlich erwiesen, einen Klassifikator nicht unendlich oft anzuwenden, sondern in einem ersten Schritt Kandidaten zu suchen und diese in einem zweiten Schritt zu klassifizieren. Dieses Verfahren wird als zweistufige Detektion bezeichnet und wird so ebenfalls beim Ballerkenner von B-Human [Röfer, Laue, Hasselbring et al., 2018] verwendet. Umgekehrt kann das Ergebnis des Detektors direkt für die Klassifikation verwendet werden, indem man dem Bild einfach die Klasse mit der höchsten Resonanz zuordnet. Die erhöhte Komplexität der Detektion macht es nicht nur schwieriger dieses Problem zu lernen, sondern bereits die Modellierung der Bounding Boxes mit ihren Klassen wird hierbei zu einem wichtigen Baustein. Zudem erfordert die nötige Regression für die Boxen mehr Ressourcen, da nicht mehr nur die Grenzen der Klassen relevant sind, sondern der genaue Wert.

Objektsegmentierung Die Objektsegmentierung muss für jedes Pixel eines Bildes eine Klasse bestimmen und somit die Objektzugehörigkeit. Dadurch erhält man nun nicht mehr nur die Bounding Box des Objektes, sondern dessen exakte Kontur, sofern diese nicht verdeckt ist. Implizit lässt sich somit aus der Anzahl der verschiedenen Klassen auch die Anzahl der Objekte bestimmen. Durch die unendlich oft wiederholte Anwendung eines Objektdetektors auf unterschiedliche Ausschnitte einer Szene kann ein Detektor als Segmentator verwendet werden. Andersherum kann das Ergebnis einer Segmentierung direkt als Detektion verwendet werden, indem man die Bounding Boxes so baut, dass sie alle Pixel einer Klasse umschließen. Genau dieses Verfahren wird auch im AutoLabeler in Kapitel 5.1 verwendet, um Trainingsdaten für einen Detektor aus einer Segmentierung zu erhalten. Im Gegensatz zur Objektdetektion fällt hier zwar die aufwendige Regression der Bounding Boxes weg, dafür müssen nun wesentlich mehr Klassen gefunden werden. Außerdem sind die zu findenden Klassen schwieriger zu unterscheiden, da gerade an den Objektgrenzen nicht klar ist, zu welchem Objekt ein Pixel gehört.

2.2 Metriken

Intersection over Union Die Intersection over Union (IoU) ist eine Metrik, welche die Übereinstimmung von Mengen betrachtet. Das beste Ergebnis von 1 wird erzielt, wenn beide Mengen gleich sind. Im schlechtesten Fall sind beide Mengen disjunkt, dann beträgt die IoU 0. Im Kontext von Objektdetektoren kann die IoU genutzt werden, um die Güte von Bounding Boxes zu bestimmen. Dazu betrachtet man die umschlossenen Pixel der Box als Menge. Dann vergleicht man die umschlossenen Pixel der Groundtruthbox mit denen der vorhergesagten Box und erhält somit den IoU-Wert. Abbildung 2.2 zeigt zwei Beispielrechnungen für den IoU-Wert bei der Roboterdetektion.

Konfusionsmatrix Grundlage der meisten Metriken für die Klassifikation bildet die Konfusionsmatrix. Sie enthält Angaben darüber, welche Klassen für die Elemente jeder Klasse vorhergesagt wurden. Im einfachsten Fall gibt es zwei Klassen: positiv und negativ. Die dazugehörige Konfusionsmatrix (Abbildung 2.3) enthält vier Eintrage, aus denen dann Metriken gebaut werden können.

Die im Folgenden aufgeführten Metriken können direkt aus den Einträgen der Konfusionsmatrix berechnet werden. Um die Lesbarkeit zu steigern werden hier und im weiteren Verlauf die folgenden Abkürzungen verwendet: True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN).

- Die **Accuracy**(Genauigkeit) gibt an, wie viel Prozent der vorhergesagten Daten richtig gelabelt worden sind.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{2.1}$$

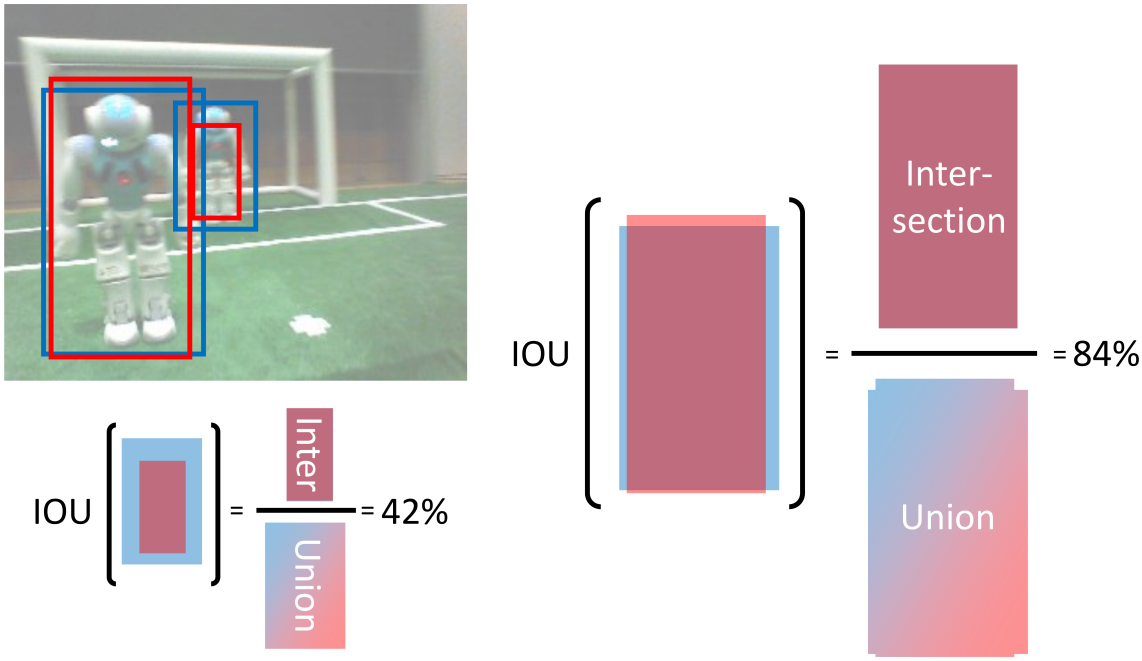


Abbildung 2.2: **IoU Beispiel.** Die Abbildung zeigt die Berechnung der IoU für die beiden im Bild oben links gezeigten Bounding Boxes. Die Größen der Boxen in der Rechnung entsprechen genau denen aus dem Bild. Es ist zu sehen, dass obwohl im kleinen Beispiel die Ground-Truth-Box die Vorhersage komplett umschließt, nur ein geringer IoU Wert erzielt wird, da die Schnittmenge nur sehr klein ist. Im großen Beispiel sind sich die beiden Boxen viel ähnlicher, weswegen ein höherer Wert erzielt werden kann.

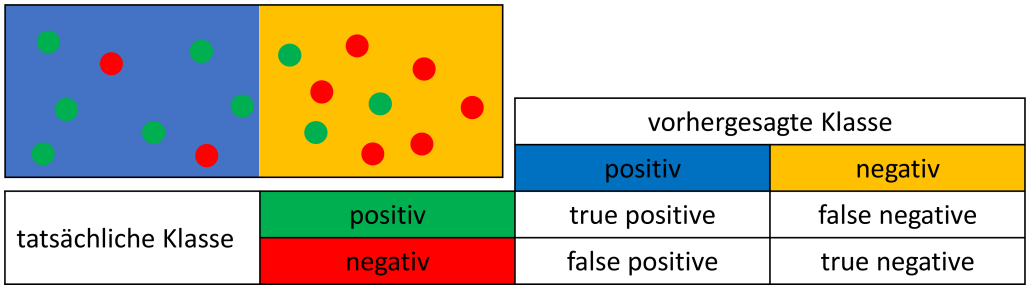


Abbildung 2.3: **Konfusionsmatrix.** Die Abbildung zeigt die Konfusionsmatrix für den binären Fall sowie ein Beispiel zur Veranschaulichung (oben links). Die Farben aus dem Beispiel entsprechen, den passenden Einträgen aus der Matrix.

- Die **Balanced Accuracy** erweitert die Accuracy um einen Ausgleich der Klassen, da ein großes Klassenungleichgewicht das Ergebnis der Accuracy verfälscht.

$$\text{Balanced Accuracy} = 0,5 \cdot \frac{\text{TP}}{\text{TP} + \text{FN}} + 0,5 \cdot \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{2.2}$$

- **Recall** betrachtet das Verhältnis zwischen den positiven Beispielen, welche auch positiv gelabelt worden sind, und der Anzahl aller tatsächlichen positiven Beispiele.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.3}$$

- **Precision** repräsentiert wie viel Prozent der positiv gelabelten Beispiele tatsächlich positiv sind.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.4}$$

- Mit $\mathbf{F_1}$ **Score** wird der harmonische Mittelwert von Recall und Precision bezeichnet.

$$\mathrm{F_1\ Score} = 2 \cdot \frac{\mathrm{Recall} \cdot \mathrm{Precision}}{\mathrm{Recall} + \mathrm{Precision}} \tag{2.5}$$

Mean Average Precision Mean Average Precision (MAP) bezeichnet eine Metrik, welche zum Standard bei der Bewertung von Objektdetektoren geworden ist. Dabei beachtet MAP nicht nur die Precision und Recall Werte, sondern auch die Reihenfolge, in der richtige und falsche Vorhersagen vorliegen. Um dies zu erreichen, werden zunächst alle Daten nach ihrer vorhergesagten Wahrscheinlichkeit, dass sie ein bestimmtes Objekt enthalten, sortiert. Im besten Fall sind die Vorhersagen nun so sortiert, dass zuerst alle tatsächlichen Objekte kommen und dann der Rest. Dieser Fall ist in Abbildung 2.4 b) repräsentiert und würde, sofern tatsächlich alle Objekte gefunden wurden, einen MAP-Wert von $1,0$ erhalten, was den höchsten MAP-Wert darstellt. Der minimale MAP-Wert von $0,0$ kann erreicht werden, falls kein Objekt gefunden wird. Der in Abbildung 2.4 a) gezeigte Fall ist ein wenig komplizierter zu berechnen und soll im Folgenden verwendet werden, um die Berechnung eines nicht trivialen MAP-Wertes zu zeigen. Nachdem die Vorhersagen sortiert worden sind, können der Recall- und Precisiongraph erzeugt werden. Dazu berechnet man, beginnend mit der Vorhersage mit der höchsten Wahrscheinlichkeit, iterativ den aktuellen Recall und Precision Wert über eine sich nach rechts ausbreitende Menge. Für das Beispiel a) ergeben sich dadurch die in Gleichung 2.6 und 2.7 dargestellten Werte, wobei angenommen wird, dass alle Objekte gefunden worden sind.

$$\text{Recall: } (0,2; 0,2; 0,4; 0,6; 0,6; 0,8; 0,8; 1.0; 1.0; 1.0) \tag{2.6}$$

$$\text{Precision: } (1,0; 0,5; 0,67; 0,75; 0,6; 0,67; 0,57; 0,63; 0,56; 0,5) \tag{2.7}$$

Mithilfe der so erhaltenen Werte kann über Gleichung 2.8 der MAP-Wert berechnet werden. Dabei ist zu beachten, dass der Begriff *Mean* sich auf unterschiedliche Klassen bezieht. Das bedeutet für den dargestellten Fall, sind Mean Average Precision und Average Precision identisch. Sollte man mehr als eine Klasse haben, dann berechnet man zunächst die AP für jede Klasse, um dann den Mittelwert über alle Klassen zu bilden. Dabei bezeichnet C die Menge aller Klassen und die Funktion *maxPrecision* liefert die maximale Precision aus der Menge aller Vorhersagen, bei denen der Recall über dem übergebenen Wert liegt oder diesem gleicht.

$$\mathrm{MAP} = \frac{\sum_{c \in C} AP(c)}{|C|} \text{ mit} \tag{2.8}$$

$$\mathrm{AP} = \frac{1}{11} \sum_{i=0}^{10} \mathrm{maxPrecision}\left(\frac{i}{10}\right) \tag{2.9}$$

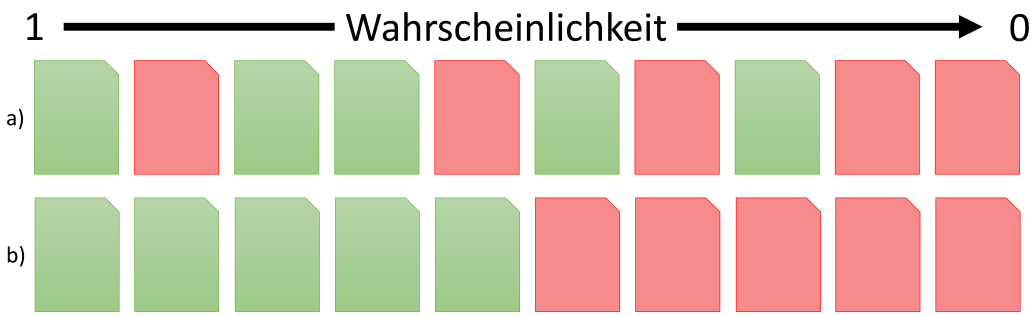


Abbildung 2.4: **Mean Average Precision Beispiel.** Zu sehen sind zwei Beispiele für die Berechnung des MAP-Wertes. Die einzelnen Vorhersagen (grün: richtig, rot:falsch) sind bereits sortiert. Es wurden jeweils alle fünf positiven Daten gefunden. Die resultierenden MAP-Werte sind a) 0,78 und b) 1.

Für das Beispiel a) ergibt dies die in 2.10 dargestellte Rechnung und somit einen MAP-Wert von 0,78.

$$\frac{1,0+1,0+1,0+0,75+0,75+0,75+0,75+0,67+0,67+0,63+0,63}{11} = 0,78 \tag{2.10}$$

2.3 Standard Platform League

Die SPL stellt eine Liga des RoboCups dar, in welcher Teams aus der ganzen Welt gegeneinander Fußball spielen. Dabei spielen sie nicht selbst, sondern sie programmieren einen standardisierten Roboter, sodass dieser Fußball spielen kann. Für die alljährlich ausgetragene Weltmeisterschaft erscheint jedes Jahr ein aktualisiertes Regelwerk, welches zunehmend höhere Ansprüche an die Teams stellt. Die folgenden Daten stammen aus dem Regelwerk des Jahres 2018 [RoboCup Technical Committee, 2018]. In einem Spiel selbst können aktuell bis zu fünf Roboter pro Team aufgestellt werden, welche untereinander kommunizieren können. Bei dem standardisierten Roboter handelt es sich um den *Nao* von *Softbank Robotics* (früher Aldebaran), welcher seit 2008 verwendet wird.

Spielfeld Das Spielfeld der SPL (Abbildung 2.5) besteht aus grünem Kunstrasen auf dem sich Linien befinden. Es ist 9m lang und 6m breit, wobei die grüne Fläche außerhalb der Linien noch 70cm weitergeht. Auf dem Spielfeld befinden sich neben den Naos noch der Schiedsrichter, der Ball und die beiden Tore. Außerhalb der grünen Fläche können beliebige Gegenstände vorkommen.

Nao Der Nao ist ein 57, 4cm großer humanoider Roboter, welcher nicht speziell für das Fußballspielen entwickelt wurde, sondern als allgemeiner Partner für alle Lebenslagen. So wird auf der Internetseite von *Softbank Robotics* damit geworben, dass der Nao für die Therapie von Autismus, an der Rezeption von Hotels sowie bei Tanzchoreographien genutzt werden kann. Darüber hinaus sind viele weitere Szenarien denkbar. In der 2018 verwendeten Version 5 besitzt der Nao 25 Freiheitsgrade, damit dieser sich möglichst menschlich verhalten kann. Im Kopf des Naos sind zwei Kameras verbaut, wovon sich eine in der Stirn befindet und geradeaus schaut. Die zweite Kamera befindet sich im vermeintlichen Mund des Roboters und schaut 39.7° nach unten. Beide Kameras sind ansonsten identisch. Sie haben einen horizontalen Öffnungswinkel von 60.97° und einen vertikalen Öffnungswinkel von 47.64°.

Des Weiteren dient ein Intel ATOM Z530 als CPU und der Arbeitsspeicher beläuft sich auf 1GB. Als Speicher stehen eine 8GB große microSDHC-Karte sowie 2GB Flashspeicher zur Verfügung.

Abbildung 2.5: **Spielelemente der Standard Platform League.** links: Übersicht des Spielfeldes der SPL. rechts: Zu sehen ist der offizielle Spielball der SPL vor dem Standardroboter ohne Trikot. In dieser Grafik ist der Roboter mit hellgrauen Akzenten präsentiert, allerdings sind auch weitere Farben, wie grau, rot, blau und orange, erlaubt. Beide Grafiken sind übernommen aus dem offiziellen Regelwerk 2018 [RoboCup Technical Committee, 2018].

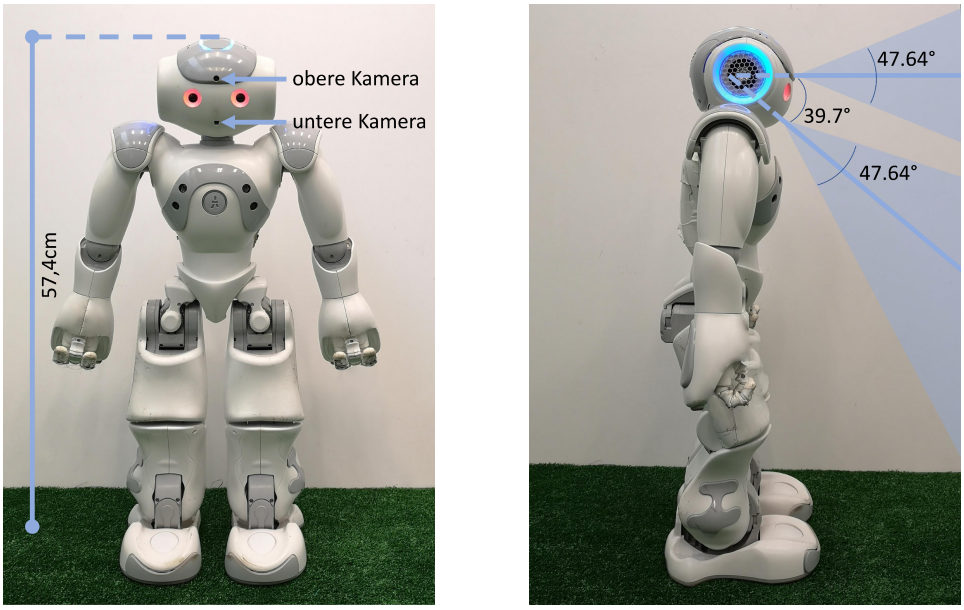


Abbildung 2.6: **Kameras des Naos.** links: Die Größe des Naos sowie die Position der beiden Kameras sind eingezeichnet. rechts: Die vertikalen Öffnungswinkel sowie der Winkel zwischen den beiden Kameras ist eingezeichnet.

2.4 Neuronale Netze

Im Rahmen dieser Arbeit wird grundlegendes Wissen über Inferenz und Training von neuronalen Netzen vorausgesetzt. Dieses kann ansonsten in [I. Goodfellow et al., 2016] nachgeschaut werden. Im Folgenden werden zunächst die in dieser Arbeit verwenden Layer erklärt. Anschließend wird auf die Python-Bibliothek Keras mit Tensorflow-Backend und die B-Human-Toolchain für neuronale Netze eingegangen.

Layer Neuronale Netze, wie sie im Kontext dieser Arbeit verwendet werden, sind lineare Abfolgen unterschiedlicher Layer, die eine Eingabe Stück für Stück zur Ausgabe umformen. Jeder Layer hat dabei seine Art der Berechnung, mit der bestimmte Anforderungen an die Eingabe einhergehen. Nachfolgend wird für die wichtigsten Layer dieser Arbeit erklärt, wie diese funktionieren und welche Parameter sie haben. Dabei wird davon ausgegangen, dass die Eingabe und Ausgabe eines jeden Layer aus n Feature Maps bestehen, welche jeweils zweidimensional sind und eine Auflösung von hxw Pixeln haben.

- Der **Convolutional-Layer** (Conv-Layer) (Abbildung 2.7) stellt den wichtigsten Layer dieser Arbeit dar, weil er für die Extraktion der Features zuständig ist. Wie der Name schon sagt, werden bei der Berechnung Faltungen ausgeführt. Die Faltungsmatrix wird dabei Filter(-kernel) genannt und besitzt drei Dimensionen, da es n zweidimensionale Feature Maps gibt. Durch die Anzahl der Feature Maps der Eingabe wird eine Dimension des Kernels bereits festgelegt, da immer über alle Feature Maps gleichzeitig gefaltet wird. Die anderen beiden können frei gewählt werden, wobei in der Praxis meistens $3 \times 3 \times n$ Filter verwendet werden. Für die Berechnung der Ausgabe wird der Filterkernel mit seinem Zentrum über alle $h \times w$ Pixel der Eingabe geschoben, wobei jeweils die Faltung ausgeführt wird. Normalerweise wird dieser dabei um jeweils ein Pixel verschoben, allerdings ist es auch möglich, diesen um mehrere Pixel zu verschieben. Dadurch verkleinert sich die Ausgabe, da weniger Werte berechnet werden. Die Schrittgröße wird als Strides bezeichnet und wird mit einem Wert für die x-Richtung und einem für die y-Richtung angegeben. Wenn die Faltung auf Pixel in der Nähe des Randes angewendet wird, kann es vorkommen, dass ein Teil des Kernels außerhalb des Bildes ist. Im Rahmen dieser Arbeit wird dies Problem dadurch gelöst, dass für Pixel außerhalb des Bildes der Wert 0 angenommen wird. Dieses Verfahren wird same-Padding genannt, da dadurch gewährleistet wird,

dass die Größe des Bildes gleich bleibt, sofern keine Strides verwendet werden, welche größer sind als 1. Die Alternative hierzu ist valid-Padding, welches die Faltung nur auf Pixel anwendet, bei denen der Kernel vollständig innerhalb des Bildes ist, wodurch die Ausgabe jedoch kleiner wird. Am Ende der Berechnung kann ein Wert, welcher Bias genannt wird, auf das Ergebnis addiert werden. Danach wird noch eine Aktivierungsfunktion auf das Ergebnis angewendet.

- **Batch Normalization** normalisiert die Eingabe nach Gleichung 2.11. Dabei werden γ und β vor dem Training gesetzt, während μ und σ gelernt werden. Dadurch, dass die Eingabe immer wieder verändert wird, hat der Layer nicht nur einen normalisierenden Effekt, sondern auch einen regularisierenden, weil der nachfolgende Layer immer auf die veränderte Eingabe reagieren muss.

$$\text{BatchNormalization}(x) = \frac{\gamma(x - \mu)}{\sigma} + \beta \tag{2.11}$$

- Der **Max-Pooling-Layer** (MaxPool-Layer) wird verwendet, um die Größe der Eingabe zu reduzieren. Dabei wird ähnlich zur Faltung ein Kernel über das Bild geschoben. Hierbei wird allerdings kein gewichteter Mittelwert aller Pixel unter dem Kernel, sondern lediglich das Maximum zurückgegeben. Normalerweise wird ein 2x2 Kernel genutzt. Um das Bild nun zu verkleinern, wird der Kernel immer um 2 Schritte verschoben, was einem (2,2) Stride entspricht. Dadurch wird die Bildgröße auf ein Viertel reduziert.

- Das **Zero-Padding** (ZeroPad) fügt oben, unten, links oder rechts Reihen bzw. Spalten von Nullen hinzu. Es wird im Kontext dieser Arbeit verwendet, um Eingaben mit ungeraden Abmessungen gerade zu machen, damit diese ohne Probleme halbiert werden können.

Keras Die *Python* Bibliothek Keras [Chollet et al., 2015] stellt ein leicht zu bedienendes Werkzeug dar, um neuronale Netze zu modellieren und zu trainieren. Das Modell steht dabei im Mittelpunkt eines jeden Projektes. Ein Modell definiert zunächst die Form der Daten, welche an das Netz übergeben werden sollen, in einem sogenannten Input. Nach dem Input können sich beliebige Abfolgen von Layern reihen, wobei jeweils die Form der Eingaben und Ausgaben jeden Layers beachtet werden muss. Sofern die Struktur des Netzes lediglich lineare Abfolgen vorsieht, kann das *Sequentiel*-Modell genutzt werden, mit dem man einfach Layer hintereinander bauen kann. Allerdings können über das allgemeine Modell nahezu beliebige Strukturen umgesetzt werden.

Steht die Struktur des Netzes, so muss es noch kompiliert werden. Beim Kompilieren wird das Modell auf das Training vorbereitet, wobei unter anderem die Anzahl der zu lernenden Parameter erfasst wird. Des Weiteren müssen beim Kompilieren der Optimierer, die Loss-Funktion sowie sonstige Metriken übergeben werden.

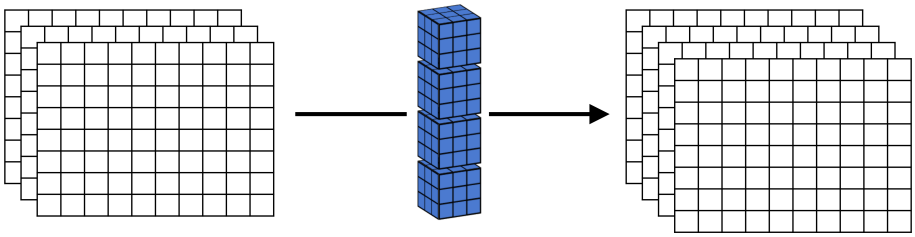


Abbildung 2.7: **Funktionsweise des Convolutional-Layers.** In der Mitte sind in blau vier 3x3x3 Filterkernel dargestellt. Die letzte Dimension ist dabei durch die drei Eingabe-Feature Maps (links) bestimmt. Da es vier Filter gibt, beträgt die Anzahl der Ausgabe-Feature Maps (rechts) ebenfalls vier.

Sobald dies geschehen ist, kann das Training über die Nutzung einer Variante der fit-Funktion begonnen werden. Dazu müssen sowohl die Trainingsbeispiele (x-Daten), als auch die gewünschten Ausgaben (y-Daten) zu diesen Beispielen bekannt sein. Diese müssen allerdings nicht statisch sein, sondern können über die Nutzung eines Generators dynamisch zur Laufzeit erzeugt werden, wodurch Speicherplatz eingespart werden kann. Zusätzlich können für das Training Callback-Funktionen übergeben werden, welche zum Beispiel das Modell abspeichern, wenn ein neues Minimum erreicht wird. Nach dem Training können über die Verwendung der Evaluationsfunktion die genauen Resultate für einen bestimmten Datensatz berechnet werden.

Für die Repräsentation der Daten und deren Berechnungen können in Keras verschiedene Backends gewählt werden. In dieser Arbeit wird das aktuell standardmäßig gesetzte Tensorflow-Backend genutzt. An den meisten Stellen im Code fällt es allerdings nicht auf, welches Backend verwendet wird, da Keras so implementiert ist, dass man das Backend wechseln kann, ohne den Code ändern zu müssen. Sollen allerdings komplexe Operationen auf Tensoren ausgeführt werden, so lässt sich die Verwendung eines spezifischen Backends kaum vermeiden. Für die Nutzung von Tensorflow [Martín Abadi et al., 2015] muss dabei eigentlich beachtet werden, dass man einen *Flow* erzeugt. Das bedeutet, dass die Eingabe solange durchgereicht und dabei umgebaut wird, bis das Ende der Berechnung erreicht ist. So kann auch im Nachhinein noch genauestens nachvollzogen werden, wie das Ergebnis zustande gekommen ist.

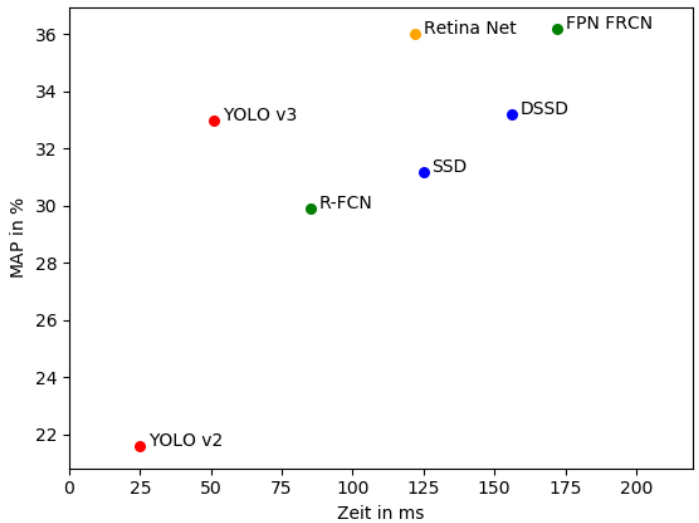
B-Human-Toolchain für neuronale Netze Um neuronale Netze effizient auf dem Nao ausführen zu können, hat B-Human eine eigene Toolchain für deren Inferenz entwickelt [Röfer, Laue, Hasselbring et al., 2018]. Da Bibliotheken, wie z.B. Keras, bereits sehr viele Funktionen mitbringen, die das Training erleichtern und optimieren, wurde sich dazu entschieden, das Training in Keras auszuführen und lediglich die fertig trainierten Modelle so zu exportieren, dass sie anschließend in dem C++-Code auf dem Nao wieder importiert werden können. Die Grundidee des Frameworks ist es dabei, dass die für die Inferenz benötigten Rechenoperation zu Beginn einmal in einen großen Block x86 Maschinencode übersetzt werden. Das Framework ist aktuell für kleinere Netze, wie den B-Human Ballerkenner [Röfer, Laue, Hasselbring et al., 2018], ausgelegt und kann bis zu 24 Layer optimieren, allerdings sind auch größere Modelle möglich.

3 Stand der Technik

3.1 Aktuelle Objektdetektoren

Die Forschung an echtzeitfähigen Objektdetektoren ist sehr aktuell. Aus diesem Grund werden immer wieder neue Varianten und Optimierungen veröffentlicht. Zudem wird die Entwicklung durch immer wieder veranstaltete Wettbewerbe gezielt gefördert. Um die Ansätze miteinander vergleichen zu können, wurden standardisierte Datensätze veröffentlicht, welche mittels einer festgelegten Bewertungsfunktion verglichen werden können, wobei es sich meistens um eine Variante der MAP (siehe Kapitel 2.2) handelt. Die beiden populärsten Datensätze, die den Standard beim Vergleich von Objektdetektoren darstellen, sind Pascal VOC 2007 [Everingham et al., 2010] und Microsoft COCO [T. Y. Lin, Maire et al., 2014]. Im Folgenden sollen Detektoren vorgestellt werden, welche gute Ergebnisse auf diesen Datensätzen erzielen konnten, da sie interessante Ansätze verwenden, welche bei der Entwicklung eines neuen Detektors im Rahmen dieser Arbeit hilfreich sein können. Kapitel 3.1 zeigt übersichtlich die Ergebnisse der unten aufgeführten Detektoren.

Faster R-CNN Veröffentlicht im Jahre 2015, stellte Faster Region based Convolutional Neural Network (R-CNN) bis 2017 das Maß aller Dinge dar, wenn man die Genauigkeit von Objektdetektoren betrachtet. Das liegt vor allem daran, dass Faster R-CNN, im Gegensatz zu allen anderen Detektoren, welche im Folgenden präsentiert werden, ein zweistufiger Detektor ist. Zweistufige Detektoren sind allgemein für ihre überlegene Genauigkeit bekannt, während sie bei der Geschwindigkeit große Schwächen aufweisen. Sie wählen in der ersten Stufe zunächst eine Menge von Boxen aus, in denen unter Umständen ein Objekt sein könnte, welche in der zweiten Stufe dann weiter klassifiziert werden. Dadurch werden die einfachen negativen Beispiele bereits in der ersten Stufe aussortiert, sodass sich der Detektor im zweiten Schritt vollkommen auf die komplizierten Exemplare konzentrieren kann. Zwar verwendet auch Faster R-CNN Mechanismen, welche ihn schneller werden lassen, wie zum Beispiel der Einsatz von *Region of Interest (RoI)-Pooling* und *Feature Sharing*: Dabei werden die Ausschnitte der Feature Maps der ersten Stufe auf eine einheitliche Größe gebracht, um diese im zweiten Schritt klassifizieren zu können, ohne dass man die Features neu berechnen müsste. Leider erreicht dieser sehr gut durchdachte Ansatz dennoch nur rund 7 Bilder pro Sekunde. Jedoch hat er das Prinzip der Anchor-Boxen eingeführt, welche auch von vielen einstufigen Detektoren genutzt werden. Nennenswerte Weiterentwicklun-



Detektor	MAP	Zeit
YOLO v2	21,6%	25 ms
YOLO v3	33,0%	51 ms
R-FCN	29,9%	85 ms
RetinaNet	36,0%	122 ms
SSD	31,2%	125 ms
DSSD	33,2%	156 ms
FPN FRCN	36,2%	172 ms

Abbildung 3.1: **Übersicht aktueller Objektdetektoren.** Es ist die Performanz der Objektdetektoren, welche in diesem Kapitel erwähnt werden, dargestellt. gleiche Farben in der Grafik repräsentieren, dass diesen Detektoren ein ähnliches Prinzip zugrunde liegt.

gen dieses Detektors sind zum einen das Feature Pyramid Network (FPN)[T. Y. Lin, Dollár et al., 2017], welches die Grundstruktur um ein neuartiges Pyramidenschema erweitert. Zum anderen erweitert Region-based Fully Convolutional Network (R-FCN)[Dai et al., 2016] die Architektur um *Position-Sensitive Score Maps*, wodurch vor allem eine höhere Geschwindigkeit erzielt werden kann.

YOLO: You Only Look Once Der Objektdetektor You Only Look Once (YOLO) [Redmon, Divvala et al., 2015; Redmon und Farhadi, 2017; Redmon und Farhadi, 2018] wurde über insgesamt drei Paper sukzessiv optimiert. Während andere Detektoren versuchen, ein ausgewogenes Verhältnis zwischen Geschwindigkeit und Genauigkeit zu erzielen, wobei mal die eine und mal die andere Seite schwerer gewichtet wird, zielt YOLO hauptsächlich darauf ab, möglichst schnell zu sein, worunter die Genauigkeit leidet. Als einer der ersten Detektoren hat YOLO ein einstufiges System präsentiert, um schnell und verlässlich Objekte zu erkennen. Zwar wurde bei der ersten Version noch auf einen *Fully-Connected* Ansatz gesetzt, um die Bounding Boxen vorherzusagen, doch schon ab der zweiten Version wurde auf Anchor Boxen gewechselt. Nach und nach wurden Neuentwicklungen in das bestehende System eingepflegt und die Performance stetig verbessert, was dazu geführt hat, dass YOLO aktuell einen der besten Objektdetektoren darstellt, welcher zudem wesentlich schneller ist, als die Konkurrenz.

SSD: Single Shot MultiBox Detector Die Veröffentlichung des Single Shot MultiBox Detectors (SSDs)[Liu et al., 2016] geschah nach dem ersten YOLO-Paper, in 2016. Der Ansatz stellt eine Alternative zum YOLO Detektor dar, welcher die Konzepte von Faster-R-CNN und YOLO aufgreift und sehr gut verständlich seine Entwicklungen erklärt. Bei SSD werden zum einen das erste Mal Anchor-Boxen mit einstufigen Erkennern kombiniert. Außerdem wird *Hard-Negative-Mining* angewendet, um dem Problem der vielen einfach negativen Boxen entgegen zu wirken, und es wird eine exzessive Augmentation proponiert. Die präsentierte Netzarchitektur verwendet das erste Mal eine Art Pyramidenschema, um auch kleinere Objekte gut erkennen zu können. Mit dem Deconvolutional Single Shot Detector (DSSD)[Fu et al., 2017] wurde zudem eine Variante des SSDs veröffentlicht, welche auf *Deconvolution* in Kombination mit *Skip-Layern* setzt, wodurch eine bessere Genauigkeit auf Kosten der Inferenzzeit erreicht wird.

Retina Net (Focal Loss) Das Paper *Focal Loss for Dense Object Detection* von T.-Y. Lin et al. [T.-Y. Lin et al., 2017] beschäftigt sich unter anderem mit der Diskrepanz zwischen einstufigen und zweistufigen Objektdetektoren. Da bis zum Zeitpunkt der Veröffentlichung der beste MAP-Wert von einem zweistufigen Erkenner erzielt worden ist, während einstufige Detektoren eine höhere Geschwindigkeit vorweisen können, wurde nach einem Verfahren gesucht, welches die einstufigen Erkenner auf das Niveau der zweistufigen anheben kann. Ihr Lösungsvorschlag ist das sogenannte *Focal Loss*, welches dem starken Ungleichgewicht zwischen den Klassen entgegenwirken soll. Um dies zu erreichen, werden Boxen, welche sehr einfach zu klassifizieren sind, weniger stark gewichtet, als ihre schwierigen Pendants, wodurch sich das neuronale Netz auf diese fokussieren kann. Des Weiteren wird in diesem Paper eine neue Architektur vorgestellt, welche zum einen das neuartige Focal Loss und zum anderen ein Pyramidenschema verwendet.

3.2 Deep Learning in der Standard Platform League

Deep Learning ist auch in der SPL ein aktuelles Forschungsgebiet, in dem derzeit viele Ansätze getestet werden. Am etabliertesten ist die Ballerkennung über neuronale Netze. Aber auch andere Aufgaben, wie Spielerkennung oder Segmentierung des Bildes werden erforscht. Im Folgenden soll ein Überblick über den aktuellen Stand der Technik geschafft werden. Dabei wird auch auf die Humanoid League geschaut, da sie sehr ähnlich zu der SPL ist. Der große Unterschied zwischen der SPL und der Humanoid League ist allerdings,

dass Letztgenannte selbstgebaute Roboter verwendet und somit jedes Team einen eigenes Robotermodell hat, welches zusätzlich zum Code entwickelt wird.

Ballerkennung Durch die Einführung eines schwarzweißen Balles im Jahre 2016 wurden die Teams vor eine neue Herausforderung gestellt. Zuvor wurde ein leuchtend orangefarbener Ball verwendet, dessen Erkennung aufgrund der Farbe relativ einfach war. Dies hat einige Teams dazu bewegt, Ansätze zu evaluieren, die auf Deep Learning basieren. Ein weiterer Grund für diese Entwicklung ist die stetige Anpassung der Regeln im Hinblick auf natürlicheres Licht, welches zuvor stark normiert war, sodass man sich gut darauf einstellen konnte.

Am bekanntesten ist die Ballerkennung des *Nao Teams HTWK* [Htwk, 2018], welche 2018 veröffentlicht wurde. Diese verwenden eine Patch Extraktion, die Bälle, Füße von Robotern und Torpfosten finden soll. Anschließend klassifiziert ein neuronales Netz unter Verwendung von Convolution und Max-Pooling diese Patches.

Des Weiteren hat auch B-Human in ihrem aktuellen Code-Release [Röfer, Laue, Hasselbring et al., 2018] einen Ballerkenner präsentiert, welcher ausgewählte Patches mithilfe von neuronalen Netzen klassifiziert. Da sich der Ball allerdings nicht immer in der Mitte der Patches befindet, gibt es eine Nachbearbeitung, welche mittels einer Hough-Transformation den Ball im Patch sucht. Außerdem wurde eine eigens entwickelte Toolchain mit veröffentlicht, welche es ermöglicht, die Inferenzzeit so stark zu reduzieren, dass der Deep Learning Ballerkenner, sogar schneller ist, als die vorherige Variante ohne Deep Learning.

In [Menashe et al., 2017] wird ein ähnlicher Ansatz verwendet, welcher allerdings mehr Aufwand in die Vorverarbeitung investiert, bevor dann eine abstrahierte Feature Map an den Klassifikator übergeben wird. Dabei zeigten die Experimente, dass eine Support Vector Machine mindestens genauso gut funktioniert wie ein neuronales Netz. Aufgrund dieser Tatsache und der exzessiven Feature Erzeugung vor der Übergabe an den Klassifikator darf hier bezweifelt werden, das überhaupt Deep Learning zum Einsatz kommt. Dies ändert allerdings nichts an den überzeugenden Ergebnissen, die dieser Erkenner liefert.

Auch in der Humanoid League, welche handelsübliche Fußbälle verwendet, werden neuronale Netze zur Erkennung von Bällen verwendet. In [Speck et al., 2017] wurde ein Ballerkenner entwickelt, welcher eine Heatmap über das gesamte Bild berechnet. Somit passt dieser in keine der zuvor beschriebenen Klassen, jedoch lässt sich der Output für die Entwicklung eines Detektors nutzen. Das dort verwendete neuronale Netz nutzt das gesamte Bild als Input und gibt für jede Reihe und jede Spalte einen Wert zurück, welcher beschreibt, ob sich dort ein Ball befinden könnte. Durch die Kombination dieser Informationen, kann die Ballposition in X- und Y-Position bestimmt werden. Die Inferenzzeit dieses Ansatzes ist nicht beschrieben, jedoch wurden schon in [Cruz et al., 2017] Zweifel an der Echtzeitfähigkeit eines solchen Systemes auf einem Nao geäußert.

Robotererkennung Wie bereits im vorherigen Abschnitt beschrieben, verwendet das *Nao Team HTWK* ihren Klassifikator auch zur Erkennung von Roboterfüßen. Aus den erkannten Füßen werden anschließend Roboter modelliert.

Allerdings gibt es auch Ansätze, welche den Roboter als Ganzes erkennen. Hier ist als erstes ein Erkenner des SPL-Teams SPQR aus dem Jahr 2017 zu nennen. Dieser verwendet zunächst klassische Bildverarbeitungsmethoden, um Kandidaten zu finden, welche dann mittels Deep Learning klassifiziert werden. Dabei wird eine Genauigkeit von 100% bei 14-22 Bildern pro Sekunde auf dem SPQR Datensatz erreicht, welcher mit dem Erkenner zusammen veröffentlicht worden ist.

Ein anderes Verfahren, welches von dem SPL Team *UChile* entwickelt worden ist [Cruz et al., 2017], verwendet ein von *B-Human* entwickeltes Verfahren zur Findung von Kandidaten und klassifiziert diese dann mithilfe spezieller Netzarchitekturen wie SqueezeNet [Iandola et al., 2016] und XNOR-Net [Rastegari et al., 2016]. Sie erreichen dabei eine Infe-

renzzeit von 0, 95ms beim dortigen Favoriten XNOR-Net, welcher zudem eine Genauigkeit von 96,6% erreicht.

Auch wenn die Humanoid League andere Roboter als den Nao verwenden, so entsprechen die Roboter der *KidSize* Klasse von der Größer her ungefähr dem Nao der SPL. Von daher lohnt sich auch hier ein Blick auf einen Ansatz dieser Liga. In [Javadi et al., 2017] werden verschiedene Netzarchitekturen auf verschiedenen System auf ihre Echtzeitfähigkeit und die dabei erzielten Ergebnisse untersucht. Dabei wird das neuronale Netz wieder als Klassifikator verwendet, nachdem zuvor Kandidaten aus einem segmentierten Bild entnommen worden sind. Die Resultate auf einem Prozessor, welcher dem des Nao ähnlich ist, zeigen, dass das LeNet mit 4,17ms zwar gute Laufzeiten besitzt, allerdings sind die Ergebnisse der Erkennung nicht gut (0,48 F_1-Score). Des Weiteren wurde auch hier das SqueezeNet verwendet, welches zwar einen sehr guten F_1-Score von 0,98 erzielen kann, aber mit 173,33ms Laufzeit wesentlich langsamer ist. Dabei beschreiben die Zeiten Berechnungszeit für einen Kandidaten.

3.3 Effiziente Inferenz neuronaler Netze

Isoliert von der Aufgabe der Objekterkennung, gibt es allgemeine Verfahren, um die Performanz von neuronalen Netzen zu verbessern. Dazu wird zunächst auf zwei Verfahren eingegangen, die bereits im vorherigen Abschnitt erwähnt wurden. Danach werden noch zwei weitere Verfahren präsentiert.

XNOR-Net Ein Ansatz um die Inferenzzeit von neuronalen Netzen zu verbessern, ist die Quantisierung. Dabei wird auf die hohe Präzision der Gleitkommazahlen verzichtet und es werden stattdessen weniger genaue Repräsentationen verwendet. XNOR-Net [Rastegari et al., 2016] treibt dies auf die Spitze, indem nur noch die beiden Zahlenwerte $+1$ und -1 verwendet werden. Dies gilt sowohl für die Eingänge von Layern, als auch für die dort verwendeten Gewichte. Die Genauigkeit sinkt dabei allerdings um ca. 10% im Falle von AlexNet, dafür ist eine Beschleunigung um das 58-fache möglich.

SqueezeNet Das in [Iandola et al., 2016] vorgestellte SqueezeNet zielt darauf ab, den Speicherverbrauch des bekannten AlexNet [Krizhevsky et al., 2017] drastisch zu reduzieren, um es für Anwendungen mit begrenzten Speicherressourcen, wie FPGAs, nutzbar zu machen. Um dies zu realisieren werden drei Strategien verfolgt:

1. Ersetzung von 3x3 Filtern durch 1x1 Filtern ⇒ Reduzierung der Parameter
2. Reduzierung von Eingangskanälen für 3x3 Filter ⇒ Reduzierung der Parameter
3. Verzögerung des Downsamplings auf tiefere Layer ⇒ Erhalt der ursprünglichen Performance

Um dies umzusetzen, wurde ein Modul für neuronale Netze entwickelt, welches diese Kriterien erfüllt. Dieses sogenannte *Fire-Module* besteht aus zwei Teilen: einem Squeeze-Layer und einem Expand-Layer. Das Squeeze-Layer besteht ausschließlich aus 1x1 Filtern, während das Expand-Layer einen Mix aus 1x1 und 3x3 Filtern enthält. Dabei wird darauf geachtet, dass die Anzahl der Filter im Squeeze-Layer die Summe an Filtern im Expand-Layer nicht überschreitet. Beiden Untermodulen ist dabei eine Rectified Linear Unit (ReLU) Aktivierung nachgeschaltet.

Das, durch den Einsatz von Fire Modulen entstandene, SqueezeNet kann die Parameter im Vergleich mit AlexNet um das 50-fache reduzieren und den Speicherbedarf des Modells auf unter 1MB absenken. Auf eine Reduzierung der Inferenzzeit wird nicht eingegangen, allerdings ist es vorstellbar, dass sich die Reduzierung der Parameter auch positiv auf diese auswirkt, sofern die genutzte Verlagerung des Downsamplings dies nicht komplett aufwiegt.

Separierbare Convolution Ein anderer Ansatz, welcher die Nutzung von 1x1 Convolutions motiviert, sind die *MobileNets* [Howard und Wang, 2017]. Durch die Verwendung der Separierbarkeitseigenschaft von Filtern kann die Anzahl der benötigten Rechenoperationen drastisch reduziert werden. Bei der Nutzung von Conv-Layern kommen normalerweise dreidimensionale Filter zum Einsatz (X,Y,Anzahl der Feature Maps). Bei der separierbaren Convolution wird hingegen zunächst je ein zweidimensionaler Filter in X-,Y-Dimension auf jede Feature Map angewendet. Im Anschluss werden die neuentstandenen Feature Maps dann zu gewichteten Summen addiert, wobei die Anzahl der Output Feature Maps der Anzahl der verschiedenen Summen entspricht. Dieser Wert kann über die Nutzung des Depth Mutlipliers reguliert werden.

Pruning Eine andere Möglichkeit die Inferenz effizienter zu gestalten, ist das Wegschneiden von unwichtigen Filtern oder anderen unwichtigen Verbindungen. Dieses Ausdünnen wird Pruning genannt. In [Molchanov et al., 2016] wird ein iteratives Verfahren für das Pruning beschrieben, welches vorsieht Layer bzw. Filter zu entfernen, anschließend Feintuning zu betreiben und dann wieder Elemente zu entfernen bis die gewünschte Größe erreicht ist. Eine wichtige Frage dabei ist, welches Element man als nächstes entfernt. Um dies herauszufinden, werden verschiedene Verfahren getestet. Das Ergebnis dieser Tests ist, dass das Taylor Verfahren am längsten eine hohe Genauigkeit beibehalten kann. Aber auch *Optimal Brain Damage* und das *Activation* basierte Pruning erzielen gute Resultate. Die Experimente zeigen zudem, wie wichtig das Feintuning zwischen den Pruningiterationen ist, da dies die Genauigkeit wieder stark anhebt, nachdem sie zuvor stetig gefallen ist.

4 Aufbau des neuronalen Netzes

Nachdem im vorherigen Kapitel verschiedene Objekterkenner präsentiert worden sind, soll daraus nun ein Objektdetektor für die Problemstellung dieser Arbeit abgeleitet werden. Dazu wird in diesem Kapitel eine Pipeline entwickelt, welche alle Schritte beinhaltet, die für Detektion von Objekten auf Bildern benötigt werden. Im nachfolgenden Kapitel wird dann darauf eingegangen, wie der entwickelte Objektdetektor trainiert werden kann.

Neben einer Grundstruktur wird der zu entwickelnde Detektor einige unterschiedliche Konfigurationen besitzen, bei denen nicht direkt zu erkennen ist, welche die beste Lösung ist. Diese unterschiedlichen Konfigurationen werden Kapitel 6 evaluiert. Da es sehr aufwändig ist, alle möglichen Konfigurationen zu testen, wird in diesem Kapitel zusätzlich ein Ursprungsmodellmodell entworfen, welches später iterativ mit der besten Konfiguration ausgestattet werden kann.

4.1 Bildvorbereitung

Die obere Kamera des Naos nimmt Bilder mit einer Auflösung von 480x640 Pixeln im YUV Farbraum auf. Die folgenden drei Schritte bereiten das Bild auf die Übergabe an das neuronale Netz vor.

Farbraumkonvertierung Die Ballerkenner der SPL haben bereits gezeigt, dass die Nutzung eines einzigen Kanales ausreicht, um Bälle zu klassifizieren. Durch diese Arbeiten motiviert, soll auch hier lediglich der Y-Kanal des YUV-Bildes verwendet werden.

Normalisierung Da die Helligkeit der Bilder, welche an das Netz übergeben werden, mitunter stark variiert, soll eine einfache Normalisierung einen Teil der Varianz entfernen. Dazu wird zunächst das Histogramm gebildet und dann jeweils die ersten 2% an beiden Seiten entfernt, damit die Normalisierung robuster wird. Danach wird eine Min-Max-Normalisierung auf dem neuen Intervall angewendet. Weitere Normalisierungen sind im eigentlichen Netz zu finden, wo allerdings auf Basis der Batch-Normalization-Layer normalisiert wird, während diese Normalisierung davon unabhängig ist.

Größenanpassung In dem letzten Schritt der Vorverarbeitung werden die Bilder auf die Eingangsgröße des neuronalen Netzes skaliert. Bei der Skalierung soll ein Verfahren verwendet werden, welches dem aus dem B-Human System möglichst ähnelt, da der Detektor später dort laufen soll. Aus diesem Grund werden Pixelgruppen gebildet und dann deren Mittelwert verwendet, sodass quasi ein Average Pooling stattfindet. Die Objektdetektoren aus dem Stand der Technik verwenden Auflösungen wie zum Beispiel 256x256 (YOLO v3), 300x300 (SSD) oder 500x500 (RetinaNet) Pixel, während die Ballerkenner 32x32 (B-Human) oder 20x20 (HTWK) Pixel nutzen. Um die richtige Größe für die Eingangsbilder zu finden, lohnt sich ein Blick in das Code Release von B-Human [Röfer, Laue, Hasselbring et al., 2018]. Dort gibt es das Thumbnailbild, welches eine Auflösung von 80x60 besitzt. Diese Auflösung bietet gleich mehrere Vorteile: Sie ist zum einen sehr klein im Gegensatz zur vollen Auflösung von 640x480, zum anderen ist die Größe exakt dreimal halbiert, was eine schnelle Berechnung ermöglicht. Des Weiteren wird diese Auflösung zur späteren Analyse der Spiele genutzt, was bedeutet, dass man noch Objekte wie Roboter gut erkennen muss. Da dieses Bild sowieso berechnet werden muss, bietet es sich an, dieses direkt als Input für das neuronale Netz zu nutzen, da somit die Zeit für das Berechnen des Bildes gespart werden kann. Allerdings lässt sich die Größe des Bildes leicht über einen Parameter anpassen, sodass weitere Formate denkbar sind, solange das Ursprungsmodellbild um eine Zweierpotenz in beiden Dimensionen verkleinert wird.

In Abbildung 4.1 ist beispielhaft eine Szene aus einem Testspiel bei verschiedenen Auflösungen präsentiert. Es ist zu sehen, dass 80x60 Pixel die letzte Auflösung darstellt,

Abbildung 4.1: **Vergleich verschiedener Auflösungen.** Die sechs Grafiken zeigen eine Szene aus einem Testspiel bei verschiedenen Auflösungen. Die Auflösungen werden ausgehend von 640x480 oben links nach unten rechts immer wieder in beiden Dimensionen halbiert.

bei der noch alle Roboter zu erkennen sind. Das bedeutet, dass im Ursprungsmodellmodell zunächst von dieser Auflösung ausgegangen wird. Dennoch sollen auch die beiden benachbarten Auflösungen evaluiert werden.

4.2 Bounding Box Vorhersage

Als nächstes wird das eigentliche neuronale Netz übersprungen, um sich zunächst einmal Gedanken um die Ausgabe des Detektors zu machen. Die Vorhersage der Position des Objektes soll über sogenannte Anchor Boxen geschehen, da diese die aktuell vorherrschende Methode sind, um Bounding Boxes zu bestimmten und in fast allen vorgestellten Objektdetektoren verwendet werden. Eine Anchor Box ist ein Prototyp für eine Bounding Box, welche dem Algorithmus ein Gefühl dafür gibt, wie groß die gesuchten Objekte sind. Dadurch muss man nicht mehr direkt die Parameter der Bounding Box lernen, sondern lediglich einen Offset zu der Anchor Box. Eine Anchor Box ist dabei an einen Pixel gebunden und kann nur innerhalb dies Pixels bewegt werden, weshalb es für jedes Pixel im letzten Layer des neuronalen Netzes eigene Anchor Boxen gibt. Da nicht alle Objekte die gleichen Ausmaße haben, wird für jedes Pixel im letzten Layer des neuronalen Netzes eine festgelegte Anzahl von Anchor Boxes bestimmt. Im Rahmen dieser Arbeit werden vier Anchor Boxen pro Pixel verwendet, damit nicht zu viele Parameter bestimmt werden müssen. Jede Anchor Box besitzt eine andere Grundform (Abbildung 4.2), sodass sich bestimmte Anchor Boxes auf spezielle Objektgrößen/-formen spezialisieren können. Um das Lernproblem zu erleichtern, sollen jedoch auch diese Offset-Parameter nicht direkt gelernt werden, sondern auf den Wertebereich einer Sigmoid-Funktion gemappt werden. Für die x,y Position werden Werte zwischen 0 und 1 bestimmt, welche die relative Position innerhalb dieses Pixels darstellen. Die Ausmaße werden ebenfalls über Werte zwischen 0 und 1 berechnet, welche mit 10 multipliziert einen Offset-Faktor ergeben (Abbildung 4.3) (siehe Gleichung 4.1 bis 4.4). Da in der Literatur für das letzte Layer meistens Größen zwischen 8 und 13 Pixeln gewählt worden sind, soll in dieser Arbeit eine Auflösung von 10x8 Pixeln verwendet werden. Dies passt von der Größenordnung zu den Varianten aus der Literatur und stellt im Falle von 80x60 Pixeln ungefähr ein Achtel der Ursprungsgröße dar, sodass man diese Größe mithilfe von Pooling gut erreichen kann.

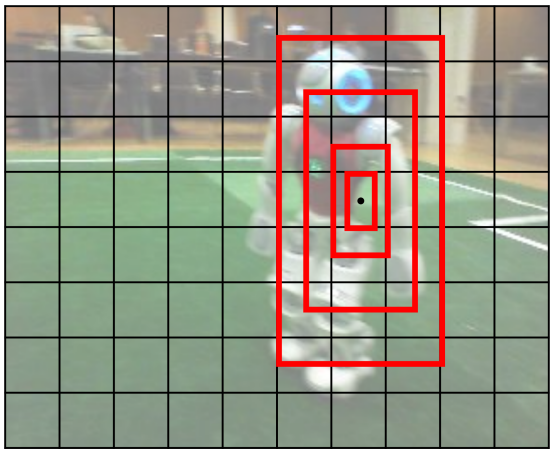


Abbildung 4.2: **Prinzip von Anchor Boxes.** Das schwarze Gitter repräsentiert die finale Auflösung von 10x8 Pixeln, bei der die Anchor Boxes bestimmt werden sollen. In Rot sind die vier Anchor Boxes dargestellt, welche es für jedes Kästchen gibt. Der schwarze Punkt markiert das Kästchen, zu dem die eingezeichneten Boxen gehören. Innerhalb dieses Kästchens können die Boxen verschoben werden.

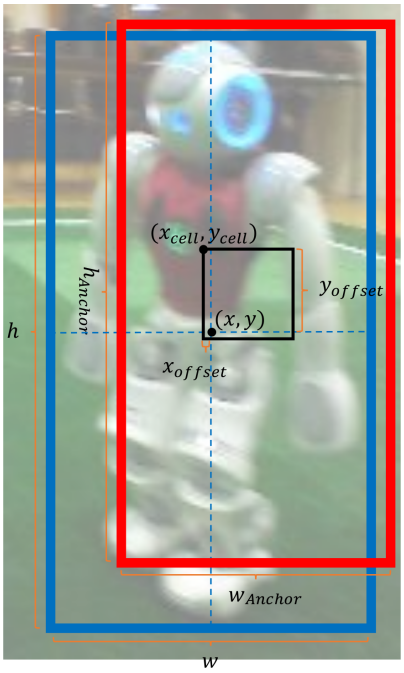


Abbildung 4.3: **Berechnung von Bounding Boxes aus Anchor Boxes.** Die rote Box repräsentiert eine Anchor Box und die blaue die tatsächliche Bounding Box des Roboters. Die schwarze Box stellt das zugehörige Pixel dar, in dem die Anchor Box verschoben werden kann.

$$x = x_{offset} + x_{cell} = sigmoid(x_{pred}) + x_{cell} \tag{4.1}$$
$$y = y_{offset} + y_{cell} = sigmoid(y_{pred}) + y_{cell} \tag{4.2}$$
$$h = sigmoid(h_{pred}) \cdot 10 \cdot h_{anchor} \tag{4.3}$$
$$w = sigmoid(w_{pred}) \cdot 10 \cdot w_{anchor} \tag{4.4}$$

4.3 Neuronales Netz

Aufgrund der Tatsache, dass es aktuell noch keinen vergleichbaren Ansatz gibt, da die Detektoren entweder zweistufig und auf Roboter spezialisiert sind oder viel zu rechenaufwändig sind, soll ein komplett neues Netz entworfen werden. Dies hat in der Vergangenheit bereits bei den Ballerkennern der SPL für gute Ergebnisse gesorgt, da man auch auf die speziellen Anforderungen eingehen kann.

Framework Bevor man mit dem Designen beginnen kann, muss man sich überlegen, welches Framework man verwenden möchte, um die neuronalen Netze zu trainieren und später zu berechnen. Dafür soll im weiteren Verlauf dieser Arbeit die Toolchain von B-Human verwendet werden. Dieses nutzt zunächst alle Vorteile von Keras in Python, um auf einfache Art und Weise das Training betreiben zu können. Anschließend können die gelernten Modelle exportiert und in C++ eingelesen werden. Dort werden die einzelnen Layer mittels hardwarenaher Programmierung stark optimiert, um eine möglichst schnelle Inferenz zu ermöglichen. Leider ist der Funktionsumfang der C++ Schnittstelle noch nicht besonders ausgereift, jedoch sollten mit Convolution, Batch-Normalization, Dense und

Pooling die wichtigsten Layer vertreten sein. Aufgrund der übersichtlichen Codebasis lässt sich das Repertoire allerdings auch gut erweitern. Die Art und Weise wie die neuronalen Netze dort berechnet werden, sorgt dafür, dass die Filter besonders gut in Vierer-Gruppen berechnet werden können, wovon bis zu sechs Stück pro Durchgang berechnet werden können. Dadurch bietet es sich an, eine Filteranzahl von 24 zu verwenden.

Feature-Extraktion Damit man am Ende die Bounding Boxes bestimmen kann, müssen zunächst Features des Eingabebildes extrahiert werden. Hier setzen alle modernen Ansätze auf eine Kombination aus Convolution und Downscaling. Um die Anzahl der möglichen Architekturen einzuschränken, werden im Folgenden zwei Module beschrieben, aus denen sich das Netz zusammensetzt (Abbildung 4.4). Dadurch wird eine grobe Struktur vorgegeben, und gleichzeitig innerhalb des Moduls Freiheiten erlaubt. Dazu wird für jedes Modul ein Baukasten definiert, welcher die verschiedenen Strukturen der einzelnen Module beschreibt. Diese einzelnen Konfigurationen werden dann in Kapitel 6 evaluiert, um einen guten Objektdetektor zu finden. Da es sehr aufwändig ist, alle möglichen Kombinationen zu testen, soll zunächst ein Ursprungsmodell gefunden werden, welches dann sukzessiv verbessert werden kann.

In [Redmon und Farhadi, 2017] wird unter anderem vorgeschlagen, zu jedem Convolutional-Layer ein Batch-Normalization-Layer einzubauen. Dieses Schema soll für beide Module angewendet werden. Für die Wahl der Aktivierungsfunktion sollen sowohl die ReLU-Funktion, als auch die LeakyReLU getestet werden, wobei das Ursprungssmodell zunächst die normale ReLU verwenden soll.

- **Feature-Modul:**
  Die Aufgabe des Feature-Moduls ist es die Features auf einer Ebene zu extrahieren. Den Kern dieses Moduls stellen somit 3x3 Convolutions dar, welche standardmäßig für diese Aufgabe verwendet werden. Es können allerdings mehrere Conv-Layer aufeinander folgen und auch die passende Anzahl an Filtern muss noch gefunden werden. Daher wird jede Konfiguration des Feature-Moduls über ein Tupel (Gleichung 4.5) beschrieben. Für das Ursprungsmodell wird zunächst nur eine Convolution mit 24 Filtern verwendet, da dies in dem genutzten Framework ein gutes Kosten/Nutzen Verhältnis aufweisen sollte.

$$\text{Konfiguration}_{\text{Feature}} = (\text{Anzahl Layer}, \text{Anzahl Filter}) \in \mathbb{N}_{>0} \times \mathbb{N}_{>0} \tag{4.5}$$

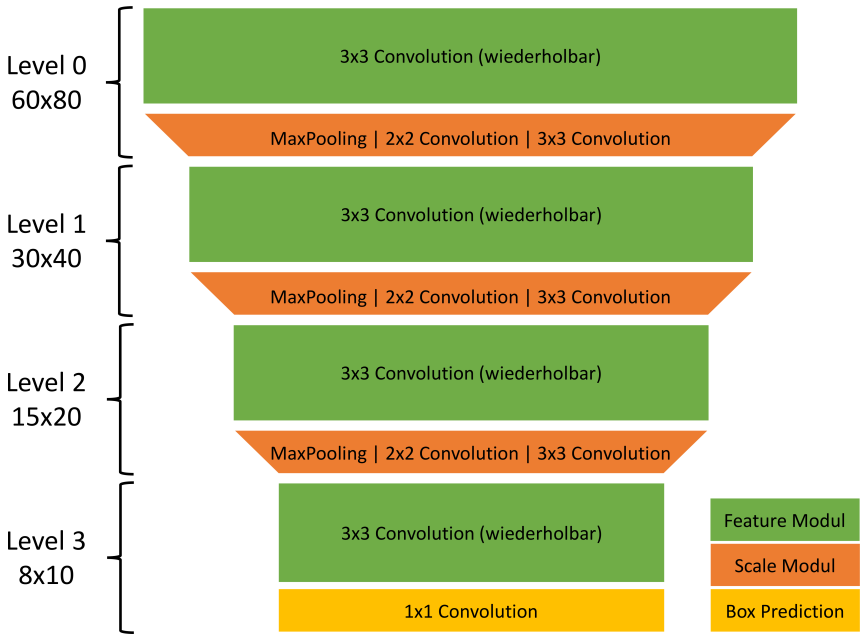


Abbildung 4.4: **Übersicht der Module des neuronalen Netzes**

- **Scale-Modul:**
  Um Features auf verschiedenen Größen extrahieren zu können, folgt jedem Feature-Modul ein Scale-Modul, welches den Übergang von einer Ebene zur nächsten herbeiführt. Das am häufigsten verwendete Verfahren für das Herunterskalieren ist MaxPooling, weshalb dieses auch im Ursprungsmodell genutzt werden soll. Sofern die Größe des Eingangs ungerade ist, wird dieser zuvor noch ein ZeroPadding Layer hinzugefügt, da ansonsten Pixel wegfallen würden. Allerdings soll neben MaxPooling auch ein anderes Verfahren getestet werden, nämlich Convolution mit einer Schrittgröße von 2, da auch dies das Bild verkleinert. Dabei soll sowohl die 2x2, als auch die 3x3 Convolution evaluiert werden. Dieses Verfahren wird unter anderem in [Springenberg et al., 2014] vorgeschlagen und könnte in diesem Szenario, wo es nur sehr wenig Parameter gibt, zusätzlichen Platz für Parameter schaffen. Somit wird auch jede Konfiguration des Scale-Moduls über ein Tupel (Gleichung 4.6) beschrieben.

$$\text{Konfiguration}_{\text{Scale}} = (\text{Typ}, \text{Anzahl Filter}) \in \{\text{MaxPool}, \text{2x2 Conv}, \text{3x3 Conv}\} \times \mathbb{N}_{>0} \tag{4.6}$$

Filterkosten Die meiste Zeit bei der Inferenz wird in die Berechnung von Conv-Layer investiert, weshalb an dieser Stelle ein kleiner Überblick über die damit verbundene Anzahl an Rechenoperationen und das Verhältnis zu den Parametern des Netzes gegeben wird. Wenn im Verlauf dieser Arbeit von Rechenoperationen die Rede ist, dann sind Mult-Adds gemeint, wie sie auch im MobileNets-Paper genutzt werden. Eine Mult-Add-Operation beschreibt dabei die in Gleichung 4.7 dargestellte Rechnung.

$$x = x + (y \cdot w) \tag{4.7}$$

Diese ist besonders praktisch für Conv-Layer, da die Faltung genau diese Operation für jedes Element im Filterkernel ausführt. Stellt man sich also vor, dass x mit dem Bias initialisiert wird, entspricht die Anzahl der Rechenoperationen für eine Filteranwendung genau der Größe des Kernels. Um die Anzahl an Rechenoperationen für einen Filter zu bekommen, muss dieses Ergebnis noch mit der Anzahl der Filteranwendungen multipliziert werden. Dies entspricht bei einer Schrittgröße von 1 und same-Padding genau der Anzahl an Pixeln pro Feature Map. Da die meisten Conv-Layer mehr als einen Filter haben, muss das Ergebnis noch mit der Anzahl der Filter multipliziert werden.

Die Anzahl der Parameter hingegen ist unabhängig von der Bildgröße und hängt somit nur von der Filtergröße und deren Anzahl ab. Pro Filter gibt es so viele Parameter wie der Kernel groß ist, plus einen für den Bias. Multipliziert mit der Anzahl der Filter erhält man somit die Anzahl der Parameter.

In dem Szenario dieser Arbeit ist jede Rechenoperation sehr teuer, weshalb es nur wenige kleine Layer gibt. Dadurch sinkt auch die Anzahl der zur Verfügung stehenden Parameter drastisch ab. Es ist allerdings wichtig, eine ausreichende Menge an Parametern zur Verfügung zu stellen, da das neuronale Netz ansonsten nicht dazu in der Lage ist, die unbekannte Funktion genau genug zu approximieren. Andersherum darf man auch nicht zu viele Parameter bereitstellen, das Netz ansonsten anfängt zu overfitten. Das Verhältnis von Rechenoperationen zu Parametern ist in Gleichung 4.8 dargestellt.

$$\frac{imageSize \cdot kernelSize \cdot filterCount}{(kernelSize + 1) \cdot filterCount} = imageSize \cdot \frac{kernelSize}{(kernelSize + 1)} \tag{4.8}$$

Da die Kernelgröße, abgesehen vom ersten Layer, immer wesentlich größer sein wird als 1, hängt das Verhältnis somit hauptsächlich von der Bildgröße ab. Das bedeutet, dass auf kleineren Bildern weniger Rechenoperation pro Parameter stattfinden. Deshalb kann es sinnvoll sein, mehr Filter in die unteren Layer zu verschieben, um eine höhere Geschwindigkeit zu erlangen. Allerdings geht mit dieser Methode auch immer die Gefahr einher, dass dem Netz auf den unteren Ebenen zu viel Flexibilität gewährt wird, sodass es wieder ins Overfitting abrutschen könnte.

Typ	Filter	Kernel	Schritte	Output
Conv	24	3x3	1	60x80
MaxPool		2x2	2	30x40
Conv	24	3x3	1	30x40
MaxPool		2x2	2	15x20
Conv	24	3x3	1	15x20
ZeroPad				16x20
MaxPool		2x2	2	8x10
Conv	24	3x3	1	8x10
Conv	20	1x1	1	8x10

Größe	Modul	Ursprungsmodell
60x80	Feature	(1,24)
	Scale	(MaxPool,24)
30x40	Feature	(1,24)
	Scale	(MaxPool,24)
15x20	Feature	(1,24)
	Scale	(MaxPool,24)
8x10	Feature	(1,24)

Tabelle 4.1: **Ursprungsmodell.** Die linke Tabelle zeigt die detaillierte Ansicht des Aufbaus des Ursprungsmodells. Die rechte Tabelle zeigt den Aufbau des Ursprungsmodells durch die eingeführten Konfigurationen. Beide Tabellen beschreiben exakt das gleiche Netz. Es ist zu beachten, dass das letzte Conv-Layer niemals in der Konfigurationsansicht vorkommen wird, da es in jeden Netz vorkommt und sich nicht ändern kann.

Pyramiden-Architektur Viele aktuelle Objektdetektoren verwenden eine Pyramiden-Architektur, bei der die Feature Maps am Ende wieder hochskaliert werden. Dazu gibt es Zwischenverbindungen, die Daten aus höher gelegenen Layern einfließen lassen. Dies soll helfen gerade kleine Objekte besser detektieren zu können. Es wird sich an dieser Stelle aber gegen die Nutzung einer solchen Architektur entschieden, da das Hochskalieren eine Operation ist, die auf dem schwachen Intel ATOM Prozessor sehr teuer ist und nur in wenigen Fällen einen wirklichen Mehrwert bringen würde. Des Weiteren würden die dazugehörigen Zwischenverbindungen die lineare Struktur des Netzes zerstören, wodurch es nötig würde, Ergebnisse zwischenzuspeichern und somit mehr Daten zu verschieben. Außerdem wird die Eingabe zum Beispiel beim YOLO v2 Netz auf dem Weg zur Ausgabe um einen Faktor 32 in jeder Dimension verkleinert, während es bei dem Detektor hier nur ein Faktor 8 ist. Dadurch dürfte das Subsampling allgemein eine kleinere Rolle spielen.

4.4 Zusammenfassung

In Abbildung 4.5 gibt es einen Überblick über die drei Komponenten des Detektors. Diese Struktur wird sich im weiteren Verlauf nicht mehr ändern. Lediglich das neuronale Netz in der Mitte wird noch weiter verfeinert. Um hierfür eine Grundlage zu haben, wurde ein Ursprungsmodell entwickelt, welches in Tabelle 4.1 zu sehen ist. Das dort verwendete Zero-Padding sorgt lediglich dafür, dass künstlich eine neue Spalte erzeugt wird, damit man von 15 auf 16 Pixel kommt. Dort ist außerdem ein Beispiel für die im Folgenden verwendete Konfigurationsschreibweise zu finden.

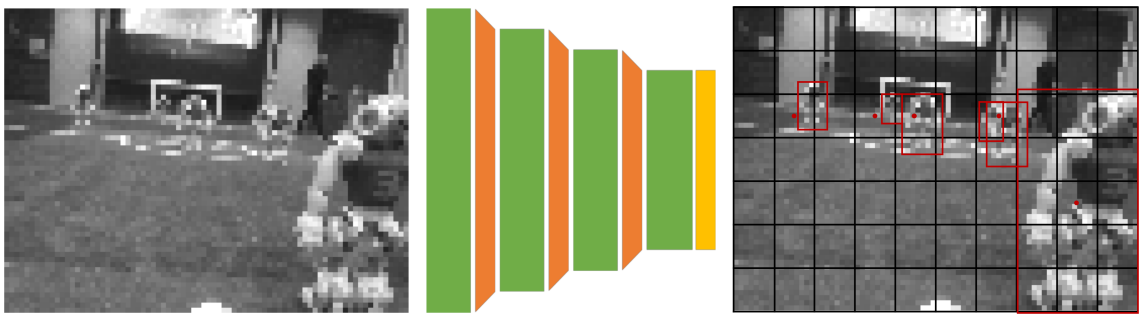


Abbildung 4.5: **Übersicht Pipeline.** Im ersten Schritt (links) wird das Bild vorbereitet. Anschließend wird es an das neuronale Netz übergeben (mittig). Zum Schluss (rechts) wird die Ausgabe des Netzes so interpretiert, dass Bounding Boxes entstehen.

5 Training des neuronalen Netzes

Nachdem im letzten Kapitel eine Grundstruktur für einen Objektdetektor entworfen worden ist, sollen nun Möglichkeiten gefunden werden, um diesen Detektor zu trainieren. Dies beginnt bei der Akquise von Bilder, geht dann weiter mit der Erzeugung von Trainingsdaten aus diesen Bildern und endet mit dem Finden passender Lossfunktionen.

5.1 Datenakquise

Bevor man anfangen kann, ein neuronales Netz zu trainieren, muss man sich Daten beschaffen, mit denen man dieses trainieren kann. Um dies zu tun, gibt es prinzipiell zwei Ansätze: entweder man generiert sich die Daten selbst oder man nutzt Daten von anderen. Da die Robotererkennung kein neues Problem in der SPL ist, haben sich auch andere Teams mit der Erzeugung von Daten beschäftigt. Aus diesem Grund soll möglichst wenig Zeit mit dem Labeln von Daten verbracht werden und stattdessen Möglichkeiten gefunden werden, um vorhandene Daten von anderen Teams zu nutzen. Dies ist besonders wichtig, da diese Daten, auch dann wenn sie nicht perfekt passen, dennoch Wissen bereitstellen, welches genutzt werden kann und sollte. Neben der Nutzung von realen Daten, sollen auch Bilder synthetisiert werden, wozu der Simulator von B-Human verwendet werden soll. Dies bietet die Möglichkeit, ohne großen Aufwand jederzeit wieder eine große Menge an Daten zu erzeugen. Allerdings muss hier darauf geachtet werden, dass die Daten der Realität möglichst ähnlich sehen, da das neuronale Netz später sonst nur Ressourcen damit verschwendet, Roboter auch in Bildern aus der Simulation detektieren zu können.

Imagetagger Damit mehr Zeit in die Entwicklung des Detektors fließen kann, wurde sich dazu entschieden, zum größten Teil auf bereits vorhandene Daten zurückzugreifen. Deshalb wurden öffentliche Datensätze von der online Labelplattform *Imagetagger* [Fiedler et al., 2018] verwendet, welche von dem Dortmunder SPL-Team, den Nao-Devils, stammen. Die Menge an erhaltenen Bildern liegt bei ca. 27.000 Bildern, welche sich auf 38 Datensätze aufteilen. Um eine solch große Anzahl an Bildern herunterladen zu können, wurde das Downloadskript von Imagetagger genutzt und so erweitert, dass es möglich ist, mehrere Datensätzen nacheinander herunterzuladen und diese zudem übersichtlich anzuordnen.

Bait: B-Human Advanced Image Tagger Bait stellt einen Fork von Imagetagger dar, welcher speziell auf die Bedürfnisse von B-Human angepasst ist. Dabei werden die Daten so organisiert, dass sich die hochgeladenen Bilder später wieder dem Log-File zuordnen lassen, aus dem sie stammen. Des Weiteren können die Daten in einem speziellen B-Human Format heruntergeladen werden, sodass die Daten direkt in den Simulator eingebunden werden können. Da Bait bisher hauptsächlich dafür verwendet worden ist, um Bälle zu labeln, mussten die Roboterdaten erst noch gelabelt werden. Dadurch, dass sich diese Bilder wieder in das dazugehörige Log-File einspeisen lassen, eignen sich diese Daten besonders gut für die spätere Evaluation, da Zusatzinformationen zur Verfügung stehen und auch der bisherige Roboterdetektor von B-Human hierauf evaluiert werden kann. Deshalb wurde im Rahmen dieser Arbeit der Datensatz eines Roboters aus einem Testspiel für die Evaluation gelabelt.

SPQR Datensatz Ein Datensatz, welcher Bilder des Naos im Zusammenhang mit der SPL enthält und schon öfters zitiert worden ist, wird von dem SPL Team SPQR bereitgestellt [Bloisi et al., 2017]. Dieser enthält ca. 2000 Bilder von verschiedenen Orten, in denen sowohl Roboter als auch Bälle und Tore gelabelt worden sind. Leider musste bei der Begutachtung des Datensatzes festgestellt werden, dass zwar viele Roboter in den Bildern gelabelt worden sind, aber nicht alle. Zudem sind in anderen Bildern Roboter teilweise mehrfach gelabelt. Dieses Phänomen ist in Abbildung 5.1 zu sehen, welche per Hand erstellt worden ist, um sicherzustellen, dass es sich nicht um einen Fehler in der Implementierung handelt. Somit kann man weder den positiven, noch den negativen Labeln

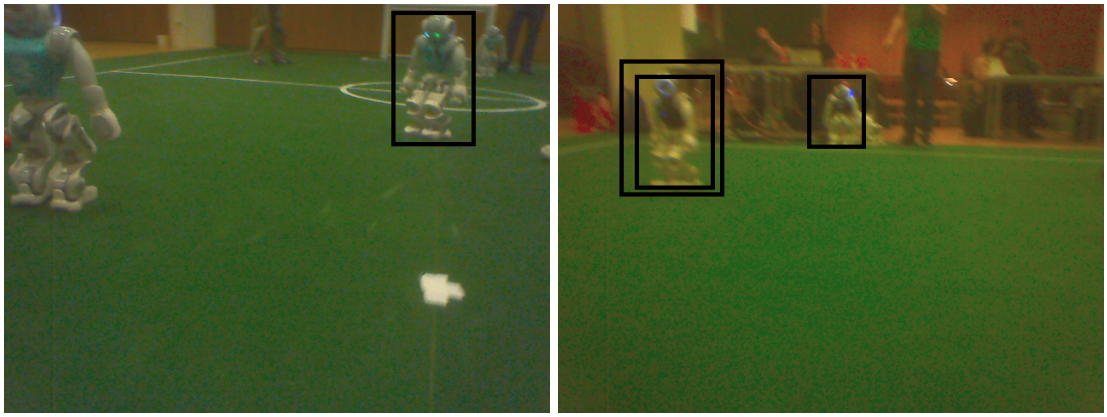


Abbildung 5.1: **SPQR Datensatz: falsche Label.** links: Im Bild sind drei Roboter zu sehen, wovon allerdings nur einer gelabelt worden ist. rechts: Im Bild sind zwei Roboter gut zu sehen, welche beide gelabelt sind, davon einer doppelt. Ein dritter Roboter ist stark verdeckt und nicht gelabelt.

vollends vertrauen. Da allerdings bereits eine große Menge an Bildern von Imagetagger zur Verfügung steht, kann auf die Nutzung dieses Datensatzes verzichtet werden. Andere Detektoren, welche auf diesem Datensatz trainiert und evaluiert worden sind, haben das neuronale Netz als Klassifikator verwendet, weshalb vermutlich nicht der Datensatz mit den ganzen Bildern, sondern die Variante mit den zugeschnittenen Bildern genutzt wurde. Ob der Fehler auch dort auftritt, wurde an dieser Stelle nicht weiter untersucht.

Autolabeler Die einfachste Art an Trainingsdaten zu gelangen, ist sie selbst zu erzeugen. Deshalb wurde im Rahmen dieser Arbeit ein Tool entwickelt, welches Bilder aus dem Simulator von B-Human automatisch annotiert. Dafür wird eine spezielle Kamera genutzt, die jedes verschiebbare Objekt der Simulation in einer anderen Farbe darstellt (siehe Abbildung 5.2). Die Farben dieser Objekte hängen von dem Objektnamen ab, sodass jedes Objekt über seine Farbe eindeutig zuordenbar ist (siehe Abbildung 5.2). Über eine Referenz zur Farbe kann somit die Postion des Objekt im Bild abgefragt werden. Zur Laufzeit findet das Modul dann die passende Bounding Box und stellt diese wiederum anderen Prozessen zur Verfügung. Da manchmal nur wenige Pixel eines Roboters zu sehen sind, weil diese zum Beispiel verdeckt sein können, werden zusätzlich Schwellwerte verwendet, um solche Fälle auszusortieren. Die normale Szene im Simulator hat keinen Hintergrund,

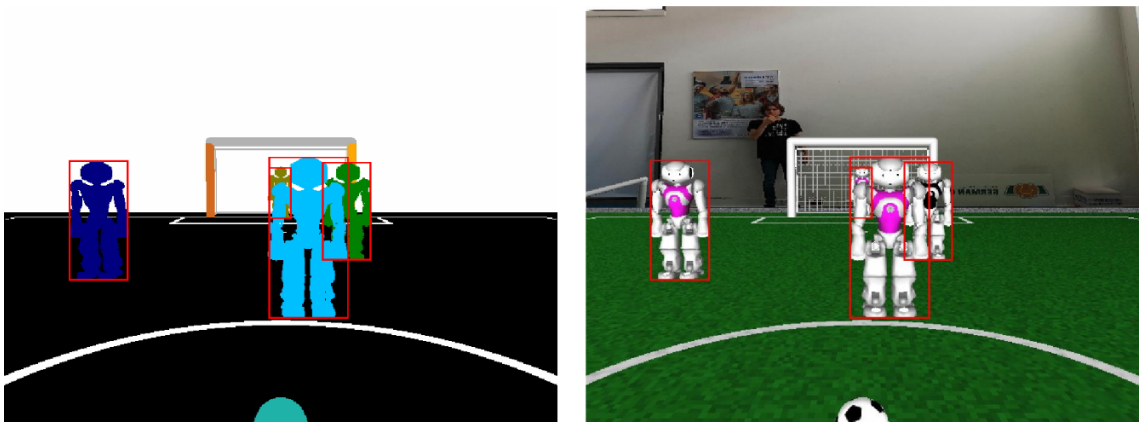


Abbildung 5.2: **Automatisch gelabeltes Bild aus dem Simulator.** Durch das links dargestellte objektsegmentierte Bild können die Bounding Boxes der dort farbigen Objekte bestimmt werden. Die rechte Seite zeigt das Originalbild mit den gefundenen Boxen. Zusätzlich zu den dargestellten Boxen für die Roboter können auch Boxen für Pfosten und den Ball berechnet werden.

Abbildung 5.3: **Simulatorszene mit Umgebungstexturen.** links: Graustufenbild aus Sicht des Naos, welches so in geringerer Auflösung an das neuronale Netz übergeben wird. rechts: Übersicht über die neue Umgebung. Die Seiten zeigen spiegelverkehrt entsprechende Sichten aus dem Projektraum von B-Human, während die Decke mit einem Deckengemälde versehen ist.

wodurch sich die Bilder aus dem Simulator in dieser Hinsicht von der Realität unterscheiden, da es dort immer Menschen und Objekte am Spielfeldrand gibt. Damit das neuronale Netz lernen kann, damit umzugehen, wurde eine Umgebung für das Spielfeld geschaffen, die einen realistischen Hintergrund erzeugen soll. Dazu wurden Leinwände rund um das Spielfeld in der Simulation platziert und diese mit Texturen aus der Realität versehen. Ein ähnlicher Ansatz wurde bereits in [Röfer, Laue, Müller et al., 2010] verfolgt, wo der Simulator um eine Zuschauermenge erweitert wurde. Da sich in der Zwischenzeit allerdings die Syntax und Befehle des Simulators geändert haben, musste die Implementierung dieses Ansatzes von Grund auf neu erstellt werden. Die fertige Szene und ein Bild aus der Sicht des Naos sind in Abbildung 5.3 zu sehen. Zwar sehen die Bilder aus der Simulation schon recht real aus, jedoch fehlen wichtige Eigenschaften, wie die Bewegungsunschärfe. Insgesamt wurden für das Training 28.000 Bilder aus einem Testspiel in der Simulation synthetisiert.

5.2 Generierung von Trainingsdaten

Daten-Augmentation Während viele Paper über Objektdetektoren ein sehr ausgiebiges Augmentieren der Daten empfehlen, wie zum Beispiel [Liu et al., 2016], soll in dieser Arbeit verstärkt darauf geachtet werden, dass die Augmentation nur Bilder erzeugt, die in ähnlicher Form tatsächlich im Spiel vorkommen könnten. Ansonsten würde der Detektor lernen Daten zu behandeln, welche bei der späteren Nutzung irrelevant sind und somit die stark begrenzten Ressourcen auf die falschen Bilder konzentrieren. Die Bilder werden zunächst im Format 640x480 geladen und die Augmentation findet auch auf dieser Größe statt. Erst nach Abschluss der Vorverarbeitung wird das Bild auf 80x60 skaliert, um möglichst wenig Informationen zu verlieren.

- **Zoom:** Beim Zoom wird das Bild um einen Faktor zwischen 1,0 und 1,5 vergrößert, welcher aus einer Gleichverteilung gesamplet wird. Da das Bild nun größer ist als 640x480, kann das Zentrum des Bildes in dem Bereich, den das Bild nun zu groß ist, verschoben werden. Die Verschiebung des Bildes wird ebensfalls aus einer Gleichverteilung gezogen. Dadurch werden insbesondere neue Bilder erzeugt, in denen die Objekte größer sind. Dies ist ein willkommenes Verhalten, da nähere Objekte tendenziell wichtiger sind. In Abbildung 5.4 ist ein Beispiel für das Zoomen zu sehen.
- **Flippen:** Eine häufig verwendete Methode zur Augmentation, ist das Drehen oder Spiegeln von Bildern. Allerdings kommt hier das zum Tragen, was bereits oben angesprochen worden ist: Es sollen keine Daten erzeugt werden, welche nicht auch in

dem eigentlichen Anwendungsszenario vorkommen können. Aus diesem Grund wird auf des Drehen verzichtet, da man das Bild nur minimal Drehen könnte, wenn man innerhalb der realen Grenzen bleiben möchte. Anders sieht es beim Spiegeln aus: zwar führt das Spiegeln an der horizontalen Achse zu unmöglichen Bildern, doch das Spiegeln an der vertikalen soll verwendet werden.

- **(Motion) Blur:** Während die Bilder aus der Realität oftmals verschwommen sind, da sich die Roboter bewegen, tritt dieses Phänomen in der Simulation nicht auf. Da die Bilder der Realität jedoch möglichst ähnlich sehen sollen, soll im Nachhinein das Bild weichgezeichnet werden, um somit zumindest den Effekt eines solchen Bildes imitieren zu können, auch wenn die Richtungen nicht stimmen, in denen sich die Unschärfe zeigt.

Nachdem die Operationen Zoom und Flippen angewendet worden sind, müssen die Positionen der Bounding Boxes angepasst werden.

Rauschen Da die Bounding Boxes in den gelabelten Daten mal mehr und mal weniger genau sind und man eigentlich nicht genau sagen kann, wann eine Bounding Box perfekt gesetzt ist, sollen die Koordinaten der Bounding Boxes mit Rauschen versehen werden. Dadurch wird zum einen die Definition für eine perfekten Bounding Box für das neuronale Netz verrauscht, sodass es hier mehr Spielraum gibt. Zum anderen wird Overfitting entgegengewirkt. Um dies umzusetzen wird der Eckpunkt oben links und der Eckpunkt unten rechts verschoben. Diese können jeweils um $\pm 5\%$ der Ursprungsgröße in der jeweiligen Dimension verschoben werden. Der Grad der Verschiebung wird dabei aus einer Gleichverteilung gesamplet, wobei vier unabhängige Samples verwendet werden. Dadurch kann die Bounding Box insbesondere größer oder kleiner werden.

Groundtruth Boxes Da die Backpropagation versucht den Fehler zwischen dem vorhersagten und dem gewünschten Ergebnis zu optimieren, muss vor der Berechnung klar sein, an welchen Stellen ein Objekt welchen Ausmaßes gefunden werden soll. Aus diesem Grund wird eine Menge von Anchor Boxes gesucht, welche die Objekte finden sollen. Dazu wählt man für jedes Objekt zunächst alle Anchor Boxes, deren IoU-Wert mit dem jeweiligen Objekt größer als 0,5 ist. Sollte es danach noch Objekte geben, denen keine Box zugewiesen worden ist, wird diesen die Anchor Box mit dem höchsten IoU zugewiesen. Bei allen diesen Anchor Boxes soll das neuronale Netz die Klasse *positiv* vorhersagen. Alle

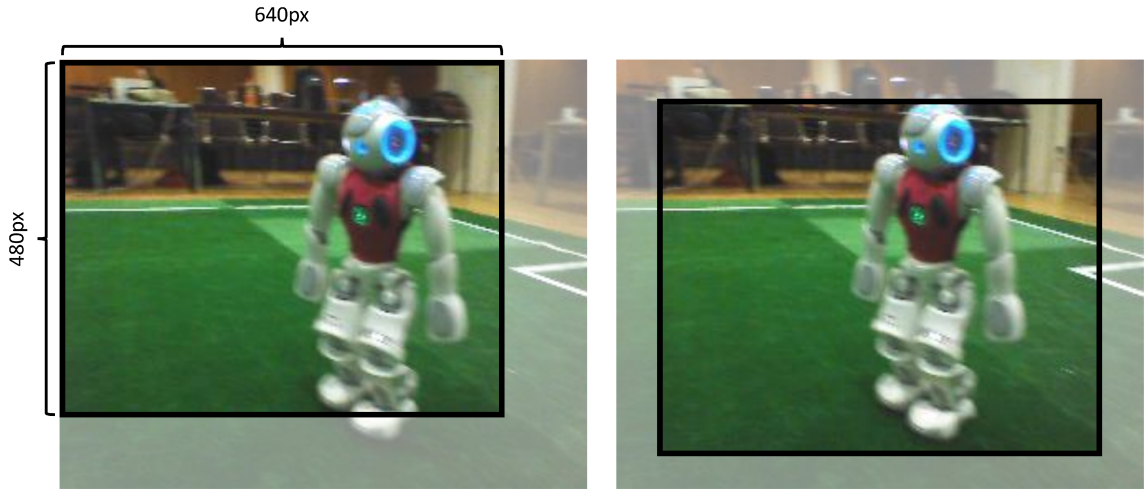


Abbildung 5.4: **Daten-Augmentation: Zoom.** Im linken Bild sieht man wie das Bild auf über 640x480 Pixel vergrößert wird. Dadurch ergibt sich ein Bereich welcher zu groß ist für das Bild. Innerhalb dieses Bereich es lässt sich das Bild verschieben, sodass ein neues Bild entsteht, wie es rechts gezeigt wird.

übrigen Boxen, die mit irgendeiner Anchor Box einen IoU-Wert haben, welcher zwischen 0,5 und 0,4 liegt, werden weder positiv noch negativ eingestuft und für die Berechnung des Losses einfach ignoriert. Alle Boxen die danach noch übrig sind, werden als *negativ* gelabelt.

Für jede zugewiesene Anchor Box werden anschließend die passenden Parameter berechnet, um die Anchor Box zur entsprechenden Ground Truth Box zu transformieren.

5.3 Lossfunktionen

Die Lossfunktion stellt den Kern eines guten neuronalen Netzes dar, weswegen es wichtig ist, eine passende Funktion zu modellieren. Für das Problem des Findens von Bounding Boxes stellt dies eine besondere Herausforderung dar, weil die Ausgänge des Netzes unterschiedliche Werte repräsentieren und somit auch unterschiedliche Lossfunktionen benötigen. Insgesamt werden fünf Ausgänge benötigt. Davon stellt der erste Ausgang die Wahrscheinlichkeit dar, dass die Box tatsächlich ein Objekt enthält. Die nächsten vier Ausgänge geben die Position und Größe der Box an. Wird mehr als eine Klasse verwendet, können weitere Ausgänge entsprechend der Anzahl an Klassen hinzugefügt werden, welche per One-Hot-Kodierung angeben, ob der Objekttyp in der Bounding Box zu finden ist. Da zunächst nur ein Objekttyp detektiert werden soll, werden auch nur Lossfunktionen für die Wahrscheinlichkeit und die Parameter der Box gesucht.

Konfidenz-Loss Für die Berechnung des Konfidenz-Losses werden die zugeordneten Ground-Truth-Boxes mit den vorhergesagten Boxen verglichen. Dabei möchte man immer eine 1 haben, wenn es für die Anchor Box eine zugeordnete Ground Truth Box gibt und ansonsten eine 0. Neutrale Boxen werden bei der Fehlerberechnung ignoriert. Der so errechnete Fehlervektor y_{err} zwischen Ziel und Vorhersage ist allgemeingültig für alle möglichen Konfidenz-Losses, die nun erklärt werden. Allerdings kann dieser Fehler anders interpretiert und gewichtet werden. Zunächst folgen allerdings ein paar vorverarbeitende Schritte, welche für alle Losses gleich sind.

1. Als erstes wird der Absolutwert der Fehler berechnet, da dies die nachfolgenden Rechnungen übersichtlicher macht: $y_{err} = abs(y_{err})$.

2. Im zweiten Schritt wird eine Gewichtung der positiven und negativen Beispiele mittels einer Alpha-Gewichtung (Gleichung 5.1) durchgeführt. Dies ist nötig, da es viel mehr negative, als positive Boxen gibt. Für die Anwendung der Alpha-Gewichtung werden einfach alle Elemente aus y_{err} mit einem Gewichtsmatrix ω_α multipliziert, der für jedes Element ein Gewicht enthält, welches von der tatsächlichen Klasse y_{true} anhängt: $y_{err} = \omega_\alpha \cdot y_{err}$.

$$\omega_\alpha = \begin{cases} 1 - \alpha \text{ falls } y_{true} = 1 \\ \alpha \text{ sonst} \end{cases} \tag{5.1}$$

3. Für das spezifische Problem der Robotererkennung wird eine weitere Gewichtung hinzugefügt, welche die Größe der Bounding Boxes berücksichtigt, da es tendenziell wichtiger ist, wenn Roboter erkannt werden, die näher sind. Aus diesem Grund werden alle positiven Boxen mit ihrer Größe gewichtet. Diese wird dabei auf einen Bereich zwischen einem Fünfzigstel und einem Achtel des Bildes zurecht geschnitten. Anschließend wird diese so normiert, dass die Summe aller Gewichte der Anzahl der positiven Boxen entspricht und das so errechnete Gewichtsmatrix ω_{area} wird mit den positiven Boxen multipliziert: $y_{err} = \omega_{area} \cdot y_{err}$.

Zusätzlich sind viele der negativen Boxen sehr einfach vorherzusagen. Das führt dazu, dass sich das Netz nicht auf die wirklich schwierigen Boxen konzentrieren kann. Dieses Problem wird in den folgenden Lossfunktionen behandelt. Die Hyperparameter der jeweiligen Lossfunktion wurden durch Experimente ermittelt.

- Um eine **Baseline** für die Performanz des neuronalen Netzes zu haben, wird eine einfache Metrik gewählt. Allgemein kann man für die Vorhersage von Wahrscheinlichkeiten zwischen 0 und 1 bzw. bei binären Klassifizierungsproblemen zwei mögliche Grundlossfunktionen verwenden: Mean Squared Error (MSE) oder Binary Cross Entropy (BCE) (Gleichung 5.2). Beide Funktionen versuchen den Fehler y_{err} zu minimieren, jedoch verwendet MSE dafür den quadrierten Fehler, während BCE den negativen logarithmischen Fehler optimiert. Es ist schwierig, die genauen Unterschiede der beiden Funktionen zu beschreiben, da das Verhalten von Problem zu Problem variieren kann. Jedoch wurde in [Golik et al., 2013] versucht experimentell die Differenzen aufzudecken, wobei herauskam, dass MSE bei guter Initialisierung der Gewichte bessere Ergebnisse erzielen kann. Allerdings bleibt MSE bei zufälligen Initialisierungen des Öfteren in lokalen Minima stecken. Da BCE besser mit zufälligen Initialisierungen umgehen kann, wird BCE im Folgenden als Baseline genutzt. Dabei wird ein α-Wert von 0,1 gewählt.

$$\mathrm{BCE}(y_{err}) = \frac{1}{|y_{err}|} \cdot \sum_{i \in y_{err}} -log(1-i) \tag{5.2}$$

- Einen anderen Ansatz verwendet das **Hard Negative Mining** des SSD (siehe Gleichung 5.3), welcher die Anzahl der zur Berechnung des Losses herbeigezogenen negativen Beispiele in Abhängigkeit von der Anzahl der positiven Exemplare beschränkt. Dies geschieht, indem man die negativen Boxen ihrem Fehler nach sortiert und dann die n größten Fehler auswählt, sodass n der dreifachen Anzahl der positiven Boxen entspricht. Dabei werden immer mindestens 200 Boxen ausgewählt. Das Loss wird dann nur noch auf den ausgewählten negativen sowie allen positiven Boxen berechnet. Der α-Wert hier ist 0,17.

$$\mathrm{HNM}(y_{err}) = \frac{\sum_{i \in pos(y_{err})} -log(1-i)}{|pos(y_{err})| + |neg(y_{err})|} + \frac{\sum_{i \in neg(y_{err})} -log(1-i)}{|pos(y_{err})| + |neg(y_{err})|} \tag{5.3}$$

- Das in [T.-Y. Lin et al., 2017] vorgestellte **Focal Loss** stellt eine Erweiterung des Cross-Entropy Losses dar, welches besonders gut mit dem zuvor geschilderten Problem umgehen können soll, indem es den Fokus verstärkt auf die schwierig zu klassifizierenden Boxen lenkt. Dies geschieht durch den Einsatz eines weiteren Faktors, welcher gut klassifizierte Samples geringer gewichtet. Außerdem soll das errechnete Loss nicht mehr über alle Boxen, sondern lediglich über alle positiven Exemplare normalisiert werden. Dies hat in den durchgeführten Versuchen allerdings keinen Vorteil gebracht, weshalb auch das Focal Loss über alle Boxen, die nicht neutral sind, normalisiert wird. In Gleichung 5.4 ist eine allgemeine Form des Focal Losses zu finden. Dabei wird ein α-Wert von 0,1 und ein γ-Wert von 1 gewählt.

$$\mathrm{FL}(y_{err}) = \frac{1}{|y_{err}|} \cdot \sum_{i \in y_{err}} -log(1-i) \cdot i^{\gamma} \tag{5.4}$$

- Zusätzlich soll ein eigener Ansatz getestet werden, welches auf dem Focal Loss basiert, aber nicht die BNE zur Gewichtung nutzt, sondern den MSE. Durch die Multiplikation mit dem neu eingeführten Faktor aus dem Focal Loss und einem γ-Wert von 1 ergibt sich hier der **Mean Absolute Cubed Error (MACE)**. Als α-Wert wird weiterhin 0.1 verwendet. Abbildung 5.5 zeigt wie unterschiedlich die Gewichtung des Fehlers für unterschiedliche BCE und MSE bei verschiedenen γ-Werten aussieht.

$$textMACE(y_{err}) = \frac{1}{|y_{err}|} \cdot \sum_{i \in y_{err}} i^2 \cdot i^1 = \frac{1}{|y_{err}|} \cdot \sum_{i \in y_{err}} i^3 \tag{5.5}$$

Bounding-Box-Loss Das Bounding-Box-Loss versucht die genaue Position und Ausmaße des Objektes zu bestimmen. Um dies umzusetzen, werden, wie beim Konfidenz-Loss, Werte zwischen 0 und 1 berechnet. Diese werden dann nach dem Formeln aus Kapitel 4.2 umgerechnet. Mit diesen Werten kann man nun auf verschiedene Arten und Weisen eine Aussage darüber treffen, wie gut die Bounding Box das Objekt abdeckt. Im Folgenden sollen drei Ansätze vorgestellt werden.

- Das **Koordinaten-Loss** (Koord-Loss) (siehe Gleichung 5.8) versucht direkt die Koordinaten zu optimieren. Deshalb gibt es hier sowohl einen Wert für die x,y Position, als auch einen für die Breite und Höhe, aus denen ein gewichteter Mittelwert berechnet wird. Die Berechnung ist auch dementsprechend simpel, da lediglich der Mean Squared Error der Differenz zwischen der gewünschten Koordinate/Größe und dem berechneten Wert bestimmt werden muss.

$$\text{Koord-Loss}_{xy}(true, pred) = \frac{\text{MSE}(true_x, pred_x) + \text{MSE}(true_y, pred_y)}{2} \tag{5.6}$$

$$\text{Koord-Loss}_{wh}(true, pred) = \frac{\text{MSE}(true_w, pred_w) + \text{MSE}(true_h, pred_h)}{2} \tag{5.7}$$

$$\text{Koord-Loss}(true, pred) = \frac{\text{Koord-Loss}_{xy}(true, pred) + \text{Koord-Loss}_{wh}(true, pred)}{2} \tag{5.8}$$

- Das ***relative Koordinaten-Loss*** (rKoord-Loss) (siehe Gleichung 5.11) verwendet den gleichen Fehler, wie das normale Koordinaten Loss, gewichtet diesen jedoch anders. Es werden zunächst alle Werte mit der Größe der Ground Truth Box in der x- bzw. y-Dimension normalisiert und dann der MAE berechnet. Dadurch können auch kleine Fehler bei kleinen Objekten stark ins Gewicht fallen, während kleine Fehler bei großen Objekten zu vernachlässigen sind. Auch hier wird abschließend ein gewichteter Mittelwert gebildet.

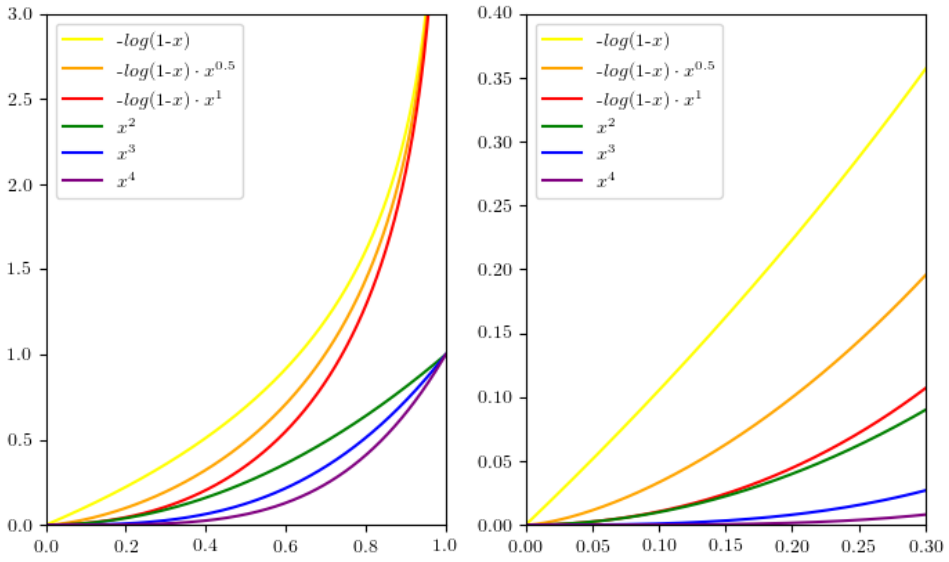


Abbildung 5.5: **Vergleich zwischen MSE und BCE.** Beide Grafiken zeigen die gleichen Funktionen, wobei rechts der Fokus mehr auf den Bereich der kleinen Fehler gelenkt ist, da diese hier weniger stark gewichtet werden sollen. Der große Unterschied zwischen MSE und BCE ist hierbei die Größe der y-Werte um einen x-Wert von 1. Während MSE ein maximales Gewicht von 1 erreicht, steigt es bei BCE exponentiell über 1 hinaus. Im Allgemeinen ist die Gewichtung bei MSE ausgewogener.

$$\text{rKoord-Loss}_{xy}(true, pred) = \frac{\text{MAE}(\frac{true_x}{true_w}, \frac{pred_x)}{true_w}) + \text{MAE}(\frac{true_y}{true_h}, \frac{pred_y}{true_h})}{2} \tag{5.9}$$

$$\text{rKoord-Loss}_{wh}(true, pred) = \frac{\text{MAE}(1, \frac{pred_w}{true_w}) + \text{MAE}(1, \frac{pred_h}{true_h})}{2} \tag{5.10}$$

$$\text{rKoord-Loss}(true, pred) = \frac{\text{rKoord-Loss}_{xy}(true, pred) + \text{rKoord-Loss}_{wh}(true, pred)}{2} \tag{5.11}$$

- Das **IoU-Loss** (siehe Gleichung 5.12)ist ein Loss, welches nicht aus der Literatur inspiriert ist, sondern im Rahmen dieser Arbeit entwickelt worden ist. Es berechnet zunächst die vorhergesagte Box, um dann den IoU-Wert zu der entsprechenden Ground Truth Box bestimmt. Dieses Berechnung wird für alle positiven Boxen ausgeführt und dann ein Mittelwert gebildet. Zuletzt muss das Ergebnis noch von 1 abgezogen werden, weil das Loss minimiert werden soll. Aufgrund der Tatsache, dass es hier nur eine Lossfunktion gibt, fällt das Finden einer passenden Gewichtung für die unterschiedlichen Parameter der Bounding Box weg.

$$\text{IoU-Loss}(true, pred) = 1 - \text{MEAN}(\text{IoU}(true_{box}, pred_{box})) \tag{5.12}$$

6 Evaluation des neuronalen Netzes

Nachdem in den letzten beiden Kapiteln alle Komponenten geliefert wurden, die gebraucht werden, um einen Objektdetektor zu trainieren, sollen in diesem Kapitel verschiedene Konfigurationen evaluiert werden. Dabei wird für jede Komponente der beste Wert gesucht. Als Metrik dient dabei immer der MAP-Wert, der sich eingestellt hat, nachdem der Lernprozess zehn Durchläufe lang kein besseres Ergebnis im Hinblick auf den F_1-Score mehr erzielen konnte. Dieses Kriterium wird verwendet, da die MAP zu aufwändig ist, als dass sie nach jeder Episode berechnet werden könnte, da dies das Training nur weiter verlangsamen würde. Der F_1-Score kommt der MAP von der Aussage schon sehr nahe, aber beachtet die Ordnung der Vorhersagen nicht. Daher scheint dies ein guter Trade-off zu sein. Um ein robusteres Ergebnis zu bekommen, werden drei unterschiedliche Testdatensätze getestet, welche unterschiedliche Schwierigkeitsgrade haben. Einer ist dem Trainingsdatensatz relativ ähnlich, einer hat schwierige Lichtverhältnisse und der letzte stammt aus einem Testspiel von B-Human, in dem sich viele Menschen über den Platz bewegen. Aus dem Mittelwert dieser drei Ergebnisse ergibt sich der MAP-Wert, welcher als Vergleichswert in diesem Kapitel genutzt wird. Dabei gilt ein Objekt als erkannt, wenn die Wahrscheinlichkeit einer Box über 50% liegt und die IoU mit der Ground-Truth-Box über einer einem Schwellwert von 50% ist. Des Weiteren werden die Experimente nicht mit allen Daten durchgeführt, die zur Verfügung stehen. Es werden lediglich 5601 Bilder aus der Realität und 3149 aus der Simulation verwendet, da das Training ansonsten zu viel Zeit in Anspruch nehmen würde.

Sobald eine neue beste Konfiguration gefunden worden ist, wird von diesem Zeitpunkt an damit weiter getestet, sodass man am Ende dieses Abschnittes einen guten Objektdetektor besitzt. Theoretisch müssten sämtliche Kombinationen von Konfigurationen verschiedener Komponenten getestet werden. Da dies allerdings zu viel Zeit in Anspruch nehmen würde, muss das beschriebene *Greedy*-Verfahren ausreichen. Das Training wie es oben beschrieben ist, dauert auf dem reduzierten Datensatz immerhin rund drei Stunden.

Umsetzung Für die Umsetzung des Objektdetektors in Python wurde keras-yolo2 (`https://github.com/experiencor/keras-yolo2`) als Grundlage verwendet. Im Verlauf dieser Arbeit wurde dieser Code immer weiter verändert und erweitert, sodass letztendlich nur noch die Infrastruktur und ein paar Hilfsfunktionen, wie das Clustering der Anchor Boxes, genutzt werden.

6.1 Allgemeine Parameter

Im ersten Teil werden zunächst allgemeine Parameter getestet. Im Zuge dessen werden zu Beginn die Lossfunktionen gegenübergestellt. Anschließend werden die besten Lossfunktionen genutzt, um die Bildgröße der Eingabe zu evaluieren. Zuletzt wird noch geprüft, ob der Einsatz der LeakyReLU-Aktivierungsfunktion einen Vorteil gegenüber der normalen ReLU-Funktion bringt.

Lossfunktionen Das erste Experiment befasst sich nicht mit der Architektur, sondern mit dem Training. Aus der Menge der vorgestellten Lossfunktionen sollen die gefunden werden, die am besten performen. Dazu wurden zunächst alle Bounding Box Lossfunktionen verglichen (siehe Tabelle 6.1). Dabei kam heraus, dass das IoU-Loss den besten MAP-Wert erzielen kann. Im zweiten Teil wurden die Konfidenz-Lossfunktionen gegenüber gestellt, wobei das IoU-Loss für die Bounding Boxes verwendet worden ist. Wie Tabelle 6.2 zu entnehmen ist, erzielt das MACE-Loss deutlich bessere Ergebnisse als die Konkurrenz. Der große Unterschied zwischen MACE und dem Focal-Loss ist erstaunlich, da nur die Gewichtung etwas gedämpfter ausfällt. Für alle folgenden Versuche wird aufgrund dieser Ergebnisse die Summe von MACE und IoU-Loss als Lossfunktion verwendet.

Bounding Box Loss	∅MAP
Mean Squared Error	0,427
Mean Absolute Percentage Error	0,423
IoU	0,457

Tabelle 6.1: Vergleich Bounding Box Losses

Konfidenz Loss	∅MAP
Binary Crossentropy	0,457
Hard Negative Mining	0,485
Focal Loss	0,471
MACE	0,502

Tabelle 6.2: Vergleich Konfidenz Losses

Bildgröße Das nächste Experiment soll die verschiedenen Bildgrößen gegenüberstellen und damit die Netzarchitektur bestimmen. Dazu wird das Ursprungsnetz aus Tabelle 4.1 so angepasst, dass es 160x80 Pixel und 40x30 Pixel große Eingaben verarbeiten kann. Die dazugehörigen Konfigurationen und Ergebnisse sind in Tabelle 6.3 zu finden. Um eine bessere Vergleichbarkeit zu schaffen, gibt es für die beiden kleineren Netze jeweils eine Version mit mehr Parametern. Dies wird durch einen zweiten Layer im ersten Feature-Modul umgesetzt.

Es zeigt sich, dass eine 40x30 Pixel große Eingabe selbst bei gleicher Parameteranzahl wesentlich schlechtere Ergebnisse erzielt. Im Gegensatz dazu ist das Ergebnis der 160x120 Pixel großen Eingabe kaum besser als die erweiterte Version des 60x80 Pixel großen Bildes. Somit wird die Beobachtung aus Kapitel 4.1 bestätigt. Zwar kann die größere Eingabe bessere Ergebnisse erzielen, jedoch braucht sie auch wesentlich länger für die Inferenz. Aus diesem Grund soll im Folgenden weiterhin eine Auflösung von 80x60 Pixel verwendet werden.

Größe	Modul	120x160	**60x80**		30x40
120x160	Feature	(1,24)	-		-
	Scale	(MaxPool,24)	-		-
60x80	Feature	(1,24)	(2,24)	(1,24)	-
	Scale	(MaxPool,24)	(MaxPool,24)		-
30x40	Feature	(1,24)	(1,24)		(2,24)
	Scale	(MaxPool,24)	(MaxPool,24)		(MaxPool,24)
15x20	Feature	(1,24)	(1,24)		(1,24)
	Scale	(MaxPool,24)	(MaxPool,24)		(MaxPool,24)
8x10	Feature	(1,24)	(1,24)		(1,24)
Gesamtparameter		22.083	22.083	16.827	16,827
∅MAP		0,582	0,561	0,502	0,456

Tabelle 6.3: Vergleich Bildgrößen

Nutzung von Simulationsdaten Im letzten Kapitel wurden neben Daten aus der Realität auch Daten aus dem Simulation erzeugt. Nachdem diese bereits in den vorangegangenen Experimenten genutzt worden sind, soll nun geprüft werden, ob diese tatsächlich einen Vorteil bringen. Dazu wird die aktuelle Architektur dreimal trainiert: nur mit realen Daten, nur mit synthetischen Daten und mit allen Daten. Die Ergebnisse sind in Tabelle 6.4 dargestellt. Es ist zu sehen, dass die ausschließliche Nutzung von Daten aus der Simulation kein gutes Ergebnis auf dem aus der Realität stammenden Testdatensatz erzielt. Kombiniert man diesen allerdings mit den Datensatz aus der Realität, so übertrifft beide Ergebnisse aus der isolierten Nutzung. Somit können die Informationen aus der Simulation einen Teil dazu beitragen, bessere Ergebnisse zu erzielen und sollen von daher auch weiterhin genutzt werden. Ein weiterer Vorteil ist, dass man den Objektdetektor aufgrund dessen auch in der Simulation gut nutzen kann. Das ist wichtig weil SPL-Teams, wie B-Human, ihre neuen Entwicklungen immer gerne erst in der Simulation testen.

Datenherkunft	⌀MAP
reale Daten	0,518
synthetische Daten	0,195
alle Daten	0,548

Tabelle 6.4: Vergleich Trainingsdatensätze

Aktivierungsfunktion	⌀MAP MaxPool	3x3Conv	2x2Conv
ReLU	0,502	0,612	0,540
Leaky ReLU	0,479	0,599	0,546

Tabelle 6.5: Vergleich Aktivierungsfunktionen

Aktivierungsfunktion Die bisherigen Experimente haben die ReLU-Aktivierungsfunktion genutzt. Das folgende Experiment soll prüfen, ob mit der LeakyReLU-Funktion bessere Ergebnisse erzielt werden können. Dazu wird der aktuelle Stand des Netzes genommen und die Aktivierungsfunktion durch eine LeakyReLU-Funktion ersetzt. Außerdem wird dies für die anderen Downsampling-Verfahren wiederholt, wobei die genauen Ergebnisse zu diesem Zeitpunkt ignoriert werden können, da der Unterschied zwischen den Aktivierungsfunktionen wichtig ist. Das Ergebnis für die ReLU-Funktion mit MaxPool liegt bereits aus dem letzten Experiment vor. Der Vergleich (Tabelle 6.5) zeigt, dass durch die Nutzung einer anderen Aktivierungsfunktion keine besseren Ergebnisse erzielt werden können. Zudem ist die LeakyReLU ein wenig rechenaufwändiger, weshalb im Folgenden weiterhin die normale ReLU-Funktion genutzt werden soll.

6.2 Modulkonfigurationen

Nachdem schon viele Hyperparameter im vorherigen Teil gefunden wurden, soll nun der Fokus auf das Feature- und Scale-Modul gelegt werden. Beide haben einen Konfigurationsraum, welcher zu groß ist, um ihn im weiteren Verlauf der Arbeit komplett zu testen. Deshalb sollen die verschiedenen Möglichkeiten hier gegenübergestellt werden, um eine gute Konfiguration zu finden, welche fortan genutzt werden kann.

Da die nun zu testenden Hyperparameter insbesondere die Inferenzzeit beeinflussen, wird nun zusätzlich die Inferenzzeit der Netze angegeben. Diese wird ermittelt, indem das jeweilige neuronale Netz in die Toolchain geladen wird und dann auf einem Laptop mit Intel i5 Prozessor und ohne Grafikkarte ausgeführt wird.

Im folgenden Experiment werden die drei verschiedenen Varianten für das Downsampling verglichen. Es wird die aktualisierte Version des Ursprungsnetzes verwendet, wobei jeweils das Scale-Modul angepasst wird. Die Ergebnisse sind in Tabelle 6.6 zu sehen. Es zeigt sich, dass die Nutzung von Conv-Layern bessere Ergebnisse liefert, aber auch deutlich rechenaufwändiger ist. Dabei scheint der der MAP-Wert mit der Anzahl der Parameter und der Inferenzzeit zu korrelieren. Um weitere Informationen über das Verhalten der drei Varianten zu erhalten, wird das Experiment mit mehr Parametern wiederholt. Dazu werden zwei zusätzliche Experimente ausgeführt. Das erste Experiment verdoppelt die Anzahl der Layer im Feature-Modul (Tabelle 6.6) und das zweite verdoppelt die Anzahl der Filter in allen Modulen (Tabelle 6.7). Es zeigt sich, dass weiterhin mit zunehmender Parameteranzahl auch der MAP-Wert zunimmt. Allerdings scheint dies bei MaxPool langsamer zu passieren als bei den anderen beiden Varianten. Zudem funktioniert die 2x2Conv-Variante erst richtig gut, sobald es eine ausreichende Menge an Parametern gibt.

Das Verhältnis von MAP-Wert und Parameteranzahl bzw. Inferenzzeit scheint einen logarithmischen Verlauf zu haben. Um eine realistische Obergrenze für diesen Detektor zu bekommen, wurde ein neuronales Netz trainiert, welches sowohl mehr Filter, als auch mehr Layer besitzt und zudem 3x3 Convolution verwendet (Scale(3x3MaxPool,48),Feature(2,48)). Das resultierende Netz besitzt 210.891 Parameter und erreicht einen durchschnittlichen MAP-Wert von 0,697.

Größe	Modul	Maxpool		3x3Conv		2x2Conv	
60x80	Feature	(1,24)	(2,24)	(1,24)	(2,24)	(1,24)	(2,24)
	Scale	(MaxPool,24)		(3x3Conv,24)		(2x2Conv,24)	
30x40	Feature	(1,24)	(2,24)	(1,24)	(2,24)	(1,24)	(2,24)
	Scale	(MaxPool,24)		(3x3Conv,24)		(2x2Conv,24)	
15x20	Feature	(1,24)	(2,24)	(1,24)	(2,24)	(1,24)	(2,24)
	Scale	(MaxPool,24)		(3x3Conv,24)		(2x2Conv,24)	
8x10	Feature	(1,24)	(2,24)	(1,24)	(2,24)	(1,24)	(2,24)
Gesamtparameter		16.827	37.851	32.595	53.619	23.955	44.979
∅Inferenzzeit		1,1 ms	4,9 ms	2,2 ms	5,5 ms	1,5 ms	5,1 ms
∅MAP		0,502	0,600	0,612	0,642	0,540	0,608

Tabelle 6.6: Vergleich Downsampling-Verfahren 1

Größe	Modul	Maxpool	3x3Conv	2x2Conv
60x80	Feature	(1,48)	(1,48)	(1,48)
	Scale	(MaxPool,48)	(3x3Conv,48)	(2x2Conv,48)
30x40	Feature	(1,48)	(1,48)	(1,48)
	Scale	(MaxPool,48)	(3x3Conv,48)	(2x2Conv,48)
15x20	Feature	(1,48)	(1,48)	(1,48)
	Scale	(MaxPool,48)	(3x3Conv,48)	(2x2Conv,48)
8x10	Feature	(1,48)	(1,48)	(1,48)
Gesamtparameter		64.731	127.371	92.811
∅MAP		0,577	0,660	0,674

Tabelle 6.7: Vergleich Downsampling-Verfahren 2

6.3 Fazit

Im ersten Teil dieses Kapitels konnten relativ eindeutig Hyperparameter gefunden werden, welche ihre Alternativen offensichtlich überragen. Im zweiten Teil war das Ergebnis nicht so eindeutig. Dennoch soll an dieser Stelle eine Konfiguration festgehalten werden, welche im Folgenden weiterverwendet werden kann.

Der Vergleich der Downsampling-Verfahren lässt vermuten, dass die MaxPool Variante die zur Verfügung stehenden Parameter nicht so gut nutzen kann, wie die anderen beiden Varianten, da selbst bei höherer Parameteranzahl schlechtere Ergebnisse erzielt werden. Beim 2x2Conv scheint es Probleme mit wenigen Parametern zu geben, da die Ergebnisse bei weniger als 48 Filtern deutlich schlechter sind als die von 3x3Conv. Erst bei 48 Filtern scheint sich hier eine Sättigung einzustellen, sodass beide gleichauf sind. Aus diesem Grund scheint 3x3Conv das beste Verfahren zu sein, obwohl es gleichzeitig das langsamste ist. Da aber die anderen beiden Verfahren mehr Parameter für die gleiche Leistung benötigen, gleicht sich dieser Nachteil wieder aus.

Die Anzahl der Filter und Layer ist kritisch zu betrachten. Im Vergleich kann sowohl die doppelte Filteranzahl, als auch die doppelte Layeranzahl hohe Ergebnisse erzielen, wobei die doppelte Filteranzahl etwas besser ist, was vermutlich der höheren Parameterzahl zuzuschreiben ist. Die hier erreichten Ergebnisse kommen der Obergrenze schon sehr nahe, jedoch sind beide Verfahren sehr langsam. Im Falle der Variante mit 48 Filtern ist diese Zunahme auf alle Convolutional-layer verteilt, während bei der doppelten Layeranzahl ein Layer hauptverantwortlich ist: die zweite Convolution am Anfang. Das liegt daran, das die erste Convolution auf dem größten Bild stattfindet, allerdings besitzt dies normalerweise auch nur einen Kanal, was die Größe ausgleicht. Durch die zusätzliche Convolution wird allerdings auf der größten Ebene eine Convolution auf einem Bild mit 48 Kanälen ausgeführt, was sehr viel Zeit in Anspruch nimmt. Verschiebt man das Layer von der ersten

in die letzte Ebene so bleibt die Anzahl der Parameter identisch, dennoch nimmt die Anzahl der Berechnungen ab, da die Filter nun auf einem kleineren Bild angewendet werden. Dadurch sinkt die Inferenzzeit auf unter die Hälfte mit 2,5 ms, während der MAP-Wert nur minimal auf 0,634 fällt.

Aufgrund dieser Ergebnisse soll im Folgenden nur noch 3x3Conv als Downsampling-Verfahren genutzt werden. Für die Layeranzahl im Feature-Modul sollen das 1223-Schema der neue Standard sein. Die Nutzungvon mehr als 24 Filtern ist mit Vorsicht zu verwenden, da die Anzahl der Rechenoperationen quadratisch ansteigt, wenn sowohl die Eingabe, als auch die Ausgabe mehr Feature Maps bekommt. 48 Filter ist dabei sehr langsam aber geringere Filteranzahlen könnten durchaus machbar sein.

Aus diesen Gründen wird das Scale-Modul auf die Parameter (3x3Conv,24) und das Feature-Modul auf (x,24) mit $x \in \mathbb{N}_{<0}$ festgelegt. Somit verbleibt die Anzahl der Layer im Feature-Modul bis auf weiteres der einzige Hyperparameter.

7 Echtzeitfähiger Objektdetektor

In Kapitel 3.3 wurden mit XNOR-Net, SqueezeNet und MobileNet drei Ansätze präsentiert, die durch eine strukturelle Umgestaltung des neuronalen Netzes eine effizientere Berechnung herbeiführen wollen. Dabei verfolgen SqueezeNet und MobileNet ein ähnliches Ziel, da beide versuchen, möglichst viele Berechnungen von 1x1 Filtern durchführen zu lassen, um somit die Nutzung von 3x3 Filtern zu reduzieren (siehe Abbildung 7.1). Das XNOR-Net hingegen macht sich die Redundanz in neuronalen Netzen zu nutzen. Durch diese wird es möglich, Eingänge und Ausgänge zu quantisieren oder gar zu binarisieren ohne größere Verluste bei der Erkennungsrate hinnehmen zu müssen. Allerdings ist die gewählte Implementierung zur Berechnung der neuronalen Netze derzeit noch nicht auf solche Datenstrukturen vorbereitet, sodass zunächst die gesamte Inferenz angepasst werden müsste. Des Weiteren soll der spätere Detektor ohnehin möglichst wenig Redundanzen aufweisen, um möglichst effizient agieren zu können. Aus diesem Grund stellt sich hier auch die Frage, inwiefern sich dieses Verfahren hier überhaupt anwenden lässt. Zudem kann das XNOR-Net Verfahren auch dann noch genutzt werden, wenn bereits Ideen aus den anderen beiden Ansätzen übernommen worden sind, da diese nicht in Konkurrenz zueinander stehen. Aus diesen Gründen beschäftigt sich diese Arbeit im Folgenden genauer mit dem SqueezeNet und dem MobileNet, um eine effizientere Berechnung zu ermöglichen.

Nachdem die beste Architektur gefunden wurde, wird evaluiert, ob durch die Verwendung von Pruning unwichtige Filter entfernt werden können, um eine noch schnellere Inferenz zu ermöglichen. Aus diesem Grund wurde die Anzahl der Filter zuvor auch nur sehr grob optimiert, da dies am Ende durch das Pruning automatisch geschieht. Des Weiteren ist zu beachten, dass die neuronalen Netze vor dem Pruning lieber etwas zu groß sein sollten, da immer etwas mehr weggeschnitten werden kann.

Die Experimente in diesem Kapitel werden, sofern nicht anders beschrieben, unter den gleichen Bedingungen ausgeführt wie in Kapitel 6.

7.1 SqueezeNet

Bei genauerer Betrachtung des SqueezeNet-Ansatzes fällt auf, dass es eigentlich gar nicht dafür entwickelt worden ist, um schnell zu sein, sondern in erster Linie um Parameter einzusparen. Das führt dazu, dass die MobileNet-Variante, welche der Größenordnung des

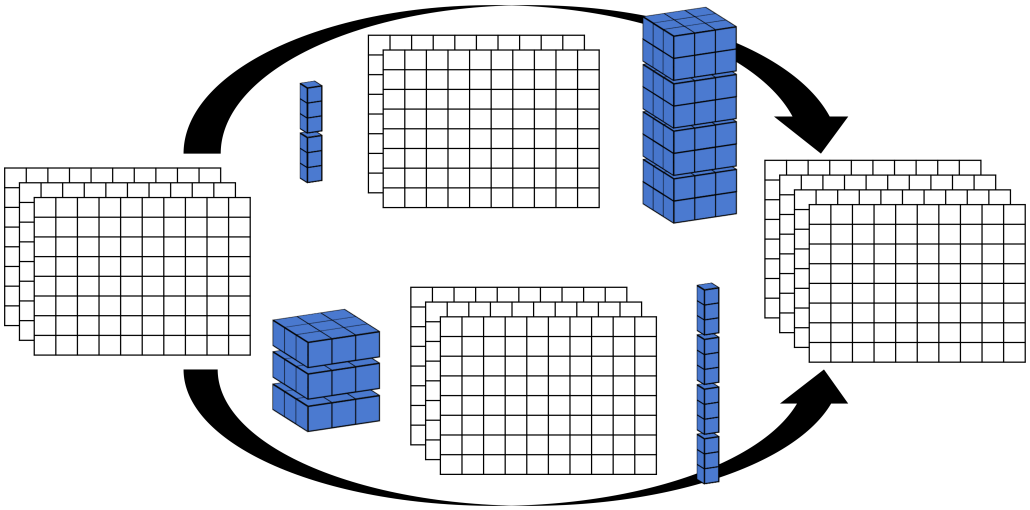


Abbildung 7.1: **SqueezeNet und MobileNet im Vergleich.** Der obere Weg zeigt die Convolution mithilfe eines Squeeze-Layers (links) und Expand-Layers (rechts). Es findet es eine Reduktion von drei Feature Maps auf zwei statt, bevor sie wieder auf vier expandiert werden. Der untere Weg repräsentiert die separierbare Convolution. Dazu werden zunächst alle Feature Maps einzeln gefiltert (links), bevor die Ergebnisse dieser Operation mittels 1x1 Convolutions zu vier Feature Maps kombiniert werden (rechts).

SqueezeNets entspricht, zwar 70.000 Parameter mehr besitzt aber dennoch ca. 22x weniger Berechnungen durchführen muss. Zudem wird in dem SqueezeNet-Paper [Iandola et al., 2016] beschrieben, dass ein Verlust der Genauigkeit eingetreten ist, welcher ausgeglichen wurde, indem das Downsampling verzögert worden ist. Dennoch soll auch das SqueezeNet im Folgenden getestet werden. Dazu wird jedoch die Einschränkung getroffen, dass im Expand-Layer nur 3x3 Convolutions vorkommen dürfen. Dies wird aus zwei Gründen getan: Als erstes soll der Performanceverlust möglichst gut reduziert werden und zweitens soll die lineare Struktur des Netzes beibehalten werden, welches bei einer Kombination von 1x1 und 3x3 Convolutions nicht möglich wäre. Im dazugehörigen Paper wird zusätzlich die Restriktion getroffen, dass die Anzahl der Filter im Squeeze-Layer kleiner sein soll, als die Anzahl der Filter im Expand-Layer. Da hier jedoch die Reduzierung der Rechenzeit im Vordergrund steht, soll diese Restriktion verschärft werden, sodass gewährleistet wird, dass nicht mehr Rechenoperationen ausgeführt werden. Dies ist genau dann der Fall, wenn Gleichung 7.4 erfüllt ist oder n_{inter} noch kleiner ist, wobei n_{in},n_{inter} und n_{out} die Anzahl der Feature Maps vor dem Squeeze-Layer, zwischen beiden Layern und nach dem Expand-Layer beschreiben. Außerdem folgt für den Fall, dass die Anzahl der Feature Maps am Anfang und Ende gleich sind, dass die Anzahl in der Mitte mindestens auf 90% reduziert werden muss.

$$1 \cdot 1 \cdot n_{in} \cdot n_{inter} + 3 \cdot 3 \cdot n_{inter} \cdot n_{out} = 3 \cdot 3 \cdot n_{in} \cdot n_{out} \tag{7.1}$$

$$\Leftrightarrow \quad n_{in} \cdot n_{inter} + 9 \cdot n_{inter} \cdot n_{out} = 9 \cdot n_{in} \cdot n_{out} \tag{7.2}$$

$$\Leftrightarrow \quad n_{inter} \cdot (n_{in} + 9 \cdot n_{out}) = 9 \cdot n_{in} \cdot n_{out} \tag{7.3}$$

$$\Leftrightarrow \quad n_{inter} = \frac{9 \cdot n_{in} \cdot n_{out}}{n_{in} + 9 \cdot n_{out}} \tag{7.4}$$

$$\Rightarrow \quad (n_{in} = n_{out} \Rightarrow n_{inter} = \frac{9 \cdot n_{in}}{10}) \tag{7.5}$$

Umsetzung Da lediglich ein 1x1 Conv-Layer vor die bisherigen 3x3 Conv-Layer geschaltet werden muss, kann die SqueezeNet-Architektur ohne weitere Anpassungen übernommen werden.

Evaluation Für die Evaluation der SqueezeNet-Architektur wird zunächst direkt vor jedes Convolutional-Layer des Feature-Moduls ein Squeeze-Layer eingefügt. Lediglich beim ersten Feature-Modul wird hierauf verzichtet, da die Anwendung auf nur eine Feature Map wenig Sinn hat. Um die Nutzung des Squeeze-Layers zu signalisieren, wird die Schreibweise $x \rightarrow y$ verwendet. Diese gibt an, dass das Squeeze-Layer x Filter besitzt und das nachfolgende Convolutional-Layer y Filter.

Im ersten Experiment wurde eine Reduktion von 24 auf 20 Feature Maps getestet, da dies die erste Konfiguration ist, welche das zusätzliche Squeeze-Layer ausgleichen sollte. Die Ergebnisse sind in Tabelle 7.1 zu finden. Es ist zu sehen, dass die verbrauchte Zeit, im Gegensatz zum Vergleichsnetz aus dem letzten Kapitel ohne Squeeze-Layer, leicht ansteigt. Dennoch fällt die MAP von 0.634 auf 0.618 ab. Um dieses Verhalten zu bestätigen, wurde das Experiment noch einmal mit einer Reduktion auf 16 Feature Maps durchgeführt. Hierbei können 0.3ms bei der Ausführungszeit gut gemacht werden, allerdings fällt der Map-Wert drastisch auf 0.554 ab. Aufgrund dieser Ergebnisse wird auf weitere Experimente verzichtet, da das Vergleichsnetz in den relevanten Kategorien besser ist. Lediglich die Anzahl der Parameter fällt von 53,619 auf 51,071 ab, was zu erwarten war, da genau dies die Stärke der SqueezeNet-Architektur ist.

Größe	Modul	20→24 (83,4%)	16→24 (66,7%)
60x80	Feature	(1,24)	(1, 24)
	Scale	(3x3Conv,24)	
30x40	Feature	(2,20→24)	(2, 16→24)
	Scale	(3x3Conv,24)	
15x20	Feature	(2,20→24)	(2, 16→24)
	Scale	(3x3Conv,24)	
8x10	Feature	(3,20→24)	(3, 16→24)
Gesamtparameter		51,071	44,323
∅ Inferenzzeit		2.7ms	2.3ms
∅ MAP		0.618	0.554

Tabelle 7.1: SqueezeNet Architektur

7.2 MobileNet

Im Gegensatz zum SqueezeNet ist die MobileNet-Architektur explizit darauf ausgelegt, auf mobilen Geräten ausgeführt zu werden. Sie teilt das normale Conv-Layer in zwei Layer auf (siehe Abbildung 7.1). Das erste Layer ist eine separierbare Convolution, welche für jede Feature Map der Eingabe einen eigenen Filter mit einer Tiefe von 1 hat. Die Filtergröße in x- und y-Dimension kann dabei frei gewählt werden, allerdings lohnt sich dies nur sofern der Filter größer ist als 1x1. Nachdem jede Feature Map mit ihrem speziellen Filter gefaltet wurde, werden die Ergebnisse zu gewichteten Summen zusammengesetzt. Dies geschieht durch eine anschließende normale 1x1 Convolution über alle resultierenden Feature Maps. Die Kombination dieser beiden Layer wird im Folgenden als SConv-Layer bezeichnet.

Der große Vorteil der SConv-Layer liegt in der drastischen Reduzierung der Rechenoperationen. Ein weiterer Vorteil der Architektur ist, dass sie ohne weitere Hyperparameter auskommt, sofern man den Width Multiplier außen vor lässt, welcher auch über die Anzahl der Filter gesteuert werden und somit ignoriert werden kann. Diese Eigenschaft macht es allerdings auch komplizierter, den Grad der Beschleunigung zu steuern, da somit wie zuvor nur noch die Anzahl der Layer und die Anzahl der Filter als Parameter bleiben.

Um den Grad der Beschleunigung zu messen, wird die Anzahl der Rechenoperationen mit der eines Conv-Layers verglichen. Nach Gleichung 7.8 bedeutet dies für die aktuell 24 Filter, dass nur 15,3% der Rechenoperationen pro Filteranwendung anfallen. Die zusätzliche Aktivierungsfunktion nach der Separable Convolution wurde hierbei weggelassen, da sie die Rechnung nur unnötig kompliziert machen würde.

$$\frac{3 \cdot 3 \cdot 1 \cdot n_{in} + 1 \cdot 1 \cdot n_{in} \cdot n_{out}}{3 \cdot 3 \cdot n_{in} \cdot n_{out}} = \frac{9 \cdot n_{in} + n_{in} \cdot n_{out}}{9 \cdot n_{in} \cdot n_{out}} \tag{7.6}$$

$$= \frac{9 + n_{out}}{9 \cdot n_{out}} \tag{7.7}$$

$$= \frac{1}{n_{out}} + \frac{1}{9} \tag{7.8}$$

Kombination An dieser Stelle könnte sich der wissbegierige Leser fragen, ob man nicht einfach beide Ansätze kombinieren könnte. Durch ein Squeeze-Layer vor der separierbaren Convolution könnte man diese besser skalierbar machen. Jedoch würde dies bedeuten, dass ausgehend von 24 Layern das Squeeze-Layer die Anzahl der FeatureMaps auf mindestens 57,8% reduzieren müsste (Gleichung 7.12). Es ist sehr unwahrscheinlich, dass eine solche Architektur noch immer die gleiche Leistung vollbringen kann, da das Squeeze-Layer einen gewaltigen Engpass darstellen würde.

$$1 \cdot 1 \cdot n_{in} \cdot n_{inter} + 3 \cdot 3 \cdot 1 \cdot n_{inter} + 1 \cdot 1 \cdot n_{inter} \cdot n_{out} = 3 \cdot 3 \cdot 1 \cdot n_{in} + 1 \cdot 1 \cdot n_{in} \cdot n_{out} \tag{7.9}$$

$$\Leftrightarrow \quad n_{in} \cdot n_{inter} + 9 \cdot n_{inter} + n_{inter} \cdot n_{out} = 9 \cdot n_{in} + n_{in} \cdot n_{out} \tag{7.10}$$

$$\Leftrightarrow \quad (n_{in} + 9 + n_{out}) \cdot n_{inter} = (9 + n_{out}) \cdot n_{in} \tag{7.11}$$

$$\Leftrightarrow \quad n_{inter} = \frac{(9 + n_{out}) \cdot n_{in}}{n_{in} + 9 + n_{out}} \tag{7.12}$$

Umsetzung Die B-Human-Toolchain für neuronale Netze besitzt noch keine Implementierung für SConv-Layer, weshalb diese zunächst implementiert werden mussten. Beim Exportieren wird dabei der *SeparableConv2D*-Layer von Keras direkt in zwei einzelne Layer aufgetrennt, um die 1x1 Convolution am Ende einfach über die bisherige Implementierung für Conv-Layer auszuführen. Für das vorangehende SConv-Layer wird ein neuer Layer für die Toolchain implementiert, welcher aus einer Kopie des bereits bestehenden Conv-Layers entstanden ist.

Evaluation Für die Evaluation der MobileNet Architektur werden Teile der Conv-Layer durch SConv-Layer ersetzt. Dabei wird das erste Layer außen vor gelassen, da der SConv-Layer bei nur einem Layer keinen Vorteil bringt. Um im Folgenden die Nutzung des SConv-Layers anzudeuten, wird die Notation x_s verwendet, wenn x Filter verwendet werden sollen.

Für das erste Experiment werden wie zuvor nur die Feature-Module mit den neuen Layern ausgestattet. Die Ergebnisse (Tabelle 7.2) zeigen, dass zwar auch hier der MAP-Wert sinkt, allerdings fällt auch die Inferenzzeit stark ab, sodass eine Vergrößerung des Netzes möglich wird. Aus diesem Grund wird das Experiment für mehr Layer und mehr Filter wiederholt. Eine Verdopplung der Layer führt dabei zu einem guten MAP-Wert von 0,616 bei einer nur um 0,3 ms langsameren Inferenzzeit. Die Erhöhung auf 36 Filter, erreicht gar eine MAP von 0,656, was dem Ergebnis von 48 Filtern mit Conv-Layern aus dem letzten Kapitel sehr nah kommt. Erstaunlicherweise werden dabei 4,6 ms weniger benötigt, welches aber dennoch zu langsam sein dürfte. Aufgrund dieser vielversprechenden Ergebnisse soll an dieser Stelle noch getestet werden, wie sich die Nutzung der SConv-Layer in den Scale-Modulen auswirkt. Dafür wird zunächst nur das erste Scale-Modul ersetzt, da dies den größten Einfluss auf die Inferenzzeit hat, und anschließend wir das komplette Netz durch SConv-Layer ersetzt. Dabei fällt auf, dass die komplette Ersetzung schneller ist und dennoch ein marginal besseres Ergebnis erzielt.

Größe	Modul	nur Feature 24 Filter		nur Feature 36 Filter	Feature/1.Scale 24 Filter	komplett 24 Filter
60x80	Feature	(1,24)		(1, 36)	(1,24)	(1,24)
	Scale	(3x3Conv,24)		(3x3Conv,36)	(3x3Conv,24_s)	(3x3Conv,24_s)
30x40	Feature	(2,24_s)	(4, 24_s)	(2, 36_s)	(2, 24_s)	(2, 24_s)
	Scale	(3x3Conv,24)		(3x3Conv,36)	(3x3Conv,24)	(3x3Conv,24_s)
15x20	Feature	(2,24_s)	(4, 24_s)	(2, 36_s)	(2, 24_s)	(2, 24_s)
	Scale	(3x3Conv,24)		(3x3Conv,36)	(3x3Conv,24)	(3x3Conv,24_s)
8x10	Feature	(3,24_s)	(6, 24_s)	(3, 36_s)	(3, 24_s)	(3, 24_s)
Gesamtparameter		22.875	28.923	48.987	18.483	9.699
∅ Inferenzzeit		1,4 ms	1,7 ms	3,4 ms	1,0 ms	0,8 ms
∅ MAP		0,584	0,616	0,656	0,575	0,576

Tabelle 7.2: MobileNet Architektur: erste Versuchsreihe

Mithilfe dieser ersten Ergebnisse soll nun eine iterative Verbesserung der Architektur erfolgen. Auf Grund der sehr guten Inferenzzeit wird als erstes das Netz, welches nur auf

SConv setzt, um 12 weitere Filter pro Layer erweitert. Das Ergebnis (Tabelle 7.3) zeigt, dass ein MAP-Wert erreicht werden kann, welcher ungefähr dem des Vergleichsnetzes entspricht, während nur 55% der Inferenzeit benötigt werden. Da sowohl die Verdopplung der Layer, als auch die Erhöhung der Filter auf 36 zu langsam waren, soll eine Mischung beider Varianten getestet werden. Deshalb wird die Anzahl der Layer mit jeder Ebene um 1 erhöht und die Anzahl der Filter auf 32 gesetzt. Dabei wird eine gute Inferenzzeit von 1,3 ms erreicht, während allerdings auch nur eine MAP von 0,596 erzielt werden kann. Zum Schluss soll noch einmal versucht werden, die Anzahl der Parameter dadurch zu erhöhen, dass in der letzten Ebene, ein normaler Conv-Layer genutzt wird. Dabei soll die „1234“Struktur erhalten bleiben. Überraschenderweise ist das Resultat schlechter als zuvor. Das liegt daran, das Modell overfittet, da so viele Parameter in den letzten Layern zu finden sind, wo sie den größten Einfluss haben. Das Overfitting ist daran zu erkennen, dass die Trainingsdaten mit einer MAP von 0,743 wesentlich besser abschneiden, als das vorherige Netz mit 0,695. Dieses Problem könnte sich auflösen, wenn auf dem gesamten Datensatz trainiert wird. Nichtsdestotrotz soll noch ein etwas kleineres Netz mit 24 Filtern trainiert werden. Dieses Netz erreicht einen MAP-Wert von 0,613, während es auch nur 1 ms für die Inferenz benötigt und ist damit schon vor dem Pruning vermutlich so gut, dass es direkt auf dem Roboter verwendet werden kann.

		komplett		ohne 4. Feature	
Größe	Modul	1223 36	1234 32	1234 32	1234 24
60x80	Feature	(1, 36)	(1, 32)	(1, 32)	(1,24)
	Scale	(3x3Conv,36_s)	(3x3Conv,32_s)	(3x3Conv,32_s)	(3x3Conv,24_s)
30x40	Feature	(2,36_s)	(2, 32_s)	(2, 32_s)	(2, 24_s)
	Scale	(3x3Conv,36_s)	(3x3Conv,32_s)	(3x3Conv,32_s)	(3x3Conv,24_s)
15x20	Feature	(2,36_s)	(3, 32_s)	(3, 32_s)	(3, 24_s)
	Scale	(3x3Conv,36_s)	(3x3Conv,32_s)	(3x3Conv,32_s)	(3x3Conv,24_s)
8x10	Feature	(3,36_s)	(4,32_s)	(4, 32)	(4, 24)
Gesamtparameter		18.855	18.299	38.751	28.995
∅ Inferenzzeit		1,5 ms	1,3 ms	1,5 ms	1,0 ms
∅ MAP		0,632	0,596	0,573	0,613

Tabelle 7.3: MobileNet Architektur: zweite Versuchsreihe

Um sich abschließend für eine Architektur entscheiden zu können, werden die drei favorisierten Ansätze genauer untersucht. Als erstes werden die bereits trainierten Netze genommen und auf dem gesamten Datensatz weiter trainiert, wobei das gleiche Abbruchkriterium gilt wie zuvor. Die Ergebnisse in Tabelle 7.4 zeigen, dass die beiden Netze mit den Conv-Layern in der letzten Ebene hier ihre Stärke ausspielen können, da ein Overfitting aufgrund der größeren Trainingsmenge schwieriger geworden ist. Die Inferenzzeiten auf dem Roboter zeigen jedoch, dass dort deutlich mehr Zeit verbraucht wird, als auf dem Laptop. Dabei bleiben die Verhältnisse zwischen den getesteten Ansätzen in etwa gleich. Aufgrund dieses deutlichen Anstiegs der Inferenzzeit soll im Folgenden die Konfiguration des kleinen Netzes verwendet werden, da die Inferenzzeit hier am geringsten ist und durch das nachfolgende Pruning vermutlich noch etwas gesenkt werden kann.

Architektur	∅Inferenzzeit auf dem Nao	∅MAP (gesamter Datensatz)
1223 36 (komplett)	7,8 ms	0,662
1234 32 (ohne 4. Feature)	7,8 ms	0,686
1234 24 (ohne 4. Feature)	6,0 ms	0,679

Tabelle 7.4: MobileNet Architektur: dritte Versuchsreihe

7.3 Pruning

Nachdem im vorherigen Abschnitt ein neuronales Netz gefunden wurde, welches sowohl schnell ist, als auch einen guten MAP-Wert erreicht, soll nun versucht werden die Anzahl der Filter zu reduzieren, ohne dabei den MAP-Wert zu stark zu senken. Dafür sollen mittels Pruning unwichtige Filter aus dem Netz entfernt werden. Abbildung 7.2 visualisiert diesen Vorgang für den Conv- und SConv-Layer. Für den zu entwickelnden Objektdetektor sollen iterativ immer wieder Bündel von vier Filtern entfernt werden, da die genutzte Toolchain für Vielfache von 4 optimiert ist. Für die Indentifikation der Filter mit dem geringsten Informationsgehalt, soll die normierte Aktivierung genutzt werden. Zwar hat sich dies in [Molchanov et al., 2016] nicht als die beste Methode herausgestellt, allerdings hat sie dennoch gute Ergebnisse erzielt und ist wesentlich einfacher umzusetzen. Da zu diesem Zeitpunkt noch nicht bekannt ist, wie gut das Pruning bei so kleinen Netzen überhaupt funktioniert, soll zunächst dieser einfache Ansatz getestet werden. Dadurch, dass immer vier Filter auf einmal entfernt werden sollen, wird der Einfluss des Auswahlalgorithmus sowieso abgeschwächt. Die Findung der nächsten zu entfernenden Filter funktioniert dabei in folgenden zwei Schritten.

1. Für jeden Layer werden die Aktivierungen aller Filter normiert, sodass ihre Summe 1 ergibt. Anschließend werden die vier Filter mit der geringsten Aktivierung gesucht und deren normierte Aktivierung aufaddiert. Zum Schluss wird der Layer mit dem geringsten Wert für das Pruning ausgewählt.

2. Da das aktuelle neuronale Netz nur sehr wenige Filter besitzt, kann die Entfernung jedes Filters einen großen Schaden anrichten, welcher ungemein größer wird, wenn vier auf einmal entfernt werden. In [Molchanov et al., 2016] wurde jedoch gezeigt, dass das Feintuning den Schaden gut auffangen kann. Um dennoch den Schaden möglichst gering zu halten, soll immer nur ein Filter entfernt werden, um anschließend Feintuning betreiben zu können. Nun kann sich beim Feintuning jedoch die Wichtigkeit der Filter ändern. Aus diesem Grund wird nach jedem Feintuning erneut der Filter mit der geringsten Aktivierung bestimmt, wobei das Layer solange dasselbe bleibt, bis vier Layer entfernt worden sind. Sobald dies der Fall ist, wird wieder bei Schritt 1 begonnen, um ein neues Layer zu finden.

Das Feintuning versucht immer den F_1-Score, welcher vor dem Pruning erreicht wurde, wieder zu erreichen. Sollte dies über zehn Lerndurchgänge nicht möglich sein, so wird

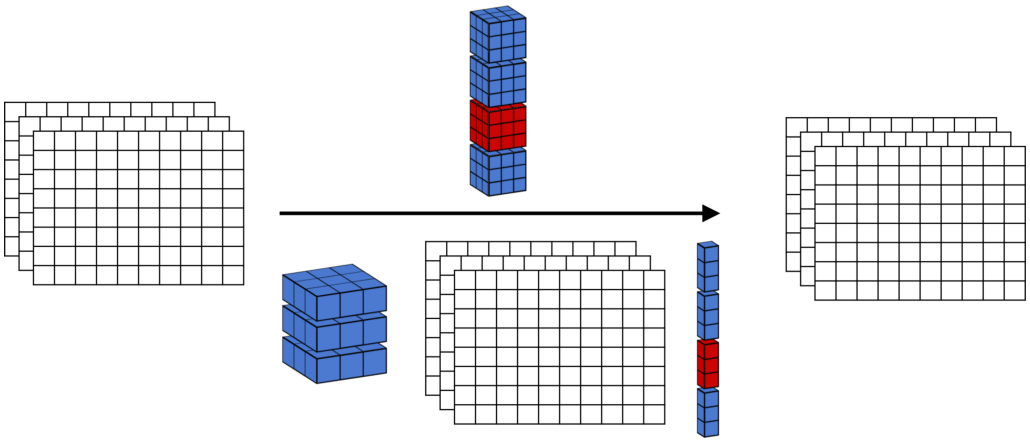


Abbildung 7.2: **Funktionsweise von Pruning.** Links im Bild sind die Feature Maps zusehen, welche als Eingang fungieren, und rechts die Ausgangs Feature Maps, nachdem das Pruning ausgeführt worden ist. In der Mitte ist oben die normale Covolution dargestellt und unten die separierbare Convolution. Die rot markierten Filter repräsentieren Filter, die für das Pruning ausgewählt worden sind. Bei der separierbaren Convolution ist zu beachten, dass man lediglich bei der zweiten Filteranwendung Pruning verwenden darf.

die Erwartungshaltung um 0,01 gesenkt. Nach jeder Entfernung von vier Filtern wird ein Checkpoint gesetzt, sodass man am Ende auswählen kann, wie viel Verlust beim MAP-Wert man eingehen möchte.

Umsetzung Für das Pruning wird die Python-Bibliothek *Keras-surgeon* (siehe `https://github.com/BenWhetton/keras-surgeon`) verwendet. Allerdings bietet diese keine Unterstützung für SConv-Layer, weshalb diese Funktionalität im Rahmen dieser Arbeit entwickelt und in die Bibliothek integriert worden ist. Dadurch ist es nun auch möglich SConv-Layer zu prunen.

Um die Aktivierung der Filter zu erhalten wurde die Python-Bibliothek *keract* (siehe `https://github.com/philipperemy/keras-activations`) verwendet.

Typ	Filter	Filter Größe	Schritte	Output
Conv	16	3x3	1	80x60
SConv	24	3x3	2	40x30
SConv	16	3x3	1	40x30
SConv	20	3x3	1	40x30
SConv	20	3x3	2	20x15
SConv	20	3x3	1	20x15
SConv	20	3x3	1	20x15
SConv	24	3x3	1	20x15
SConv	24	3x3	2	10x8
Conv	24	3x3	1	10x8
Conv	24	3x3	1	10x8
Conv	24	3x3	1	10x8
Conv	24	3x3	1	10x8
Conv	20	1x1	1	10x8

Tabelle 7.5: Netz nach dem Pruning

Netz	∅MAP	F_1-Score
vor Pruning	0,675	0,845
nach Pruning	0,646	0,847
neu trainiert	0,635	0,838

Tabelle 7.6: Ergebnisse des Prunings

Evaluation Für die Evaluation dieses Ansatzes wurde das Netz gepruned, das im vorherigen Abschnitt ausgewählt worden ist. Allerdings wurde es zwischenzeitlich weiter trainiert auf dem gesamten Datensatz, sodass vor Beginn des Prunings ein F_1-Score von 0.845 und eine MAP von 0,687 vorliegen. Das Pruning wurde dabei solange ausgeführt, wie das Netz sich wieder auf einen F_1-Score von mindestens 0,84 erholen konnte. Dies war bis zur der Entfernung des achten Bündels der Fall. In Tabelle 7.5 ist die Architektur des Netzes nach dem Pruning zu sehen. Es fällt auf, dass lediglich am Anfang Filter entfernt worden sind. Dies ist ein gewünschtes Verhalten, da diese Filter aufwändiger sind. Tabelle 7.6 zeigt die Ergebnisse vor und nach dem Pruning. Zusätzlich wurde das Netz mit dem Aufbau, welcher durch das Pruning erreicht wurde, noch einmal von Grund auf neu trainiert. Es fällt auf, dass das Netz, auf das Pruning angewendet worden ist, leicht bessere Ergebnisse erzielt, als das neu trainierte. Vermutlich würde dieser Effekt zunehmen, wenn mehr Filter gepruned werden würden. Des Weiteren bietet Pruning aber auch den Vorteil, dass man sich weniger Gedanken um die Filteranzahl machen muss, da man einfach mit sehr vielen Filtern starten und dann die unwichtigen abschneiden kann. Da der MAP-Wert durch dieses Verfahren kaum schlechter geworden ist, aber gleichzeitig die Inferenzzeit auf dem Nao von 6 ms auf 4,5 ms gesunken ist, soll das Netz aus Tabelle 7.5 von nun an verwendet werden.

8 Depth Learning

Der Nao besitzt zwei Kameras, wovon die obere ungefähr in die Richtung zeigt, in welche der Kopf gerichtet ist, während die untere wesentlich tiefer blickt (siehe Kapitel 2.3). Das führt dazu, dass die untere Kamera meistens nur den Boden zeigt und selten auch Bälle oder Füße von Robotern. Allerdings sind die Roboter so groß, dass sie im normalen Spielverlauf niemals nur in der unteren Kamera zu sehen sind. Da die untere Kamera somit keinen großen Beitrag zur Detektion leisten kann, wird komplett darauf verzichtet ihr Kamerabild auszuwerten. Ein weiterer Grund der dafür spricht, nur die obere Kamera zu benutzen, ist die Einsparung der Inferenz für das untere Bild. Allerdings wurde das Bild der unteren Kamera bisher dazu verwendet, um anhand der Fußposition die Entfernung der Roboter zu bestimmen. Sofern nur die obere Kamera genutzt wird, befinden sich die Füße allerdings oftmals nicht im Bild. Um den Verlust dieser Funktion aufzuwiegen, soll der Detektor um eine Distanzbestimmung erweitert werden.

Datenakquise Die benötigten Trainingsdaten können allerdings nicht so einfach per Hand gelabelt werden, da man mit dem bloßen Auge nicht erkennen kann, wie weit ein Roboter entfernt ist. Allerdings existiert dieses Problem lediglich in der Realität, da in der Simulation die Positionen aller Roboter bekannt sind. Das Autolabeltool aus Kapitel 5.1 kann somit relativ einfach um die Distanz erweitert werden. Dazu wird für jeden gefundenen Roboter die Position auf dem Spielfeld ausgelesen und dann in das Koordinatensystem des Roboters verschoben, dessen Kamera gerade ausgewertet wird.

Generative Adversarial Networks Nun steht man vor dem Problem, dass man die Distanzwerte nur für die Simulationsbilder hat, während die Detektion später hauptsächlich in der Realität stattfinden soll. Um nun das neuronale Netz mit den Distanzwerten aus der Simulation trainieren zu können, darf das neuronale Netz die beiden Quellen nicht mehr unterscheiden können. Sobald dies der Fall ist, kann man das neuronale Netz einfach nur noch auf den Simulationsdaten trainieren und die Ergebnisse werden implizit auch für die Bilder aus der Realität besser.

Das Problem Bildquellen ununterscheidbar zu machen, ist bereits von den Generative Adversarial Networks (GANs) [I. J. Goodfellow et al., 2014] bekannt. Die Lösung dort ist, dass man die Bilder aus der Simulation durch einen Generator laufen lässt, dessen Aufgabe es ist, das Bild möglichst real aussehen zu lassen. Anschließend folgt ein Diskriminator, der versucht den realen Bildern und der Ausgabe des Generators die richtigen Klassen zuzuweisen. Dieser Aufbau ist in Abbildung 8.1 zusehen und wird iterativ in zwei sich abwechselnden Phasen trainiert:

1. Der Diskriminator wird eingefroren. Anschließend wird der Generator daraufhin trainiert, dass der Diskriminator in seinem aktuellen Zustand die beiden Quellen nicht mehr unterscheiden kann.

2. Der Generator wird eingefroren. Anschließend wird der Diskriminator daraufhin trainiert, dass er die beiden Quellen möglichst gut trennen kann.

Diese Art des Trainings ist extrem kompliziert, da die Stärken der beiden Gegenspielen (Generator und Diskriminator) sehr ausgeglichen sein müssen, da sonst einer von beiden so gut wird, dass der andere keine Chance mehr hat. Allerdings möchte man, dass beide sich gegenseitig immer wieder neuen Herausforderungen stellen, da es nur so möglich wird, dass man an Ende einen guten Generator bekommt. Um zu präsentieren, welche eindrucksvollen Möglichkeiten GANs besitzen, wird kurz das CycleGAN [Zhu et al., 2017] vorgestellt. Mit dieser Art GAN ist es beispielsweise möglich, Pferde in Bildern in Zebras zu transformieren und andersherum (siehe Abbildung 8.2 & 8.3). Da es sehr mühsam wäre, für jedes Pferdebild ein exakt gleiches Bild mit einem Zebra zu finden, sind CycleGANs dazu in der Lage, ohne eine direkte Zuordnung der Bilder zu lernen. Das heißt, man benötigt lediglich einen Datensatz mit Pferdebildern und einen mit Zebrabildern. Der Begriff Cycle kommt

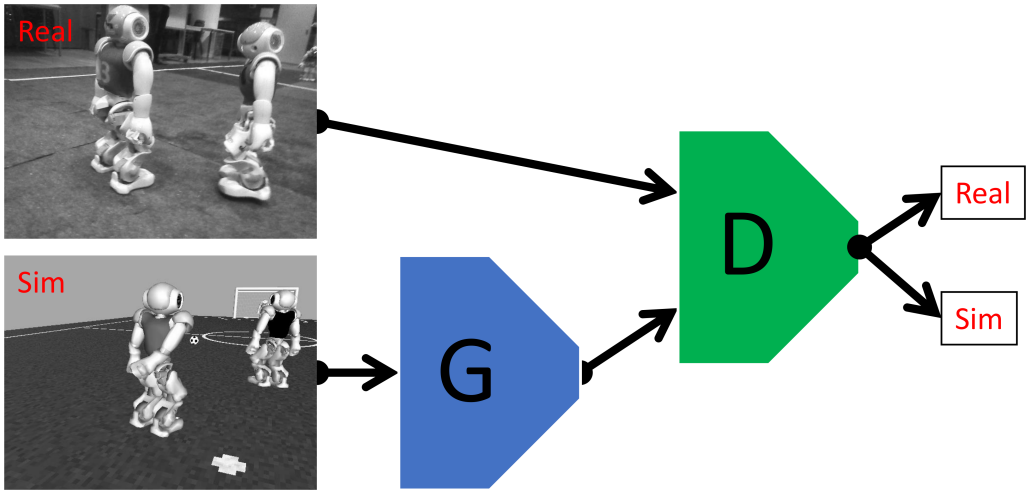


Abbildung 8.1: **Aufbau von Generative Adversarial Networks.** Bilder aus der Simulation (Sim) werden zunächst an den Generator (G) übergeben. Anschließend versucht ein Diskriminator (D) den Bildern aus der Realität (Real) und den generierten die richtigen Label zuzuordnen.

daher, dass man für das Training zum Beispiel ein Pferdebild nimmt, dieses mit einem Generator in ein Zebrabild umwandelt, und mit einem zweiten Generator wieder zurück. Dabei ist es das Ziel den Fehler zwischen den Ursprungsbild und dem generierten Bild zu minimieren.

Simulation oder Realität? Nun ist das aktuelle Netz so trainiert, dass es sowohl Bilder aus der Realität, als auch Bilder aus dem Simulator verarbeiten kann. Damit das Netz dies kann, muss es versuchen Features zu finden, welche sowohl in der Realität, als auch in der Simulation zu finden sind. Ist dies nicht möglich, müssten zwei unterschiedliche Feature-Mengen gefunden werden. Da in Kapitel 6.1 bereits gezeigt worden ist, dass die Nutzung von Simulationsdaten zu einem besseren Ergebnis für Daten aus der Realität beiträgt, kann man davon ausgehen, dass sich ein großer Teil der Features für beide Quellen gültig ist. Daher kann man insbesondere annehmen, dass die Features mit zunehmender Tiefe des Netzes immer uneindeutiger werden. Um dies zu bestätigen, soll ein Versuch gemacht werden, bei dem das Netz nach dem vorletzten Layer abgeschnitten wird und stattdessen ein Diskriminator trainiert wird, welcher versuchen soll, die dort vorliegenden Feature Maps in Simulation und Realität zu trennen. Der Aufbau den man dadurch erhält (siehe Abbildung 8.5), ähnelt dem eines GANs, mit dem einzigen Unterschied, dass sowohl die Simulationsbilder, als auch die Bilder der Realität durch den Generator müssen, bevor sie zum Diskriminator kommen. Die Ergebnisse in Tabelle 8.1 zeigen, dass durch das spezielle GAN-Training die Accuracy von 75,7% auf 64,2% gedrückt werden kann. Da ein eine zufällige Zuteilung der Label im Schnitt eine Accuracy von 50% erreichen sollte, ist dieses Ergebnis schon sehr ordentlich. Der in Abbildung 8.4 dargestellte Verlauf des Trainings zeigt, dass sich zwar die Accuracy der Trainingsdaten stetig verbessert, aber die Accuracy der Testdaten hat einen willkürlichen Verlauf hat. Das deutet darauf hin,

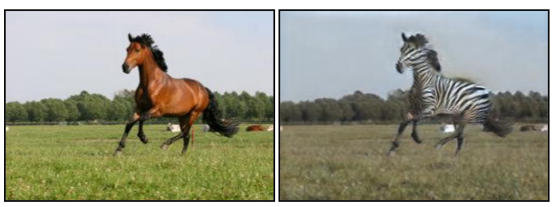


Abbildung 8.2: **CycleGAN: Pferd zu Zebra.** Entnommen aus [Zhu et al., 2017]

Abbildung 8.3: **CycleGAN: Zebra zu Pferd.** Entnommen aus [Zhu et al., 2017]

dass hier keine sinnvollen Features für die Unterscheidung der beiden Quellen gefunden werden, sondern stark auf die Trainingsmenge overfittet wird. Um sicherzustellen, dass beide Quellen ungefähr gleich gut zugeordnet werden können, ist in Tabelle 8.1 zusätzlich die Accuracy für den Teil der Trainingsdaten, die aus der Simulation stammen, dargestellt. Es ist zu sehen, dass dieser Wert bei beiden Versuchen in etwa gleich ist, was zum einen darauf hindeutet, dass durch das GAN-Training insbesondere die Accuracy der realen Daten gesunken ist. Zum anderen bedeutet dies, dass die Accuracy-Werte relativ nah beieinander sind. Für die Auswertung konnte auf die Nutzung der Balanced Accuracy verzichtet werden, da ein ausgeglichenes Verhältnis zwischen Daten aus der Realität und Daten aus der Simulation bestand. Diese Ergebnisse bedeuteten letztendlich, dass das GAN-Training verwendet werden sollte und die erzeugten Features vor dem letzten Layer keinen Aufschluss mehr darüber geben, ob ein Bild real oder synthetisch ist.

Art	Accuracy (Trainingsdaten)	Accuracy (nur Simulationsdaten)
ohne GAN-Training	75,7%	58,4%
mit GAN-Training	64,2%	58,2%

Tabelle 8.1: Trennbarkeit von Simulation und Realität

Aus der Simulation in die Realität Um die derzeitige Architektur dahingehend zu erweitern, dass auch Entfernungen vorhergesagt werden können, wird für jede Bounding Box nun zusätzlich noch die Entfernung als sechster Wert mit vorhergesagt. Dabei wird die Entfernung bei 10m abgeschnitten und der Bereich für das Lernen auf das Intervall zwischen 0 und 1 abgebildet, sodass auch hier zum Lernen eine Sigmoid-Funktion verwendet werden kann. Um nun die Vorhersage der Distanz zu trainieren, wird die Lossfunktion um einen Term für die Distanz erweitert. Dieser berechnet den MSE, allerdings nur für Bilder aus der Simulation. Für Bilder aus der Realität ist das Distanz-Loss immer 0. Würde man mit dieser Lossfunktion jetzt das neuronale Netz einfach so weiter trainieren, würde das dazu führen, dass die Distanzwerte für die synthetischen Daten gute Ergebnisse liefern, während für die realen Daten sehr schlechte Ergebnisse zu erwarten sind. Das liegt daran, dass sich das Netz explizit darauf konzentrieren kann, die Features zu suchen, die bei der Distanzbestimmung auf den synthetischen Daten helfen. Um dies zu verhindern, wird der Teil des Netzes, der im letzten Abschnitt als Generator definiert worden ist, eingefroren. Dadurch ist der letzte Layer dazu gezwungen, aus den vorliegenden Features die Distanz

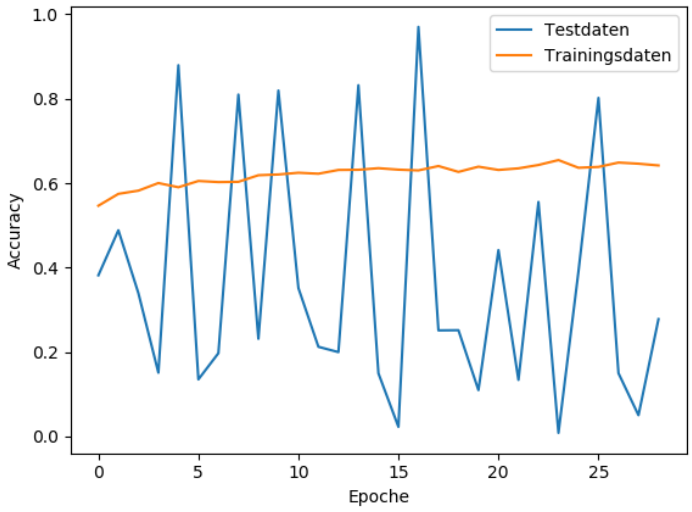


Abbildung 8.4: Verlauf des GAN-Trainings

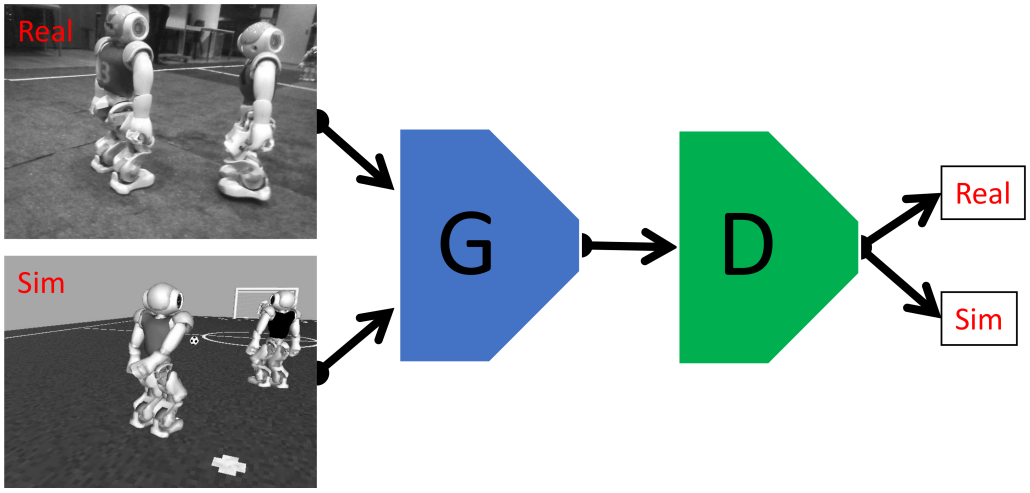


Abbildung 8.5: **Alternativer Aufbau von Generative Adversarial Networks.** Bilder aus der Simulation (Sim) und die realen Bilder (Real) werden zunächst an den Generator (G) übergeben. Anschließend versucht ein Diskriminator (D) den generierten Bildern die richtigen Label zuzuordnen.

zu bestimmen. Da diese Features aber keine Hinweise mehr darüber enthalten, woher die Daten stammen, wird nicht nur die Distanzberechnung für Daten der Simulation optimiert, sondern auch die für die realen Daten. Selbst wenn es gewollt wäre, könnte man nun nicht mehr nur die synthetischen Daten verbessern. Allerdings funktioniert dieses Verfahren nur dann, wenn die neue Zielvariable, welche nur Label aus der Simulation besitzt, aus den bisherigen Features berechnet werden kann. Da für die Bestimmung der Bounding Box ein gewisses Verständnis für die Distanz vorhanden sein muss, ist es sehr wahrscheinlich, dass sich auch die Distanz mit diesen Features berechnen lässt. Vermutlich ist es allerdings nicht möglich zusätzlich noch Bälle oder Torpfosten zu erkennen, ohne dafür passende Features extrahiert zu haben. Das Distanzloss konnte beim Training auf $7,7116e-4$ reduziert werden. Dies lässt vermuten, dass sich die Features der Bounding-Box-Vorhersage auch gut für die Bestimmung der Distanz nutzen lassen. Für realen Bilder wurde lediglich die Plausibilität geprüft, also ob Robotern die weiter weg stehen eine höhere Distanz zugeordnet wird und ob die Distanzen ungefähr stimmen. Dabei kam heraus, dass die Plausibilität in fast allen Fällen gegeben ist. Eine genauere Evaluation der Distanzbestimmung in der Realität folgt im nächsten Kapitel. Es sei zu beachten, dass durch die genutzten Verfahren der MAP-Wert auf 0,650 gefallen ist. Dieser Verlust wird allerdings durch die neue Distanzvorhersage aufgewogen.

9 Nachverarbeitung

Bevor im nächsten Kapitel die Evaluation gemacht werden kann, wird der Detektor noch um eine Nachverarbeitung ergänzt, welche ein paar Fehlerkennungen entfernt, wozu unter anderem domänenspezifisches Wissen genutzt wird. Für die Anwendung des Objekterkenners auf dem Nao können einige einfache Annahmen getroffen werden, indem ein Teil der erkannten Boxen aussortiert werden:

Non Maximum Suppression Nachdem das neuronale Netz Bounding Boxes vorhergesagt hat, müssen diese noch aussortiert werden, da das Lernproblem so gestaltet worden ist, dass mehr als eine Box für jedes Objekt erkannt wird. Da viele der vorgestellten Objektdetektoren dies genauso machen, soll auch hier der gleiche Lösungsansatz hier verwendet werden: Non Maximum Suppression (NMS). Die NMS sortiert alle Bounding Boxes aus, welcher von einer anderen Box verdeckt werden, die eine höhere Wahrscheinlichkeit besitzt ein Objekt zu beinhalten. Dabei meint verdeckt, dass die beiden Boxen einen gemeinsamen IoU-Wert von über 0,3 haben.

Begrenzung des Spielfeldes Außerdem wird noch ein Standardverfahren aus der SPL verwendet, wobei Objekte außerhalb des Spielfeldes ignoriert werden, da diese für das Spiel irrelevant sind. Im B-Human-Framework gibt es bereits Routinen, die die Spielfeldbegrenzung erkennen. Dieses Wissen soll genutzt werden, um Bounding Boxes auszusortieren, die ihren untersten Punkt außerhalb der Spielfeldbegrenzung haben, wobei ein kleiner Übergangsbereich von 20 Pixeln erlaubt wird. Damit werden zwar auch Roboter aussortiert, die außerhalb des Feldes sind, allerdings sind diese sowieso nicht wichtig für das Spielgeschehen. Viel wichtiger ist hierbei, dass sämtliche Zuschauer und sonstige Objekte neben dem Spielfeld aussortiert werden, da das neuronale Netz aufgrund seiner geringen Größe und Auflösung hier hin und wieder Fehlerkennungen macht.

Entfernungsabgleich Neben den beiden etablierten Verfahren, soll noch ein neues Verfahren getestet werden, welches die vorhergesagte Entfernung betrachtet. Da es nun die Möglichkeit gibt, die Distanz zu schätzen, soll diese nun genutzt werden, um Bounding Boxes auszusortieren, bei denen die geschätzte Distanz zu stark von der gemessenen Distanz abweicht. Um die Distanz überhaupt zu messen, wird das Standard-Verfahren aus dem B-Human-Framework genutzt. Dieses betrachtet die Stelle im Kamerabild, an der das jeweilige Objekt den Boden berührt, und projiziert diesen Punkt auf das Spielfeld. Der Nachteil an dieser Variante ist, dass die Stelle, an der sich Objekt und Boden berühren, im Kamerabild sein muss. Sofern dies nicht der Fall ist, kann die Entfernung nicht gemessen werden. Es wird davon ausgegangen, dass dies Kriterium erfüllt ist, wenn die Bounding Box mindestens 50 Pixel vor dem unteren Bildrand endet. Dann kann der unterste Punkt in der Mitte der Bounding Box verwendet werden, um ihn auf das Spielfeld zu projizieren und somit dessen Entfernung zu bestimmen. Wenn nun eine der beiden Distanzen mehr als das Doppelte der anderen beträgt, wird die Bounding Box aussortiert. Dieses Verfahren soll im Folgenden evaluiert werden. Insbesondere soll dabei evaluiert werden, ob dieses Verfahren die Entfernung von Objekten außerhalb des Spielfeldes überflüssig macht, da bei diesen Objekten das Größenverhältnis nicht passen dürfte.

10 Evaluation des Objektdetektors

Dieses Kapitel befasst sich mit der finalen Auswertung des entwickelten Objektdetektoren. Bislang wurde der Detektor lediglich auf exportierten Bildern in Python getestet und ist nur für die Messung der Inferenzzeit in das B-Human-Framework exportiert worden. Nun soll schließlich auch die Erkennung auf dem Nao getestet werden.

10.1 Vergleich mit YOLO

Zu Beginn dieser Evaluation soll eine finale Evaluation in Python stattfinden. Dazu wird das entwickelte neuronale Netz mit der Netzarchitektur von YOLO v2 [Redmon und Farhadi, 2017] verglichen. Es wird die kleinste mögliche Eingabebildgröße verwendet, welche sich auf 224x224 beläuft. Dadurch ist der Layer, in dem die Bounding-Box-Vorhersage geschieht, 7x7 Pixel groß. Es wird jedoch eine Anpassung vorgenommen: Die Eingabe soll kein RGB-Bild sein, sondern lediglich ein Graustufenbild. Dies sorgt zum einen für faire Vergleichsbedingungen und zum anderen reduziert dies den Speicher, der während des Trainings auf der Grafikkarte benötigt wird. Da lediglich eine NVIDIA GeForce GTX 1060 mit 6 GB Speicher zur Verfügung steht, muss die Batchgröße dennoch von 128 auf 32 reduziert werden, was das Training stark verlangsamt. Ansonsten wird das neuronale Netz unverändert aus der genutzten Grundlage keras-yolo2 übernommen. Für das Training werden die Lossfunktionen und Trainingsdaten verwendet, welche im Verlauf dieser Arbeit entwickelt und genutzt worden sind. Es wird auf die Vorhersage der Distanz verzichtet. Für den Vergleich wird jedoch das neuronale Netz verwendet, welches sich nach der Distanzvorhersage ergeben hat, da dies die finale Version darstellt, welche auch später auf dem Roboter genutzt werden soll. Um beide Netze zu vergleichen, wird die MAP bei unterschiedlichen IoU-Schwellwerten von 0,1 bis 0,9 in Tabelle 10.1 gegenübergestellt. Es ist zu sehen, dass bis 0,2 beide Netze einen ähnlich guten Wert erzielen können. Erst mit einem größer werdenden Schwellwert kann sich die YOLO v2 Architektur absetzen, wobei die maximale Differenz bei einem Schwellwert von 0.7 erreicht wird. Das lässt vermuten, dass das größere Netz von YOLO v2 lediglich eine bessere Bounding-Box-Regression besitzt, während die Klassifizierung, ob ein Pixel einen Roboter enthält, bei beiden ähnlich ist.

	∅MAP								
Architektur	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
diese Arbeit	0,773	0,776	0,761	0,725	0,652	0,527	0,347	0,156	0,026
YOLO v2	0,773	0,779	0,774	0,764	0,740	0,689	0,560	0,318	0,067

Tabelle 10.1: Vergleich mit YOLO v2

10.2 Verschiedene Distanzen

Um zu testen, wie gut der Objektdetektor auf dem echten Roboter performt, soll zunächst im Stand für verschiedene Entfernungen getestet werden, ob ein Roboter erkannt wird und ob die geschätzte Entfernung stimmt. Da die Roboter im Spiel die unterschiedlichsten Posen annehmen können, wird werden für jede Entfernung zudem unterschiedliche Positionen getestet. In Abbildung 10.1 ist der Versuchsaufbau zu sehen. Für jede der fünf Entfernungen, die dort zu sehen sind, werden neun verschiedene Posen getestet:

1. Der Roboter steht in der gleichen Pose wie in Abbildung 2.6 in der mittleren Position frontal ausgerichtet.

2. Der Roboter steht in der gleichen Pose wie in Abbildung 2.6 in der mittleren Position seitlich ausgerichtet.

3. Der Roboter steht in der gleichen Pose wie in Abbildung 2.6 in der linken Position frontal ausgerichtet. Dies soll prüfen, ob die Erkennung und insbesondere die Distanzschätzung auch am linken Bildrand funktioniert.

4. Der Roboter steht in der gleichen Pose wie in Abbildung 2.6 in der rechten Position frontal ausgerichtet. Dies soll prüfen, ob die Erkennung und insbesondere die Distanzschätzung auch am rechten Bildrand funktioniert.

5. Der Roboter steht in der Gorilla-Pose (Abbildung 10.2 a)) in der mittleren Position in einem 45° Winkel ausgerichtet. Diese Pose soll prüfen, ob auch Roboter erkannt werden, die gerade dabei sind aufzustehen.

6. Der Roboter liegt wie in Abbildung 10.2 c) auf dem Boden in der mittleren Position in einem 45° Winkel ausgerichtet. Diese Pose soll prüfen, ob auch Roboter erkannt werden, die am Boden liegen.

7. Der Roboter steht in der gleichen Pose wie in Abbildung 2.6 mit einem Ball vor seinen Füßen liegend in der mittleren Position frontal ausgerichtet. Diese Pose soll prüfen, ob sich der Detektor durch den Ball ablenken lässt.

8. Zwei Roboter stehen, wie in Abbildung 10.2 d) gezeigt wird, in der mittleren Position frontal ausgerichtet. Es soll geprüft werden, ob die beiden Roboter einzeln erkannt werden und inwiefern sich dies auf die Entfernungsschätzung auswirkt.

9. Der Roboter liegt wie in Abbildung 10.2 b) auf dem Boden in der mittleren Position frontal ausgerichtet. Diese Pose soll prüfen, ob auch Roboter erkannt werden, die gerade eine Parade gemacht haben.

Des Weiteren wurden die Posen 1 und 2 auch in einem Abstand von 30 cm evaluiert, um zu prüfen, ob ein Roboter erkannt wird, wenn er direkt vor dem erkennenden Roboter steht. Eine Messung läuft dabei so ab, dass ein Roboter auf der jeweiligen Position in der jeweiligen Pose platziert wird, während der Rest des Feldes leer ist (Abbildung 10.3). Um einen Vergleichswert zu haben, wird für alle Posen zusätzlich geprüft, ob der aktuelle Robotererkenner von B-Human [Röfer, Laue, Hasselbring et al., 2018] die Roboter erkennt. Die detaillierten Ergebnisse dieses Experimentes sind in Anhang A zu finden.

Die Ergebnisse zeigen, dass der aktuelle Robotererkenner von B-Human die Roboter zwar nicht in der Nahdistanz erkennt, aber dafür zuverlässig im Bereich von 1,5 m bis 3 m arbeitet. Aufgrund des Verfahrens, welches für die Erkennung genutzt wird, ist die Pose des Roboters zweitrangig, sodass alle Posen gleich gut erkannt werden. Allerdings werden am Boden liegende Roboter oft als 2 Roboter erkannt.

Der Objektdetektor, der in dieser Arbeit entwickelt worden ist, erkennt Roboter bis einschließlich 4,5 m, solange diese nicht flach auf dem Boden liegen, wie es bei Pose 6 der Fall ist. In dem Fall ist eine Erkennung nur bis 1,5 m möglich. Ab 6 m werden nur noch vereinzelt Roboter erkannt. Bei einer Entfernung von 1,5 m ist es sogar möglich, die beiden Roboter aus Pose 8 zu unterscheiden.

Wie zu erwarten war, nimmt die Genauigkeit der Distanzbestimmung mit zunehmender Entfernung ab. Dies wird vor allem an der schlechten Auflösung liegen. Bis 3m ist die Entfernungsschätzung sehr genau, solange die Roboter nicht verdeckt sind oder vor anderen Robotern stehen. Die gemessene Distanz weicht oftmals stärker von der tatsächlichen Distanz ab, hat dafür allerdings nicht so große Ausreißer bei den angesprochenen Fällen, was zu einer geringeren Standardabweichung führt. Es zeigt sich, dass die Nutzung des Mittelwertes beider Werte die Standardabweichung senken kann, auch wenn im Bereich bis 3m der Mittelwert etwas schlechter wird. Im Nahbereich gibt es einen sehr großen Fehler für den seitlich stehenden Roboter, während der frontal stehende Roboter sehr gut geschätzt wird. Es kann sein, dass hier zu wenig Trainingsdaten vorliegen. Es fällt zudem auf, dass die Entfernungsschätzung anfällig ist, für Veränderungen der Kopfneigung. Das bedeutet, dass sich die geschätzte Entfernung sich verändert, wenn der Kopf gesenkt oder angehoben

wird. Auch dies ist vermutlich auf das Fehlen von Trainingsdaten zurückzuführen, da der Roboter diese Winkel nur sehr selten im Spiel einnimmt.

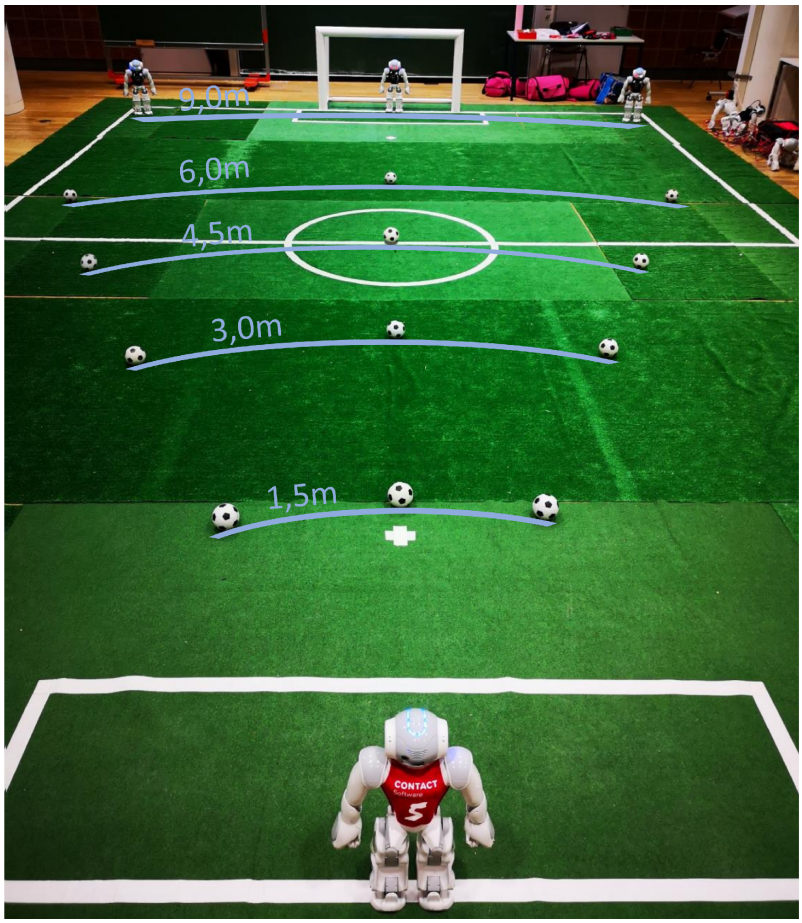


Abbildung 10.1: **Versuchsaufbau: verschiedene Distanzen.** Die zwölf Bälle und die drei schwarzen Roboter auf dem Spielfeld markieren die verschiedenen Positionen, auf denen die Erkennung evaluiert wird. Dabei werden die linken und rechten Position lediglich für die Posen 3 und 4 benötigt. Der rote Roboter stellt den Roboter dar, auf dem der Objektdetektor läuft.

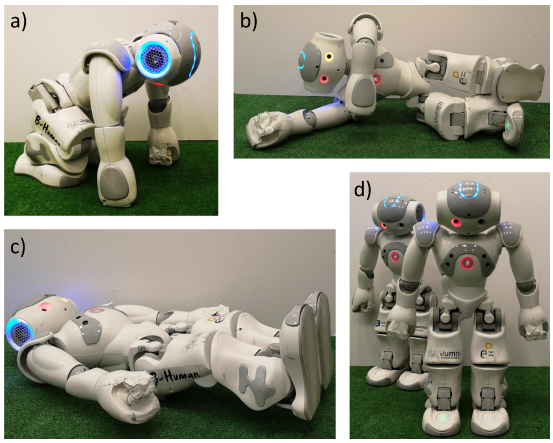


Abbildung 10.2: **Evaluationsposen.** Die vier Bilder zeigen verschiedene Posen, wie sie bei der Evaluation verwendet werden. a) Gorilla-Pose b) Torwart-Parade c) gefallener Roboter d) zwei Roboter (einer verdeckt)

Abbildung 10.3: **Evaluation einer Distanz und Pose.** Für die Auswertung einer bestimmten Position und Pose befinden sich lediglich zwei Roboter auf dem Feld: der erkennende Roboter und der zu erkennende Roboter.

10.3 Performanz im Spiel

Als nächstes soll die Performance im Spiel gemessen werden. Um dies zu tun, wird ein gelabelter Datensatz von einem Testspiel von B-Human verwendet. Dieser enthält sowohl Roboter, die nah dran sind, als auch solche, die weit entfernt sind. Das Testspiel fand in dem gleichen Raum statt, der in Abbildung 10.1 und 10.3 zu sehen ist.

Um die Auswertung direkt im B-Human-Framework laufen zu lassen, wurde das Modul, dessen bisherige Aufgabe lediglich darin bestand Label zu verwalten und anzuzeigen, erweitert. Dazu wurden sämtliche Metriken, welche bisher in Python genutzt worden sind, noch einmal in C++ implementiert, sodass sich zwei Mengen von Labeln, wobei eine als Ground Truth und eine als Vorhersage interpretiert wird, automatisch auswerten lassen. Diese Ergebnisse können über einen Zeitraum gesammelt und der Mittelwert im Simulator angezeigt werden.

Für die Evaluation werden der Recall, die Precision, die durchschnittliche IoU aller erkannten Boxen sowie der MAP-Wert für das gesamte Spiel berechnet. Es werden mehrere Durchgänge gemacht, bei denen die folgenden Varianten von Erkennern getestet werden sollen.

1. Der aktuelle Robotererkenner von B-Human wird getestet, um einen Vergleichswert zu bekommen. Allerdings erkennt dieser neben Robotern auch Pfosten, weshalb diese als neutrale Element gezählt werden. Des Weiteren bestimmt er nicht die exakte Bounding Box, sondern nur den Bereich, in dem die Füße sind (siehe Abbildung 10.4). Dadurch wird der IoU-Wert oftmals sehr gering sein. Um dieses Problem zu lösen, wird die Erkennung künstlich auf die dreifache Breite erweitert und der IoU, welcher erreicht werden muss, damit ein Objekt als erkannt gezählt wird, wird auf 10% gesenkt.

2. Der hier entwickelte Roboterdetektor wird zunächst nur mit NMS als Nachverarbeitung verwendet.

3. Der hier entwickelte Roboterdetektor wird mit NMS und der Begrenzung durch den Feldrand als Nachverarbeitung verwendet.

4. Der hier entwickelte Roboterdetektor wird mit NMS und dem Entfernungsabgleich als Nachverarbeitung verwendet.

5. Der hier entwickelte Roboterdetektor wird mit NMS als Nachverarbeitung verwendet. Es wird allerdings ein IoU-Wert von 0,3 als Schwellwert genutzt.

Abbildung 10.4: **Vergleich der Bounding Boxes.** Im linken Bild ist eine Bounding Box des Roboterdetektors dieser Arbeit zu sehen. Im rechten Bild ist die erkannte Box des aktuellen Robotererkenners von B-Human zu sehen. Die Zahl über der Box zeigt die geschätzte Distanz an, welche nur für den hier entwickelten Roboterdetektor existiert.

Alle fünf Versuche werden zunächst für das gesamte Feld ausgeführt. Anschließend werden diese noch einmal für die Bereiche bis 6 m und bis 3 m wiederholt. Die Precision und MAP sind jeweils nur für die Evaluation über das gesamte Feld angegeben. Das liegt daran, dass für Tests in den Bereichen bis 6 m beziehungsweise bis 3 m zwar die Ground-Truth-Boxen aussortiert werden, die außerhalb dieser Bereiche liegen, da deren Entfernung relativ genau bestimmt werden kann. Für die vorhergesagten Boxen ist dies jedoch nicht möglich, da diese falsch oder ungenau sein können. Dadurch wird das Ergebnis der Precision schlechter und verliert seine Aussagekraft. Da die Precision für die MAP genutzt wird, gilt dies auch für diese. Zudem wird für Versuch 1 keine MAP berechnet, da dort keine Gewichtung der Erkennungen ausgeben wird.

Die Ergebnisse in Tabelle 10.2 zeigen, dass der Ansatz dieser Arbeit im Vergleich mit dem aktuellen Verfahren von B-Human eine rund 20% geringe Precision erreicht. Allerdings liegt dies nicht daran, dass es so viele Fehlerkennung gibt, sondern daran, dass der Schwellwert mit einem IoU von 0,5 wesentlich strenger gewählt worden ist. Wie Versuch 5 zeigt, steigt die Precision um 10%, wenn ein Schwellwert von 0,3 verwendet wird. Allerdings ist die Precision beim B-Human-Verfahren dennoch rund 12% besser. Dafür erkennt dieses Verfahren auch nur 20% der Roboter, während der Ansatz dieser Arbeit fast 60% erreicht. Auch in den näheren Distanzen ist dieser Ansatz hier überlegen. Beide Ansätze können ihre Performance verbessern, wenn sie auf einen kleinen Bereich angewendet werden. Auch der IoU-Wert der erkannten Boxen ist beim Ansatz dieser Arbeit besser und wird zudem in den näheren Distanzen besser. Allerdings ist der Erkenner von B-Human auch nicht darauf ausgelegt, die Bounding Box zu erkennen.

Die beiden Nachverarbeitungen, die in Versuch 3 und 4 getestet worden sind, können beide keinen wirklichen Mehrwert erzielen, da sie zwar die Precision steigern, können aber im Gegenzug auch der Recall sinkt. Allerdings gilt dies vor allem auf weiten Distanzen, sodass es eine Überlegung wert ist, diese Verfahren lediglich im Nahbereich zu nutzen.

	Recall			Precision	IoU			MAP
Versuch	gesamt	bis 6 m	bis 3 m	gesamt	gesamt	bis 6 m	bis 3 m	gesamt
1	0,204	0,285	0,583	0,972	0,657	0,657	0,662	-
2	0,583	0,716	0,904	0,754	0,717	0,730	0,806	0,506
3	0,574	0,705	0,889	0,762	0,718	0,731	0,806	0,497
4	0,558	0,700	0,904	0,778	0,725	0,734	0,806	0,488
5	0,662	0,781	0,929	0,856	0,683	0,705	0,793	0,591

Tabelle 10.2: Vergleich mit B-Human im Spiel

10.4 Pfostenerkenner

Um zu zeigen, dass der gelernte Objektdetektor nicht nur für die Detektion von Robotern verwendet werden kann, soll mit dem gleichen neuronalen Netz ein Pfostenerkenner gelernt werden. Um dies zu realisieren, werden allerdings sehr viele gelabelte Bilder von Pfosten benötigt, welche es so nicht gibt. Aus diesem Grund wird der Pfostenerkenner nur auf Daten aus der Simulation trainiert. Dazu wurden 40.000 Bilder in der Simulation erstellt, wovon ungefähr ein Zehntel als Testdaten fungieren soll. Für den Roboterdetektor wurden Label, die auf der finalen Auflösung kleiner als einen Pixel sind, ignoriert. Dies ist für den Pfostenerkenner nicht möglich, da die Pfosten sehr dünn sind und somit ein Großteil der Daten aussortiert werden müsste. Des Weiteren müssen noch die Anchor Boxes angepasst werden, wozu die Clustering-Funktion von keras-yolo2 verwendet wird. Als letztes wird noch das Verhältnis von positiven zu negativen Daten für die Alpha-Gewichtung des Konfidenz-Losses von 0,1 auf 0,3 angehoben, um wieder ein Ausgleich zwischen Recall und Precision herzustellen. Ansonsten müssen keine weiteren Änderungen vorgenommen werden.

Die erzielten MAP-Werte bei verschiedenen Schwellwerten sind in Tabelle 10.4 zu sehen. Es zeigt sich, dass ohne viel Anpassungen gute Ergebnisse erzielt werden können. Der Grund dafür, dass diese besser sind als beim Robotererkenner, ist vermutlich, dass die Bilder für das Training und die Auswertung lediglich aus der Simulation stammen und somit etwas leichter zu erkennen sind.

	∅MAP								
Objekttyp	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
Robotererkenner	0,773	0,776	0,761	0,725	0,652	0,527	0,347	0,156	0,026
Pfostenrerkenner	0,891	0,879	0,870	0,853	0,741	0,659	0,509	0,264	0,031

Tabelle 10.3: Mean Average Precision des Pfostenerkenners

Außerdem wurden drei Versuche gemacht, um die Performance des Pfostenerkenners im Spiel zu testen:

1. Als Vergleichswert wird der Pfostenerkenner von B-Human [Röfer, Laue, Hasselbring et al., 2018] auf dem in Kapitel 10.3 verwendeten Datensatz evaluiert. Analog zum Robotererkenner werden hier Roboter als neutrale Elemente betrachtet.

2. Auch wenn der Pfostenerkenner dieser Arbeit lediglich in der Simulation trainiert worden ist, soll auch dieser auf dem gleichen Datensatz evaluiert werden.

3. Zusätzlich soll der hier entwickelte Pfostenerkenner noch in der Simulation getestet werden. Dazu wird die Standard-Testszene von B-Human verwendet, welche um die Anpassungen, die im Rahmen dieser Arbeit entstanden sind, erweitert. Ein Aufbau dieser Szene ist in Abbildung 5.3 (rechts) zu finden.

Für Versuch 1 wird wieder ein IoU-Schwellwert von 0,1 verwendet, während für die anderen beiden Versuche ein Wert von 0,3 verwendet wird, da Pfosten sehr oft verdeckt und weit entfernt sind. Die Ergebnisse in Tabelle 10.4 zeigen, dass in der Simulation, wie zu erwarten war, sehr gute Ergebnisse erzielt werden. Erstaunlich ist jedoch, dass selbst auf dem Datensatz aus der Realität fast 45% der Pfosten erkannt werden und zudem nur 30% der erkannten Pfosten False Positives sind. Damit ist der Pfostenerkenner aus dieser Arbeit in beiden Kategorien besser als der aktuelle Pfostenerkenner von B-Human.

Versuch	Recall	Precision	IoU	MAP
1	0,028	0,462	0,397	
2	0,447	0,706	0,457	0,339
3	0,754	0,782	0,552	0,685

Tabelle 10.4: Mean Average Precision des Pfostenerkenners

10.5 Diskussion

Die Ergebnisse aus Kapitel 10.1 haben gezeigt, dass selbst ein wesentlich größeres Netz mit einer größeren Eingabe von 224x224 kaum bessere Ergebnisse erzielen kann als das Netz, das in dieser Arbeit entwickelt worden ist. Allerdings sollte hier beachtet werden, dass das Training für beide Netze abgebrochen worden ist, nachdem sich zehn Episoden lang keine Verbesserung des F_1-Scores eingestellt hat. Es ist somit möglich, dass gerade die YOLO v2 Architektur noch etwas Luft nach oben hat. Ein großer Sprung dürfte hier allerdings ausbleiben. Von daher sind die Ergebnisse des Netzes aus dieser Arbeit schon bemerkenswert und zeigen, dass es sich lohnt Aufwand in das Design der Architektur zu investieren, da dadurch viele Ressourcen bei der Inferenz gespart werden können.

Die Evaluation der verschiedenen Positionen und Posen offenbarte, dass die maximale Entfernung, auf der Roboter robust erkannt werden können, zwischen bei 4,5 m

liegt. Damit wird die Reichweite des bisherigen Robotererkenners von B-Human um 1,5 m übertroffen. Allerdings gilt dies nicht für alle Posen. Besonders große Probleme haben hier die Roboter gemacht, die auf dem Boden liegen. Das könnte daran liegen, dass sie aufgrund ihrer geringen Höhe im Kamerabild mit Linien verwechselt werden oder ihre Darstellung im finalen Layer des Detektors einfach zu klein ist. Allgemein lässt sich allerdings sagen, dass der Ansatz dieser Arbeit alle möglichen Posen erkannt hat und dies für die meisten Posen sogar bis zu einer Distanz von 4,5 m. Dieses Ergebnis wird auch noch einmal durch die Evaluation auf dem Testspieldatensatz bestätigt, da dort über 90% der Roboter in einem Bereich von unter 3 m erkannt werden. Hierbei erzielt der Objekterkenner dieser Arbeit eine wesentlich höhere Erkennungsrate als der aktuelle Ansatz von B-Human, während es nur eine geringfügig schlechtere Precision gibt, welche allerdings vor allem daran liegt, dass die Bounding Boxes zu ungenau sind, und deswegen als False Positive gezählt werden. Jedoch gibt es auch tatsächliche False Positives, welche zum einen durch Menschen zustande kommen, die sich entweder am Feldrand oder auf dem Feld befinden. Zum anderen entstehen False Positves durch Artefakte, welche sich aus ungünstigen Zusammensetzungen des Bildes ergeben. Hierbei kann der aktuelle Robotererkenner von B-Human punkten, da dieser nur dann False Positves erkennt, wenn Menschen sich tatsächlich auf dem Spielfeld befinden, was normalerweise nicht der Fall ist. Ein weiterer Schwachpunkt des entwickelten Objektdetektors sind Roboter, die hintereinander stehen. Bis zu einer Distanz von 1,5 m können diese zwar noch auseinandergehalten werden, darüber hinaus verschmelzen diese aber häufig und sorgen für eine ungenaue Bounding Box. Hier kann in Zukunft noch für bessere Ergebnisse gesorgt werden. Ein neues Feature, welches der aktuelle Robotererkenner von B-Human nicht bietet, ist die Erkennung von Robotern auf sehr nahe Distanzen bei denen der Roboter fast das ganze Bild verdeckt. Insgesamt betrachtet ist das Ergebnis der Erkennung allerdings sehr zufriedenstellend, da der aktuelle Robotererkenner von B-Human in fast allen Kategorien geschlagen wird, sodass dieser lediglich eine bessere Precision aufweisen kann. Allerdings könnte eine intensivere Beschäftigung mit der Nachverarbeitung hier für Abhilfe sorgen, da die hier getesteten Verfahren zwar einige False Positves aussortieren konnten, aber auch ein Teil der True Positives aussortiert worden ist, weshalb dieses Ergebnis nicht überzeugen konnte.

Die Schätzung der Distanz der erkannten Roboter wurde in Kapitel 10.2 evaluiert. Zudem wurde sie mit einer Messung der Entfernung aufgrund der unteren Kante der erkannten Bounding Box verglichen. Obwohl die Trainingsdaten für die Vorhersage der Distanz lediglich aus der Simulation stammen, konnten gerade im Bereich bis 3 m gute Ergebnisse erzielt werden. Der Mittelwert über alle Posen weicht hierbei nur leicht von der tatsächlichen Distanz ab und ist vor allem genauer als die Messung der Distanz. Allerdings führen manche Posen dazu, dass es stärkere Abweichungen bei der Vorhersage gibt, sodass in diesen Fällen die Messung die besseren Ergebnisse liefert. Dies führt zu einer geringen Standardabweichung bei der gemessenen Distanz. Es hat sich jedoch gezeigt, dass der Mittelwert aus beiden Methoden in den meisten Fällen die beste Schätzung darstellt. Des Weiteren ist aufgefallen, dass die Vorhersage der Distanz von der Neigung des Kopfes abhängt, was hoffentlich durch weitere Trainingsdaten mit anderen Kopfhaltungen verbessert werden kann. Gleiches gilt auch für seitlich stehende Roboter im Nahbereich. Zusammenfassend scheint die Schätzung der Distanz allerdings ein brauchbares neues Feature zu sein, welches zwar noch etwas Verbesserungspotential in manchen Situationen hat, aber in den meisten Fällen hilfreiche Informationen liefert.

Abschließend wurde der entwickelte Roboterdetektor in einen Pfostenerkenner umgewandelt, wobei neben den offensichtlichen Anpassungen der Trainingsdaten und Anchor Boxes lediglich die Gewichtung angepasst werden musste. Zusätzlich mussten aufgrund der geringen Breite der Pfosten auch solche Label für das Training zugelassen werden, die im Vorhersagelayer weniger als einen Pixel groß sind. Obwohl der Pfostenerkenner nur auf Simulationsdaten trainiert worden ist, da keine gelabelten Daten aus der Realität vorliegen, können dennoch überraschend gute Ergebnisse auf den Testdaten aus der Realität erzielt werden. Dabei wird sogar die Erkennungsrate des aktuellen Erkenners von B-Human übertroffen. Zwar war dieses Experiment lediglich dafür gedacht, um zu zeigen,

dass der Objektdetektor nicht nur Roboter erkennen kann, aber eventuell kann durch die Nutzung von ein paar Trainingsdaten aus der Realität ein ordentlicher Pfostenerkenner entstehen.

11 Ergebnisse

11.1 Fazit

Zu Beginn dieser Arbeit wurde die Frage gestellt, ob es möglich sei, aktuelle Objektdetektoren auf mobile Roboter wie den Nao zu übertragen und dabei echtzeitfähig zu bleiben. Um dies zu überprüfen, wurde in dieser Arbeit ein Objektdetektor entwickelt, dessen Ziel es ist, durch die Nutzung von Deep Learning selbst auf mobilen Robotern in Echtzeit Objekte zu erkennen.

Dazu wurden zunächst aktuelle Objektdetektoren analysiert, um sich einen Überblick über die verschiedenen Methoden der Objektdetektion mittels Deep Learning zu verschaffen. Des Weiteren wurden aktuelle Entwicklungen aus der SPL betrachtet, um das Wissen über Objektdetektoren auf mobile Roboter übertragen zu können. Aus diesem Pool an Methoden und Ideen wurden die vielversprechendsten ausgewählt, um ein neuronales Netz und dessen Trainingsprozess zu designen. Durch eine erste Evaluation des neuronalen Netzes konnten dann weitere Annahmen gemacht werden, sodass eine grobe Architektur entstanden ist, die allerdings noch nicht echtzeitfähig war. Um dieses neuronale Netz möglichst effizient zu gestalten, wurden verschiedene Methoden evaluiert, wobei vor allem die separierbare Convolution und das Pruning einen Zuwachs bei der Geschwindigkeit gebracht haben. Um die unter Kamera des Nao nicht verwenden zu müssen, wurde anschließend eine Distanzschätzung entwickelt, die es schafft, mit einem Verfahren ähnlich zu den bekannten GAN-Architekturen allein auf Traningsdaten aus der Simulation Distanzwerte für die Realität vorherzusagen.

Dabei sind neben vielen Hilfmitteln wie dem Autolabeler, dem Evaluationsmodul für das B-Human-Framework, weiteren Layern für die B-Human-Toolchain für neuronale Netze, und der Implementierung von SConv-Layern für keras-surgeon auch wissenschaftlich relevante Neuentwicklungen gemacht worden. Dazu zählen zum einen die Nutzung von der IoU und des MACE als Lossfunktionen. Zum anderen zählt dazu aber auch die effiziente Netzarchitektur und die GAN-Variante, um allein aufgrund von Traingsdaten aus der Simulation sinnvolle Aussagen über die Realität treffen zu können. Diese Entwicklungen sind keinesfalls auf die Verwendung in einem Roboterdetektor beschränkt, sondern haben das Potential auch bei anderen Detektionsproblemen das Mittel der Wahl zu sein.

Zum Schluss soll nun anhand den fünf Kriterien, die am Anfang der Arbeit festgehalten worden sind, überprüft werden, ob dieser Objektdetektor den Anforderungen dieser Arbeit gerecht geworden ist.

Allgemeinheit Während der gesamten Entwicklungen wurde es vermieden, auf Naospezifische Eigenschaften einzugehen. Einzig die Größe und das Format des Naos findet sich in den Anchor Boxes wieder, welche für jedes andere Objekt leicht erstellt werden können. Insbesondere hat das Experiment aus Kapitel 10.4 gezeigt, dass der Roboterdetektor sich leicht zu einem Pfostenerkenner umbauen lässt.

Echtzeitfähigkeit Der finale Objektdetektor verbraucht auf dem Nao durchschnittlich 4,5 ms. Dabei ist eingerechnet, dass die Berechnung für die untere Kamera übersprungen wird und deswegen zu vernachlässigen ist. Die Inferenzzeit für das obere Bild beträgt somit 9 ms. Zwar liegt dieser Wert über den geforderten 7 ms, allerdings erfüllt der Objektdetektor die Vorgabe für Echtzeitfähigkeit vom Anfang. Da jede Kamera des Naos für sich mit 30 Hz betrieben wird und sich nur im Verbund 60fps ergeben, erfüllen die durchschnittlichen 4,5 ms dieses Kriterium vollkommen.

Robustheit Die Fähigkeit mit schwierigen Bildern umgehen zu können, hat der Roboterdetektor in Kapitel 10.3 beweisen. Zusätzlich wurde dies mit den Testdaten in Keras Kapitel 10.1 bestätigt, da auch hier gute Ergebnisse erzielt werden konnten.

Distanz Wie die Evaluation aus Kapitel 10.3 ergeben hat, werden im Bereich bis 6 m 72% der Roboter und im Bereich bis 3 m 90% der Roboter erkannt. Somit sind die Anforderungen an die Distanz erfüllt.

Präzision Die Evaluation aus Kapitel 10.3 hat gezeigt, dass die Precision im Gegensatz zum Recall mit 75% knapp an den Vorgaben von 80% scheitert. Allerdings hat die Reduzierung des Schwellwertes auch gezeigt, dass die Precision nicht daran scheitert, dass falsche Roboter erkannt werden, sondern, dass die Roboter nicht genau genug erkannt werden.

Unterm Strich wurden bis auf die Präzision alle Vorgaben erreicht und die Präzision ist nur knapp gescheitert. Somit wurde das Ziel dieser Arbeit erreicht und es kann abschließend behauptet werden, dass es zwar möglich ist, einen Deep Learning basierten, echtzeitfähigen Objektdetektor für mobile Roboter zu entwickeln, dies jedoch viel Zeit und Hingabe in Anspruch nimmt, da jedes Detail des Detektors gut durchdacht sein muss, um eine überzeugende Erkennungsrate bei einem so kleinen neuronalen Netz erreichen zu können.

11.2 Ausblick

Da sich der neu entwickelte Roboterdetektor sich anders verhält als der bisherige Robotererkenner von B-Human, muss das Modell der erkannten Roboter angepasst werden, bevor dieser ordentlich verwendet werden kann. Dazu zählt zum einen die Tatsache, dass nun wesentlich mehr Roboter erkannt werden, aber zum anderen auch, dass etwas mehr False Positives erkannt werden. Zusätzlich müssen neue Features wie die Erkennung im Nahbereich und die Schätzung der Distanz so integriert werden, dass auch andere Module davon profitieren können.

Allerdings weist auch der Roboterdetektor an sich noch Schwächen auf: Zum Beispiel können bessere Ansätze für die Aussortierung von False Positives gefunden werden sowie eine bessere Erkennung von Robotern, die auf dem Boden liegen. Zudem kann die Auseinanderhaltung von Robotern, die sich teilweise verdecken, genauer untersucht werden. Da in dieser Arbeit bereits anfangen worden ist, einen Pfostenerkenner zu entwickeln, kann dieses Bestreben fortgesetzt werden. Vor allem wäre es in dieser Hinsicht interessant zu prüfen, ob es möglich ist, den Roboterdetektor dahingehend zu erweitern, dass er sowohl Roboter, als auch Pfosten detektieren kann. Theoretisch wäre auch die Detektion von Bällen möglich, wobei es nicht unwahrscheinlich ist, dass diese zu klein sind, um auf dem 80x60 Pixel umfassenden Bild auf größere Entfernungen erkannt zu werden.

Für die Schätzung der Distanz können weitere Trainingsdaten gesammelt werden, um Sonderfälle zu vermeiden und die Schätzung robuster gegen Neigungen des Kopfes zu machen. Des Weiteren kann evaluiert werden, ob sich neben der Distanz weitere Eigenschaften der Roboter vorhersagen lassen. Dazu zählt insbesondere die Rotation der Roboter, da es einen großen Vorteil im Spiel bringen würde, wenn man wüsste, wie der Roboter ausgerichtet ist, um besser abschätzen zu können, welche Aktion der Roboter als nächstes ausführen wird.

Des Weiteren kann untersucht werden, ob sich der entwickelte Objektdetektor auch für die Nutzung in anderen Anwendungsgebieten eignet. Es wäre zum Beispiel denkbar, diesen in Staubsaugerrobotern zu nutzen, um Haustiere von Möbeln zu unterscheiden. Zudem könnte eine etwas stärke CPU, die Nutzung in Drohnen möglich machen, sodass Menschen oder Gebäude bereits von weitem erkannt werden können. Unabhängig von dem spezifischen Anwendungsfeld sollten die Lossfunktionen, welche im Rahmen dieser Arbeit entstanden sind, daraufhin überprüft werden, ob diese auch in Objektdetektoren aus dem Stand der Technik gute oder gar bessere Ergebnisse erzielen können, als die Lossfunktionen, die bisher genutzt werden.

Literatur

Albani, Dario et al. (2017). „A Deep Learning Approach for Object Recognition with NAO Soccer Robots“. In: S. 392–403. ISBN: 978-3-319-68792-6. DOI: 10.1007/978-3-319-68792-6_33. URL: http://link.springer.com/10.1007/978-3-319-68792-6_33.

Badrinarayanan, Vijay, Alex Kendall und Roberto Cipolla (2015). „SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation“. In: *CoRR* abs/1511.00561. arXiv: 1511.00561. URL: http://arxiv.org/abs/1511.00561.

Bloisi, Domenico et al. (2017). „Machine Learning for RealisticBall Detection in RoboCup SPL“. In: July. arXiv: 1707.03628. URL: http://arxiv.org/abs/1707.03628.

Chollet, François et al. (2015). *Keras*. https://keras.io.

Cruz, Nicolás, Kenzo Lobos-tsunekawa und Javier Ruiz-del-solar (2017). „Using Convolutional Neural Networks in Robots with Limited Computational Resources : Detecting NAO Robots while Playing Soccer“. In: *RoboCup 2017 Symposium*. arXiv: 1706.06702.

Dai, Jifeng et al. (2016). „R-FCN: Object Detection via Region-based Fully Convolutional Networks“. In: ISSN: 15206149. DOI: 10.1109/ICASSP.2017.7952132. arXiv: 1605.06409. URL: http://arxiv.org/abs/1605.06409.

„Echtzeit, Echtzeitsysteme, Echtzeitbetriebssysteme“ (2005). In: *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 39–73. ISBN: 978-3-540-27522-0. DOI: 10.1007/3-540-27522-3_3. URL: https://doi.org/10.1007/3-540-27522-3_3.

Everingham, Mark et al. (2010). „The pascal visual object classes (VOC) challenge“. In: *International Journal of Computer Vision* 88.2, S. 303–338. ISSN: 09205691. DOI: 10.1007/s11263-009-0275-4.

Fiedler, Niklas, Marc Bestmann und Norman Hendrich (2018). „ImageTagger: An Open Source Online Platform for Collaborative Image Labeling“. In: *RoboCup 2018: Robot World Cup XXII*. Springer.

Fu, Cheng-Yang et al. (2017). „DSSD : Deconvolutional Single Shot Detector“. In: arXiv: 1701.06659. URL: http://arxiv.org/abs/1701.06659.

Golik, Pavel, Patrick Doetsch und Hermann Ney (2013). „Cross-entropy vs. Squared error training: A theoretical and experimental comparison“. In: *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH* 2.2, S. 1756–1760. ISSN: 19909772. DOI: 10.1145/1102351.1102422. arXiv: arXiv:1503.01842v1.

Goodfellow, Ian J et al. (Juni 2014). „Generative Adversarial Networks“. In: S. 1–9. arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661.

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

He, Kaiming et al. (März 2017). „Mask R-CNN“. In: arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870.

Howard, Andrew G und Weijun Wang (2017). „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications“. In: *arXiv preprint arXiv:1704.04861*. arXiv: 1704.04861v1.

Htwk, Nao-team (2018). „Team Research Report“. In: URL: http://robocup.imn.htwk-leipzig.de/documents/TRR_2017.pdf?lang=de.

Iandola, Forrest N. et al. (2016). „SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size“. In: S. 1–5. ISSN: 0302-9743. DOI: 10.1007/978-3-319-24553-9. arXiv: 1602.07360. URL: http://arxiv.org/abs/1602.07360.

IGN (2016). *Person of Interest: Season 5 - Extended Trailer*. URL: https://www.youtube.com/watch?v=7I8OeQs7cQA.

Javadi, Mohammad et al. (2017). „Humanoid Robot Detection using Deep Learning : A Speed-Accuracy Tradeoff“. In:

Krizhevsky, Alex, Ilya Sutskever und Geoffrey E Hinton (Mai 2017). „ImageNet classification with deep convolutional neural networks“. In: *Communications of the ACM*

60.6, S. 84–90. ISSN: 00010782. DOI: 10.1145/3065386. URL: http://dl.acm.org/citation.cfm?doid=3098997.3065386.

Lin, Tsung Yi, Piotr Dollár et al. (2017). „Feature pyramid networks for object detection“. In: *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-Janua, S. 936–944. ISSN: 0006-291X. DOI: 10.1109/CVPR.2017.106. arXiv: 1612.03144.

Lin, Tsung Yi, Michael Maire et al. (2014). „Microsoft COCO: Common objects in context“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8693 LNCS.PART 5, S. 740–755. ISSN: 16113349. DOI: 10.1007/978-3-319-10602-1_48. arXiv: 1405.0312.

Lin, Tsung-Yi et al. (Aug. 2017). „Focal Loss for Dense Object Detection“. In: arXiv: 1708.02002. URL: http://arxiv.org/abs/1708.02002.

Liu, Wei et al. (2016). „SSD: Single shot multibox detector“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9905 LNCS, S. 21–37. ISSN: 16113349. DOI: 10.1007/978-3-319-46448-0_2. arXiv: 1512.02325.

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Menashe, Jacob et al. (2017). „Fast and Precise Black and White Ball Detection for RoboCup Soccer“. In: July.

Molchanov, Pavlo et al. (2016). „Pruning Convolutional Neural Networks for Resource Efficient Inference“. In: 2015, S. 1–17. ISSN: 0004-6361. DOI: 10.1051/0004-6361/201527329. arXiv: 1611.06440. URL: http://arxiv.org/abs/1611.06440.

Mühlenbrock, Andre und Tim Laue (2018). „Vision-Based Orientation Detection of Humanoid Soccer Robots“. In: *RoboCup 2017: Robot World Cup XXI*. Hrsg. von Hidehisa Akiyama et al. Cham: Springer International Publishing, S. 204–215. ISBN: 978-3-030-00308-1.

Rastegari, Mohammad et al. (März 2016). „XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks“. In: S. 1–17. arXiv: 1603.05279. URL: http://arxiv.org/abs/1603.05279.

Redmon, Joseph, Santosh Divvala et al. (2015). „You Only Look Once: Unified, Real-Time Object Detection“. In: ISSN: 01689002. DOI: 10.1109/CVPR.2016.91. arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

Redmon, Joseph und Ali Farhadi (2017). „YOLO9000: Better, faster, stronger“. In: *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-Janua, S. 6517–6525. ISSN: 0146-4833. DOI: 10.1109/CVPR.2017.690. arXiv: 1612.08242.

— (Apr. 2018). „YOLOv3: An Incremental Improvement“. In: ISSN: 0146-4833. DOI: 10.1109/CVPR.2017.690. arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767.

Ren, Shaoqing et al. (Juni 2015). „Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6, S. 1137–1149. ISSN: 01628828. DOI: 10.1109/TPAMI.2016.2577031. arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

RoboCup Technical Committee (Mai 2018). *RoboCup Standard Platform League (NAO) Rule Book*. URL: http://spl.robocup.org/wp-content/uploads/2018/04/SPL-Rules_small.pdf.

Röfer, Thomas, Tim Laue, Arne Hasselbring et al. (2018). *B-Human Team Report and Code Release 2018*. Only available online: http://www.b-human.de/downloads/publications/2018/coderelease2018.pdf.

Röfer, Thomas, Tim Laue, Judith Müller et al. (2010). *B-Human Team Report and Code Release 2010*. Only available online: http://www.b-human.de/downloads/bhuman10_coderelease.pdf.

Russakovsky, Olga et al. (Dez. 2015). „ImageNet Large Scale Visual Recognition Challenge“. In: *International Journal of Computer Vision* 115.3, S. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. arXiv: 1409.0575. URL: http://link.springer.com/10.1007/s11263-015-0816-y.

Speck, Daniel et al. (2017). „Ball Localization for Robocup Soccer using Convolutional Neural Networks“. In:

Springenberg, Jost Tobias et al. (2014). „Striving for Simplicity: The All Convolutional Net“. In: *CoRR* abs/1412.6806. arXiv: 1412.6806. URL: http://arxiv.org/abs/1412.6806.

Taigman, Yaniv et al. (Juni 2014). „DeepFace: Closing the Gap to Human-Level Performance in Face Verification“. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, S. 1701–1708. ISBN: 978-1-4799-5118-5. DOI: 10.1109/CVPR.2014.220. arXiv: 1501.05703. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6909616.

Zhu, Jun-yan et al. (März 2017). „Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks“. In: arXiv: 1703.10593. URL: http://arxiv.org/abs/1703.10593.

Abkürzungsverzeichnis

BCE Binary Cross Entropy

Conv Convolutional

DSSD Deconvolutional Single Shot Detector

FN False Negative

FP False Positive

FPN Feature Pyramid Network

GAN Generative Adversarial Network

IoU Intersection over Union

MACE Mean Absolute Cubed Error

MAP Mean Average Precision

MaxPool Max-Pooling

MSE Mean Squared Error

NMS Non Maximum Suppression

R-CNN Region based Convolutional Neural Network

R-FCN Region-based Fully Convolutional Network

ReLU Rectified Linear Unit

RoI Region of Interest

SConv Separable Convolutional

SPL Standard Platform League

SSD Single Shot MultiBox Detector

TN True Negative

TP True Positive

YOLO You Only Look Once

ZeroPad Zero-Padding

Abbildungsverzeichnis

Tabellenverzeichnis

A Versuchsergebnisse: Verschiedene Distanzen

Entfernung: 0,3m	1	2	3	4	5	6	7	8	9
Erkannt (dieser Ansatz)	1	1	-	-	-	-	-	-	-
Erkannt (B-Human)	0	0	-	-	-	-	-	-	-
Distanz (Vorhersage)	27	60	-	-	-	-	-	-	-
Distanz (Gemessen)	-	-	-	-	-	-	-	-	-

Entfernung: 1,5m	1	2	3	4	5	6	7	8	9	Mittelwert	Standardabweichung
Erkannt (dieser Ansatz)	1	1	1	1	1	1	1	2	2		
Erkannt (B-Human)	1	1	1	1	1	0,5	1	1	2	1,43333333	0,195505044
Distanz (Vorhersage)	1,3	1,5	1,4	1,5	1,2	1,1	1,7	1,5	1,7	1,26666667	0,129099445
Distanz (Gemessen)	1,3	1,3	1,2	1,1	-	-	1,4	1,3	1,1	1,36111111	0,1751983
Distanz (Mittelwert)	1,3	1,4	1,3	1,3	1,2	1,1	1,55	1,4	1,7		

Entfernung: 3,0m	1	2	3	4	5	6	7	8	9	Mittelwert	Standardabweichung
Erkannt (dieser Ansatz)	1	1	1	1	1	0	1	1,5	1,5		
Erkannt (B-Human)	1	0	1	1	1	1,5	1	1,5	1,5	2,9625	0,509901951
Distanz (Vorhersage)	2,6	2,3	2,6	2,8	2,7	-	3,9	3,5	3,3	2,7	0,533853913
Distanz (Gemessen)	2,3	2,2	2,5	2,2	2,5	-	3,1	3	3,8		
Distanz (Mittelwert)	2,45	2,25	2,55	2,5	2,6	-	3,5	3,25	3,55	2,83125	0,486055552

Abbildung A.1: **Versuchsergebnisse: Verschiedene Distanzen (0,3 m - 3,0 m).** Die Spalten stehen für die Posen 1-9. Die Werte für die Kategorie „Erkannt“ repräsentieren die Anzahl der erkannten Roboter. Dabei bedeutet zum Beispiel der Wert 1,5, dass in manchen Frames 1 Roboter und in anderen 2 Roboter erkannt wurden. Die Werte für die Distanzen sind in Meter angeben.

Entfernung: 4,5m	1	2	3	4	5	6	7	8	9		
Erkannt (dieser Ansatz)	1	1	1	1	0	0	1	1	1,5		
Erkannt (B-Human)	0	0	0	0	0	0	0	1	0	Mittelwert	Standardabweichung
Distanz (Vorhersage)	6	5,5	4,1	4,8	-	-	5	5	5,5	5,12857143	0,607100839
Distanz (Gemessen)	4,4	4,1	3,5	2,6	-	-	3,9	3,9	3,8	3,74285714	0,595069024
Distanz (Mittelwert)	5,2	4,8	3,8	3,7	-	-	4,45	4,45	4,65	4,43571429	0,494568715

Entfernung: 6,0m	1	2	3	4	5	6	7	8	9
Erkannt (dieser Ansatz)	0	0	0,5	1	0	0	0	0	0
Erkannt (B-Human)	0	0	0	0	0	0	0	0	0
Distanz (Vorhersage)	-	-	-	6	-	-	-	-	-
Distanz (Gemessen)	-	-	-	4,3	-	-	-	-	-

Entfernung: 9,0m	1	2	3	4	5	6	7	8	9
Erkannt (dieser Ansatz)	0	0	0	0	0	0	0	0	0
Erkannt (B-Human)	0	0	0	0	0	0	0	0	0
Distanz (Vorhersage)	-	-	-	-	-	-	-	-	-
Distanz (Gemessen)	-	-	-	-	-	-	-	-	-

Abbildung A.2: **Versuchsergebnisse: Verschiedene Distanzen (4,5 m - 9,0 m).** Die Spalten stehen für die Posen 1-9. Die Werte für die Kategorie „Erkannt“ repräsentieren die Anzahl der erkannten Roboter. Dabei bedeutet zum Beispiel der Wert 1,5, dass in manchen Frames 1 Roboter und in anderen 2 Roboter erkannt wurden. Die Werte für die Distanzen sind in Meter angeben.