Abschlussbericht Projektgruppe 630

Objektdetektion in Bildern in Echtzeit unter Ressourcenbeschränkung

Barth, Matthias Brömmel, Piet Feiniger, David
Kanwischer, Alexander Nentwich, Franz
Phan, Thanh Long Schawohl, Sabrina Schürk, Peter
Seidl, Tristan Wehrmaker, Marvin Wilking, Rahel

30. September 2020

Inhaltsverzeichnis

1 Einleitung

Unter dem Begriff „Künstliche Intelligenz (KI)“ haben sich in den letzten Jahren diverse Verfahren und Produkte, sowohl im Privat- als auch im Industriesektor, etabliert. Sei es in Form von Sprachassistenten im häuslichen Umfeld, teilautonom fahrender Fahrzeuge oder spezieller Verarbeitungsprozesse für Smartphone-Kameras. Hierbei steht die benötigte Rechenleistung entweder in dezentralen Rechenzentren oder in Form von spezieller Hardware, wie zum Beispiel einer Neural Processing Unit (NPU), direkt im Gerät zur Verfügung. So besitzen bereits Smartphones spezielle Prozessoren mit integrierten NPUs um benötigte Berechnungen für Deep-Learning-Algorithmen effizient auszuführen[Syn].

Die Robot World Cup Initiative (RoboCup) bietet eine Plattform für die Forschung im Bereich Robotik und KI, welche sich den menschlichen Fußball als Vorbild nimmt. Ziel ist es ein Fußballspiel mit einem Team vollständig autonomer humanoider Roboter gegen ein Team menschlicher Fußballspieler nach offiziellen FIFA-Regeln zu gewinnen[Rob]. Die Technische Universität Dortmund ist mit ihrem Team „Nao Devils Dortmund“[Naoa] Teil dieser Initiative und forscht aktiv in der Standard Platform League (SPL). Hierbei wird standardisierte Hardware verwendet, wodurch für jedes Team identische Voraussetzungen existieren und ein Fokus auf der Software-Entwicklung liegt[Spl]. Derzeit wird der NAO-Roboter der Firma SoftBank Robotics in Version 6 verwendet[Naob].

Anders als bei eingangs erwähnten Anwendungen im Bereich KI, besitzt der NAO V6 nur beschränkte Rechenleistung, vor allem keine spezialisierte NPU. Es stellt somit eine besondere Herausforderung dar, unter den gegebenen Ressourcen alle notwendigen Anforderungen einer dynamischen Echtzeit-Umgebung zu verarbeiten. Aktuelle Forschungsthemen aus dem Bereich der KI lassen sich somit nicht ohne Weiteres auf einen NAO V6 übertragen.

1.1 Motivation und Zielsetzung

Im Zuge der Projektgruppe 630 aus dem Wintersemester 2019/2020 sowie Sommersemester 2020 werden diverse Verfahren betreffend der Objektdetektion in Bildern in Echtzeit unter Ressourcenbeschränkung betrachtet. Übergeordnete Zielsetzung ist hierbei eine verbesserte Erkennung aller relevanten Objektklassen im Spielverlauf. Dazu sollen andere Roboter, Spielfeldmarkierungen, der Spielball sowie die Tore erkannt werden und in die Entscheidungsfindung des NAO-Roboters einfließen. Eine besondere Herausforderung bei der Verwendung solcher Verfahren ist die notwendige Einhaltung einer Laufzeitbeschränkung. Auf die Dynamik eines Fußballspiels muss von einem NAO-Roboter in Echtzeit reagiert werden. Bei einer Bildaufnahme von 30 FPS entsteht somit ein maximales Zeitfenster von 33 ms, indem sämtliche Entscheidungen getroffen bzw. Handlungen eingeleitet

werden müssen. Dabei sollen aktuell genutzte Verfahren aus der KI, wie beispielsweise ein adaptiertes YOLOv2-Netz zur Ballerkennung, gegen modernere Ansätze geprüft werden. Erstmalig werden auch Ansätze der Bildsegmentierung auf einem NAO-Roboter evaluiert, ob Spielfeldmarkierungen durch ein DNN-gestütztes Verfahren erkannt werden können. Hierzu wird bislang ein heuristisches Scan-Linien-Verfahren verwendet.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit behandelt in Kapitel 2 die Verfolgung der maßgeblichen Zielsetzung. Dabei werden in Abschnitt 2.2 drei aktuelle Ansätze der Objektdetektion vorgestellt, auf eine Verwendung mit einem NAO-Roboter geprüft und abschließend gegeneinander verglichen. In Abschnitt 2.3 werden hingegen Ansätze zur Bildsegmentierung vorgestellt. Da es sich hierbei um eine erstmalige Verwendung handelt, wird einleitend auf unterschiedliche Verfahren zur Erstellung von Trainingsdaten eingegangen. Hierbei werden synthetisch erzeugte Trainingsdaten in Abschnitt 2.3.1 und manuell aufbereitete Daten in Abschnitt 2.3.2 vorgestellt. Letzte dienen als Grundlage um in Abschnitt 2.3.2.3 automatisiert weitere Trainingsdaten zu erzeugen. Die Abschnitte 2.3.3 bis 2.3.6 stellen anschließend diverse Ansätze der Bildsegmentierung vor. Hervorzuheben ist die Adaption eines U-Net-Ansatzes in Abschnitt 2.3.6, da dieser in Abschnitt 2.4 auf dem NAO-Roboter integriert wurde. Von der eigentlichen Zielsetzung dieser Arbeit losgelöst, werden in Kapitel 3 Anpassungen am Verhalten der NAO-Roboter vorgestellt. Abschließend findet sich in Kapitel 4 eine Zusammenfassung der wichtigsten Erkenntnisse sowie ein Ausblick ausgewählter Themen.

2 Vision

Die bisherige Bildverarbeitung eines NAO-Roboters basiert maßgeblich auf einem Scan-Linien-Ansatz für Ball- und Linienerkennung, sowie einer YOLOv2-Adaption zur Ballerkennung (siehe Abschnitt 2.1). In diesem Kapitel wird die Adaption und Bewertung diverser aktueller Ansätze der Object Detection (siehe Abschnitt 2.2) und erstmalig Image Segmentation (siehe Abschnitt 2.3) auf dem NAO-Roboter vorgestellt. Dabei stellen die beschränkten CPU-Ressourcen des NAO V6 die maßgebliche Herausforderung bei der Adaption der gewählten Ansätze dar, da diese auf weitaus leistungsstärkeren Systemen entwickelt und genutzt werden. Darüber hinaus wird ein Experiment vorgestellt, ob es mit Hilfe eines neuronalen Netzes (NN) möglich ist, die Position eines NAO-Roboters auf dem Spielfeld anhand seines Kamerabildes zu bestimmen (siehe Abschnitt 2.5).

2.1 Aufzeigen von aktueller Pipeline im Roboter

Die derzeitige Architektur (siehe Abb. 2.1) zum Erkennen von Robotern und Bällen nutzt dafür verschiedene Techniken. Die Eingabe besteht aus den Kamerabildern und den dazugehörigen Kammeramatrizen. Für die Erkennung von Robotern wird Objekt Detection mit einem Netz, das vom Prinzip ähnlich wie Yolov2 ist, verwendet. Durch einen Kalman-Filter werden dann die Roboter Standpunkte in Weltkoordinaten vorhergesagt. Mithilfe des Scan Lines Verfahrens werden zum einen Linien erkannt, die mitunter zur Lokalisierung verwendet werden. Zum anderen findet das Scan Lines Verfahren auch andere Objekte im Bilde, welch als interessante Regionen gekennzeichnet werden. Diese Regionen werden einem CNN übergeben, welches auf die Erkennung von Bällen ausgelegt ist. Für die Vorhersage der Ballposition, wird der aktuelle Ball einem Kalman-Filter übergeben.

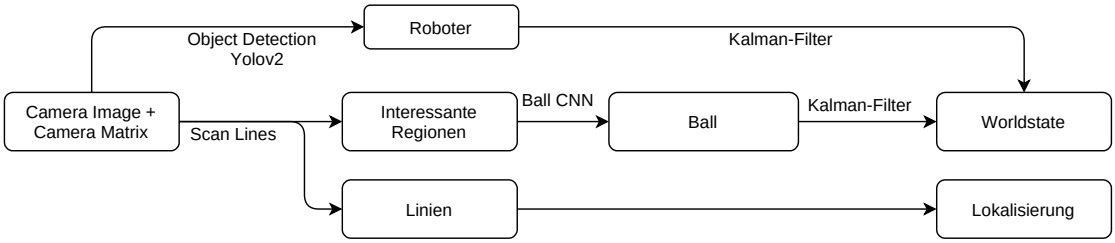


Abbildung 2.1: Bisheriger Ansatz zur Erkennung von Objekten und zur Lokalisierung

Die Idee, dargestellt in Abb. 2.2 , ist es diese Architektur zu vereinfachen. So soll über Objekt Detection Roboter, Bälle und andere relevante Objekte wie z.B. Tore

erkannt werden. Für Roboter und Bälle soll weiterhin mit den vorhandenen Kalman-Filtern die Weltkoordinaten ermittelt werden. Anstelle von Scan Lines soll mit Image Segmentation die Linien erkannt werden. Die dabei erkannten Linen sollen daraus mit einem Linien Algorithmus in eine vom Roboter erfassbare Form gebracht werden. Die Roboter lokalisieren sich weiterhin anhand der gefundenen Linien.

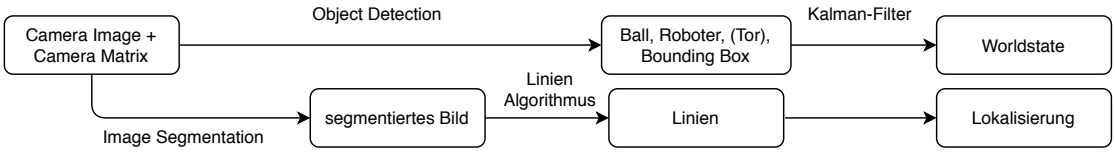


Abbildung 2.2: Neuer Ansatz zur Erkennung von Objekten und zur Lokalisierung.

2.2 Object Detection

Im Bereich der Objektdetektion wurden drei Ansätze verfolgt, die jeweils auf verschiedenen Verfahren basieren. Wichtige Auswahlkriterien sind dabei die Echtzeiterkennung und Techniken, die das Erkennen von weit entfernten Objekten verbessern. Zum einen wurde ein Ansatz auf Basis des YOLOv3 verfolgt. Da in der aktuell verwendeten Bildverarbeitung ein Netz, welches auf YOLOv2 basiert, eingesetzt wird, wurde dieses Netz als neuere Version dessen in Erwägung gezogen. Diese neuere Version ist außerdem besser darin kleine Objekte zu erkennen als YOLOv2[RF18b]. Als weiterer Ansatz wurde der Single Shot MultiBox Detector (SSD) verwendet. Dieser zeichnet sich dadurch aus, dass Bounding Boxes aus verschiedenen Auflösungen vorhergesagt werden. Es wird überprüft, ob dies die Erkennung kleiner Bälle verbessert. Außerdem wurde ein Ansatz auf Basis des CenterNet verfolgt. Dieses sagt die Bounding Boxes über eine Vorhersage der Objektmittelpunkte hervor. Das CenterNet ist schneller als YOLOv3 und erkennt im COCO-Datensatz kleine Objekte etwas besser als dieses[ZWK19]. Ein weiterer Ansatzpunkt, welcher nicht weiter verfolgt wurde, ist die Betrachtung der Bilder als Sequenzen und damit das Nutzen rekurrenter Netzstrukturen, da dies parallel bereits an anderer Stelle umgesetzt wurde[Moo20].

Für die Experimente im Bereich der Objektdetektion wurde sich auf die Erkennung von Robotern und Bällen in der oberen Kamera eingeschränkt. Dies dient dem Ziel die Erkennung kleiner, weit entfernter Bälle zu verbessern, da diese in den Bildern der unteren Kamera nicht auftreten. Eine Verbesserung der Erkennung ermöglicht eine schnellere Reaktion der Roboter auf die aktuelle Spielsituation. Mit dieser Einschränkung wurden im Labeling-Tool ImageTagger[1] Datensätze herausgesucht die für diese Klassen vollständig gelabelt sind. Dabei wurden möglichst neue Datensätze präferiert, um größtenteils Bilder des NAO V6 zu verwenden. Die Aufteilung dieser in Trainings-, Validierungs-, und Testdatensatz ist in Tabelle 2.1 zu sehen. Des Weiteren wurde manuell ein Datensatz

[1] https://imagetagger.bit-bots.de/

zusammen gestellt, der fast ausschließlich Bilder enthält, auf denen der Ball weit entfernt ist. Dieser Datensatz dient dazu, dass evaluiert werden kann, wie gut die trainierten Netze in solchen Situationen den Ball erkennen.

	Training	Validierung	Test
IDs der ImageTagger-Datensätze	401	464	403
	414		408
	466		412
	468		
	527		
Anzahl Bilder	3078	671	935

Tabelle 2.1: Übersicht der Zusammenstellung der genutzten Datensätze für die Objekterkennung.

2.2.1 YOLOv3

In diesem Abschnitt wird das Verfahren You Only Look Once (YOLO) zur Objektdetektion präsentiert. Dieses ist die dritte Version und wurden von Joseph Redmon und Ali Farhadi im Jahr 2018 vorgestellt[RF18b]. Die Arbeitsweise von YOLO kann wie folgt kurz zusammengefasst werden. Es wird ein einziges neuronales Netz für das ganze Bild angewendet. Das Bild wird dabei in die kleinen Regionen eingeteilt. Für jede kleine Region werden die Bounding Boxen und Konfidenzwerte berechnet. Weiterhin wird für jede Bounding Box eine Wahrscheinlichkeit von den definierten Klassen berechnet. Die meisten berechneten Bounding Boxen werden wegen niedriger Konfidenzwerte durch die Non-Maximum Suppression (NMS) unterdrückt. In der Abb. 2.3 wird die Arbeitsweise von YOLO dargestellt. Für den NAO-Roboter wird bereits ein auf YOLOv2 basierendes Netz eingesetzt. In dieser Projektgruppe wird ein CNN-Netz, welches auf YOLOv3 basiert, für Roboter entwickelt. In Tabelle 2.2 wird dargestellt, dass YOLOv3 nicht nur eine gute Genauigkeit liefert, sondern auch deutlich schneller ist als andere Netze. Die Laufzeit ist ein wichtiger Faktor für den Wahl des Netzes wegen der Beschränkung von der Hardware des Roboters. Im Vergleich mit YOLOv2 ist aber YOLOv3 zweimal langsamer, siehe Tabelle 2.3. Allerdings ist YOLOv3 besser als YOLOv2 für das Detektieren von den kleinen Objekten, weil YOLOv3 Shortcut Connections verwendet, mit denen die feinen Informationen der vorherigen Schichten abgerufen werden können. Diese Schichten ermöglichen, dass das Netz mehr Informationen von der Feature-Map bekommen kann.

2.2.1.1 Datensatz

Der Datensatz, welcher für die Experimente benutzt wird, stammt aus der Seite ImageTagger[2]. Dabei werden die Bilder aus den Bildmengen 401, 414, 466 und 468 für Training

[2]https://imagetagger.bit-bots.de/

Methode	**mAP**	**Zeit**
SSD321	45,4 %	61 ms
DSSD321	46,1 %	85 ms
R-FCN	51,9 %	85 ms
SSD513	50,4 %	125 ms
DSSD513	53,3 %	156 ms
FPN FRCN	59,1 %	172 ms
RetinaNet-50-500	50,9 %	73 ms
RetinaNet-101-500	53,1 %	90 ms
RetinaNet-101-800	57,5 %	198 ms
YOLOv3-320	51,5 %	22 ms
YOLOv3-416	55,3 %	29 ms
YOLOv3-608	57,9 %	51 ms

Tabelle 2.2: Performance von YOLOv3 auf dem COCO Datensatz im Vergleich mit den anderen Netzen. Die Zahlen wurden aus [RF18b] genommen.

Model	**FPS**
YOLOv2 608x608	40
YOLOv3 608x608	20

Tabelle 2.3: Performance von YOLOv2 und YOLOv3 mit der Eingabegröße von 608x608 auf dem COCO Datensatz. Die Zahlen wurden aus [RF18a] genommen.

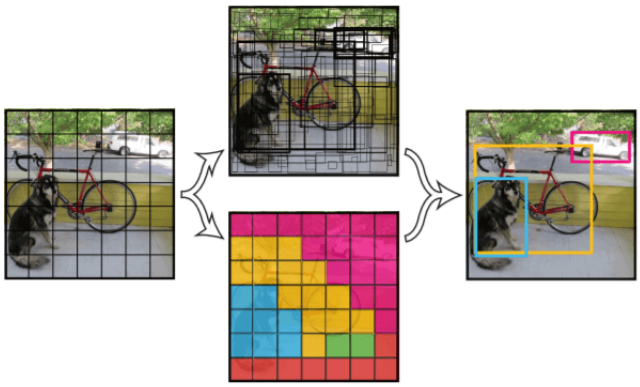


Abbildung 2.3: Die Arbeitsweise von YOLO. Das Bild wird in die kleinen Regionen aufgeteilt (das linke Bild). Dann wird für jede Region eine Bounding Box berechnet (das mittlere obere Bild). Weiterhin wird die vorhergesagte Klasse für die jeweilige Region berechnet. Aus der Kombination von der Bounding Box und vorhergesagten Klasse resultiert das Ergebnis (das rechte Bild)[Red+15].

und Validierung bezogen, wobei nur Bilder mit Ball oder Roboter genutzt werden. Diese Bilder wurden von der oberen Kamera des NAO-Roboters aufgenommen. Insgesamt umfasst der Datensatz über 2.500 Bilder. Für das Training wurde der Datensatz zufällig in 80 % Trainings- und 20 % Validierungsdaten aufgeteilt.

Abbildung 2.4: Beispielbilder aus dem aufbereiteten Datensatz.

2.2.1.2 Experimente und Auswertungen

Für die Evaluierung wurde bisher das YOLO-Netz mit den unterschiedlichen Architekturen versucht. Nach den verschiedenen Durchläufen wurde die Netzarchitektur, welche in der Tabelle B.1 beschrieben wird, als das Standardnetz für das Experiment gewählt. Für das Experiment wurde das Netz auf eine Batchsize von 64 und eine Lernrate von 0,001 eingestellt. Das Netz wurde in 15.000 Batches trainiert. Während des Trainings wurden die Gewichte des Netzes zwischengespeichert. Dabei wurde das Gewicht, welches das beste Ergebnis auf den Evaluationsdaten ergab, immer gespeichert. Damit erhält das Netz zum Ende des Trainings das beste Gewicht. Für die Berechnung der Average

Roboter AP	62,31 %
Ball AP	44,28 %
mAP	53,29 %

Tabelle 2.4: Die Ergebnisse von dem trainierten YOLO 240x180.

	Roboter AP	Ball AP	mAP
YOLOv3 240x180	62.31%	44.28%	53.29%
YOLOv3 320x240	64.77%	56.80%	60.78%

Tabelle 2.5: Vergleichen der Qualität von YOLOv3 240x180 mit YOLOv3 320x240.

Precision (AP) und mean Average Precision (mAP) wurden nur die Bounding Boxen berücksichtigt, welche mit den Ground Truth Bounding Boxen einen IoU-Wert größer als 0,5 haben. Das Ergebnis von dem trainierten Netz wird in Tabelle 2.4 präsentiert. Diese Werte zeigen, dass das Netz die NAO-Roboter besser als die Bälle erkennt. Allerdings ist der AP-Wert der NAO-Roboter nur 62,31 %. Dieser Wert ist geringer als erwartet. In Abb. 2.5 werden einige Testbilder von dem trainierten YOLO 240x180 mit einem Konfidenzthreshold von 0,5 vorgestellt. In den dargestellten Testbildern werden nur nahe Bälle erkannt. Wenn der Ball zu weit entfernt ist, kann das Netz diesen nicht mehr detektieren. Auch weit entfernte NAO-Roboter werden gar nicht oder nur sehr ungenau erkannt. Für das zweite Experiment wurde ein zweites Netz YOLOv3 320x240 mit einer Netzeingabe von 320x240 Pixeln trainiert. Die Architektur von YOLOv3 320x240 wird in Tabelle B.1 dargestellt. Für das Training vom zweiten Netz wurden auch die Batchsize von 64 und Lernrate von 0,001 verwendet. Das Netz wurde in insgesamt 15.000 Batches trainiert. In Tabelle 2.5 werden die Ergebnisse von YOLOv3 240x180 mit den Werten von YOLOv3 320x240 verglichen. Die Werte zeigen, dass mit einer größeren Netzeingabe das Netz eine bessere Genauigkeit liefern kann. Im Weiteren wurde das trainierte Netz YOLOv3 240x180 mit dem DCG kompiliert und die Laufzeit auf dem NAO-Roboter getestet. Die Laufzeit beträgt circa 30 ms und ist noch deutlich zu hoch, da auf dem NAO-Roboter noch weitere Routinen ausgeführt werden müssen.

2.2.1.3 Fazit

Für die trainierten Netze wurde bisher nur die Laufzeit auf dem Roboter getestet und sie wurde noch nicht in einem Spiel geprüft. Deswegen gibt es keine Aussage, ob diese Netze ein besseres Ergebnis als das aktuelle Netz auf dem Roboter liefern kann. Dies soll als nächstes geprüft werden. In Tabelle 2.6 wurde die Laufzeit von YOLOv3 240x180 mit der Laufzeit von dem aktuellen Netz auf dem Roboter verglichen. Die Werte zeigen, dass das aktuelle Netz circa 4,5 Mal schneller als YOLOv3 240x180 läuft. Somit ist das aktuelle Netz eine bessere Variante für die Objektdetektion in Echtzeit.

Abbildung 2.5: Einige Testbilder mit dem trainierten YOLOv3 240x180.

Netz	**Laufzeit**
aktuelles Netz auf NAO	6,5 ms
YOLOv3 240x180	30 ms

Tabelle 2.6: Vergleich der Laufzeit von YOLOv3 240x180 mit dem aktuellen Netz auf dem NAO-Roboter.

2.2.2 SSD

Ein weiterer Ansatz für die Objektdetektion ist der Single Shot MultiBox Detector (SSD)[Liu+16]. Eine Besonderheit dessen ist, dass Bounding Box Kandidaten in verschiedenen Bildauflösungen vorhergesagt werden, statt nur in der letzten Netzwerk-Schicht diese Kandidaten zu erzeugen. Dies begünstigt das Erkennen von Bällen, da diese bei kleinen Bildauflösungen auf wenige Pixel skaliert sind, was das Erkennen dieser erschwert. Ein weiterer Grund für die Wahl des Ansatzes ist die Geschwindigkeit des Netzes, welche für die Ermöglichung der Echtzeiterkennung relevant ist. Da das Netz später unter beschränkten Ressourcen arbeiten muss, ist ein grundsätzlich schnelles Netz vorteilhaft.

2.2.2.1 Technische Umsetzung

Die technische Umsetzung des Ansatzes basiert auf einem bereits bestehenden Repository[deG19]. Das ausgewählte Netz ist hierbei eine Implementierung des SSD300 Ansatzes. Dieses erwartet Eingabebilder der Größe 300x300 Pixel. Zum Training wurden die Datensätze wie in Tabelle 2.1 aufgelistet verwendet und entsprechend skaliert. Die Annotationsdateien wurden in das PASCAL VOC Format überführt. Aufgrund von Beschränkungen des Repositories, wurden aus dem Datensatz außerdem Bilder, die keine Annotationen enthalten, entfernt. Es werden vier Klassen verwendet, die Klassen Ball, Roboter, Goalpost und Penaltycross.

2.2.2.2 Training und Ergebnisse

Das Netz wurde 12.000 Iterationen lang mit einer Batchsize von 4 trainiert, wobei nach 8.000 Iterationen keine weitere Verbesserung des Losses zu erkennen war. Die Lernrate wurde auf 0,0001 gesetzt. Die Average Precision (AP) lag für die Klassen Roboter bei 92 %, Ball 24 %, Penaltycross 26 % und Goalpost bei 54 %. Dabei handelt es sich um das beste erzielte Ergebnis. Roboter erkennt das Netz also sehr gut, Bälle allerdings deutlich schlechter. Das entspricht nicht dem Ziel die Ballerkennung zu verbessern.

2.2.2.3 Fazit

Das Netz ist gut darin Roboter zu erkennen, allerdings wesentlich schlechter darin Bälle zu erkennen. Die Bilder werden durch die Skalierung auf 300x300 Pixel für den Input bereits stark reduziert. Eine höhere Auflösung könnte die Erkennung der Bälle verbessern. Dafür könnte ein SSD512 Netz ausprobiert werden. Allerdings ist dabei auch zu beachten, dass eine höhere Auflösung auch eine höhere Laufzeit verursachen würde. Eine Verkleinerung der Netzstruktur wurde bisher noch nicht vorgenommen, wodurch der Ansatz auf dem Roboter nicht einsatzfähig ist.

2.2.3 CenterNet

Das CenterNet ist eine Netzstruktur, die im April 2019 von Zhou et al. entwickelt wurde[ZWK19]. Statt einer direkten Vorhersage von Bounding Boxes, wie bei anderen Netzen für Objektdetektion, ist es das Ziel eines CenterNet die Mittelpunkte von Objekten über die Vorhersage einer Heatmap zu bestimmen. Die Breite und Höhe des Objekts werden regressiert. Aus der vorhergesagten Heatmap werden die Peaks als Mittelpunkte extrahiert, um daraus Objekte der jeweiligen Klasse vorherzusagen. Durch den Fokus auf die Vorhersage der Mittelpunkte ist eine bessere Unterscheidung sich überlappender Objekte möglich. Dies ist eine Situation, welche häufig auf Roboter und Bälle zutrifft, weshalb dies das Potential bietet die Erkennung dieser zu verbessern.

Der von den Autoren bereitgestellte Code[Zho19] umfasst die Verwendung diverser Netzwerke als Basisnetzwerk des Ansatzes. Das CenterNet, mit dem ResNet-18 Netz als Basis, erreicht bei den Experimenten von Zhou et al. auf dem COCO-Datensatz 142 FPS und 28,1 mAP. Bei Verwendung des DLA-34 Basisnetzwerks läuft das CenterNet auf

gleicher Hardware zwar nur mit 52 FPS, erreicht dafür aber 37,4 mAP. Damit ist es außerdem bei besserem mAP-Wert mehr als doppelt so schnell wie YOLOv3 auf dieser Hardware.[ZWK19]

2.2.3.1 Deep Layer Aggregation

Für die Experimente wurde als Basisnetz das DLA-34 Netzwerk verwendet. Das ResNet-18 Netz ist zwar mehr als doppelt so schnell, wofür allerdings die Genauigkeit leidet. Das Hourglass-Netzwerk, ein weiteres Basisnetz, ist bereits auf der deutlich stärkeren Hardware, welche für die Experimente von Zhou et al. verwendet wurde, nicht in der Lage in Echtzeit zu laufen.[ZWK19]

Das DLA-34 Netzwerk von Yu et al.[Yu+18] nutzt Deep Layer Aggregation (DLA), um die verschiedenen Ebenen eines Netzwerkes komplexer zu aggregieren als mit reinen Shortcut Connections. Dies dient dem Zweck, die Informationen aus frühen Layern mit denen aus späteren besser verbinden zu können. Die Aggregation besteht dafür sowohl aus einer iterativen, als auch einer hierarchischen Aggregation der Ebenen. Die Kombination dieser beiden verbessert sowohl die örtliche, als auch die semantische Verbindung zwischen den Ebenen und somit die Vorhersage. Die Einbeziehung von Informationen der frühen Layer mit einer hohen Auflösung begünstigt die Erkennung von kleinen, weit entfernten Bällen.

Für die Experimente wurde das DLA-Netzwerk außerdem ohne einige von Zhou et al. vorgenommene Modifikationen verwendet, da diese unter anderem die Verwendung von Deformable Convolutions[Dai+17] umfassen, welche noch nicht in Keras bzw. TensorFlow umgesetzt sind und somit nicht ohne erheblichen Aufwand dorthin übersetzt werden könnten. Dies ist relevant, da das Netz in Keras vorliegen muss, um vom Devils Code Generator (DCG) kompiliert werden zu können.

2.2.3.2 Experimente und Ergebnisse

Für die mit dem CenterNet durchgeführten Versuche wurde die PyTorch-Implementierung des ursprünglichen CenterNet auf die benötigten Funktionalitäten reduziert und angepasst, um die Roboterbilder zu verarbeiten. Die Labels wurden dafür in das COCO-Format[3] konvertiert.

Es werden drei Experimente vorgestellt. Für alle Experimente entsprechen die meisten Parameter den Default-Einstellungen des Ursprungscodes, geänderte Parameter werden im Weiteren erläutert. Es wird mit Bildern in voller Auflösung, also 640x480, trainiert. Das erste Experiment wurde über 160 Epochen mit einer Batchsize von 8 und einer Lernrate von 1,5e-5 trainiert. Im zweiten Experiment wird versucht, die Modellgröße zu verringern. Die Channels der Layer werden jeweils um die Hälfte reduziert. Dies hat die Parameteranzahl fast geviertelt, wobei sie noch immer bei mehreren Millionen liegt. Die restlichen Trainingsparameter wurden gleich belassen und es wurde für 200 Epochen trainiert. Die Modifizierung des Netzwerkes hat allerdings zur Folge, dass das zuvor

[3] https://cocodataset.org/#format-data

verwendete Pretraining auf ImageNet[4]-Daten nicht weiter verwendet werden kann. Das zuletzt durchgeführte Experiment nutzt das gleiche Modell wie das vorige Experiment und fast gleiche Trainingsparameter, lediglich die Lernrate wurde verdoppelt. Das Training dauerte 148 Epochen.

Beispiele für die Ergebnisse der Netze auf nicht für das Training verwendeten Bildern sind in Abb. 2.6, 2.7 und 2.8 dargestellt. Das erste der Netze liefert die besten Ergebnisse, was der Größe und dem Pretraining zuzuschreiben sein dürfte. Bei gleicher Größe, liefert das dritte Netz etwas bessere Ergebnisse als das zweite, trotz eines kürzeren Trainings. Festzustellen ist außerdem, dass im dritten Bild (siehe Abb. 2.8) das erste und dritte Netz jeweils den liegenden Roboter erkennen, auch wenn dieser in der Ground Truth nicht markiert ist. Alle Netze erkennen den groß im Bild auftretenden Roboter im zweiten Bild (siehe Abb. 2.7) schlecht oder gar nicht, was auch in anderen Bildern auftritt. Der Ball in diesem Bild wird nicht erkannt, ist allerdings bei Betrachtung der Heatmap auch schwierig zu erkennen, da der Mittelpunkt des Balles sehr nah am Mittelpunkt des Roboters liegt und in diesem untergeht. Positiv ist allerdings das Erkennen des Balles vom ersten Netz und insbesondere dem dritten Netz in dem ersten und dritten Bild.

Die Roboter- und Ballerkennung der Netze scheinen anhand visueller Evaluation gut zu sein, wenn auch nicht perfekt. Allerdings liegen keine Metriken für die Erkennung vor, die dies objektiv bestätigen würden.

2.2.3.3 Umstieg auf Keras

Die Übertragung des CenterNet-Ansatzes nach Keras ist ein wichtiger Schritt, um die Netze mit dem DCG für den Roboter kompilieren zu können und die Laufzeit auf diesen zu evaluieren. Dabei wurde von einem Git-Repository[xua19], welches den CenterNet-Rahmen in Keras bereits enthält ausgegangen und das DLA-34 Netz darin übertragen. Dies ist jedoch nicht vollständig möglich, da das PyTorch-Modell ConvTranspose2d Layer verwendet, dessen Parameter „goups“der Output-Dimension entspricht. Im Keras-Äquivalent, dem Conv2DTranspose Layer, existiert dieser Parameter nicht, das Verhalten des Layers stimmt also nicht genau überein. Das sich in der PyTorch-Implementierung des CenterNet die Form der Netzausgabe und der Ground Truth Daten unterscheidet macht eine Übergabe letzterer als Netzeingaben sowie die Berechnung der Loss-Funktion als Teil der Netzstruktur notwendig. In der Keras-Version konnten noch keine Ergebnisse gewonnen werden.

2.2.3.4 Fazit

Auch wenn der Ansatz subjektiv gute erste Ergebnisse in der PyTorch-Implementierung geliefert hat, muss dabei beachtet werden, dass dies noch mit für den Roboter viel zu großen Netzen erreicht wurde. Außerdem liegen für die Netze keine Metriken vor, die einen leichten Vergleich mit anderen Netzen möglich machen würden. Vor der Verkleinerung der Netze wurde der Fokus darauf gelegt, das Netz nach Keras zu übertragen, da es andernfalls nicht möglich ist, es mit dem DCG für die Roboter zu kompilieren. Diese

[4]http://www.image-net.org/

Abbildung 2.6: Ergebnisse der CenterNet-Experimente an einem ersten Beispielbild. Von links nach rechts: Ground Truth Heatmap, Predicted Heatmap, Ground Truth Bounding Boxes, Predicted Bounding Boxes. Von oben nach unten: Die drei Experimente in der Reihenfolge der Durchführung. Die Bilder werden durch diverse Maßnahmen augmentiert, weshalb die Bildausschnitte nicht absolut identisch sind. Das Bild stammt aus den Testdaten und wurde nie zum Training verwendet.

Übertragung war jedoch nicht ohne Aufwand und auch nicht völlig unterschiedsfrei möglich, da das Netz einen Parameter benötigt, welcher kein Äquivalent in Keras hat. Es war daher bislang nicht möglich, eine kleinere Form des Netzes auszuprobieren, weshalb keine Aussage darüber getroffen werden kann, ob das Netz in einer kleineren Form weiterhin gute Ergebnisse erzielen kann. Außerdem ist unklar, ob die Struktur des Netzes ohne weitere Probleme in eine Form gebracht werden kann, die eine Kompilierung mit dem DCG möglich macht.

2.2.4 Vergleich der Verfahren

Weder der SSD-Ansatz noch der CenterNet-Ansatz konnten für den Gebrauch auf dem Roboter evaluiert werden. Im Falle des SSD wurden bereits mit der vollen Netzgröße keine guten Ergebnisse für die Ballerkennung erreicht. Für die Ergebnisse des CenterNet liegen keine Metriken vor, welche die Güte der großen Netze zeigen. Von guten großen Netzen kann außerdem nicht direkt auf die Eignung auf den Robotern geschlossen werden, da für

Abbildung 2.7: Ergebnisse der CenterNet-Experimente an einem zweiten Beispielbild. Von links nach rechts: Ground Truth Heatmap, Predicted Heatmap, Ground Truth Bounding Boxes, Predicted Bounding Boxes. Von oben nach unten: Die drei Experimente in der Reihenfolge der Durchführung. Die Bilder werden durch diverse Maßnahmen augmentiert, weshalb die Bildausschnitte nicht absolut identisch sind. Das Bild stammt aus den Testdaten und wurde nie zum Training verwendet.

diese die Netzstrukturen signifikant verkleinert werden müssten, um dem Anspruch der Echtzeiterkennung zu genügen. Der YOLO-Ansatz hingegen hat ein Netz hervorgebracht, welches auf der Hardware der Roboter in Echtzeit laufen kann. Dabei wurde jedoch noch nicht berücksichtigt, dass neben der reinen Netzlaufzeit auch die Laufzeit für die Linienerkennung, das Verhalten und weiterer Komponenten bedacht werden muss. Das Netz erreicht für Roboter eine AP von 62,31 % und für Bälle eine AP von 44,28 %. Für dieses Netz wurde noch kein Vergleich mit der aktuellen Bilderkennung durchgeführt, weshalb nicht festgestellt werden kann, ob diese Werte eine Verbesserung darstellen. Das Netz hat außerdem ebenfalls Schwierigkeiten damit, weit entfernte Bälle und auch Roboter zu erkennen. Zusätzlich wurde das Netz zwar für den Roboter kompiliert, aber noch nicht in das Framework eingebaut, was es nicht direkt einsatzfähig macht.

Abbildung 2.8: Ergebnisse der CenterNet-Experimente an einem dritten Beispielbild. Von links nach rechts: Ground Truth Heatmap, Predicted Heatmap, Ground Truth Bounding Boxes, Predicted Bounding Boxes. Von oben nach unten: Die drei Experimente in der Reihenfolge der Durchführung. Die Bilder werden durch diverse Maßnahmen augmentiert, weshalb die Bildausschnitte nicht absolut identisch sind. Das Bild stammt aus den Validierungsdaten und wurde nie zum Training verwendet.

2.3 Image Segmentation mittels CNN

Nachteil der Object Detection im Anwendungsfall des NAO-Roboters ist die fehlende Möglichkeit Spielfeldlinien zu erkennen, da diese, durch die grundsätzliche Arbeitsweise einer Object Detection, nicht sinnvoll per Bounding Box erfasst werden können. Bounding Boxes werden als Rechtecke um gefundene Objektklassen gezogen und umschließen diese somit. Spielfeldlinien erstrecken sich hingegen über das komplette Spielfeld, wodurch eine Bounding Box in solchen Fällen ebenfalls das komplette Spielfeld erfasst. Innerhalb der Image Segmentation ist es hingegen möglich inhaltlich zusammenhängende Zonen bzw. Pixelbereiche in einem Bild zusammenzufassen und als eine Klasse, ohne Verwendung von einer Bounding Box, darzustellen (siehe Abb. 2.9). Damit ist es möglich, auch Spielfeldlinien durch ein neuronales Netz (NN) zu erkennen. Angetrieben durch die gesteigerte CPU-Leistung des NAO V6 und insbesondere die zusätzlich verfügbaren und bislang wenig genutzten CPU-Kerne wurde die Möglichkeiten der Image Segmentation

erstmalig auf dem NAO V6 getestet.

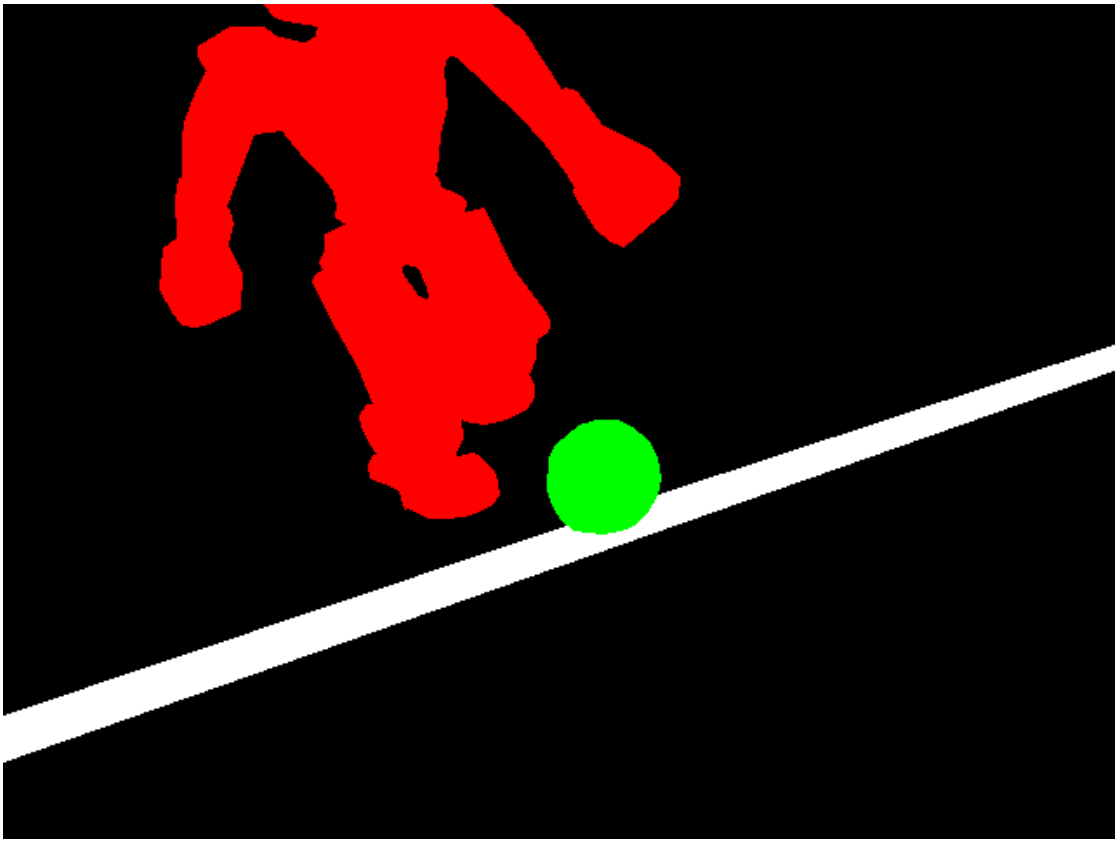


Abbildung 2.9: Segmentiertes Bild mit unterschiedlichen Farben je Objektklasse.

Innerhalb dieses Kapitels wird Evaluation diverser Ansätze zur Linienerkennung inklusive und exklusive weiterer Objektklassen, wie beispielsweise Roboter oder Tore, vorgestellt. Ebenfalls vorgestellt wird Erstellung von Trainingsdaten, sowohl durch manuelle Arbeit, als auch durch automatische Prozesse auf Basis des Projektes UERoboCup oder selbst trainierter NN.

2.3.1 Unreal Engine

Um ein Neuronales Netz zu trainieren, ist eine Große Menge an Trainingsdaten notwendig. Trainingsdaten mit passgenauen Masken für eine Segmentierung im Roboterfußball sind jedoch nicht in hinreichender Menge vorhanden und das manuelle Erstellen solcher Daten ist deutlich aufwändiger als Erstellen von Daten mit Bounding Boxes für die Objekterkennung.

Daher wurde eine automatische Methode zum Erstellen von Trainingsdaten in Betracht gezogen. Hierfür wurde das auf der Unreal Engine basierende Projekt UERoboCup

[Hes+17][5] verwendet. Dies erzeugt automatisch Szenen aus der Roboterperspektive, sowie die dazugehörigen Ground Truth Informationen für die Segmentierung.

Beim Test der so trainierten Netze auf echten Daten waren die Ergebnisse allerdings nicht zufriedenstellend. Dies wird darauf zurückgeführt, dass die generierten Bilder im Vergleich zu den Spielfeldaufnahmen zu homogene Spielsituationen darstellen, zu sauber und zu homogen belichtet sind.

Um die Nutzbarkeit der erzeugten Daten auf echten Bildern zu verbessern, wurden Fotos aus der Arena als Hintergrund eingefügt, Feldlinien transparenter gemacht und Pixelrauschen und Unschärfe eingefügt, siehe Abb. 2.10.

Abbildung 2.10: Beispiele von generierten Bildern ohne vorgenommene Anpassungen(links) und mit vorgenommen Anpassungen (rechts).

Trotz der vorgenommenen Anpassungen liefern die in der Engine gerenderten Bilder weiterhin keine zufriedenstellenden Ergebnisse auf echten Daten. Daher werden die mit UERoboCup erzeugten Bilder derzeit nicht verwendet.

2.3.2 Datensatz

2.3.2.1 COCO Annotator

Wie im vorherigen Abschnitt 2.3.1 erwähnt, standen der Projektgruppe keine Daten für das Trainieren von Segmentierungsnetzen zur Verfügung und es wurde ein neuer Datensatz erstellt [**segDataset**]. Das von der vorherigen Projektgruppe verwendete Tool zum Labeln der Daten (ImageTagger) ist nicht für das Erstellen von Trainingsdaten für die Segmentierung geeignet, da es nicht den notwendigen Funktionsumfang besitzt. Es musste daher ein Tool gefunden werden, das die Anforderungen dieser Projektgruppe erfüllt.

Für die Segmentierung ist es notwendig, dass nicht nur rechteckige Bereiche, sondern auch Polygone markiert werden können. Daraus leitet sich die erste Anforderung an das gesuchte Tool zum Labeln von Trainingsdaten für die Segmentierung ab. Zusätzlich zur

[5] https://github.com/TimmHess/UERoboCup

Polygonmarkierung ist es wichtig einzelne Roboter unterscheiden zu können. Das Tool soll demnach das Labeln verschiedener Instanzen einer Klasse ermöglichen. Weiter sollte beim Labeln eine komfortable Zusammenarbeit gewährleistet sein. Dazu gehört, dass eine gleichzeitige Bearbeitung der Daten möglich ist und alle Teammitglieder zu jeder Zeit auf alle Daten zugreifen können. Das Tool sollte also online zugänglich und für jedes Teammitglied erreichbar sein. Zu den weiteren geringer priorisierten Kriterien zählen leichte Bedienung, dass es sich um ein Open-Source-Tool handelt und dass das Tool ohne Kosten genutzt werden kann.

Schnell wurde eine große Sammlung an Tools zum Labeln von Bildern gefunden[Bro19]. Die aufgeführten Kriterien schränkten die Liste der Tools schnell auf eine kleine Auswahl ein. Nach einer Auswertung dieser Kandidaten fiel die Wahl auf das Tool COCO Annotator[Bro20]. Der COCO Annotator erfüllt alle oben beschriebenen Anforderungen und ermöglicht dem Team eine flexible Zusammenarbeit beim Labeln, da er browserbasiert funktioniert und somit von allen Teammitgliedern genutzt werden kann. Um die gleichzeitige Zusammenarbeit aller Projektgruppenmitglieder beim Annotieren zu ermöglichen, ist der COCO Annotator auf einem Server im Roboter Arena Netzwerk eingerichtet worden. Auf diese Weise können alle Mitglieder der Projektgruppe über einen VPN-Zugang zum Server gleichzeitig im selben Datensatz Bilder annotieren und haben jederzeit Zugriff auf den aktuellen Stand des Datensatzes. Der COCO Annotator wird in Form mehrerer Docker Container auf einer virtuellen Maschine bereitgestellt.

2.3.2.2 Erzeugung von Ground Truth Informationen

Die Bilder zum Annotieren wurden aus den Logdateien der Roboter von Spielen aus vorherigen Events (RoboCup 2019 und GermanOpen 2019) extrahiert. Es wird geprüft, ob sie eine reguläre Spielsituation zeigen und in den Datensatz aufgenommen werden können. Die Projektgruppe hat gemeinsam die gewählten Bilder segmentiert. Dabei wird in jedem Bild die Grundlinien (line), die Mittelkreislinie (centercircle), der Strafstoßpunkt (penaltycross), der Ball (ball), die Roboter (robot) und das Tor (goal) markiert. Beim Annotieren der Objekten ist es einfacher, wenn sich die Markierungen überlappen dürfen. Dies ist möglich, wenn beim Annotieren die Reihenfolge beachtet wird, um so die Überlappungskonflikte zu lösen. Da die Spielfeldmarkierungen und das Tor immer hinter den Robotern und dem Ball sind, ist hier die Reihenfolge irrelevant. Abb. 2.11 zeigt den Fortschritt bei korrekter Vorgehensweise.

Da die Annotationen durch unterschiedliche Personen persönlichen Standards unterliegen, haben sie unterschiedliche Qualität und Genauigkeit. Um dies zu verhindern wurde der Datensatz noch einmal korrigiert und die Genauigkeit vieler Bilder verbessert.

Der COCO Annotator bietet die Möglichkeit, die Labeldaten im namensgebenden COCO-Format zu exportieren. Der Export liefert eine JSON-Datei mit den Segmentierungen als Polygone auf dem Bild. Ein Python-Skript bereinigt vorhandene Überlappungen der Polygone. Aus den Polygonen zur Segmentierung lassen sich nun dynamisch die Masken für das Training von neuronalen Netzen benutzen. In Tabelle 2.7 ist der Aufbau des Datensatzes dargestellt.

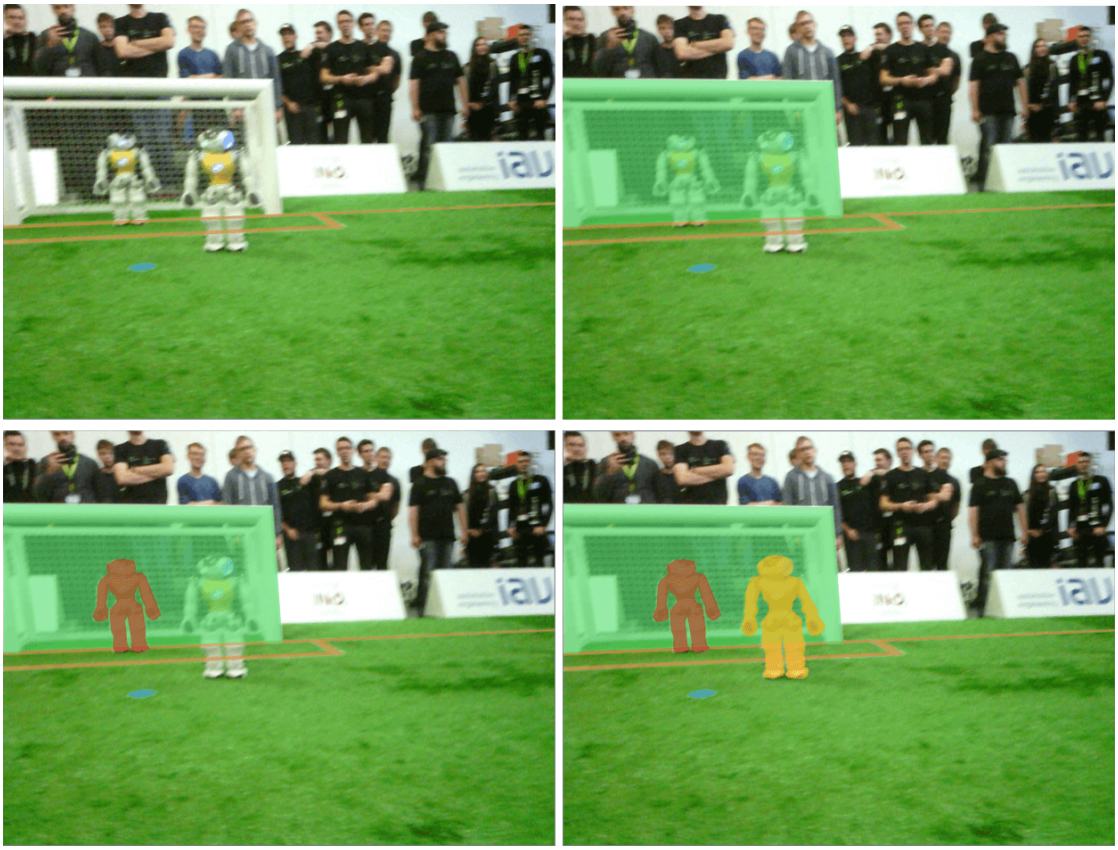


Abbildung 2.11: Beispiel für die korrekte Vorgehensweise beim Labeln von Bildern.

In der Abb. 2.12 ist ein Beispiel für zwei vollständig segmentierte Bilder. Die Überlappungen der Annotationen wurden hier noch nicht korrigiert.

Spiel	Bilder	Annotationen	line	ball	robot	cc	goal	pc
BandB	233	1427	231	76	811	136	119	48
HTWK-Leipzig	5	14	4	3	1	0	0	0
rUNSWift	326	1505	320	87	752	141	144	55
Team-Team	151	786	147	49	419	67	71	27
HULKs	464	2624	456	126	1360	285	278	113
Alle Spiele	1179	6356	1162	345	3347	633	616	247

Tabelle 2.7: Aufbau des Datensatzes. cc steht für centercircle und pc für penaltycross.

Abbildung 2.12: Beispiele segmentierter Bilder. Roboter haben als verschiedene Instanzen auch verschiedene Farben, gehören aber zur gleichen Klasse

2.3.2.3 Verwendung von großen Netzen zur automatischen Segmentierung

Ein größerer Datensatz mit mehr Bildern hilft beim Trainieren und Evaluieren von neuronalen Netzen, doch das Labeln der Bilder ist ressourcenaufwändig. Hieraus ergibt sich die Möglichkeit, Bilder automatisch labeln zu lassen. Dabei kann ein NN genommen werden, bei dem eine schnelle Inferenz nicht wichtig ist.

Zum automatischen Annotieren wurden zwei verschiedene Netze trainiert, welche nicht ressourcenbeschränkt sind. Das Erste NN ist eine FPN Semantic Segmentation Neural Network[Yak19], welches sich gut eignet, verschiedene Klassen von Objekte zu segmentieren. Dies wurde für alle Klassen verwendet, welche nur einmal in einem Bild vorkommen können (line, centercircle, penaltycross und goal). Von allen weiteren kommen verschiedene Instanzen des gleichen Objekts vor. Für diese wurde ein Instance Segmentation Neural Network[Wu+19] trainiert. Diese Aufteilung wurde vorgenommen, da das Netz zur Instanzerkennung Probleme bei Objekten hat, die über das ganze Bild gehen. Dies ist zum Beispiel bei den Grundlinien oft der Fall. Mit diesen Netzen wurden nun 8.821 weiter Bilder automatisch annotiert und zum Datensatz hinzugefügt.

Es wurde ein kleineres U-Net Semantic Segmentation Neural Network[Yak19] auf drei verschiedenen Wegen trainiert, um zu evaluieren, inwieweit die automatisch annotierten Bilder beim Training hilfreich sind. Für einen Einblick in die Architektur siehe Abschnitt 2.3.6. Beim Training der Netze unterscheidet sich nur die Zusammensetzung der Trainingsdaten. Alle Netze wurden für 100 Epochen mit 500 Schritten pro Epoche und einer Batchsize von 4 trainiert. Das erste Netz *(auto)* wurde mit 40 Epochen automatisch und 60 Epochen manuell gelabelten Bildern trainiert. Das zweite *(man)* wurde nur mit den manuell gelabelten Bildern trainiert. Das letzte Netz *(only_auto)* wurde nur mit automatisch gelabelten Bildern trainiert. Zum Evaluieren der Netze wurden die Bilder aus dem Spiel gegen Team-Team benutzt.

In der Abb. 2.13 werden alle drei Trainingsansätze verglichen. Dabei ist vor allem der F1 Score aus den Validierungsbildern interessant. Hier lässt sich sehen, dass alle drei

Werte am Ende nah bei einander sind. In der Tendenz sieht man, dass das kombinierte Training am besten ist. Das Training mit nur automatisch gelabelten Bildern schneidet am schlechtesten ab. Der Validierungs-F1-Score verbessert sich nach 60 Epochen kaum noch.

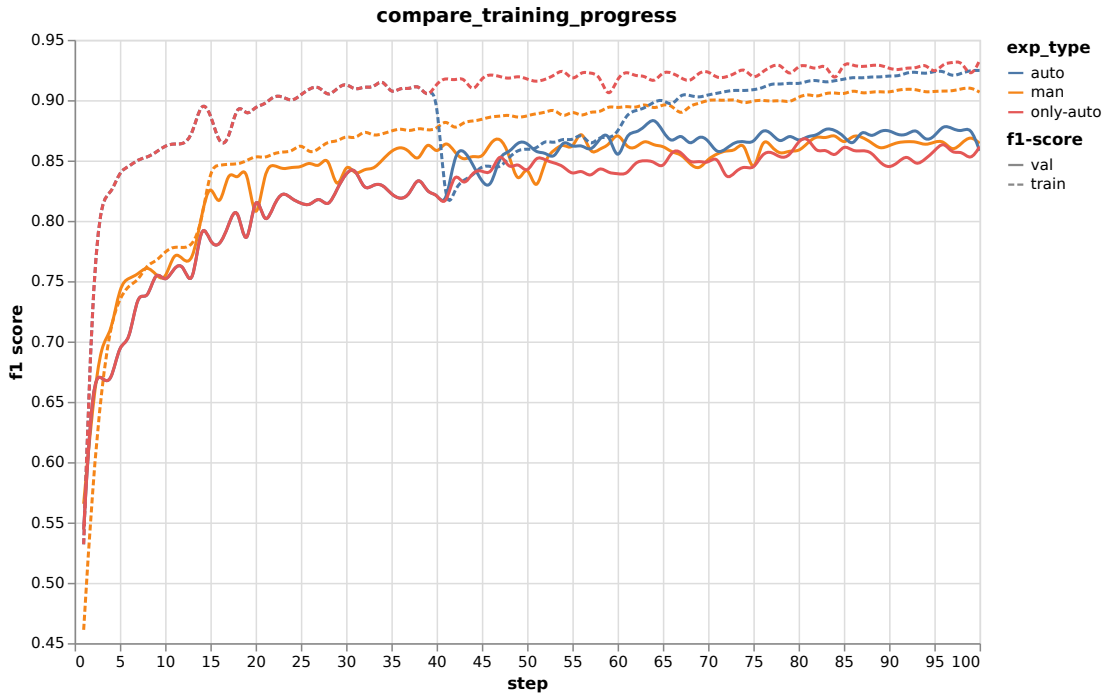


Abbildung 2.13: Vergleich der Trainings und Validierungs-F1-Score.

In den Abb. 2.14, 2.15 und 2.16 kann man den Validierungs-F1-Score der drei Experimente aufgeteilt nach den sechs Klassen sehen. Es ist zu erkennen, dass die verschiedenen Klassen unterschiedlich gut gelernt werden und das Hinzufügen der automatisch annotieren Bilder jeweils einen unterschiedlichen Effekt hat. Bei den Klassen robot, centercircle und goal gibt es eine kleine Verbesserung von *auto* gegenüber *man*, welches wiederum besser als *only_auto* ist. Bei den Klassen ball, line und penaltycross lässt sich keine Tendenz ausmachen und alle Netze schneiden ähnlich gut ab.

Beispielbilder der Netze sind in Abb. 2.17 und weitere im Anhang Abb. A.1, A.2 und A.3 zu sehen.

Die Experimente haben gezeigt, dass es bei der Hälfte der Klassen einen Vorteil hat mit den automatisch gelabelten Bilder zu trainieren. Es bietet sich somit an, die automatisch gelabelten Bilder zum Vortrainieren dieser Klassen zu benutzen. Aus diesem Grund wurden die Bilder zum Datensatz hinzugefügt. Damit kommen weitere 8.821 Bilder mit 61.224 Annotationen aus 10 verschiedenen Spielen zum Datensatz hinzu. In Zukunft können die Netze zur automatischen Segmentierung weiter verbessert werden, um die Qualität der automatisch annotierten Bilder zu verbessern.

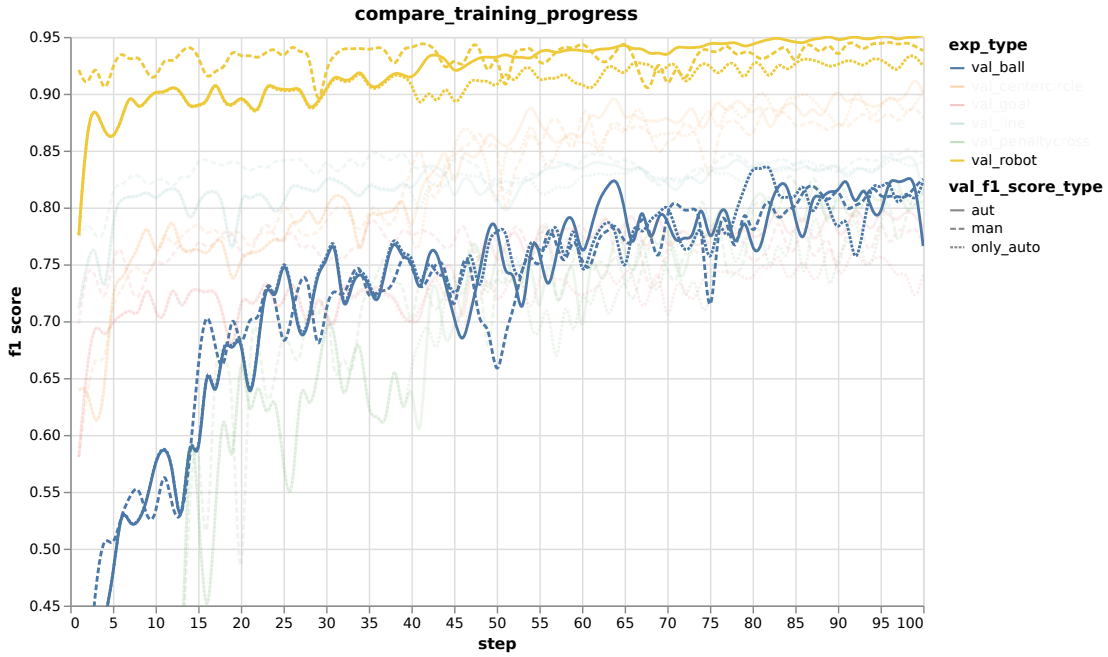


Abbildung 2.14: Vergleich des Validierungs-F1-Score von ball und robot.

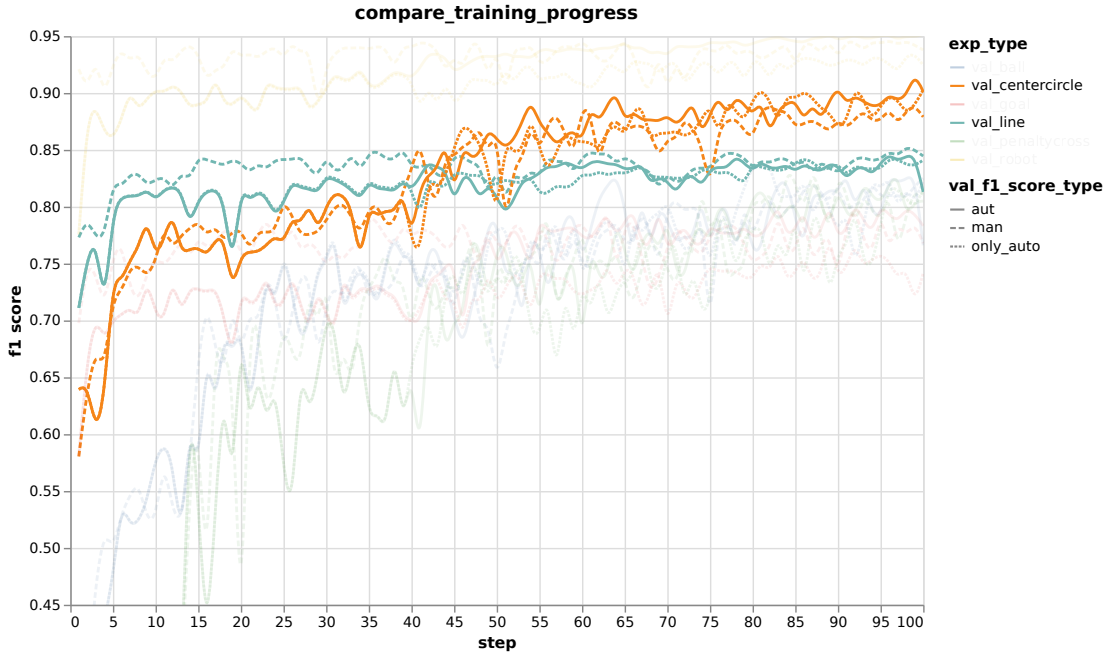


Abbildung 2.15: Vergleich des Validierungs-F1-Score von centercircle und line.

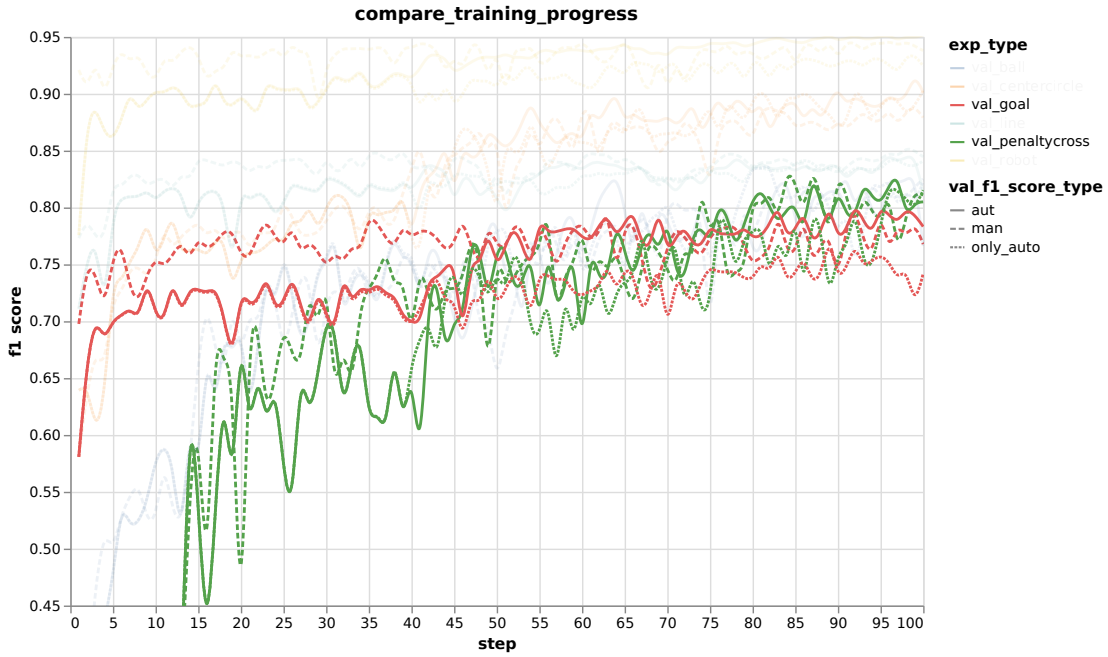


Abbildung 2.16: Vergleich des Validierungs-F1-Score von goal und penaltycross.

2.3.3 Verbesserte Ballerkennung mit Visual Mesh

Visual Mesh[HC18] ist ein Verfahren, bei dem aus einer Bildeingabe ein Pixelbild in Form eines Netzes des Bildes generiert wird, welches dann zum Trainieren eines CNN benutzt wird. Abb. 2.18 zeigt ein Bild von einem Spiel, mit den aus den Kamerakoordinaten berechneten Mesh, nur die Bildpixel die von dem in weiß eingezeichneten Netz überlagert werden, werden als Eingabe für das CNN verwendet. Die Objekterkennung erfolgt anhand der Struktur, die das Objekt in dem Netz hat. Das Netz wird über die Kameramatrix, Art und Ausrichtung des Bildes erzeugt. Eine Eigenschaft des Netzes ist, dass es nach hinten engmaschiger wird, sodass kleine Objekte dieselbe Anzahl an Punkten im Netz benutzen wie nahe, große Objekte. So besitzen alle Objekte vom selben Typ, die selbe gelernte Struktur. Da nur die Punkte vom Netz und nicht das gesamte Bild benutzt wird, wird die Eingabe in das CNN von vornherein verkleinert.

2.3.3.1 Auswertung

Der Datensatz zum Trainieren mit VisualMesh enthält alle manuell und automatisch gelabelten Bilder mit Bällen. Für die Erstellung des Datensatzes werden die Bilder, die Masken und die Kamerainformationen benötigt. Dies muss alles in das richtige Format gebracht werden und anschließend daraus Datein, im Format tfrecord, zum Trainieren und Testen erzeugt werden. Es wurden 60% der Daten zum Trainieren und jeweils 20% zum Validieren und Testen des Netzes benutzt.

Die hier abgebildeten Bilder sind das Ergebniss eines Netzes mit einer Ball-Precision

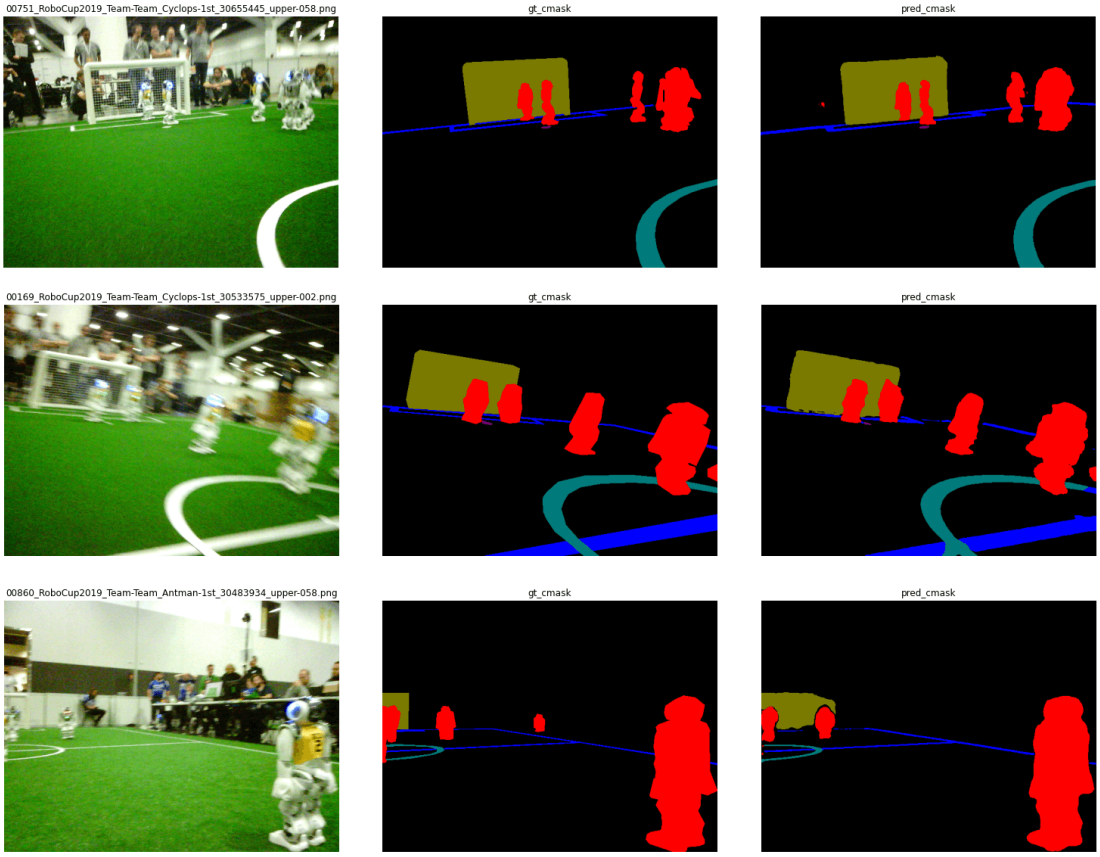


Abbildung 2.17: Vorhergesagte Bilder aus dem Validierungsdaten. Von oben nach unter: *auto*, *man*, *only_auto*.

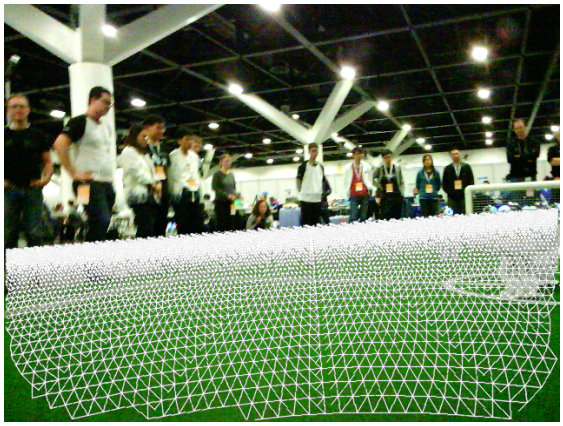


Abbildung 2.18: Beispielbild eines Mesh von einem Roboterbild

von 0,87 und einem Ball-Recall von 0,75. Interessante Bereiche werden in den Trainingsergebnissen farbig markiert. Schwarz steht dabei für Umgebung und weiß für Bälle. Auch wie sicher sich das Netz in der Erkennung der Objekte ist wird in drei Stufen angezeigt, volle Sättigung ist 90% sicher, 75% ist der nächste Sättigungsgrad und die schwächste Sättigung steht für 50% Sicherheit.

Objekte wie Roboterfüße, werden als interessanter Bereich erkannt, und richtiger Weise als Umgebung eingestuft. In Abb. 2.19 wird ein naher Ball erkannt. Dabei ist zu sehen, dass die äußere Umgebung des Balles zunächst als Umgebung erkannt wird (äußere schwarze Umrandung), welche immer unsicherer erkannt wird (Umrandung in schwarz mit geringerer Sättigung), je näher es an den Ball herangeht. Der Ball wird zunächst unsicher (schwache grau/weiße Umrandung direkt um den Ball) und später sicher (innerste weiße Umrandung im Ball) erkannt. Abb. 2.20 zeigt, dass nach dem selben Schema auch weit entfernte Bälle erkannt werden.

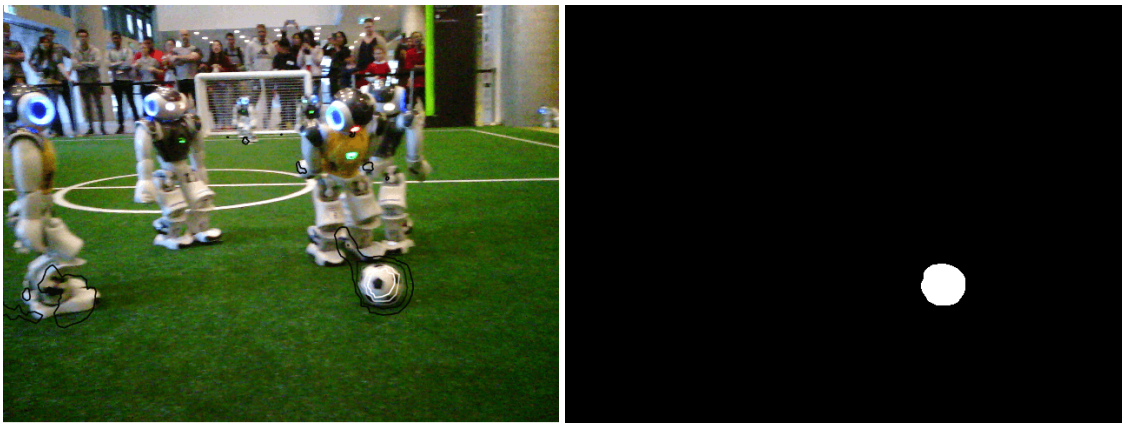


Abbildung 2.19: Beispiel naher Ball, links: Ergebniss trainiertes CNN, rechts: Ground Truth Daten.

Das Ausführen von VisualMesh auf dem Roboter mit der vorhandenen CPU Engine kam zu folgenden Ergebnissen. Die Laufzeit und die Resultate sind stark von den gewählten Mesh Einstellungen abhängig. Der Radius des im Roboterfußball benutzten Balles sind 0.05m. Bei diesem Radius benötigt das VisualMesh pro Durchlauf ca. 700 ms und kann auf ca. 7,5m Entfernung noch gut erkennen, auch wenn der Ball in der Ecke neben einem Roboter liegt. Wenn der Radius auf 0.093m eingestellt wird, verkürzt sich die Laufzeit auf ca. 300ms, jedoch wird der Ball nur noch auf eine Entfernung von 5m sauber erkannt. VisualMesh hat auch GPU unterstütze Engines eingebaut, die weniger Laufzeit benötigen sollen. Jedoch wird dafür OpenCL benötigt. Diese Engines zu testen steht noch aus. Als Ergebniss lässt sich jedoch festhalten, der Ansatz erzielt sehr gute Ergebnisse, auch für weit entfernte Bälle, allerdings ist die Laufzeit der CPU Engine deutlich zu hoch.

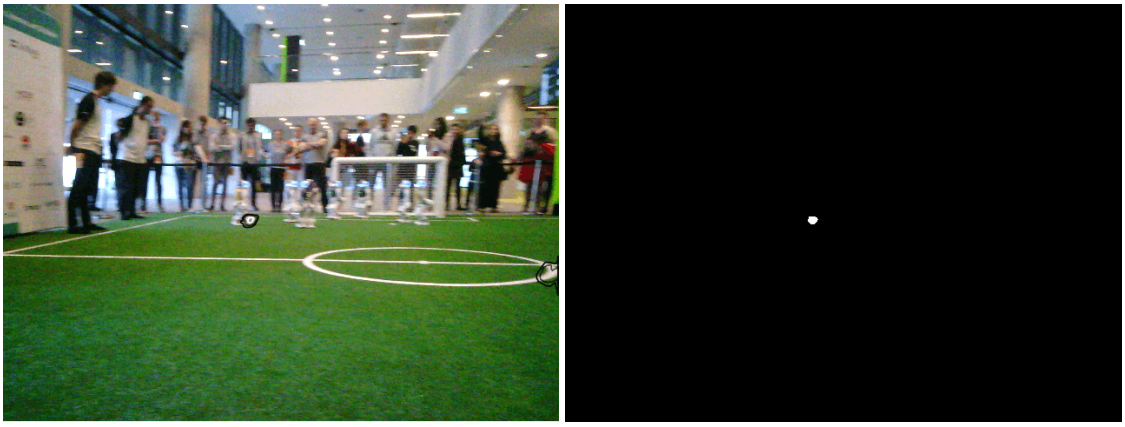


Abbildung 2.20: Beispiel ferner Ball, links: Ergebniss trainiertes CNN, rechts: Ground Truth Daten.

2.3.4 Linienerkennung durch LaneNet

Die aktuelle Linienerkennung des NAO-Roboters basiert auf einem Scan-Linien-Ansatz [Sch+16]. Diese Scan-Linien werden innerhalb einer Vorverarbeitung horizontal und vertikal aus dem Kamerabild bezogen und anschließend in Segmente aufgeteilt. Diese Segmentente dienen unter anderem zur Erkennung von Bällen oder Linien. Als Alternative wurde LaneNet von Neven u. a. für die Linienerkennung im Detail betrachtet. LaneNet realisiert eine Instanzsegmentierung für Fahrbahnmarkierungen aus dem Straßenverkehr[Nev+18]. Dieser Ansatz zeichnet sich, neben der semantischen Instanzsegmentierung, über eine Fortführung von verdeckten Fahrbahnmarkierungen aus. Dazu wird nach einer gemeinsamen Encoder-Einheit mit zwei getrennten Decoder-Einheiten gearbeitet. Durch eine Migration des LaneNet-Ansatzes auf den NAOs sollen Feldlinien durch das CNN getrennt voneinander erkannt werden, so dass eine nachträgliche Unterscheidung per Nachverarbeitung nicht mehr notwendig ist. Zusätzlich können durch die erkannten Linientypen direkt Informationen für die Lokalisierung des NAOs auf dem Spielfeld gewonnen werden.

Eine Evaluation erfolgte mit einer inoffiziellen aber frei zugängliche Implementierung auf Basis von TensorFlow[MC18], sowie 400 repräsentativer Bilder mit einer Auflösung von 1280x960 Pixeln samt Bitmap-Maske. Die Pixel-Auflösung stellt die maximal mögliche Kamera-Leistung des NAO bei 30 Frames per second (FPS) dar, womit die Fähigkeiten des gewählten Ansatzes unter optimalen Bedingungen evaluiert werden soll. Regelmäßige Abstürze innerhalb der Trainingsphase, sowie eine schlechte Trainierbarkeit, verhindern jedoch eine qualifizierte Bewertung. Die genauen Gründe dafür sind derzeit unbekannt, da aufgrund der Komplexität des Ansatzes und einer ungewissen Realisierung auf den beschränkten Ressourcen eines NAO-Roboters der LaneNet-Ansatz einvernehmlich verworfen wurde. Stattdessen soll ein selbst entwickeltes Netz mit möglichst großer Bildauflösung zur Linienerkennung eingesetzt werden. Dieser Ansatz wird in Abschnitt 2.3.5 vorgestellt.

2.3.5 Linienerkennung bei möglichst hoher Bildauflösung

Ziel dieses Ansatzes ist die Erstellung eines CNN zur Linienerkennung, welches mit einer möglichst hohen Bildauflösung arbeitet. Derzeit arbeitet ein CNN auf einem NAO-Roboter mit einer stark reduzierten Auflösung (zum Beispiel 160x120 Pixel) um die Laufzeitschranke von 30 FPS einhalten zu können. Durch die Verkleinerung der Bilder können jedoch weit entfernte Objekte im Bild stark verkleinert werden, mit anderen Objekten verschmelzen oder gar verloren gehen, so dass sie von einem CNN nicht mehr erkannt werden. Vor allem weit entfernte Feldlinien oder Bälle sind von diesem Problem betroffen.

Da eine einzelne 2D Separable Convolution mit beispielhaft acht 3x3 Kerneln bei der Verarbeitung der maximalen Auflösung eine Ausführungszeit von gerundet 37 ms besitzt, müssen Möglichkeiten genutzt werden um die Bildgröße unter möglichst geringem Verlust von Detailinformationen zu reduzieren. Hierzu wurden zwei Ansätze verfolgt: Dilation-basierende CNNs bei maximaler Bildauflösung sowie vorab skalierte Integralbilder[VJ]. Letztere sollen sicherstellen, dass weit entfernte Linien, durch ihren hohen RGB-Wert, beim skalieren der Bilder nicht verloren gehen. Nebenläufig wurden zusätzlich weitere Bild-Modifikationen durchgeführt und überprüft, ob diese einen positiven Effekt auf die Lern- und Inferenzfähigkeit des Netzes haben:

- Umstellung auf Grau-Bilder
- Entfernung des grünen Farbkanals
- Schwärzung von grünen Pixeln[6]
- Anwendung von Sobel-Operatoren[Sob14]

Durch diese Modifikationen sollen störende bzw. nicht relevante Informationen per Vorverarbeitung bereits aus dem Bild zu entfernt werden. Eine Modifikation der Bilder zeigt jedoch keinen nennenswerten Vorteil gegenüber reinen Integralbildern. Stattdessen erzeugen einige Modifikationen kontraproduktive Effekte, so beispielhaft die Schwärzung der grünen Pixel in einem Bild um eine bessere Abhebung zwischen Spielfeld und Feldlinien zu ermöglichen. Zwar wird eine weit entfernte Spielfeldlinie aufgrund des verstärkten Kontrastes besser erkannt, jedoch muss der Schwellenwert zur Einordnung als grünes Pixel je nach Untergrund, Helligkeit oder Einfallswinkel des Lichtes immer wieder neu kalibriert werden. Aus diesem Grund werden als Trainingsdaten entweder nur Bilder in originaler Auflösung oder als Integralbild mit 320x240 Pixel oder 160x120 Pixel verwendet. Im Datensatz stehen 1.179 Bilder aus dem COCO Annotator zur Verfügung (siehe Tabelle 2.7 auf Seite 22), welche durch eine horizontale Spiegelung zusätzlich augmentiert wurden. Eine Aufteilung erfolgt in 70 % Trainings-, 20 % Validierungs- sowie 10 % Testdaten und ist für alle Netzstrukturen identisch, so dass ein direkter Vergleich der Netzstrukturen möglich ist.

[6] Wenn Grünwert des Pixels Maximum darstellt und eine ausreichend große Distanz zum Rot- **und** Blauwert aufweist (z.B. $grün - rot > 40$)

Die in Anhang B gezeigten drei Netze stellen hierbei für die genannten Bildgrößen die jeweils besten Kandidaten dar und sind in einen teilautomatischen Prozess entstanden. Dazu sind unterschiedliche Kombinationen und Konfigurationen von Layern automatisch kombinatorisch aufgebaut, trainiert und getestet worden. Die Netzstrukturen sind dabei bewusst einfach gehalten um einen simplen Aufbau gegenüber komplizierten Strukturen zu prüfen (vgl. U-Net-Adaption in Abschnitt 2.3.6). Die Qualität eines Netzes ist auf Basis von Loss und Accuracy der Validierungsdaten berechnet und durch die Division von Accuracy durch Loss als vergleichbare Kennzahl verwendet. Early Stopping[GBC16] wird als Schutz vor Overfitting eingesetzt.

Durch objektive Betrachtung dieser Werte, sowie subjektive Analyse der Inferenzen der trainierten Netze, lässt kein Favorit ermitteln (vgl. Tabelle 2.8 sowie Anhang A auf Seite 76). NetCand01 fällt negativ durch eine erhöhte Anzahl an Störungen in den Bildern auf, vor allem im direkten Vergleich zu NetCand02 (vgl. Abb. 2.21). Hier wird vermutet, dass durch die Reduktion der Auflösung potenziell störende Bildelemente, wie zum Beispiel Rasenstrukturen, homogenisiert werden und somit für ein CNN einfacher auszuschließen sind. Insgesamt zeigt sich jedoch auch, dass je kleiner die Bildgröße des Inputs als auch die Arbeitsgröße innerhalb des Netzes, desto schlechter bzw. unsicherer werden weiter entfernte Linien erkannt. Die Kandidaten NetCand01 und NetCand02 überschreiten die Laufzeit bis um das Doppelt. Eine Reduktion der Netztiefe, zur Verringerung der Laufzeit, führt jedoch zu drastisch schlechteren Ergebnissen, wodurch diese beiden Kandidaten im Vergleich zu einer Netzstruktur mit Bilddaten von 160x120 Pixeln letztendlich unterliegen.

(a) Originalbild (b) Ergebnis NetCand01 (c) Ergebnis NetCand02

Abbildung 2.21: Inferenz-Vergleich zwischen NetCand01 und NetCand02 mit reduzierter Störung durch Reduktion von Bildauflösung.

Insgesamt muss festgestellt werden, dass die getesteten Netzstrukturen der Adaption des U-Net aus Abschnitt 2.3.6 in der Qualität und trotz höherer Auflösung nicht überlegen sind. Der U-Net-Ansatz erkennt auf einer geringeren Auflösung und vergleichbarer Laufzeit zum einen Linien ähnlich gut, zusätzlich jedoch noch weitere Objekte wie Roboter, Bälle oder Tore. Aufgrund der Ressourcenbeschränkung des NAO wird eine Adaption des U-Net im direkten Vergleich favorisiert, wodurch nur ein CNN genutzt wird, welches zusätzlich weitere Objektklassen erkennt.

Name	Laufzeit	Auflösung	Training		Validation		
			Accuracy	Loss	Accuracy	Loss	Kennzahl
NC01	60 ms	640x480	0,979401	0,062504	0,980879	0,058190	16,856532
NC02	53 ms	320x240	0,981583	0,052909	0,982677	0,050080	19,622294
NC03	23 ms	160x120	0,979757	0,057468	0,980829	0,053254	18,417879

Tabelle 2.8: Vergleich von unterschiedlichen Netz-Kandidaten für Linienerkennung, auf sechs Nachkommastellen gerundet, Name gekürzt.

2.3.6 Objekterkennung mittels U-Net

Um alle möglichen Objekte auf dem Feld in Echtzeit zu erkennen wurde eine Architektur gewählt, welche auf dem U-Net-Ansatz basiert[RFB15]. Zunächst wurde ein mit etwa einer Millionen Parametern großes Modell trainiert, folgend Modell 1, welches anschließend verkleinert wurde zur Einhaltung der Laufzeitbeschränkung eines NAO-Roboters. Ein U-Net ist ein Faltungsnetz, bestehend aus Down- und Upsampling-Operationen, sowie der Konkatenation von Features. Da beim Downsampling nach und nach Informationen verloren gehen, werden in den meisten Upsampling Schichten zusätzliche Features vorheriger Schichten passender Größe verwendet. Durch das Hinzugeben der Features früherer Schichten können Merkmale verwendet werden, welche in späteren Schichten verloren gegangen sind.

Zunächst wurde mit dem Datensatz trainiert, welcher mit Hilfe der Software-Lösung UERoboCup erzeugt wurde (siehe Abschnitt 2.3.1). Die Bilder sind mit 3D-Modellen synthetisch erzeugt worden und jedem Pixel wurde ein Klassenlabel zugeordnet. Zu den Objektklassen, die vom Netz gelernt werden sollen, gehören Background, Line, Ball, Robots, Centercircle, Goal und Penaltycross. Als Kostenfunktion wird die Binary Cross Entropy genutzt. Für das Training wurden insgesamt 5400 Bilder verwendet, mit einem 80:20 Train-Test-Split. Sie wurden auf die Größe 160x120 Pixel skaliert und als Augmentierung mit einer Wahrscheinlichkeit von 0,5 vertikal gespiegelt. Die Inferenzzeit ist mit etwa 1,2 Sekunden auf dem NAO noch deutlich zu lang. Die Netzstruktur von Modell 1 ist in Tabelle B.3 zu finden.

Zur Evaluation wurde jeweils pro Klasse Precision und Recall berechnet, sowie der daraus resultierende F1 Score. Bereits nach 10 Epochen Training liefert das Modell 1 F1 Scores von mindestens 0,76 in allen Klassen (siehe Tabelle 2.9). Test Predictions dieses Modells sind zu sehen in Abb. A.4. Zu erkennen sind Verwechslungen in den Klasse Centercircle und Line, als auch in Penaltycross und Ball. Angewendet auf realistische Bilder der Kamera des NAO-Roboters konnte das trainierte Netz keine Ergebnisse liefern. Es wird vermutet, dass dies am komplexeren Aufbau dieser Bilder liegt. Diese zeichnen sich durch wechselnde Hintergründe, unterschiedliche Lichtverhältnisse, Unschärfe und Rauschen gegenüber synthetisch erzeugten Bildern aus.

Nachdem der Datensatz mit etwa 1.090 manuell gelabelten Bildern zur Verfügung stand (siehe Abschnitt 2.3.2) wurde das Modell erneut trainiert. Es liefert für fast alle Klassen schlechtere Ergebnisse (siehe Tabelle 2.10). Dies liegt vermutlich an dem teilweise sehr

Klasse	Recall	Precision	F1 Score
Background	0,99	0,99	0,99
Line	0,83	0,77	0,80
Ball	0,88	0,82	0,85
Robots	0,98	0,93	0,95
Centercircle	0,68	0,86	0,76
Goal	0,74	0,97	0,84
Penaltycross	0,78	0,86	0,82

Tabelle 2.9: Validierungsmetriken von U-Net-Modell 1, trainiert auf synthetisch hergestellten Daten durch UERoboCup.

starken Rauschen und der geringeren Menge an Trainingsdaten. Das Penaltycross wird nicht mehr erkannt, aber auch in den Klassen Ball, Centercircle und Line gibt es einen Rückgang vom F1 Score von mindestens 0,2. Testbilder zeigen, dass auf realistischen Daten diese Klassen oftmals vom Netz verwechselt werden (siehe Abb. A.5). Außerdem sind die Masken teilweise lückenhaft. Das Problem scheint häufig aufzutreten, wenn andere Roboter sehr nahe zu sehen sind. Dies deutet auf einen Mangel von solchen Szenarien im Datensatz hin.

Klasse	Recall	Precision	F1 Score
Background	0,99	0,99	0,99
Line	0,46	0,82	0,59
Ball	0,49	0,75	0,59
Robots	0,88	0,92	0,90
Centercircle	0,83	0,36	0,50
Goal	0,84	0,88	0,86
Penaltycross	0,00	0,00	0,00

Tabelle 2.10: Metriken von U-Net-Modell 1 trainiert auf realistischen Daten.

Um schlechte Train-Test-Splits vorzubeugen, wurde des Weiteren mit Cross-Validation trainiert für jeweils 100 Epochen mit $k = 4$. Folgende Ansätze zur Reduktion der Laufzeit und Verbesserung der Erkennungsqualität wurden betrachtet. Unter anderem wurde die Anzahl der verwendeten Filter stark reduziert und einige Schichten entfernt. Dabei wurden mal mehr und mal weniger Down- und Upsampling Operationen verwendet. Bei tieferen Netzen musste die Anzahl der Filter stärker reduziert werden, als bei weniger tiefen Netzen, um die Laufzeitbeschränkung nicht zu übersteigen. Diese Netze konnten deutlich bessere F1 Scores vorweisen, besonders in der Klasse Ball, als weniger tiefe Netze. Es konnte kein Netz gefunden werden, welches mit nur zwei Downsampling Schichten einen F1 Score von über 0,05 in der Klasse Ball erreicht. Leaky ReLUs haben sich als Aktivierungsfunktion der Hidden-Layer als vorteilhaft erwiesen und konnten die F1 Scores

verbessern. Bisher lag die Größe des Inputs bei 160x120 Pixel, welche sich nur drei mal ganzzahlig halbieren lässt. Deshalb wurden einige Alternativen getestet, wie zum Beispiel 128x96 Pixel.

Das folgende Modell 2 besteht aus vier Down- und drei Upsampling Schichten (zu finden in Tabelle B.4) und verwendet einen Input der Größe 128x96. Die Laufzeit des Modell 2 beträgt auf dem NAO V6 durchschnittlich 24 ms. Die F1 Scores sind in allen Klassen zurückgegangen (siehe Tabelle 2.11). Besonders die Ballerkennung hat unter dem Kapazitätsverlust zu leiden, aber auch die Erkennung des Centercircles.

Klasse	Recall	Precision	F1 Score
Background	0,91	0,97	0,94
Line	0,44	0,46	0,45
Ball	0,10	0,35	0,15
Robots	0,83	0,72	0,77
Centercircle	0,19	0,35	0,25
Goal	0,43	0,57	0,49
Penaltycross	0,00	0,00	0,00

Tabelle 2.11: Metriken von U-Net-Modell 2 trainiert auf realistischen Daten.

Da der manuell gelabelte Datensatz zu dem Zeitpunkt nur maximal in 20 % der Bilder Bälle enthält, könnte dieser Mangel zu dem schlechten F1 Score der Klasse Ball von 0,15 beitragen. Es konnten bereits große Netze verwendet werden, um Bälle in Bildern automatisiert zu labeln (siehe Abschnitt 2.3.2.3). Zu diesem Zeitpunkt konnten allerdings die restlichen Klassen noch nicht zufriedenstellend automatisiert gelabelt werden. Daher wurde eine spezielle Augmentierungsmethode entwickelt, welche nur die automatisch gelabelten Bälle verwendet. Etwa 3.000 solcher Bälle wurden mit Hilfe der generieten Maske aus dem Bild geschnitten. Pro Trainingsbild wurde anschließend mindestens ein Ball und maximal vier Bälle zufällig hinzugefügt. Dadurch wurde der Anteil der Ballpixel künstlich im Trainingsdatensatz erhöht. Die Bälle wurden mit Skalierungen, Rotationen und Rauschen augmentiert (siehe Abb. 2.22. Dabei wurde darauf geachtet, das besonders kleine Bälle nicht verkleinert und besonders Große nicht vergrößert wurden.

Die Ergebnisse zeigen eine starke Verbesserung der Ballerkennung, als auch in allen anderen Klassen, außer in Penaltycross und Centercircle (siehe Tabelle 2.12). Die Vergrößerung des Inputs auf 160x120 Pixel führt nur zu minimalen Verbesserungen, die Inferenzzeit verlängert sich dadurch aber um etwa 9 ms.

Die Erkennung des Centercircles und des Penaltycrosses ist sehr schlecht mit niedrigen F1 Scores von 0,25 und 0,0. Test-Predictions zeigen, dass es oftmals Verwechslungen mit der Klasse Line gibt. Um die Metriken von Line durch diese Verwechslungen nicht zu verschlechtern, können die Masken von Centercircle und Penaltycross der Klasse Line hinzugefügt werden. Dadurch erhöht sich der Recall um 0,09, die Precision um 0,27 und der F1 Score um 0,17.

Die verwendete Kostenfunktion, die Binary Cross Entropy, hat bisher alle Klassen gleich

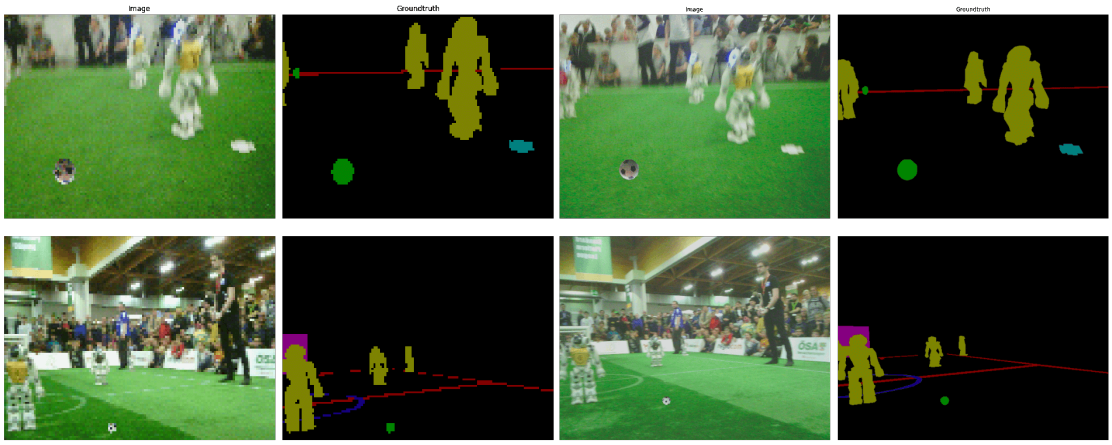


Abbildung 2.22: Trainingsbilder mit zusätzlichen automatisiert gelabelten Bällen. Links mit der Auflösung 128x96 Pixel, rechts mit 640x480 Pixel.

Klasse	Recall	Precision	F1 Score
Background	0,98	0,97	0,98
Line	0,55	0,60	0,57
Ball	0,34	0,42	0,37
Robots	0,86	0,87	0,87
Centercircle	0,19	0,43	0,25
Goal	0,57	0,70	0,63
Penaltycross	0,00	0,00	0,00
L+C+P	0,64	0,87	0,74

Tabelle 2.12: Metriken von U-Net-Modell 2 trainiert auf realistischen Daten mit zusätzlichen Bällen. Die Klasse L+C+P kombiniert die Klassen Line, Centercircle und Penaltycross.

gewichtet. Ein erhöhtes Ballgewicht konnte auch hier die Ballerkennung von Model 2 leicht verbessern, von einem F1 Score von 0,37 auf 0,43, bei maximaler Schwankung von 0,01 im F1 Score der anderen Klassen.

Ein solches Segmentierungsnetz kann helfen, die Objekterkennung der verschiedenen Klassen zu verbessern oder gar erst zu ermöglichen. So ist beispielsweise bislang noch keine Torerkennung vorhanden, kann aber mit Hilfe eines solchen Netzes implementiert werden. Da das Modell 2 mit 24 ms auf dem NAO V6 recht schnell ist, kann die Inferenz in Echtzeit erfolgen.

2.4 Implementierungsansätze

In Abschnitt 2.3 sind verschiedene Ansätze vorgestellt worden, neuronale Netze zur Segmentierung zu nutzen, um Objekte auf dem Spielfeld zu erkennen. Die Erkennung von Feldlinien und Toren sind hierbei besonders vielversprechend.Für Feldlinien ermöglicht die Segmentierung erstmals, ein neuronales Netz für die Erkennung anzuwenden. Für Tore waren erste Ergebnisse der Segmentierung vielversprechend, siehe Abschnitt 2.3.6, zumal zuvor keine funktionierende Torerkennung implementiert war.Der folgenden Abschnitt befasst sich mit der Implementierung der Linienerkennung und Torerkennung mit Hilfe von Neuronalen Netzen und es wird evaluiert, wie sich diese Ansätze auf die Erkennung auswirken und wie gut diese für die Anwendung geeignet sind.

2.4.1 Linienerkennung

Der Ansatz der Segmentierung wird für verschiedene Erkennungsaufgaben in Erwägung gezogen. Für die Linienerkennung ist ein Segmentierungsansatz sinnvoll, weil Linien sich sehr schlecht durch Bounding Boxes beschreiben lassen.
Um diesen Ansatz zu testen wurde ein neuronales Netz in die Linienerkennung implementiert und Auswirkungen auf die Linienerkennung evaluiert.

Die bisherige Linienerkennung der NaoDevils funktioniert in Kürze folgendermaßen: Linienpunkte werden gefunden, indem das Bild mit Scan-Linien abgetastet wird. Gefundene Linienpunkte werden zu Segmenten verbunden und Zusammenpassende Liniensegmente werden zu Linien zusammengeführt.

Die Linienerkennung mit einem NN unterscheidet sich dadurch, dass das Finden von Linienpunkten nicht mehr verschiedene Scan-Linien verwendet werden, sondern ein CNN, das eine Segmentierungsmaske ausgibt, mit deren Hilfe Linienpunkte ermittelt werden.

Das CNN arbeitet auf dem auf 160×120 Pixel reduzierten Kamerabild und gibt eine Maske in Form eines Arrays mit Fließkommazahlen aus, die ab einer bestimmten Akzeptanzschwelle für die Erkennung verwendet werden, siehe Abb. 2.23.

Durch zeilenweises und spaltenweises Iterieren über die Ausgabemaske werden Linienpunkte von Feldlinien als Mittelpunkte zusammenhängender Pixel bestimmt.

Die so gefundenen Punkte bilden häufig Stufen, die sich untereinander schwer zu Feldlinien verlängern lassen. Daher werden die aus der Maske erkannten Linienpunkte durch Abgleich mit den Bilddaten optimiert und falsch erkannte Punkte entfernt,(siehe Abb. 2.24). Die so erkannten Punkte werden an die weitere Linienerkennung übergeben, die unverändert abläuft.

Um ein Gütemaß für die Segmentierung zu erhalten, werden die originale Linienerkennung und die Linienerkennung mit dem CNN auf einem Datensatz von 139 zufällig ausgewählten Bildern miteinander verglichen, siehe Tabelle 2.13.
Hierbei zählt die teilweise Erkennung erkennbarer Linienabschnitte, wobei jedes Stück Linie zwischen zwei Ecken oder Kreuzungen als einzelner Abschnitt zählt.
Insgesamt ist mit dem CNN ein deutlich höherer Recall zu bemerken, wobei die Precisi-

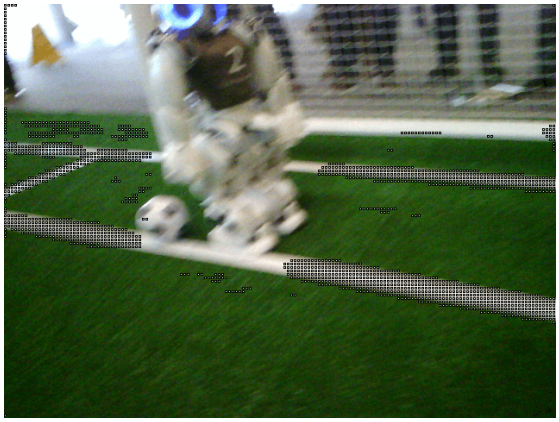


Abbildung 2.23: Segmentierungsmaske auf das Bild übertragen.

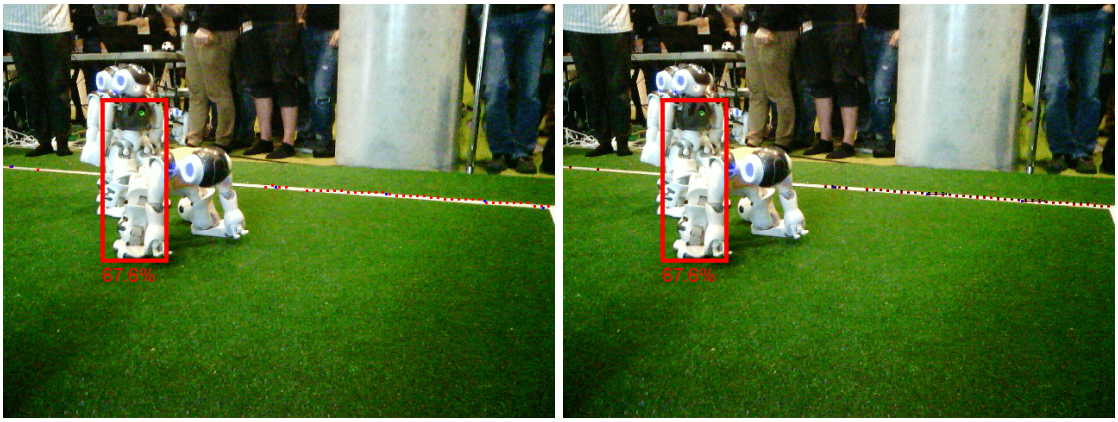


Abbildung 2.24: Linienpunkte aus der Segmentierung (links) und verbessert (rechts).

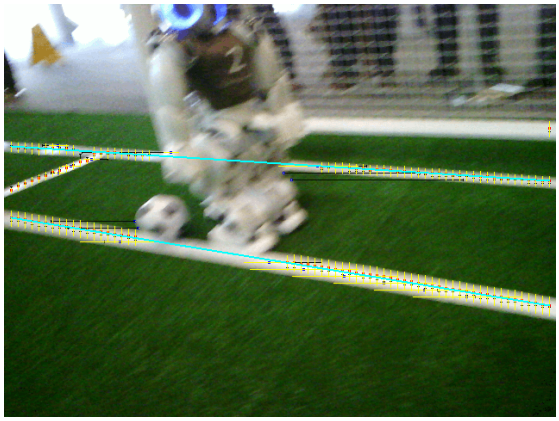


Abbildung 2.25: Gefundene Linien.

on gleich bleibt, siehe Tabelle 2.13. Die Laufzeit des neuronalen Netzes allein beträgt allerdings 41 ms auf einem NAO V6, was die Zielsetzung von 30 ms für die gesamte Bildverarbeitung deutlich überschreitet.

	Recall	Precision	F1 Score
CLIPPreprocessor	0,2941	0,9721	0,4516
CNN	0,4078	0,9791	0,5758

Tabelle 2.13: Auswertung der Linienerkennung mit Scan-Linien und mit neuronalem Netz im Vergleich.

Obwohl die gemessene Präzision beider Verfahren etwa identisch ist, unterscheidet sich die Art der häufigsten False Positives zwischen den beiden. Mit Scan-Linien werden häufiger Feldrand und Tor als Linien gewertet, während es mit dem CNN häufiger zur Falscherkennung von Mittelkreisen und zu lang gezogenen Linien kommt (siehe Abb. 2.26).

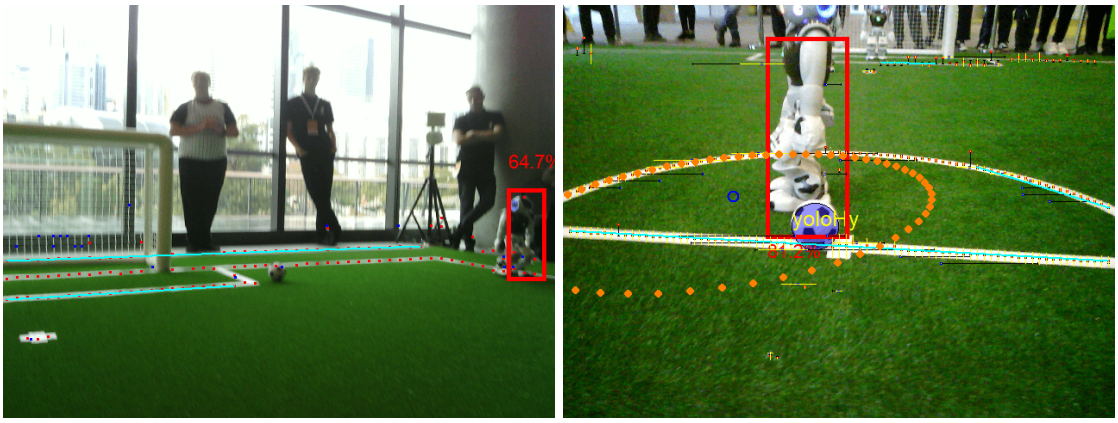


Abbildung 2.26: Typische False Positives für die Linienerkennung mit Scan-Linien (links) und mit dem CNN (rechts).

Insgesamt führt das Anwenden eines Convolutional Neural Networks zu einer Verbesserten Erkennungsrate. Die Laufzeit ist derzeit allerdings deutlich erhöht, sodass die Praxistauglichkeit in Frage steht.

2.4.2 Torerkennung

Ähnlich wie bei der Linienerkennung soll auch bei der Torerkennung ein Ansatz mittels eines neuronalen Netzes ausprobiert werden. Im Gegensatz zur Linien- gibt es bei der Torerkennung bisher keine funktionierende Lösung. Die Erkennung von Toren ermöglicht, diese als weitere Information zur Lokalisierung der Roboter bereitzustellen. Die Mindestanforderungen an einen Ansatz mittels neuronalen Netzes sind daher gering. Ziel dieses Ansatzes ist es vorerst nicht, eine Perfekte Lösung für die Torerkennung zu entwickeln, sondern überhaupt gelegentlich Tore erkennen zu können, da das gelegentliche Erkennen

von Toren und das Bereitstellen einer zusätzlichen Markierung für die Lokalisierung in jedem Fall eine Verbesserung zur aktuellen Situation darstellt. In späteren Arbeitsschritten kann diese Lösung dann beliebig ausgebaut werden. Kurzgefasst funktioniert die Torerkennung wie folgt: Erst werden die Kanten der vom Netz segmentierten Torfläche in Bildpunkte umgewandelt. Anschließend wird – wie bei der Linienerkennung – auch bei der Torerkennung das Bild mittels Scan-Linien abgetastet und versucht anhand von Helligkeitsunterschieden der Grauwerte jedes einzelnen Bildpunktes heuristisch die Torpfosten zu ermitteln.

2.4.2.1 Funktionsweise

Die CNN-Torerkennung ist ähnlich aufgebaut wie die CNN-Linienerkennung. Das Netz macht eine Ausgabe in Form eines Arrays, das numerische Werte für die erkannten Klassen (0 = Hintergrund, 1 = Linie, 2 = Tor) enthält. Das Netz erkennt im Idealfall die von beiden Pfosten, der Latte und dem Boden umrahmte Fläche und stellt diese in der Ausgabe als nahezu viereckige Fläche abzüglich Überlappungen mit Robotern oder Ball dar. Im Rahmen der Torerkennung soll versucht werden, aus dieser Ausgabe die Position der Fußpunkte der Torpfosten eines vom Netz erkannten Tores festzustellen. Um zusätzliche Iterationen über dasselbe Array zu vermeiden, geschieht dieser Erkennungsprozess im selben Schritt wie die Feststellung der Linienpunkte (siehe Abschnitt 2.4.1). Es wurde eine neue Repräsentation GoalEdgePointsPercept (GEPP) eingeführt, die als Bindeglied dient, um die Funktionalität der Torerkennung in ein eigenes Modul auslagern zu können. Das GEPP enthält alle Punkte der Außenkanten der vom Netz erkannten Torfläche und hält zu jedem Punkt neben der Position im Bild noch den Grauwert und die Information, ob der Punkt am linken oder rechten Rand der segmentierten Torfläche liegt.

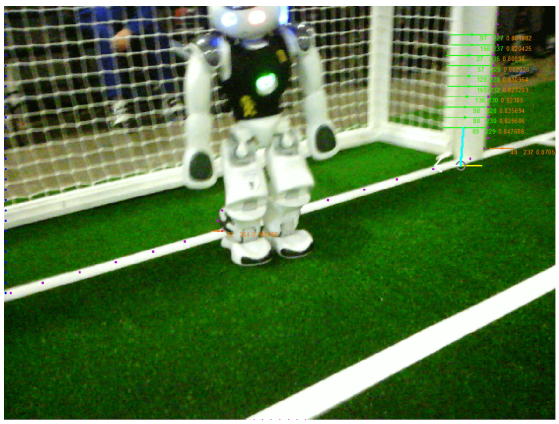


Abbildung 2.27: Erkannter Torpfosten mit markiertem Fußpunkt. Der Fußpunkt ist durch einen Kreis markiert. Der Pfeil zeigt in Richtung des Torinneren

Der erste Schritt in der Torerkennung ist es, die Außenkanten der Torpfosten jeder Zeile der Segmentierungsmaske festzustellen. Aufgrund der Form und Lage der segmentierten Torfläche fallen die Kantenpunkte je nach Perspektive auf das Tor unterschiedlich aus.

So kann es dazu kommen, dass manche der erkannten Kantenpunkte entlang der Torlinie liegen und daher vorerst irrelevant für die Feststellung der Fußpunkte der Torpfosten sind. Weiter werden Torkanten auch am Bildrand festgestellt, wenn die Segmentierungsmaske angibt, dass sich das Tor bis an den Bildrand erstreckt. Da der Torpfosten in diesem Fall höchstwahrscheinlich außerhalb des Bildes liegt, werden auch Punkte am Bildrand als irrelevant erachtet. Die festgestellten Punkte werden mittels GEPP in das neue Modul GoalPerceptor übertragen, welches die restliche Torerkennungslogik enthält. Die oben genannten irrelevanten Punkte werden entfernt.

Da die gleiche Segmentierungsmaske für die Linien- und die Torerkennung verwendet wird, entsteht auch hier das Problem, dass die Maske die Kanten aufgrund ihrer geringeren Auflösung nicht sehr genau abbilden kann. Es ist daher auch hier notwendig, die bisher gefundenen Punkte im Nachhinein durch ein Abtasten der Grauwerte zu verfeinern beziehungsweise diese auf die Torpfostenmitte zu korrigieren, da der Fußpunkt in der Mitte des Pfostens liegen soll. Im Anschluss wird versucht, den Fußpunkt ausfindig zu machen. Dazu wird entlang der gefundenen Punkte abwärts gescannt, bis der Scan auf das Spielfeld trifft. Wurde auf diesem Weg das untere Ende des Torpfostens gefunden, werden die Koordinaten dieses Punktes im Bild in Spielfeldkoordinaten relativ zum Roboter umgerechnet. Anschließend sind alle notwendigen Informationen vorhanden, um eine Instanz eines Torpfostens zu erstellen und in das CLIPGoalPercept zu übertragen.

2.4.2.2 Fazit

Mit der vorgestellten Funktionsweise ist es grundsätzlich möglich, Torpfosten zu erkennen und somit eine weitere Quelle von Informationen für die Lokalisierung der Roboter bereitzustellen. Das gesetzte Ziel für die Verbesserung der Torerkennung ist somit erreicht. Ausgereift ist die Torerkennung damit aber noch nicht. Nach wie vor gibt es einige Situationen, in denen das Tor nicht erkannt werden kann. Bei der Linienerkennung ist die Vorgehensweise, die Grauwerte der Bildpunkte abzutasten, vielversprechend, da sich die helle Linie in den meisten Fällen deutlich vom grünen Boden absetzt. Die Torpfosten hingegen befinden sich von Spiel zu Spiel vor wechselnden Hintergründen. So kann es sein, dass hinter weißen Torpfosten eine weiße Bande das Spielfeld begrenzt oder sich Zuschauer hinter dem Tor befinden. Dies ist einer der Gründe, weshalb die Torerkennung noch nicht einwandfrei funktioniert. Ein weiterer Grund ist, dass zur Angabe der Position des Pfostens sein Fußpunkt im Sichtfeld des Roboters liegen muss. Der Torpfosten kann also erkannt, aber seine Position nicht angegeben werden. Dazu kommt es immer dann, wenn der Torpfosten von einem anderen Objekt verdeckt wird oder der Roboter zu nah am Torpfosten steht.

2.4.2.3 Ausblick

Im vorangegangen Abschnitt sind die Schwierigkeiten erläutert worden, an denen die umgesetzte Torerkennung noch scheitert. In der Zukunft könnte man aber durchaus auch Lösungen für diese Probleme finden und umsetzen. Beispielsweise wäre eine denkbare Lösung gegen die Ungenauigkeit der Segmentierungsmaske ein weiteres Netz, das nur den

Bereich des Tores in höherer Auflösung scannt. Auf diese Weise bekäme man genauere Punkte und das Tor könnte auch bei wechselnden Hintergründen gut erkannt werden.

Das Problem der verdeckten Torpfosten könnte angegangen werden, indem eine Gerade entlang der Punkte an der Torlinie gefunden wird. Wird dann die Gerade, die aus einem teilweise erkannten Torpfosten resultiert, verlängert und schneidet diese mit der Torlinie, könnte auf diese Weise der Fußpunkt des Pfostens ermittelt werden.

2.5 Orientierung im Raum mittels CNN

Auf dem NAO ist es wichtig, dass die Lokalisierung schnell ist, um in Echtzeit zu laufen. Anstatt wie in Abschnitt 2.3 Linien zu erkennen und sich anhand dieser zu orientieren wird bei der Orientierung im Raum mittels CNN versucht sich direkt anhand des Bildes zu orientieren. Es wird eine kompakte 2D Repräsentation des Bildes erlernt, welches dann als Position und Rotation der Kamera interpretiert wird. Da hierbei kein Upscaling der 2D Repräsentation des Bildes berechnet werden muss, ist die Hypothese, dass dieser Ansatz schneller ist.

2.5.1 Trainingsdaten

Die Daten zum Training des neuronalen Netzes werden über zwei Wege erzeugt. Der erste Weg sind synthetischen Masken. Dafür wird das Feld in einer 3D Engine[Mat20] nachgebaut. In dieser wird eine zufällige Orientierung der Kamera in der unteren Hälfte des Spielfeldes ausgewählt, bei der Linien, Mittelkreislinien und das Tor sichtbar sind. Diese Einschränkung wird vorgenommen, damit sich eindeutig und einfacher die Position und Rotation bestimmen lässt. Mit dieser Methode wurden 10.000 Masken mit Kamera Position und Rotation für das Training und 1.000 für die Evaluation erstellt. In Abb. 2.28 ist ein Beispiel aus dem Datensatz dargestellt.

Die zweite Möglichkeit benutzt segmentierte Bilder. Hierbei wird aus der Segmentierungsmaske die Position in einer Hälfte des Spielfeldes und die Rotation des Roboters bestimmt. Damit lässt sich ein neuronales Netz mit der Maske oder dem Originalbild als Input und der Position und Rotation als Output auf realistischen Bildern und Masken trainieren und testen. Die Orientierung der Kamera wird in zwei Schritten bestimmt. Zuerst werden Key Points auf der Maske gesucht, deren Position auf dem Feld bekannt ist. Wenn vier oder mehr Key Points gefunden wurden, kann mit ihnen die zugehörige Homographie-Matrix und mit dieser die Position und Rotation der Roboterkamera approximiert werden. Da diese durch ungenaue Key Points oft noch nicht präzise ist, wird als zweiter Schritt eine Suche in der Nähe der geschätzten Position und Rotation durchgeführt. Dabei wird die zu bewertende Orientierung in einer 3D Engine gerendert und das Resultat wird mit der Maske verglichen. Als beste Position und Rotation wird die mit dem kleinsten Tversky Loss ausgewählt. Dieser Fehler kann auch als Metrik zur Güte der gefundenen Position und Rotation benutzt werden. In Abb. 2.29 sind zwei Beispiele der Positions- und Rotationsbestimmung dargestellt.

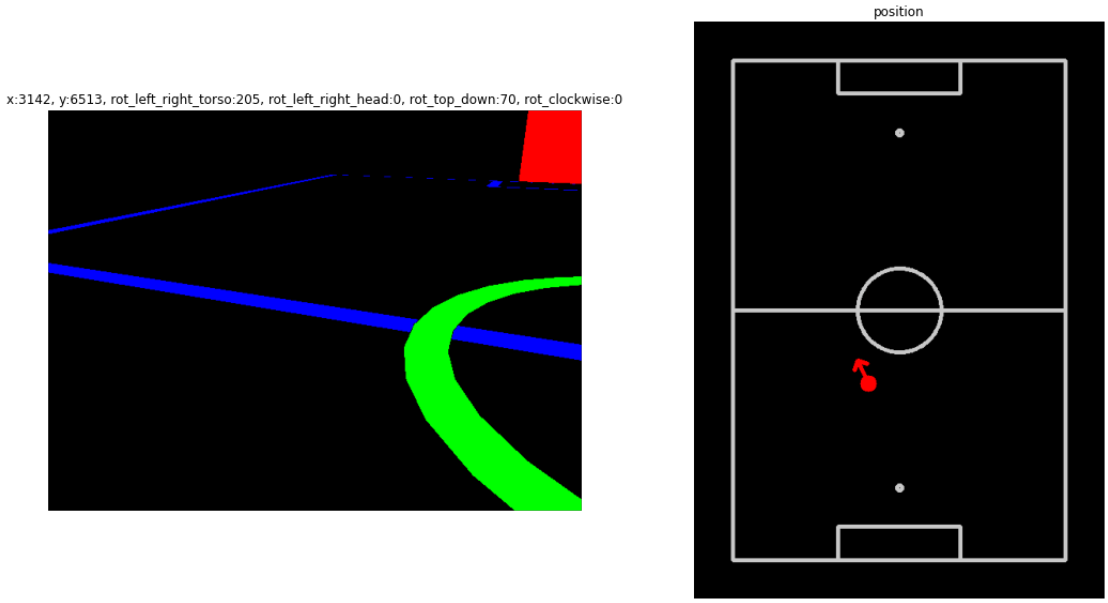


Abbildung 2.28: Beispielbild aus dem synthetischen Datensatz. Links: Maske aus der Sicht des Roboters. Rechts: Die Position und Rotation des Roboters auf dem Feld.

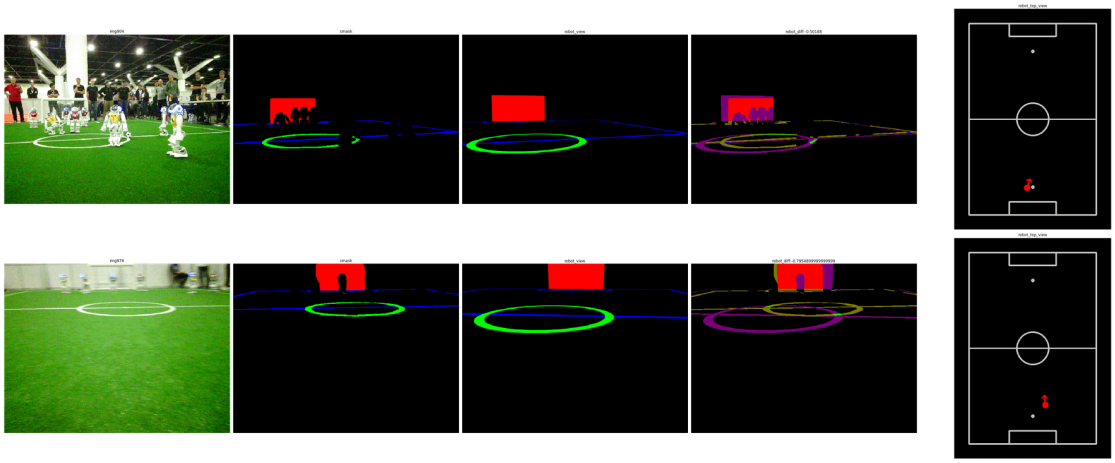


Abbildung 2.29: Zwei Beispiele aus dem Lokalisierungsdatensatz. Von links nach rechts: Kamerabild, segmentiertes Bild, Sicht aus der geschätzten Position und Rotation, Differenz der beiden Masken und die geschätzte Position und Rotation im Feld.

2.5.2 Durchführung

Das neuronale Netz besteht aus einem CNN, dass die Repräsentation lernt und einem Fully Connected Neural Network, welches die Position und Rotation als Regression lernt. Der Aufbau des Modells ist in Abb. 2.30 zu sehen. Als Optimizer wurde RMSProp mit einer Lernrate von 0,001 gewählt. Es wurde mit einer Batchsize von 16 für 8.000 Iterationen trainiert.

Zur Evaluation wurde der mittlere absolute Fehler über alle Validierungsdaten der synthetischen Masken berechnet. Es ergibt sich ein Fehler von 438 mm entlang der langen Seite des Feldes, mit einer Länge von 6.000 mm, ein Fehler von 317 mm entlang der kurzen Seite des Feldes, mit einer Länge von 4.500 mm und ein Rotationsfehler von 24 Grad. In Abb. 2.31 sind zwei Positionsschätzungen dargestellt und weitere im Anhang in Abb. A.12.

2.5.3 Fazit

Das Experiment zeigt, dass aus einer Segmentierungsmaske mit einem neuronalen Netz eine akzeptable Kameraposition geschätzt werden kann. Eine Verbindung der Schätzungen mit einem Kalman-Filter, wie es im Moment im NAO der Fall ist, würde die Positions- und Rotationsbestimmung wahrscheinlich noch bedeutend verbessern. Um zu untersuchen, ob dies auch mit dem Originalbild aus dem Spiel anstatt der Maske funktioniert, muss der zweite Datensatz präziser werden. Unter diesen Bedingungen kann sich zeigen, ob die Orientierung im Raum mittels CNN auf dem NAO schneller und genauer ist als der bisherige Ansatz.

Es gibt die Möglichkeit dem Modell zusätzliche Metainformationen des Roboters zu geben. Die Neigung des Kopfes des Roboters wäre zum Beispiel eine Information, die dem Roboter helfen könnte, seine Position und Rotation zu bestimmen. Diese Informationen ließen sich nach dem Glätten der 2D Repräsentation hinzufügen. Ein Ansatz, um nicht eindeutig Positionen zu bestimmen, ist eine Positionswahrscheinlichkeitsverteilung. Dies könnte mit einer Position-Heatmap als Ausgabe realisiert werden.

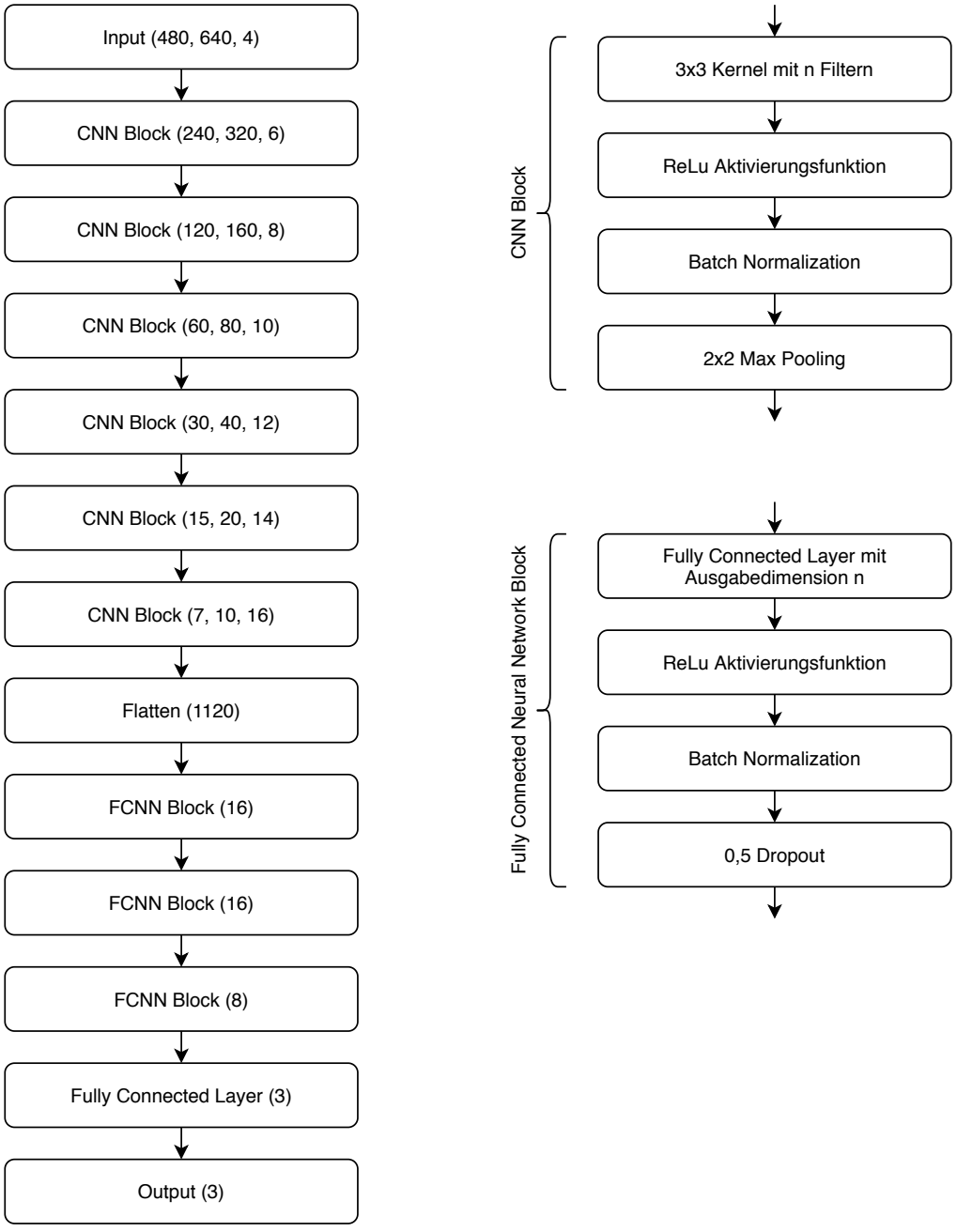


Abbildung 2.30: Aufbau des neuronale Netzes.

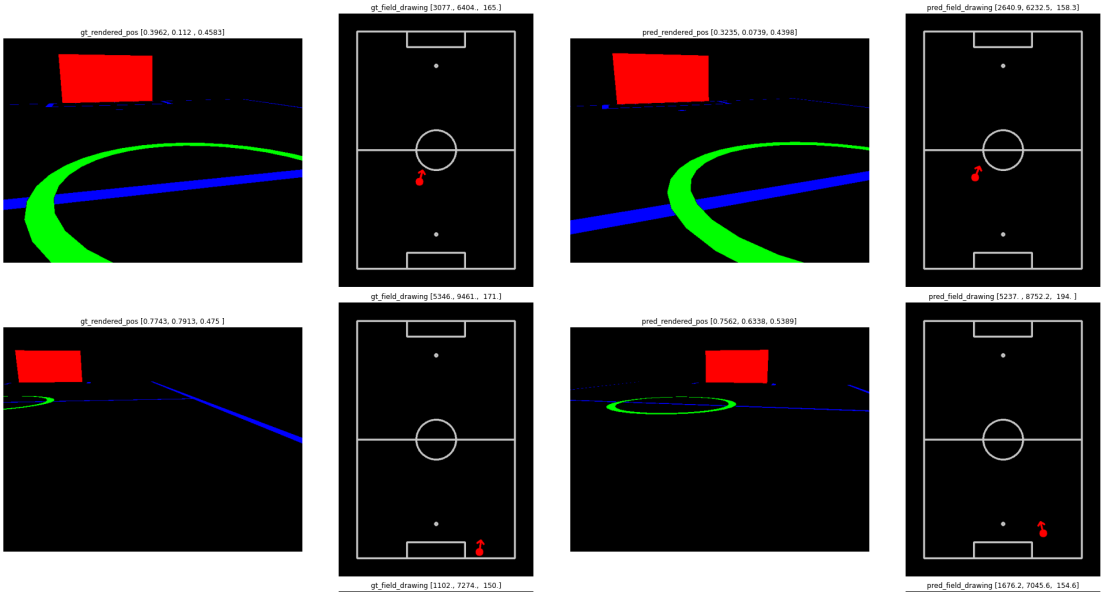


Abbildung 2.31: Zwei Positionsschätzungen des Modells aus dem Validierungsdatensatz. Von links nach rechts: die Sicht des Roboters aus der echten Position, die Position des Roboters, die Sicht des Roboters aus der geschätzten Position und die geschätzte Position.

3 Verhalten

Im Verhalten wurde sich mit zwei verschiedenen Arten von Aufgabenstellungen auseinandergesetzt. Zum einen mit Funktionen, die unabhängig von der Rolle eines Roboters sind. Dazu gehört die Suche nach Bällen, wenn keinem aus dem Team die Ballposition bekannt ist und das Anpassen der Ballverfolgung um rollende Bälle schneller erfassen zu können. Zum anderen wurde sich damit beschäftigt die Rollen des Center (Mittelfeldspieler), des Ballchaser (ballführender Spieler) und des Keeper (Torwart) zu überarbeiten. Im Folgenden werden die Ergebnisse präsentiert. Dabei ist der Aufbau des Spielfeldes den aktuellen Regeln zu entnehmen[Rul].

3.1 Balltracking

Zur Positionsbestimmung des Balles wird ein Kalman-Filter[Wil09] verwendet. Dieser eignet sich besonders dafür, Objekte mit gleichbleibender Geschwindigkeit in Echtzeit zu verfolgen. Da ein Spiel allerdings häufig einen Wechsel von liegenden und rollenden Bällen beinhaltet, wird die genaue Position des Balles, vor allem beim Übergang von einem liegenden zu einem rollenden Ball, erst zu spät bekannt. Dies kann dazu führen, dass Roboter nicht oder falsch reagieren.

3.1.1 Kalman-Filter

Bei einem Kalman-Filter handelt es sich um ein mathematisches Verfahren, welches mithilfe von bayesschen Schätzungen und dem regelmäßigem Sehen eines Objektes, seine Position vorhersagen kann.

Im Framework wird ein Multi-Hypothesen Kalman-Filter verwendet. Auf das Behandeln der Nichtlinearität wird nicht weiter eingegangen, da der verfolgte Ansatz sich vor allem auf das Vorhandensein verschiedener Hypothesen stützt. Der Kalman-Filter durchläuft in jedem Frame fünf Phasen und benutzt dafür die Ergebnisse, die er im vorherigem Frame produziert hat. Zuerst wird aus der Position des Balles im letzten Frame, und allen existierenden Hypothesen berechnet, wo sich der Ball nach Verfolgen dieser Hypothese aktuell befinden würde. Im zweiten Schritt werden die Sensordaten des Roboters genommen und die erkannte Ballposition mit den Ballpositionen der Hypothesen verglichen. Hypothesen mit ungültigen Ballpositionen fallen raus und die Wahrscheinlichkeiten für die Akzeptanz der Hypothesen wird angepasst. Wenn die Sensordaten mehrere theoretische Ballpositionen ergeben, werden dementsprechend aus diesen möglichen Positionen und den verbliebenen Hypothesen, neue Hypothesen erzeugt. Im vierten Schritt werden Hypothesen mit zu niedriger Wahrscheinlichkeit aussortiert und ähnliche zusammengefasst, um die Anzahl an Hypothesen gering zu halten. Zum Schluss wird die Hypothese

mit der höchsten Wahrscheinlichkeit gesucht, welche das neue Ballmodell stellt. In dem Framework gibt es zusätzlich auch noch Hilfsfunktionen wie z.B eine Kickerkennung, die die Wahrscheinlichkeiten der Hypothesen nochmal verändern.

Die Roboter berechnen die Ballposition lokal für sich. Allerdings wird auch noch ein Teamballmodell und ein Remoteballmodell berechnet, die unter den Robotern kommuniziert werden. Wichtig ist, dass der im nachfolgenden beschriebene Ansatz sich nur für lokal erstellte Hypothesen eignet und implementiert wurde, da die Berechnung in Echtzeit erfolgen soll und die Roboter untereinander nicht durchgehend kommunizieren dürfen.

3.1.2 Lösungsansatz

Die Idee ist, für jeden Frame den aktuellen Ballpercept mit den Vorherigen zu vergleichen und wenn sich eine Bewegung in eine bestimmbare Richtung ergibt, daraus eine Geschwindigkeit zu bestimmen. Als Position wird die vom letzten Percept genommen. Aus dieser Geschwindigkeit und Position wird eine künstliche Hypothese erstellt. Da dies direkt am Anfang des Zyklus passiert, durchläuft die künstliche Hypothese auch den Verifizierungsprozess durch Überprüfung durch die Sensordaten. Sie stellt nur das nächste Ballmodell, wenn sie auch als beste Hypothese gewertet wird. Diese künstlich errechnete Hypothese ist im Folgenden die SupportHypothese.

3.1.3 Umsetzung und Probleme

Es gab ein Problem mit dem Simulator, welcher zwar bei normalen Simulationen die Debugdrawings angezeigt hat, dies aber beim Abspielen von Logs nicht mehr tat. Dies ist durch die falsche Initialisierung der Debugdrawings in der ursprünglichen Version aufgetreten. Logs sind aufgezeichnete Daten von einem Spiel, durch das abspielen kann Verhalten in Spielsituationen simuliert werden. Debugdrawings sind für den Simulator konzipierte farbliche Unterstützungen, so wird in diesem Fall jede SupportHypothese als rosa Ball auf dem Spielfeld angezeigt. Durch die falsche Initialisierung der Debugdrawings wurde beim Abspielen der Logs die SupportHypothesen nicht angezeigt.

Der Ansatz ist, dass mindestens drei Ballpercepts innerhalb einer halben Sekunde vorhanden sein müssen, die sich in dieselbe Richtung bewegen, um daraus eine Geschwindigkeit des Balles errechnen zu können. Für die Umsetzung wurde ein Ringbuffer benutzt. Das Element, welches gerade als Anfangspunkt der Bewegung gilt, wird als first-Element abgespeichert. Das dem Ringbuffer als letztes hinzugefügte Element, welches sich allerdings dort am Anfang befindet, ist das last-Element. Zuerst wird überprüft ob der zeitliche Abstand der Erstellung des first-Element und dem last-Element nicht mehr als eine halbe Sekunde beträgt. Wenn der Abstand größer ist, kann das first-Element nicht richtig sein und das nächste Element wird betrachtet. Im zweiten Schritt wird überprüft, ob die Gerade, die vom first- und last-Element gebildet wird, das aktuell betrachtete Element enthält. Dazu wird aus den Positionen des first- und last-Elements eine Geradengleichung der Form $y = m * x + b$ aufgestellt. Zur Überprüfung, ob das aktuelle Element ungefähr auf dieser Geraden liegt, wird die X-Position in die Geradengleichung eingesetzt und überprüft ob der errechnete Y-Wert der Y-Position plus/minus einem

ε-Abstand entspricht. Der ε-Abstand kann frei gewählt werden und liegt bei 10 cm, um Messungenauigkeiten auszugleichen. Bei ersten Experimenten hat sich dieser ε-Abstand als gut herausgestellt. Diese Überprüfung kann auch mithilfe von linearer Regression gemacht werden, was vielleicht für bessere Ergebnisse sorgt. Unter der Bedingung, dass mindestens drei Elemente übrig geblieben sind, die Zeit und Raum Bedingungen erfüllen, wird die vermutliche Geschwindigkeit des Balles berechnet. Dabei ist zu beachten, dass aufgrund der Möglichkeit von verschieden großen Zeitabständen, Strecke über Zeit berechnet werden muss. Dies geht einfach aus der Differenz der Positionen und Zeitpunkte vom first- zum last-Element. Damit wird zum Schluss eine Hypothese erstellt, mit der Validität von $minValidity + \frac{(1 - minValidity)}{2}$, wobei minValidity eine frei wählbar Konstante ist, welche die minimale Validität angibt, ab wann eine Hypothese akzeptiert werden kann. Dieser Wert hat sich als gut bewiesen, da die SupportHypothesen damit nicht zu stark gewichtet werden aber auch nicht sofort wieder aussortiert werden. Wichtig ist es nachträglich die Sensorupdates auf den Wert der minimal geforderten Sensorupdates, der angibt ab wann eine Hypothese akzeptiert werden kann, zu setzen, da die neu erstellte Hypothese sonst keine Beachtung im Bewertungsprozess bekommt und nicht akzeptiert werden kann. Dies würde dazu führen, dass sich im Vergleich zum ursprünglichem Ansatz im nächsten Frame nichts verändern würde, was den Sinn und Zweck der SupportHypothese überflüssig machen würde.

3.1.4 Auswertung

Für die Auswertung wurden von verschiedenen Logs Replays im Simulator durchgeführt. Das selbe Log einmal mit und ohne Verwendung der SupportHypothese. Diese Replays wurden einfachheitshalber über Screen-Recording aufgenommen, sodass man für die Auswertung die Videos einer bestimmten Spielsituation abspielen und vergleichen kann. Beim parallelen Betrachten der Replays mit und ohne SupportHypothese konnte man sehen, dass es keinen Unterschied in der Wahl des Ballmodelles gibt. Die nicht vorhandenen Reaktionen der Roboter lassen sich demnach nicht durch das Hinzufügen von künstlichen Hypothesen beheben.

Für zukünftige Projektgruppen ist allerdings festzuhalten, dass wenn ein Roboter auf einen liegenden Ball zuläuft, die Ballgeschwindigkeit manchmal negativ wird.

3.2 Ballsuche

Die bisherige Ballsuche verwendet ein statisch gestaltetes Patrouillieren über vordefinierte Positionen auf dem Spielfeld. Diese sind so verteilt, dass durch die entstehenden Pfade beinahe das gesamte Spielfeld in das Blickfeld der Roboter gelangt. Bei diesem Vorgehen wird allerdings die aktuelle Spielsituation nicht berücksichtigt. Es stellt sich deshalb die Frage, ob eine Analyse dieser und einer entsprechend auf die gewonnenen Informationen angepasste Ballsuche dazu führt, dass der Ball zuverlässiger, aber auch schneller gefunden wird.

Im Folgenden wird daher ein neuer dynamischer Ansatz vorgestellt, der zur Klärung dieser Frage beitragen soll. Bei diesem wird das Spiel in drei mögliche Szenarien unterteilt, in denen der Ball gesucht werden kann. Zu diesen gehören Standardsituationen, wie zum Beispiel Eckbälle oder Elfmeter, dann die Spielsituationen, bei denen das eigene Tor erhöht in Gefahr ist und zuletzt alle weiteren Spielsituationen, welche als der normale Spielablauf bezeichnet werden.

Unverändert zur alten Ballsuche gilt, dass immer erst dann nach dem Ball gesucht wird, wenn das gesamte Team diesen verloren hat. Als Ergänzung wird außerdem die bereits bestehende eigene zusätzliche Ballsuche des Ballchasers übernommen. Des Weiteren trifft das nachfolgende nur auf Feldspieler zu. Der Keeper wird nicht in die Ballsuche mit eingebunden, da es zu jedem Zeitpunkt seine höchste Priorität ist das Tor zu verteidigen (siehe Abschnitt 3.3.1).

3.2.1 Ballsuche für Standards

Die Ballsuche für Standards findet immer dann Anwendung, wenn eine Standardsituation nach Spielregeln ausgeführt werden muss. Ziel der Standardsituation ist es, den gewöhnlichen Spielfluss wiederherzustellen. Dabei ist durch die Regeln stets eingegrenzt, wo der Ball liegen muss. Sollte der Ball vor oder während einer Standardsituation verloren gehen, zum Beispiel durch Aufnahme durch den Schiedsrichter, so kann er sich nur im Rahmen der durch die Regeln definierten Positionen befinden. Für die Ballsuche für Standards macht es deshalb nur Sinn, an den entsprechenden Stellen zu suchen. Für jede Standardsituation gestaltet sich die Observierung der jeweiligen relevanten Bereiche anders, weshalb in der Regel immer die Spieler zur Suche verwendet werden, die ohnehin in der Nähe sein sollen.

3.2.2 Ballsuche bei Gefahr

Die Ballsuche bei Gefahr ist die defensivste aller Ballsuchvarianten. Sie wird nur dann angewandt, wenn das Tor sich in erhöhter Gefahr befindet (siehe Abschnitt 3.2.4). In diesem Szenario wird davon ausgegangen, dass dem gesamten Team die Position des Balls unbekannt ist, dem Gegner allerdings nicht. Es ist zu erwarten, dass das gegnerische Team die Kontrolle über den Ball erlangt und sich auf das Tor zubewegt. Aufgrund dieser Annahme ist es die Aufgabe des Teams den Ball zu suchen während es gleichzeitig versucht das Tor zu schützen. Dafür begibt sich das gesamte Team in die eigene Hälfte zurück und nimmt dort eine defensive V-Formation ein (Abb. 3.1). Sinn dieser Aufstellung ist es, dass der Gegner keinen Bereich in der eigenen Hälfte unbemerkt mit dem Ball passieren kann. Die beiden offensivsten Rollen positionieren sich nahe der Mittellinie aber möglichst weit außen, damit dem gegnerischen Team kein Angriff über die Flügel gelingen kann. Die beiden defensivsten Rollen positionieren sich eng zusammengezogen in der Nähe der Goalbox. Durch die Zustellung des Raumes soll es dem Gegner so schwerer gemacht werden einen Ball aufs Tor zu bringen. Gleichzeitig wird durch die V-Formation darauf geachtet, dass der Keeper weiterhin ein möglichst freies Sichtfeld hat, damit er auch Schüsse von außerhalb so früh wie möglich erfassen und entsprechend agieren kann.

Abbildung 3.1: Aufstellung im Rahmen der Ballsuche bei Gefahr.

Alle Positionen innerhalb der Aufstellung sind durch Parameter variabel, wodurch diese an verschiedene Teams angepasst werden kann. Es muss allerdings darauf geachtet werden, dass sich die beiden offensivsten Spieler nicht in der Penaltybox befinden, da sonst die neue Strafraumregelung verletzt wird.

Zu Beginn der Ballsuche begibt sich immer derjenige Spieler zu einer Position in der Aufstellung, der ihr gerade am nächsten ist. Dadurch ist sichergestellt, dass die Aufstellung möglichst schnell eingenommen wird. Sollte es aufgrund einer Unterzahl nicht möglich sein alle Positionen zu belegen, so werden die Positionen ihrer Wichtigkeit nach in aufsteigender Reihenfolge vernachlässigt. Die beiden defensiven Position werden dabei als wichtiger eingestuft als die beiden offensiven Positionen.

3.2.3 Normale Ballsuche

Die normale Ballsuche stellt den Umgang mit einem Ballverlust in allen weiteren Spielsituationen dar, welche keine besonderen Eigenschaften aufweisen. Im Folgenden wird die Basis dieser Ballsuche, die sogenannte Field Coverage, und das Vorgehen unter ihrer Verwendung zur Auffindung des Balls erläutert.

3.2.3.1 Field Coverage

Bei der Field Coverage wird der Ansatz des Bremer Teams aus dem Team Report[Bü+17] verwendet. Ziel dieser Field Coverage ist es, dass kontinuierlich nachgehalten wird, welche Bereiche des Spielfelds ein Roboter einsehen kann und daraus zu schließen wo der Ball sich am wahrscheinlichsten befindet. Das Spielfeld wird dazu in 216 Quadrate (18 x 12) unterteilt, für die nachgehalten wird, ob der Roboter diese einsehen kann und wie wahrscheinlich es ist, dass dort der Ball liegt. Jeder Roboter verfügt dabei über seine eigene Field Coverage. Das Bremer Team unterteilt dabei die Field Coverage in die lokale Field Coverage und die globale Field Coverage welche den folgenden Zweck verfolgen:

In der lokalen Field Coverage wird nachgehalten, wann ein Roboter welche Quadrate gesehen hat. Dazu bekommt das jeweilige Quadrat immer den Zeitstempel des momentanen Frames zugeteilt, wenn der Roboter eine freie Sicht auf diesen Bereich hat. Des Weiteren wird der Zeitstempel eines Quadrates auf 0 gesetzt, wenn sich der Ball laut dem Ballmodell innerhalb dieses Quadrates befindet.

Die globale Field Coverage ist eine Erweiterung der lokalen Field Coverage unter Einbezug der lokalen Field Coverages aller anderen Mitspieler. Hierfür müssen alle Roboter ihre lokale Field Coverage per Broadcast kommunizieren. Dadurch ergibt sich laut dem Bremer Team bei allen ungefähr die gleiche globale Field Coverage. Sie weisen allerdings darauf hin, dass bei einem Timestamp von 4 Bytes die Kommunikationsbeschränkungen durch die Regeln es unmöglich machen den gesamten Inhalt einer jeden lokalen Field Coverage unkomprimiert zu versenden. Wo der maximale Payload liegt, welcher dabei überschritten wird, ist allerdings nicht aufgeführt, sondern lediglich, dass er dem SPLStandardMessage Protokoll entsprechen muss. Der Änderungshistorie des offiziellen Releases des GameControllors[Has18] zufolge, lag dieser zum Zeitpunkt des Team Reports bei 780 Bytes. 2018 wurde die Größe auf 474 Bytes reduziert und ist seitdem unverändert. Sie verwenden deshalb eine Kompression, bei der sowohl die Größe der Timestamps reduziert wird als auch pro Kommunikationszyklus von jedem Roboter stets nur eine Reihe, also 18 Quadrate, versendet werden. Mit ihrem Vorgehen ergibt sich dann ein Payload von 76 Bytes.

Obwohl die Field Coverage des Bremer Teams eine sinnvolle Unterstützung für die Ballsuche ist, müssen kleine Anpassungen an das eigene Team vorgenommen werden. Es ist nicht akzeptabel, dass mit der vorliegenden Kompressionslösung nur 18 der 216 Quadrate pro Sekunde versendet werden. Es würde somit 12 Sekunden dauern, bis die lokale Field Coverage gänzlich übermittelt ist. Auch mit anderen Kompressionsmöglichkeiten, welche ähnlich der des Bremer Ansatzes sind, bei denen aber auf eine partielle Übertragung verzichtet wird, ist es nicht möglich die Obergrenze für den Payload zu unterschreiten.

Dies hängt vor allem damit zusammen, dass seit 2017 die Grenze fast um 41 % der damaligen Grenze reduziert wurde, als auch damit, dass neben den Informationen für die lokale Field Coverage in einem Kommunikationszyklus auch noch weitere relevante Informationen übertragen werden müssen. Es kann für die lokale Field Coverage also nicht der Großteil des Payloads verwendet werden.

Stattdessen wird der Bremer Ansatz so erweitert, dass jeder Spieler nicht nur seine eigene lokale Field Coverage berechnet, sondern auch die seiner Mitspieler. Verwendet wird dafür die sogenannte RobotMap, die stets an alle Mitspieler kommuniziert wird. Sie enthält eine Liste der Positionen aller Roboter auf dem Spielfeld, sprich Hindernisse, die dem jeweiligen Spieler bekannt sind. Die Kommunikation der lokalen Field Coverage eines jeden Mitspieler ist dadurch hinfällig. Auch wird der Inhalt der lokalen Field Coverage abgeändert. Es wird nun lediglich berechnet, wann welches Quadrat gesehen wurde und dann mit einem entsprechenden Timestamp versehen. Ob ein Hindernis die Sicht auf ein Quadrat verdeckt, wird anhand der RobotMaps entschieden. Die Berücksichtigung des Balles erfolgt nicht mehr.

Alle errechneten lokalen Field Coverages werden dann wie im Bremer Ansatz zusammengefügt zu der globalen Field Coverage. Dabei wird für jedes Quadrat immer der letzte Timestamp genommen, damit die globale Field Coverage die aktuellsten Informationen über das Spielfeld aus Sicht des gesamten Teams enthält. Dieser Timestamp ist innerhalb der globalen Field Coverage der sogenannte Coverage Value. Je niedriger dieser Wert, desto höher die Wahrscheinlichkeit, dass dort der Ball liegt, da dieses Quadrat eine gewisse Zeit nicht mehr eingesehen wurde. Der Coverage Value kann niedrig sein indem das Quadrat lange von niemanden gesehen wurde oder aber der Ball laut dem Ballmodell innerhalb des Quadrates war. Letzteres setzt den Coverage Value auf 0. Außerdem kann es sein, dass die globale Field Coverage gerade erste initialisiert wurde. Bei der Initialisierung erhalten alle Quadrate einen negativen Wert, welcher am Mittelpunkt des Spielfelds minimal ist und sich nach außen hin 0 annähert. Dies hat den Hintergrund, dass es beim Anstoß am wahrscheinlichsten ist, dass der Ball am Mittelpunkt liegt. Des Weiteren erhält ein Quadrat gemäß des Bremer Ansatzes den Timestamp des aktuellen Frames und somit einen gleichwertigen Coverage Value, wenn sich die eigene Roboterposition auf diesem befindet. Würde der Ball auf diesem Feld liegen, hätte der Roboter ihn entdeckt und die Suche wäre vorbei. Die Abb. 3.2 und 3.3 veranschaulichen, wie die aktuelle Implementation der Field Coverage aussieht.

3.2.3.2 Anwendung

Das Spielfeld wird erneut in Bereiche, sogenannte Areale, unterteilt. Die Unterteilung ist dabei wesentlich grobgranularer. Aktuell wird das Spielfeld in 4 x 3 Areale unterteilt. Im Rahmen der Suche soll der Roboter die verschiedenen Areale überprüfen, ob in ihnen der Ball liegt. Zur Bestimmung der Reihenfolge der Überprüfungen findet die globale Field Coverage nun Anwendung.

Jedes Areal erhält einen gewissen Wert, über den dann die Reihenfolge bestimmt wird. Dies ist der sogenannte Areal Value. Er ergibt sich aus den Coverage Values aller Quadrate, die sich innerhalb dieses Areals befinden, und den Wegkosten (travelCost) um

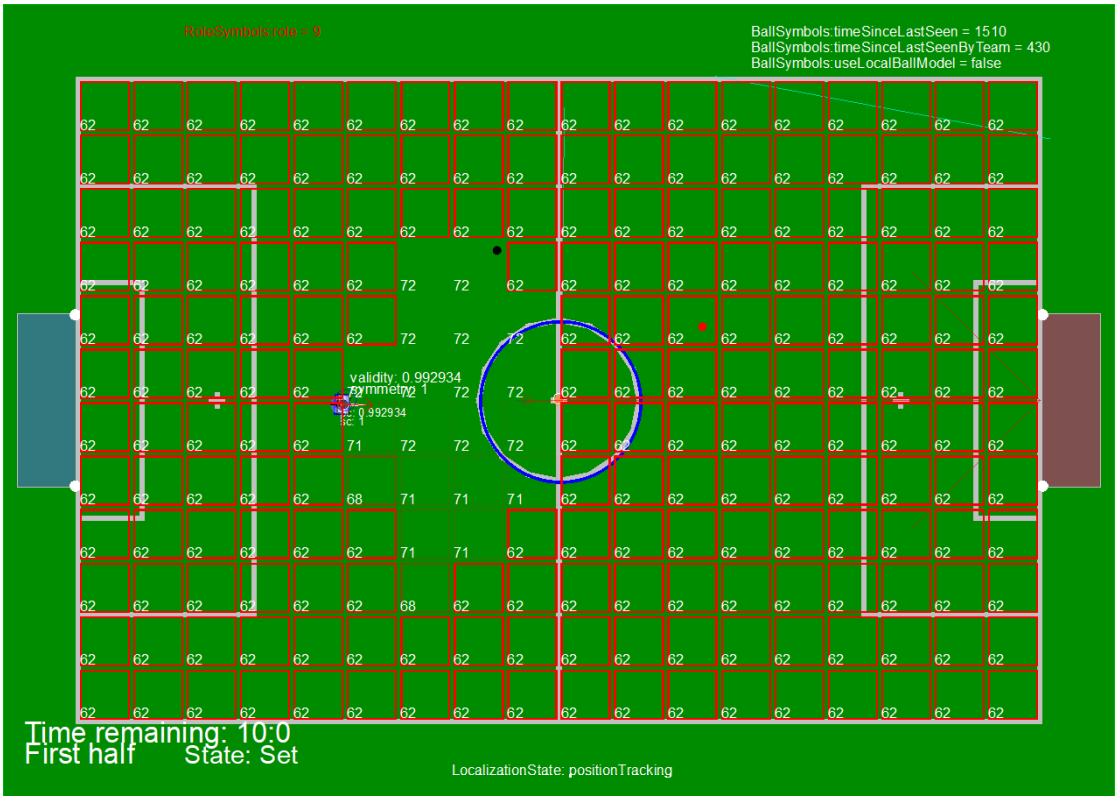


Abbildung 3.2: Lokale Field Coverage mit Timestamps.

zu diesem Areal zu gelangen. Die Wegkosten entsprechen der Distanz zu dem nächsten Punkt des Areal. Es ergibt sich:

$$ArealValue = costMod * travelCost + covMod * \sum CoverageValue$$

Ein niedriger Coverage Value ist gut, da die Wahrscheinlichkeit den Ball dort zu finden hoch ist und niedrige Wegkosten sind gut, damit der Roboter ein nahes Areal wählt. Folglich soll das Areal mit dem niedrigsten Areal Value zuerst überprüft werden. Der Modifikator costMod ist der Wegkosten Modifikator und der covMod ist der Coverage Value Modifikator. Sie dienen dazu das Verhältnis anzupassen um die Ballsuche zu optimieren und sind parametrisierbar.

Vor der Berechnung der Areal Values muss allerdings darauf geachtet werden, dass die Coverage Values noch normalisiert werden. Dies ist erforderlich, da die Coverage Values aus den Timestamps hervorgehen und somit nach oben unbeschränkt sind. Für die Wegkosten gibt es hingegen ein Maximum. Würde keine Normalisierung der Coverage Values vorgenommen werden, würden die Wegkosten auf Dauer an Relevanz verlieren.

Zur Überprüfung eines Areals bewegt sich der Roboter zum nächstliegenden Punkt des Areals und von dort aus zum entferntesten Punkt der Arealsgrenze auf der er sich befindet. Es wird sich davon versprochen, dass auch hinter Hindernisse innerhalb des

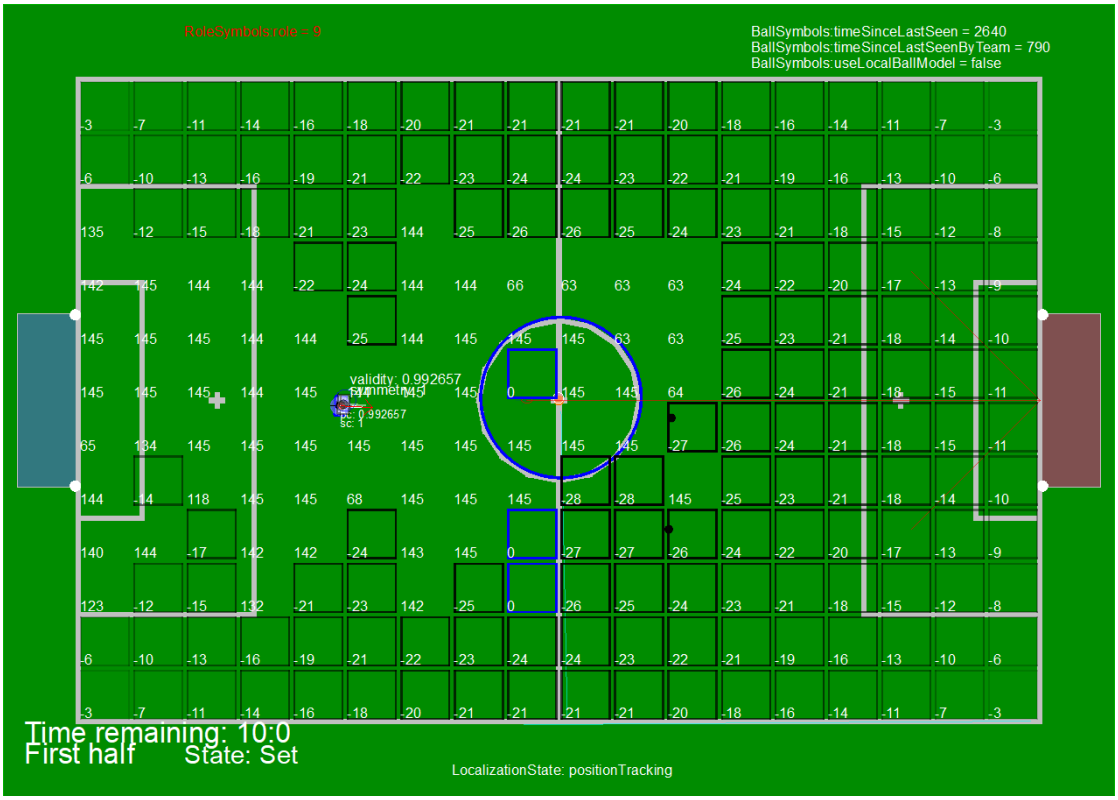


Abbildung 3.3: Globale Field Coverage mit Timestamps.

Areals geblickt werden kann. Die normale Ballsuche unter Verwendung der Areale ist in Abb. 3.4 veranschaulicht.

Es ist wichtig, dass wenn der Roboter sich einmal für ein Areal zur Überprüfung entschieden hat, er auch bei dieser Entscheidung bleibt. Würde er bei jeder Aktualisierung der globalen Field Coverage, also jedes Frame, eine neue Entscheidung treffen, besteht die Chance, dass der Roboter lediglich zwischen den Arealen pendelt, ohne ein Areal gänzlich zu überprüfen. Stattdessen darf eine erneute Entscheidung erst erfolgen wenn die Überprüfung eines Areals abgeschlossen ist.

3.2.4 Verfahren zur Auswahl der Ballsuche

Die Auswahl, welche Ballsuche ausgeführt werden soll, wird in jedem Frame anhand der aktuellen Spielsituation neu entschieden. Dies ermöglicht es, dass eine Ballsuche auch unterbrochen werden kann, wenn sich das Szenario ändert und eine andere Ballsuche besser angebracht ist. Aus der normalen Ballsuche heraus kann so zum Beispiel ein Übergang zur Ballsuche bei Gefahr erfolgen.

Die eigentliche Auswahl folgt dabei einer Priorisierung. Es wird zunächst überprüft, ob eine Standardsituation vorliegt und danach, ob das Tor in Gefahr ist. Erstere Information wird aus den GameInfos gewonnen. Sie hat die höchste Priorität, da eine Standardsituation

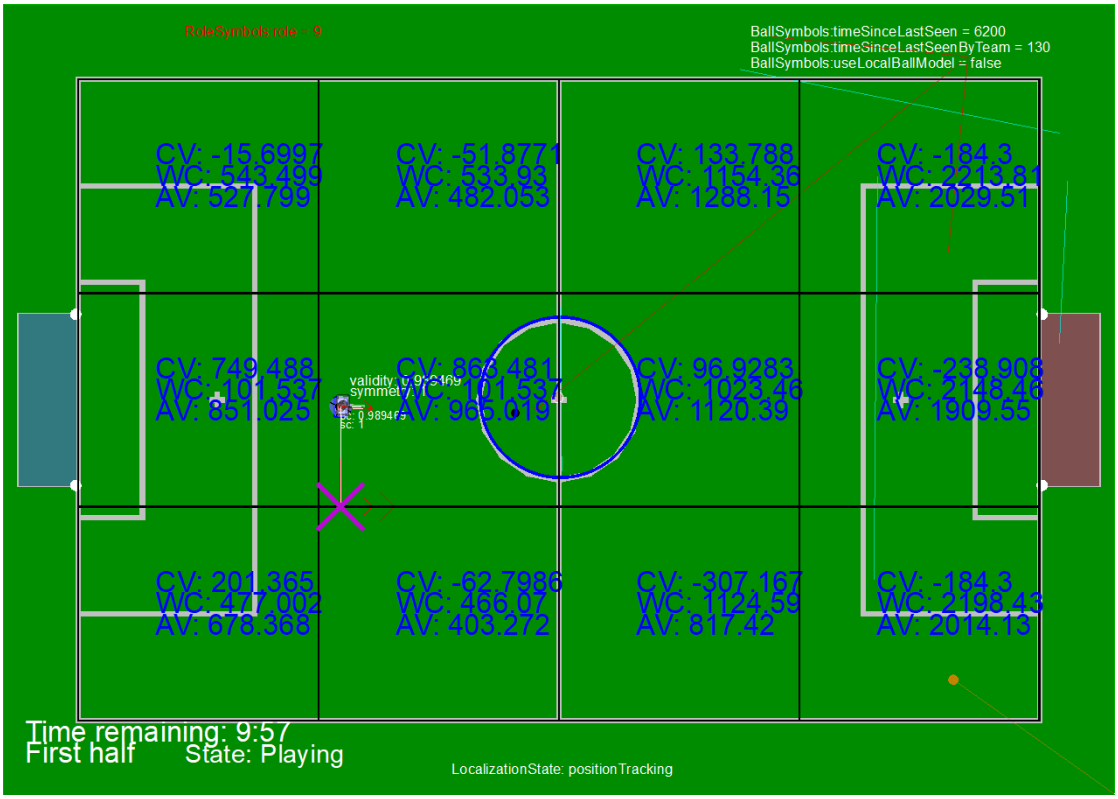


Abbildung 3.4: Areale bei der normalen Ballsuche und Ziel zur Untersuchung eines Areals. (CV = Coverage Value; WC = Wegkosten; AV = Areal Value)

und eine Gefahrensituation gleichzeitig auftreten könnten, bei einer Standardsituation aber wie schon zuvor erklärt aufgrund der Regeln klar ist, wo der Ball liegen kann. Mit einer gezielten Ballsuche an den entsprechenden Stellen kann der Ball deshalb vermutlich schneller aufgefunden werden, als wenn das Team eine vordefinierte defensive Aufstellung einnimmt.

Die Frage nach der Gefahr für das Tor ist die Nächstrelevante. Der erste Ansatz war es zu überprüfen, ob sich mindestens zwei gegnerische Spieler in unserer eigenen Hälfte befinden und davon mindestens einer in der Penaltybox. Allerdings wird hierfür eine teamübergreifende Auflistung aller Gegenspieler benötigt – eine Art globale RobotMap. Da diese nicht existiert, jeder Mitspieler allerdings seine eigene RobotMap kommuniziert, wurde versucht diese Informationen zu einer approximierten globalen RobotMap zusammenzusetzen. Dies hat sich aber als nicht machbar erwiesen. Das Problem war, dass ein Gegenspieler sich aus der Sicht zweier unterschiedlicher Mitspieler niemals exakt an derselben Position befindet. Nun müsste man beim Zusammenlegen der einzelnen RobotMaps aller Mitspieler bei solchen Fällen immer entscheiden, ob es sich um denselben Gegner in beiden RobotMaps handelt und nur eine geringe Positionsabweichung aufgrund von Sensorungenauigkeiten vorliegt oder aber ob es wirklich zwei verschiedene Gegen-

spieler sind, die nur sehr nah beieinanderstehen. Dieses Problem war über eine simple Approximation mit einem Threshold für Positionsabweichungen nicht entscheidbar. Was stattdessen benötigt wird, ist eine globale RobotMap, welche zum Beispiel mithilfe eines Kalman-Filter erstellt werden könnte. Dies überschritt allerdings den Rahmen sowohl thematisch als auch in zeitlicher Hinsicht. Stattdessen gilt das Tor als in Gefahr, wenn die Position, an welcher der Ball laut Ballmodell zuletzt gesehen wurde, innerhalb der eigenen Hälfte ist.

In allen weiteren Spielszenarien wird auf die normale Ballsuche zurückgegriffen. Sie ist immer die letzte Option, die ausgeführt wird, wenn die aktuelle Situation keine spezielle Handhabung erfordert.

3.2.5 Evaluation

Bei ersten Tests noch während der Implementierung ist bereits aufgefallen, dass das Team bei der normalen Ballsuche in ungünstigen Situationen dazu tendiert, den Keeper im Tor allein zu lassen. Da das gegnerische Team allerdings keinen offenen Rückraum vorfinden soll, wurde resultierend daraus entschieden, dass niemals alle Feldspieler nach dem Ball suchen, sondern dass der hinterste Spieler in der normalen Ballsuche an der Goalbox bleibt und den Keeper bei potenziellen Angriffen unterstützt.

Bezüglich der Dauer zur Berechnung der eigenen lokalen Field Coverage wurden 53 Mikrosekunden gemessen. Die gleiche Berechnung muss ebenfalls zusätzlich für alle Mitspieler vorgenommen werden. Da mit Daten gleicher Struktur gearbeitet wird, ist zu erwarten, dass die Dauer sich linear verhält. Bei der Berechnung aller lokalen Field Coverages für jeden der fünf Spieler ist also mit einer Gesamtdauer von 265 Mikrosekunden zu rechnen. Hinzu kommt noch die Dauer für die Berechnung der globalen Field Coverage und weitere Operationen. Ziel ist es bei unter einer 1 Millisekunde pro Modul zu bleiben. Das Modul zu Berechnung der Field Coverage würde dieses Kriterium trotz zusätzlicher Operationen erfüllen. Es steht also eine Kompressionsmöglichkeit, welche eine vollständige Übertragung der lokal Field Coverage nur alle 12 Sekunden vorsieht, dafür allerdings nur wenige Bytes an Payload benötigt, einer Variante gegenüber, die mehr Laufzeit benötigt, allerdings die Limitationen diesbezüglich nicht überschreitet. Folglich wurde sich dafür entschieden, dass die zusätzliche Laufzeit in Kauf genommen wird, um sich jeglichen zusätzlichen Payload zu ersparen und eine sekündlich vollständige Aktualisierung aller lokalen Field Coverages zu ermöglichen.

Des Weiteren ist nach eigener Einschätzung die Bewertung, wann das Tor in Gefahr ist, nicht die beste Lösung, da so in der Hälfte aller Fälle defensiv gespielt wird. Sobald eine globale RobotMap hinzugefügt wurde, sollte in zukünftigen Implementierungen zugegriffen werden. Allerdings gilt auch bei der aktuellen Implementierung, dass man im schlimmsten Fall nur zu defensiv spielt. Ausgehend davon, dass sich dies auf eine Spielsituation bezieht, in der eine Unsicherheit an Informationen herrscht, wird es als sicherer betrachtet, wenn man zu defensiv, statt zu offensiv spielt.

Eine Bewertung wie sich die neue Ballsuche nun gegenüber der alten Ballsuche verhält, und somit die Beantwortung der einleitenden Frage, wurde nicht vorgenommen. Der Hintergrund dafür wird im Folgenden erklärt. Als Metrik zur Bewertung der Qualität

der Ballsuche wurde zu Beginn die Zuverlässigkeit der Ballsuche, sprich wie häufig wird der Ball gefunden, als auch die zeitliche Dauer den Ball zu finden, benannt. Diese Metrik könnte für beide Ballsuchen erfasst werden und dann zur Bewertung miteinander verglichen werden. Allerdings stellt sich die Frage, welche Spielsituationen für die Ballsuche zur Messung hergestellt werden. Hier konnte keine Entscheidung gefällt werden, da gänzlich unklar war, wie repräsentativ diese für ein reales Spiel wären. Im schlimmsten Fall könnte es dazu kommen, dass aufgrund einer solchen Bewertung die Aussage getroffen wird, dass die neue Ballsuche besser ist, dies allerdings nicht auf reale Spiele zutrifft. Es wurde sich stattdessen dafür entschieden, die Qualität der neuen Ballsuche bei realen Testspielen zu bestimmen. Dann liegen Daten aus einem echten Spiel mit echten Gegnern vor. Allerdings kam es aufgrund der aktuellen Pandemie nicht zu Testspielen mit anderen Mannschaften und entsprechende Daten konnten nicht erfasst werden. Eine entsprechende Bewertung steht also noch aus und sollte vorgenommen werden, sobald dies wieder möglich ist.

Allerdings lässt sich schon jetzt die Aussage treffen, dass der Erfolg der Ballsuche sehr stark von zuvor definierbaren Parametern abhängig ist. Ein Parameterpaar sticht dabei besonders hervor, da es beim Testen der Funktionalität der Implementierung vermehrt Probleme bereitet hat: das Verhältnis von der Summe der Coverage Values eines Areals und den Wegkosten. Gewichtet man die Coverage Values eines Areals zu stark, untersucht das gesamte Team ausschließlich dieses Areal. Gewünscht ist aber, dass die Roboter sich sinnvoll über das gesamte Spielfeld aufteilen. Zum aktuellen Zeitpunkt wurde kein gutes Verhältnis gefunden. Es wird daher empfohlen, dieses Parameterpaar kontinuierlich anzupassen, sodass sich die aus realen Spielen gewonnene Bewertung verbessert.

3.3 Anpassungen Rollenverhalten

Die elementaren Bausteine des Verhaltens sind die Rollen. Jeder Roboter auf dem Spielfeld kriegt abhängig von der momentanen Spielsituation eine Rolle zugeteilt. Jede Rolle ist dabei auf eine oder mehrere Aufgaben spezialisiert. Die Rolle Keeper ist zum Beispiel darauf ausgelegt, das Tor zu verteidigen und die Rolle Ballchaser hat die Aufgabe, den Ball ins gegnerische Tor zu treiben. Das allgemeine Spielverhalten ist somit abhängig von dem individuellen Verhalten der einzelnen Rollen. Es ist daher von großer Relevanz, dass dieses gut durchdacht ist und hervorragend funktioniert. Jedoch hat sich in der Vergangenheit vermehrt gezeigt, dass das Verhalten einiger Rollen nicht gut zum Spiel passt oder aber dass Rollen sich fehlerhaft verhalten. Es ist deshalb das Ziel, alle Rollen von Grund auf zu überarbeiten. Begonnen wird dabei mit dem Keeper, dem Ballchaser und dem Center. Die durchgeführten Änderungen werden im Folgenden beschrieben.

3.3.1 Keeper

Der Keeper ist einer der wichtigsten Rollen auf dem Spielfeld. Dementsprechend muss sichergestellt sein, dass er zuverlässig seine Aufgabe erfüllt: das Tor zu schützen. Dieser Aufgabe kommt der aktuelle Keeper allerdings nicht nach. Wenn der Ball sich aufs Tor

zubewegt, springt er nicht um ihn zu halten oder er verlässt manchmal das Tor, obwohl dies nicht gewünscht ist. Die aktuelle Version wird deshalb komplett verworfen und der Keeper wird mit einem neuen Verhaltenskonzept neu erarbeitet, welches im Folgenden vorgestellt wird.

3.3.1.1 Relevanter Bereich und Situationen

Der neue Ansatz basiert auf einer genauen Definition, welchen Bereich der Keeper verteidigen soll. Dieser Bereich wird als die KeeperZone bezeichnet. Da dies der für den Keeper relevante Bereich ist, wird sich darauf festgelegt, dass der Keeper sich nur innerhalb dieser Zone bewegen darf. Sie wird als Oval so aufgespannt, dass sie beide Torpfosten sowie den Mittelpunkt der Goalboxlinie nahe des Penaltycross kreuzt.

Das Spiel ist nun in verschiedene Situationen aufgeteilt, in denen der Keeper sich unterschiedlich verhalten muss:

- Der Ball liegt außerhalb der Zone
- Der Ball liegt innerhalb der Zone
  - Der Keeper steht zwischen Ball und Tor
  - Der Ball liegt zwischen Keeper und Tor
- Der Ball wird geschossen
- Der Ball wird nicht gesehen

Eine detaillierte Erklärung zu dem jeweiligen Verhalten findet sich in den entsprechenden Abschnitten.

3.3.1.2 Der Ball liegt außerhalb der KeeperZone

In dieser Spielsituation besteht nicht die akute Gefahr, dass der Ball zeitnah ins Tor gedribbelt wird. Allerdings kann es passieren, dass ein Gegner einen Schuss von weiter außerhalb ausführt. Um dem Gegner solch einen Schuss zu erschweren, bewegt sich der Keeper nach vorne bis an den Rand der Zone. Dadurch ist es ihm möglich einen viel größeren Bereich des Tors abzudecken. Er positioniert sich dabei so, dass er sowohl stets auf dem Rand der KeeperZone steht, als auch gleichzeitig auf einer Linie vom Ball zum Mittelpunkt des Tors. Letzteres erhöht die Wahrscheinlichkeit, dass der Keeper bei einem Schuss den Ball durch Hinsetzen oder -werfen abfängt, da er durch die Positionierung dem Ball stets so nah wie möglich ist. Eine beispielhafte Positionierung lässt sich in Abb. 3.5 sehen.

3.3.1.3 Der Ball liegt innerhalb der KeeperZone

Alternativ kann sich der Ball auch innerhalb der Zone befinden. Hier muss nun entsprechend der relationalen Position des Balles zum Keeper und Tor gehandelt werden. Der

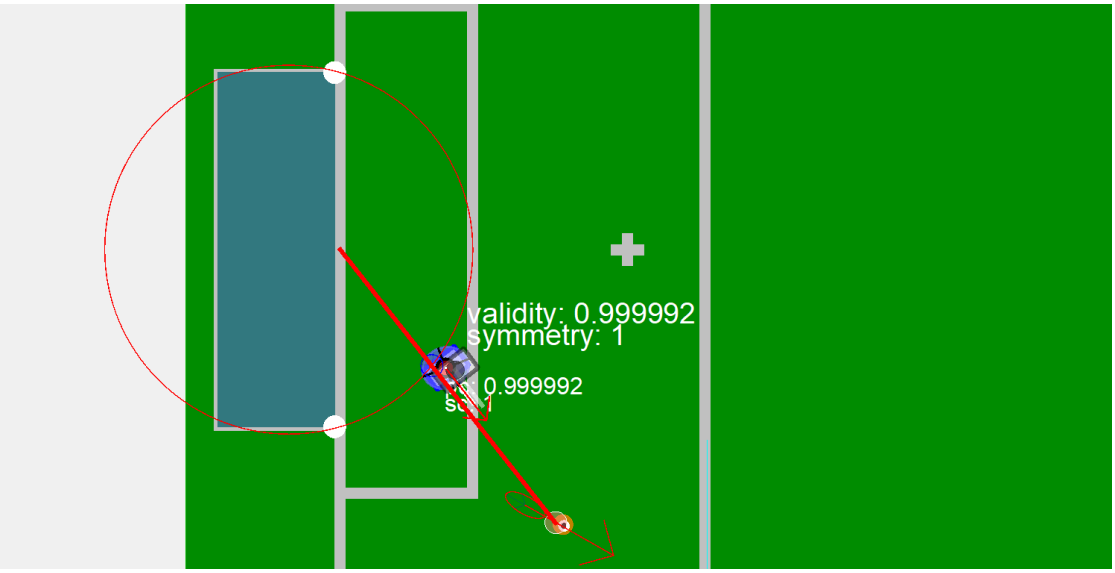


Abbildung 3.5: Beispielhafte Positionierung wenn der Ball außerhalb der KeeperZone ist.

einfachere Fall ist es, wenn der Keeper sich zwischen Ball und Tor befindet. Der Keeper ist dann immer noch in der Lage, sich schützend zu positionieren, um zu verhindern, dass der Ball ins Tor gelangt. Da dieser sich allerdings bereits innerhalb der KeeperZone befindet, macht es wenig Sinn, dass der Keeper sich aus dem Tor herausbewegt. Es ist vom Gegner zu erwarten, dass er einen Schuss aufs Tor bringt oder aber den Ball ins Tor dribbelt. Da unklar ist, welche der beiden Möglichkeiten gewählt wird, dennoch aber gegen beide verteidigt werden muss, positioniert der Keeper sich auch hier dem Ball möglichst nahe. Es wird die Position auf der Grundlinie gewählt, welche die kürzeste Distanz zum Ball hat. Wird gedribbelt, so bewegt der Keeper sich auf der Grundlinie mit dem Ball mit, bis es zu einem Kontakt kommt. Der Keeper blockiert somit aktiv das Dribbling des Gegners. Wird Geschossen, so muss der Gegner sehr stark nach links oder nach rechts schießen, damit er am Keeper vorbeischießt. Ein erfolgreicher Torschuss wird dadurch deutlich erschwert.

Es muss allerdings auch bedacht werden, was passiert, wenn kein Gegner oder Mitspieler in Reichweite ist, um den Ball zu spielen. In diesem Fall kommt der Keeper hervor, um den Ball außer Reichweite des Tors zu schießen.

Der Ball kann sich aber auch zwischen Keeper und Tor befinden. Hier sei angenommen, dass der Keeper den Ball sieht. Für das Gegenteil sei auf Abschnitt 3.3.1.5 verwiesen. Der Keeper befindet sich hier in einer Position, in der er das Tor nicht schützen kann. Es ist daher sein Ziel, so schnell wie möglich wieder zwischen Ball und Tor zu gelangen. Sein Zielpunkt für die Positionierung ist deshalb der nächste Punkt zum Ball, welcher sich auf der Geraden vom Mittelpunkt des Tores zum Ball befindet. Sollte er diesen Punkt erreichen, so kann er sich wie zuvor beschreiben auf eine passende Position auf der Grundlinie begeben, allerdings rückwärts, damit er stets einen Blick auf den Ball

hat. Die Schwierigkeit, die nun im Raum steht, ist, dass der Keeper an diesen Zielpunkt gelangen muss, ohne dabei gegen den Ball zu laufen. Andernfalls besteht die Gefahr, dass der Keeper den Ball aus Versehen ins eigene Tor befördert. Um dies zu verhindern, wird ein kreisförmiger Bereich um den Ball herum definiert, den der Keeper nicht betreten darf. Auf dem Weg zum Zielpunkt bewegt der Keeper sich so, dass er möglichst den direkt Weg nimmt. Sollte er dabei allerdings durch den gesperrten Bereich laufen wollen, so bewegt er sich entlang des Randes, bis er wieder auf einen geraden Weg zum Zielpunkt gelangt.

3.3.1.4 Der Ball wird geschossen

Das Verhalten bei einem Schuss ist unabhängig von der aktuellen Position des Keepers. Zunächst wird ausgewertet, ob ein Handeln überhaupt erforderlich ist oder ob der Keeper gemäß seines sonstigen Verhaltens weiter versucht seine Positionierung zu optimieren. Erforderlich ist es dann, wenn die vorhergesagte Ballposition sich hinter dem Keeper befindet und der Ball sich beim Überqueren der eigenen Grundlinie zwischen den beiden Torpfosten befinden wird. Sollte ein Handeln erforderlich sein, wird als Nächstes überprüft, ob ein passendes Handeln zum Erfolg führen würde. Dies ist nur dann der Fall, wenn der Keeper mit einem Sprung oder Hinsetzen überhaupt in Reichweite des Balls gelangt. Führt Handeln zu Erfolg, wird entsprechend gehandelt. Es kann gesprungen werden oder aber der Keeper kann sich setzen und die Arme ausbreiten. Letzteres ist dabei zu bevorzugen, da das Aufstehen nach einem Sprung wesentlich mehr Zeit in Anspruch nimmt. Sollte das Hinsetzen allerdings nicht zu einem erfolgreichen Halten des Balls führen, sondern nur der Sprung, so wird gesprungen. Wenn keine der beiden Aktionen zum Erfolg führen würde, dann kann trotzdem gesprungen werden, oder aber es wird sich weiter positioniert. Dies ist parametrisierbar und muss nach eigenem Ermessen eingestellt werden.

3.3.1.5 Der Ball ist nicht sichtbar

Wie aus Abschnitt 3.2 hervorgeht, nimmt der Keeper nicht an der Ballsuche des Teams teil. Stattdessen hat er sein eigenes Verhalten wie er damit umgeht, wenn er den Ball nicht sieht. Es ist für den Keeper zwar wichtig, dass der Ball so schnell wie möglich wieder in sein Blickfeld gelangt, allerdings hat es noch eine viel höhere Priorität, das Tor optimal zu schützen. Der Keeper positioniert sich deshalb auf der Mitte seiner eigenen Grundlinie. Auf dieser Position ist es ihm möglich einen großen Bereich des Spielfelds einzusehen, wodurch er den Ball schnell wieder erfassen kann, und gleichzeitig schützt er das Tor auch optimal, da er sowohl für die Verteidigung zum linken als auch zum rechten Torpfosten dieselbe Zeit benötigt.

Hierbei muss aber beachtet werden, wie er sich auf diese Position begibt. Um möglichst viel vom Spielfeld zu sehen, bewegt er sich rückwärst mit dem Blick zur Mitte des Spielfelds. Dabei besteht die Gefahr, dass der verlorene Ball hinter dem Keeper liegt und er bei der Rückwärtsbewegung unabsichtlich ein Eigentor erzielt. Um dies zu verhindern, wird bei der Ballsuche des Keepers berücksichtigt, wo sich der Ball zuletzt befand.

Befand er sich zuletzt in der gegnerischen Hälfte, ist es sehr unwahrscheinlich, dass er aus ihr bis hinter den Keeper gelangt ist, ohne dass ein Mitspieler aus dem Team dies gesehen hätte. Der Keeper kann sich deshalb, wie zuvor erklärt, rückwärts auf die Zielposition begeben.

Befand er sich hingegen in der eigenen Hälfte, so ist das Unterfangen risikoreicher. Um aber auch hier ein Eigentor zu vermeiden, bewegt der Keeper sich zunächst von seiner aktuellen Position aus einige Schritte nach vorne, dreht sich dann in Richtung Tor und begibt sich anschließend auf die Zielposition. Dies vermeidet, dass der Keeper den Ball unabsichtlich ins Tor befördert. Es erfolgen vor der Rotation zunächst einige Schritte nach vorne, damit es bei ihr nicht zu einem Eigentor kommt, wenn der Ball ungünstig nah hinterm Keeper liegt.

3.3.1.6 Evaluation

Da der Keeper sich noch in der Implementierung befindet, wird im Rahmen der Evaluation der aktuelle Grad der Erfüllung der vorgesehenen Funktionalität überprüft. Ein Vergleich mit dem ersetzten Verhalten des alten Keepers und eine entsprechende Bewertung findet nicht statt.

Erste Tests zur Überprüfung der Funktionalität haben gezeigt, dass die aktuelle Implementierung des Konzepts noch sehr viele Probleme aufzeigt. Eines der größten Probleme ist die Delokalisierung. Solange sich der Keeper so positioniert, dass er einen Blick weg vom eigenen Tor hat, treten keine Probleme auf. Dies ist beispielsweise der Fall, wenn der Keeper sich auf dem Rand der KeeperZone positioniert, wenn der Ball außerhalb von ihr liegt. Sobald der Keeper sich allerdings dem Tor zuwendet, interpretiert er seine eigene Position innerhalb von wenigen Sekunden falsch und verhält sich dementsprechend fehlerhaft. Dies geht sogar so weit, dass aufgrund des falschen Verhaltens Eigentore geschossen werden oder aber das Tor dem Gegner vollständig geöffnet wird. Eine mögliche Lösung diese Delokalisierung zu vermeiden, wäre natürlich zu sagen, dass der Keeper sich niemals zum Tor wenden soll, allerdings besteht dann besonders bei einer unbekannten Ballposition die Gefahr, dass der Keeper bei einer Rückwärtsbewegung den Ball selbst ins Tor befördert. Es ist bei Verlust des Balls zwingend erforderlich, dass der Keeper sich wie zuvor vorgestellt dem Tor zuwendet, was natürlich in Konkurrenz damit steht, sich nicht zu delokalisieren. Eine Lösung wie beides sichergestellt werden kann, liegt zum aktuellen Stand nicht vor.

Des Weiteren macht die automatische Positionskorrektur der Roboter Probleme. Diese greift immer dann, wenn ein Roboter delokalisiert war und sich neu lokalisiert hat sowie vice versa. Das führt dazu, dass der Roboter über eine neue Position verfügt, die er einnehmen möchte. Dabei kam es bei den Implementierungstests vermehrt dazu, dass der Keeper sich so bewegt hat, dass er mit dem Ball kollidierte, was wiederum vereinzelt zu Eigentoren geführt hat.

Als letztes war es während der Implementierungstests auffällig, dass bei Situationen, in denen der Ball zwischen Keeper und Tor liegt, die existierende Pfadplanung nicht gut funktioniert. Obwohl der Pfadplanung ein Punkt gegeben wird, zu dem auf direktem Wege ohne Kollision gelangt werden kann, plant die Pfadplanung einen Weg, der erst

vom Ziel hinfort führt und dann durch den Ball hindurch zum Ziel. Es war gewünscht, dass der Keeper sich hinter den Ball bewegt, dabei aber die Sperrzone um den Ball herum vermeidet. Dementsprechend werden der Pfadplanung nur Wegpunkte zum Ziel übergeben, die direkt ohne das Passieren des gesperrten Bereiches erreicht werden können. Die Pfadplanung sorgt aber dafür, dass es zu dem direkten Gegenteil kommt. Ersten Einschätzungen zufolge könnte dies daran liegen, dass zu nah an einem Hindernis, dem Ball, vorbei gegangen werden soll und die Pfadplanung damit falsch umgeht. Dieses Fehlverhalten muss näher untersucht werden, um herauszufinden, wie es zu beheben ist.

Alle weiteren zuvor beschriebenen Funktionalitäten scheinen wie geplant zu funktionieren. Es steht natürlich noch aus, was in einem echten Spiel passiert und wie sich das neue Konzept gegenüber dem alten Verhalten schlägt, aber für das erste Fazit während der Implementierung lässt sich Folgendes sagen: Der Keeper kommt einem Großteil der für ihn vorgesehenen Funktionalität nach. Dies zeugt zumindest davon, dass es technisch möglich ist diese umzusetzen. Allerdings sind auch noch gravierende Fehler vorhanden, die zwingend behoben werden müssen, bevor der neue Keeper einsatzbereit ist. Erst wenn dies der Fall ist, kann der neue Keeper bewertet werden und es kann eingeschätzt werden, ob das neue Konzept etwas bringt, oder aber wesentliche Mängel aufweist und abgeändert werden muss.

3.3.2 Ballchaser

Der Ballchaser wurde nicht neu konzeptioniert, es wurden lediglich Verbesserungen vorgenommen. Konkret geht es darum, das Verhalten des Ballchasers bei gegnerischem Ballbesitz zu verbessern. Die Idee dabei ist das Verhaltensweise aus dem Sport zu implementieren und somit den Gegner in den Laufweg zu laufen und nicht wie bisher sich nur auf den Ball zuzubewegen unabhängig davon, ob ein gegnerischer Roboter ihn führt. Somit versucht dieser Ansatz den Laufweg des Gegners zu erkennen und diesen zu blockieren. Des Weiteren ist zu beachten, dass der Gegner in der Nähe des Tores schießen könnte, deshalb wurde an dieser Stelle ein anderes Verhalten implementiert und der Ballchaser versucht das eigene Tor verdecken, indem er sich zwischen Ball und Tor bewegt. Im Folgenden Abschnitt beschreibt der Begriff Striker den gegnerischen Roboter mit der geringsten Distanz zum Ball.

3.3.2.1 Striker Interception

Zunächst stellt sich die Frage, wann es sinnvoll ist, in den Laufweg des Strikers zu laufen oder sich das normale Hinbewegen zum Ball besser eignet. Bei diesem Ansatz wird davon ausgegangen, dass die Roboter beider Teams sich gleich schnell fortbewegen. Deshalb werden die beiden Distanzen vom Ballchaser zum Ball und vom Striker zum Ball verglichen. Nur wenn die Distanz vom Striker zum Ball kleiner ist, wird eine Interception in Betracht gezogen. Befindet der Ballchaser sich in der unmittelbaren Nähe des Balles wird das normale Zweikampfverhalten ausgeführt.

Bei der Interception gibt es zwei unterschiedliche Verhalten, die davon abhängig sind, wo der Ball sich befindet. Das ist damit begründet, dass in der Nähe des eigenen Tores der

Striker schießen könnte und deshalb die Abdeckung des eigenen Tores höchste Priorität hat.

Befindet sich der Ball in der Nähe des eigenen Tores, hat der Ballchaser die Priorität, dieses zu schützen. Dies geschieht, indem der Ballchaser sich auf die Gerade zwischen Ball und Tor hinbewegt (siehe Abb. 3.6). Sollte sich auf dieser Geraden schon ein Roboter befinden, läuft der Ballchaser auf den Ball zu. Erreicht der Ballchaser diese Gerade bewegt er sich auf den Ball zu, um diesen dem Striker abzunehmen.

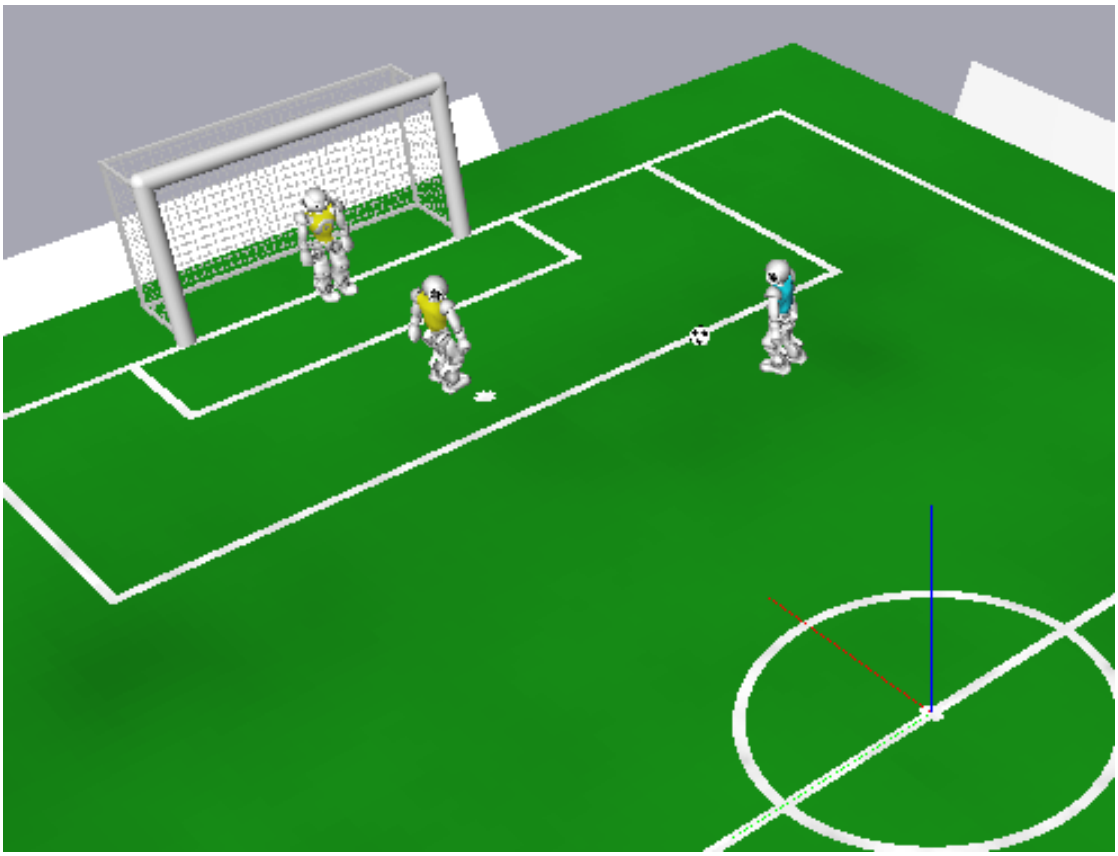


Abbildung 3.6: Beispiel für eine Interception in der Nähe des eigenen Tores. Der Roboter im blauen Trikot ist der Ballchaser und der im gelben Trikot ist der Striker.

Befindet sich der Ball nicht in der Nähe des eigenen Tores, versucht der Roboter den Laufweg des gegnerischen Roboters anzulaufen.

3.3.2.2 Evaluation

Zur Evaluation wurden drei Testszenarios jeweils zehn mal mit dem alten Ballchaser und zehn mal mit dem neuen Ballchaser durchgeführt. Im folgenden werden die Szenarios aus der Sicht des eigenen Tores beschrieben. Des weiteren befindet sich der Ball in jedem

	neuer Ballchaser	alter Ballchaser
Striker kam frei zum Schuss	0	4
Striker wurde beim Schuss gestört	0	4
Striker wurde Ball abgenommen	10	2

Tabelle 3.1: Vergleich zwischen neuem und alten Ballchaser bei Test Szenario 1.

	neuer Ballchaser	alter Ballchaser
Striker kam frei zum Schuss	5	6
Striker wurde beim Schuss gestört	4	4
Striker wurde Ball abgenommen	1	0

Tabelle 3.2: Vergleich zwischen neuem und alten Ballchaser bei Test Szenario 2.

Szenario vor dem Striker. In dem ersten Szenario startet der Ballchaser mittig zwischen Penaltybox und Mittellinie auf Höhe des linken Schnittpunktes zwischen Mittelkreis und Mittellinie und der Striker startet auf der Mittellinie sehr weit links (siehe Abb. 3.7 links). Beim zweiten Testszenario befindet sich der Ballchaser auf dem Schnittpunkt von Mittellinie und Mittelkreis, der Ball befindet sich auf dem vordersten Punkt des Mittelkreises (siehe Abb. 3.7 rechts). Auf Abb. 3.6 ist Szenarios drei zu sehen. Hier startet der Ballchaser vor dem Elfmeterpunkt und der Ball liegt auf der Penaltyboxkante auf Höhe der linken Goalboxkante. Die folgenden Tabellen zeigen die Ergebnisse.

Die Tabelle 3.1 zeigt sehr positive Resultate. Der neue Ballchaser hat jeden Ball gewonnen. Das bedeutet, der Ball lag vor den Füßen des Ballchasers und er konnte den Ball führen.

Bei Tabelle 3.2 sieht man kaum Unterschiede, weil beim Vorbeilaufen der Ballchaser in der Nähe des Balles kam und damit das normale Zweikampfverhalten ausgeführt wurde.

In Tabelle 3.3 ist zu erkennen, dass der neue Ballchaser bessere Ergebnisse in der Nähe des eigenen Tores hat.

Als Fazit lässt sich sagen, dass deutliche Verbesserungen in den ausgewählten Situationen zu erkennen sind. Jedoch wurden die Test im Simulator durchgeführt. Eine klare Aussage, ob der neue Ballchaser einen positiven Effekt auf das Spiel hat, muss in echten Spielen getestet werden.

	neuer Ballchaser	alter Ballchaser
Striker kam frei zum Schuss	3	10
Striker hat den Ballchaser angeschossen	7	0

Tabelle 3.3: Vergleich zwischen neuem und alten Ballchaser bei Test Szenario 3.

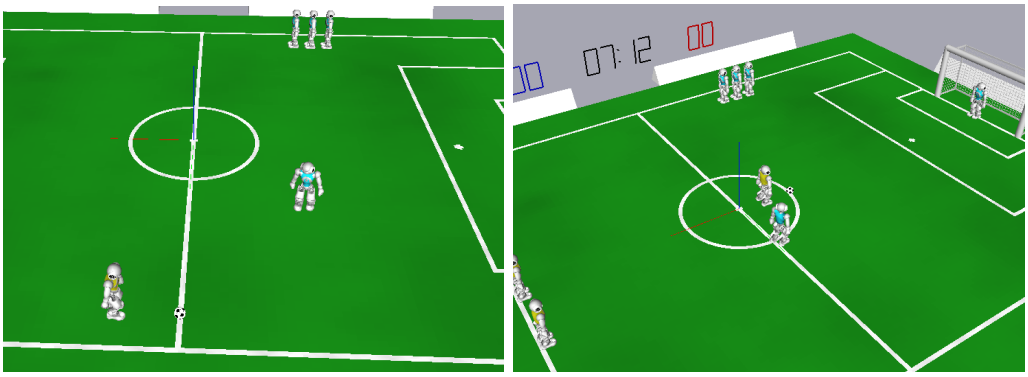


Abbildung 3.7: Das linke Bild zeigt das Test Szenario 1 und das rechte Bild zeigt das Test Szenario 2. Der Roboter im blauen Trikot ist der Ballchaser und der im gelben Trikot ist der Striker

3.3.3 Center

Im Zuge der Neukonzeptionierung der Rollen wurde der Center neu implementiert. Der Fokus lag dabei die Wartbarkeit zu verbessern und den Center aktiver zu gestalten. Des Weiteren sind die neue Penaltybox und die damit entstanden Regeln zu beachten. Diese besagen, dass nur drei Roboter eines Teams sich gleichzeitig in der Penaltybox befinden dürfen. Die Missachtung der Regel hat eine Zeitstrafe zu folge[Rul].

3.3.3.1 Verhalten

Das Verhalten des Centers ist von der Position des Balles und des Ballchasers abhängig. Das Feld wird in drei Bereiche aufgeteilt. Diese Bereiche sind eigene Penaltybox, gegnerische Hälfte und eigene Hälfte ohne Penaltybox. Die eigene Penaltybox darf der Center nicht betreten, weil sich dort wie oben erwähnt nur drei Roboter eines Teams gleichzeitig befinden dürfen. Diese drei sind der Defender, der Keeper und der Ballchaser. Befindet sich der Ball in der eigenen Hälfte, aber nicht in der Penaltybox, so ist der Center der Backup-Ballchaser. Das bedeutet, er läuft hinter dem Ballchaser her und kann so im Falle eines Ballverlustes eingreifen. Befinden sich der Ball und Ballchaser in der gegnerischen Hälfte bleibt der Center an der Mittellinie und bewegt sich hinsichtlich der Y-Achse auf gleicher Höhe zum Ballchaser. In der eigenen Hälfte soll der Center den Backup-Ballchaser darstellen, das heißt leicht versetzt hinter dem Ballchaser herlaufen. Wenn der Ballchaser in der gegnerischen Hälfte ist, soll der Center sich auf derselben Höhe hinsichtlich der y-Achse wie der Ball befindet.

3.3.3.2 Standartsituationen

Das Verhalten wurde bei Standardsituationen auch angepasst. Im Folgenden werden die wichtigsten Änderungen aufgezählt:

- Beim Anstoß des eigenen Teams stellt sich der Center zwischen dem Receiver und dem Ballchaser nach hinten versetzt, da ein Passspiel ausgeführt werden soll, um bei einem misslungenen Pass bestmöglich reagieren zu können.
- Bei einer eigenen Ecke stellt der Center sich zentral vor die gegnerischen Penaltybox, so kann er nach einem Pass den Ball direkt auf das Tor Schießen.
- Bei einer gegnerischen Ecke verteidigt der Center vor der eigenen Penaltybox.

3.3.3.3 Evaluation

Beim Testen des Centers ist aufgefallen, dass dieser mit den Ballchaser kollidiert. Daher wurde ein stärkerer Fokus darauf gelegt, dass der Center dem Ballchaser ausweicht. Bevor der Center sich positioniert überprüft er nun, ob sich der Ballchaser in der Nähe befindet. Ist das der Fall, bewegt der Center sich hinsichtlich der y-Achse von dem Ballchaser weg. Allerdings gibt es dabei eine Ausnahme, wenn der Center sich zwischen Ballchaser und Ball befindet, läuft dieser orthogonal zu der Linie zwischen Ballchaser und Ball Richtung eigenes Tor.

3.3.3.4 Fazit

Alle zuvor beschriebenen Funktionalitäten sind planmäßig umgesetzt worden. Zusätzlich wurde die Wartbarkeit durch bessere Struktur deutlich vereinfacht. Ein direkter Vergleich zu dem Verhalten des alten Center ist nicht sinnvoll, da dieser sehr passiv agierte und sich die Funktionsweisen somit sehr unterscheiden.

4 Fazit und Ausblick

Ziel dieser Arbeit war die Verbesserung der Objektdetektion in Bildern in Echtzeit unter Ressourcenbeschränkung. Dabei stellt die Einhaltung der Laufzeitbeschränkung mit der zur Verfügung stehenden Rechenleistung die größte Herausforderung dar. Zwar stehen aus der aktuellen Forschung im Bereich der KI diverse Ansätze zur Verfügung, welche ohne Berücksichtigung der Laufzeit sehr gute Ergebnisse erbringen. Jedoch zeigte sich, dass sich diese Ergebnisse erheblich verschlechtern, sobald die notwendigen Anpassungen an die Laufzeit durchgeführt wurden.

Für die untersuchten Ansätze der Objektdetektion konnte keine aussagekräftige Evaluation durchgeführt werden, da die notwendige Laufzeitschranke nicht erfüllt wird. Ohne einen direkten Vergleich auf einem NAO-Roboter mit dem bereits implementierten YOLOv2-Ansatz kann keine Aussage darüber getroffen werden, ob die gewählten Ansätze effektiv auf die Ressourcenbeschränkung angepasst werden können. Im Bereich der Bildsegmentierung können erste funktionsfähige Ansätze vorgewiesen werden. Wenngleich weit entfernte Objekte ein Problem darstellen, konnte gezeigt werden, dass Bildsegmentierung auf dem NAO-Roboter grundsätzlich eine mögliche Technik darstellt. Das adaptierte U-Net konnte im Bereich der Ballerkennung nur bedingt überzeugen, lieferte jedoch gute Werte im Bereich Linien-, Roboter und erstmalig Torerkennung. Es ist somit möglich mit einem NN mehrere Objektklassen gleichzeitig zu erkennen. Aufgrund der aktuellen unterschiedlichen Arbeitsweise mit Scan-Linien-Verfahren zur Linienerkennung sowie Ball- und Robotererkennung, kann abschließend noch nicht genau beurteilt werden, ob ein einzelnes NN all diese Aufgaben in gleicher oder besserer Qualität erfüllen kann. Erste Ergebnisse sprechen hier zumindest nicht von einer unmöglichen Aufgabe. Hier empfiehlt sich die Verfolgung des Ansatzes mit einer vollen Integration auf den NAO-Roboter um anschließend beide Vorgehensweisen qualifiziert vergleichen zu können.

Insgesamt kann festgestellt werden, dass die zuverlässige Erkennung von weit entfernten Objekten ein ungelöstes Problem darstellt. Durch die notwendige Rechenleistung im Bereich Objektdetektion oder Bildsegmentierung müssen die Eingabedaten soweit verkleinert werden, dass die Objekte in der Ferne schlecht bis gar nicht erkannt werden. Der vorgestellte Ansatz des Visual Mesh stellt hier einen Ansatz vor, wie dieses Problem umgangen werden kann, jedoch konnte dieser Ansatz noch nicht auf den NAO-Roboter übertragen werden. Daher ist noch keine Einschätzung möglich, ob eine Realisierung unter der Laufzeitbeschränkung möglich ist. Es empfiehlt sich jedoch diesen Ansatz weiter zu verfolgen.

Für den Ansatz Orientierung im Raum mittels CNN wurden erste Experimente durchgeführt. Diese zeigen, dass unter Einschränkungen die Position aus synthetischen Masken mit einem CNN geschätzt werden kann. Zur Evaluation des Verfahrens auf dem NAO-Roboter sind jedoch noch weitere Experimente notwendig. Da die ersten Ergebnisse

vielversprechend sind, empfiehlt es sich diesen Ansatz weiter zu verfolgen.

Unabhängig der vorgestellten Ergebnisse empfiehlt sich die Evaluation weiterer Möglichkeiten zur Portierung und Beschleunigung der verwendeten Netze. Der bislang verwendete DCG stellt sich durch seinen Funktionsumfang beim Design der Netzstrukturen als limitierender Faktor heraus. Unabhängig von dieser Arbeit durchgeführte erste Tests mit Tensorflow Lite, einem Framework für maschinelles Lernen auf Geräten mit reduzierter Rechenleistung, zeigten potenziell bessere Laufzeitergebnisse auf dem NAO-Roboter, bei größerem Funktionsumfang. Die notwendige Version dieses Frameworks erschien erst gegen Ende dieser Arbeit, so dass die möglicherweise positiven Ergebnisse nicht direkt geprüft werden konnten. Wegen der jedoch ersten positiven Tests sollte diese Evaluation durchgeführt werden, da dadurch potenziell nicht realisierbare Ansätze dieser Arbeit möglich werden könnten.

Zusammenfassend kann gesagt werden, dass die Objektdetektion unter Ressourcenbeschränkung möglich ist, jedoch starke Anforderungen an die Entwicklung und Integration eines NN stellt.

A Abbildungen

Datensatz

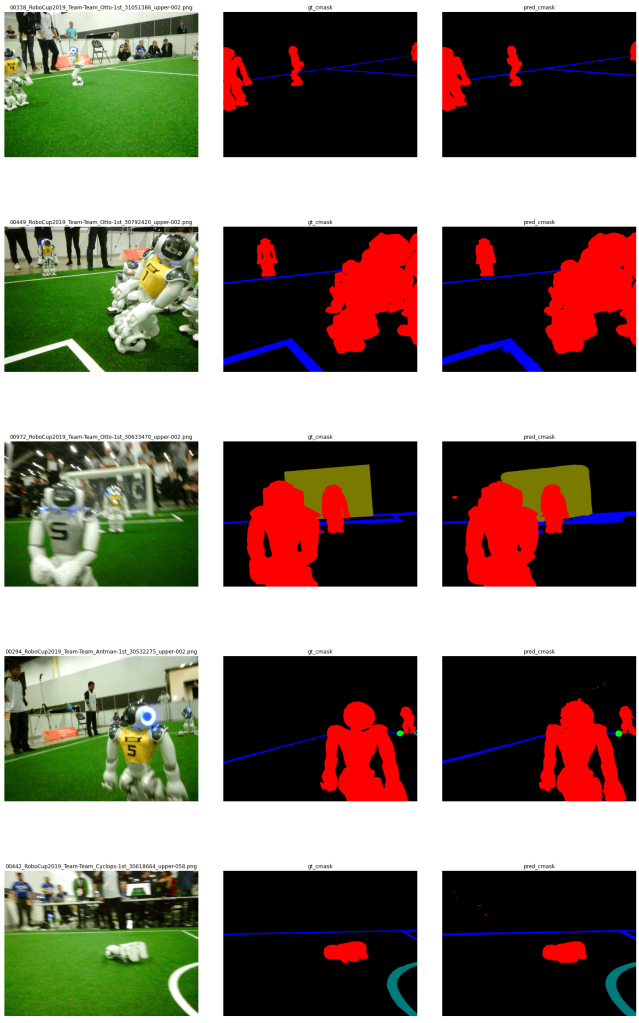


Abbildung A.1: Validierungsbilder aus dem Training mit automatisch annotierten Bildern am Anfang.

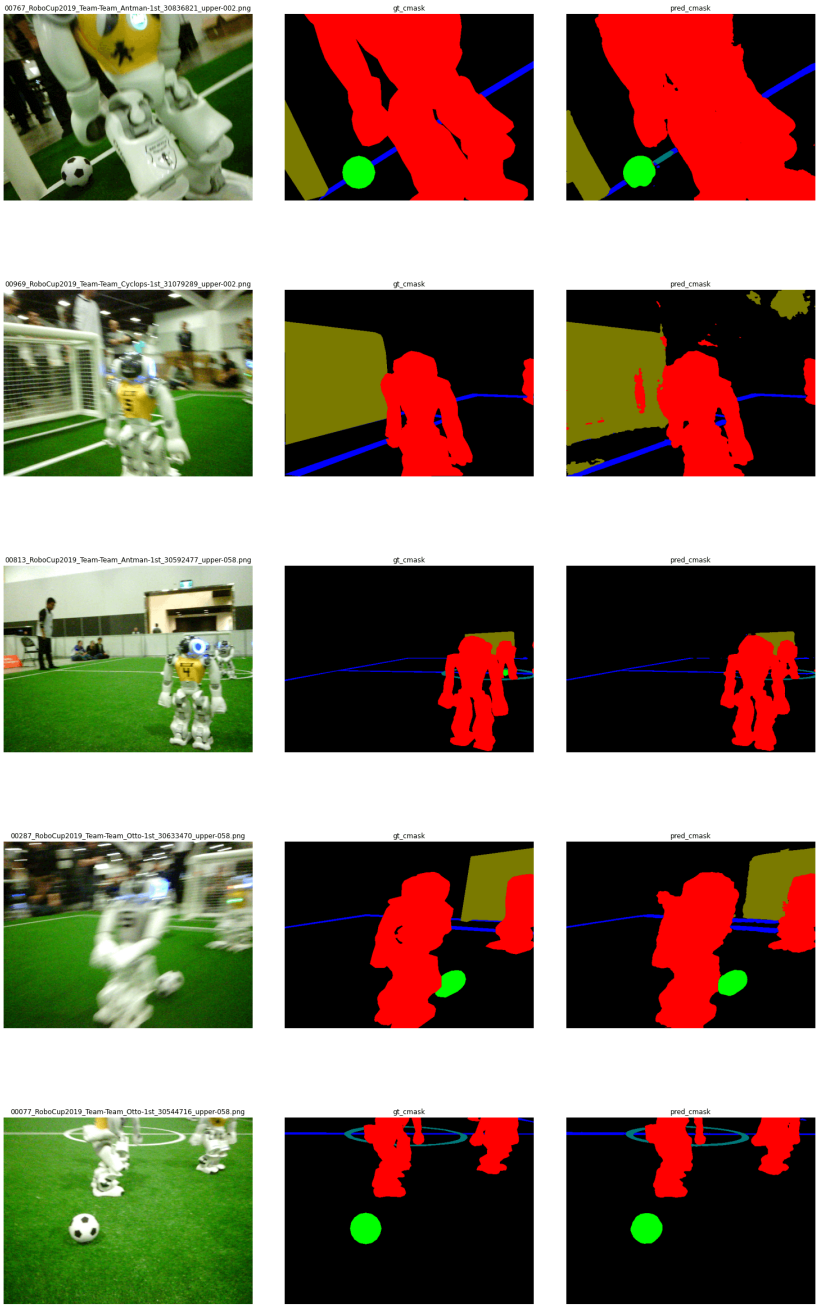


Abbildung A.2: Validierungsbilder aus dem Training mit nur manuell annotierten Bildern.

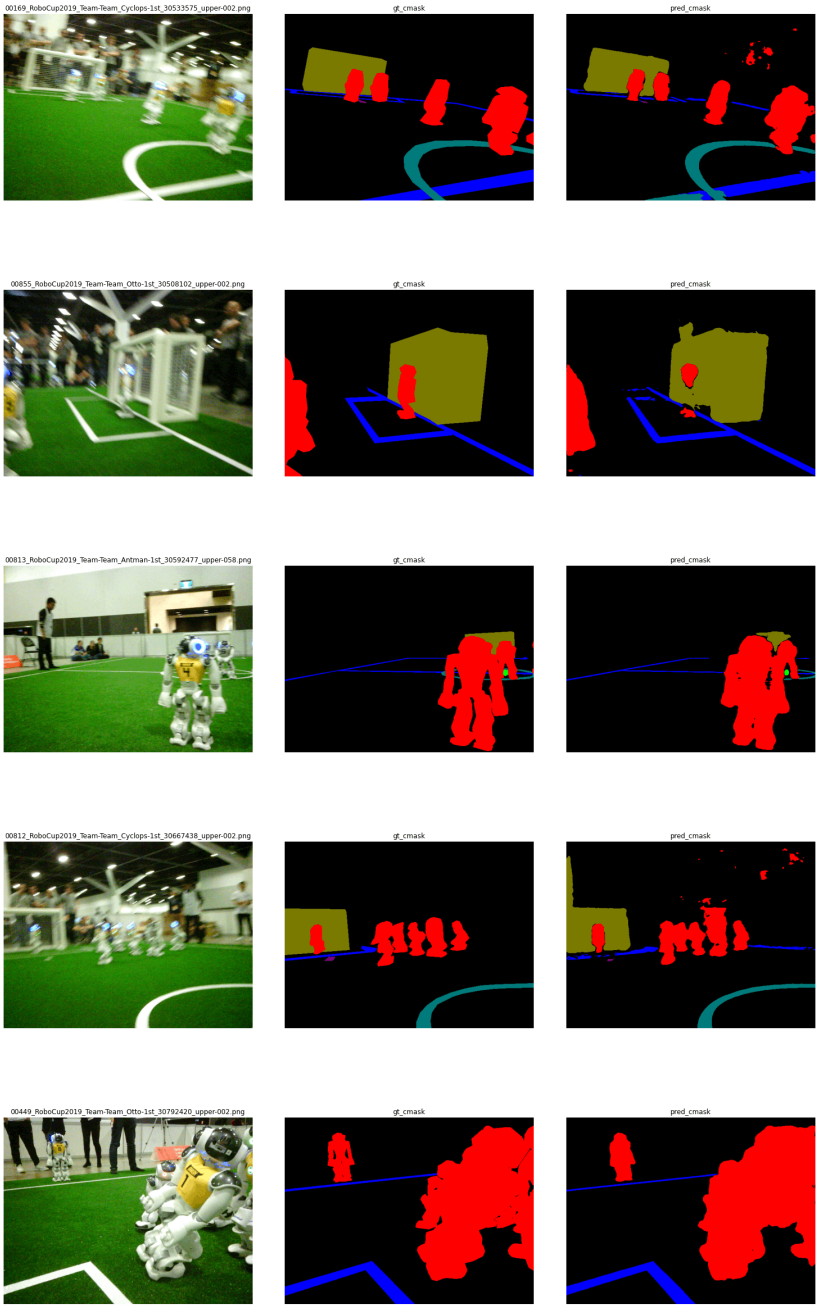


Abbildung A.3: Validierungsbilder aus dem Training mit nur automatisch annotierten Bildern.

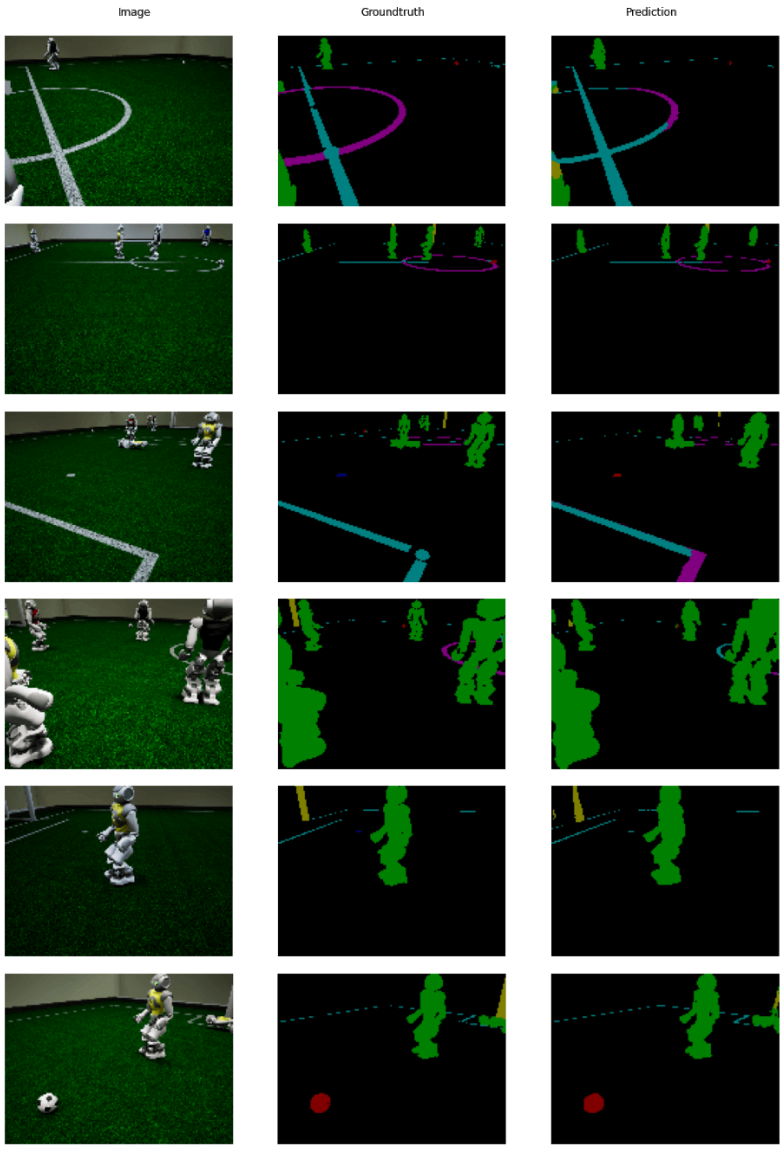


Abbildung A.4: Test Predictions von U-Net Model 1 auf dem Unreal Engine Datensatz.

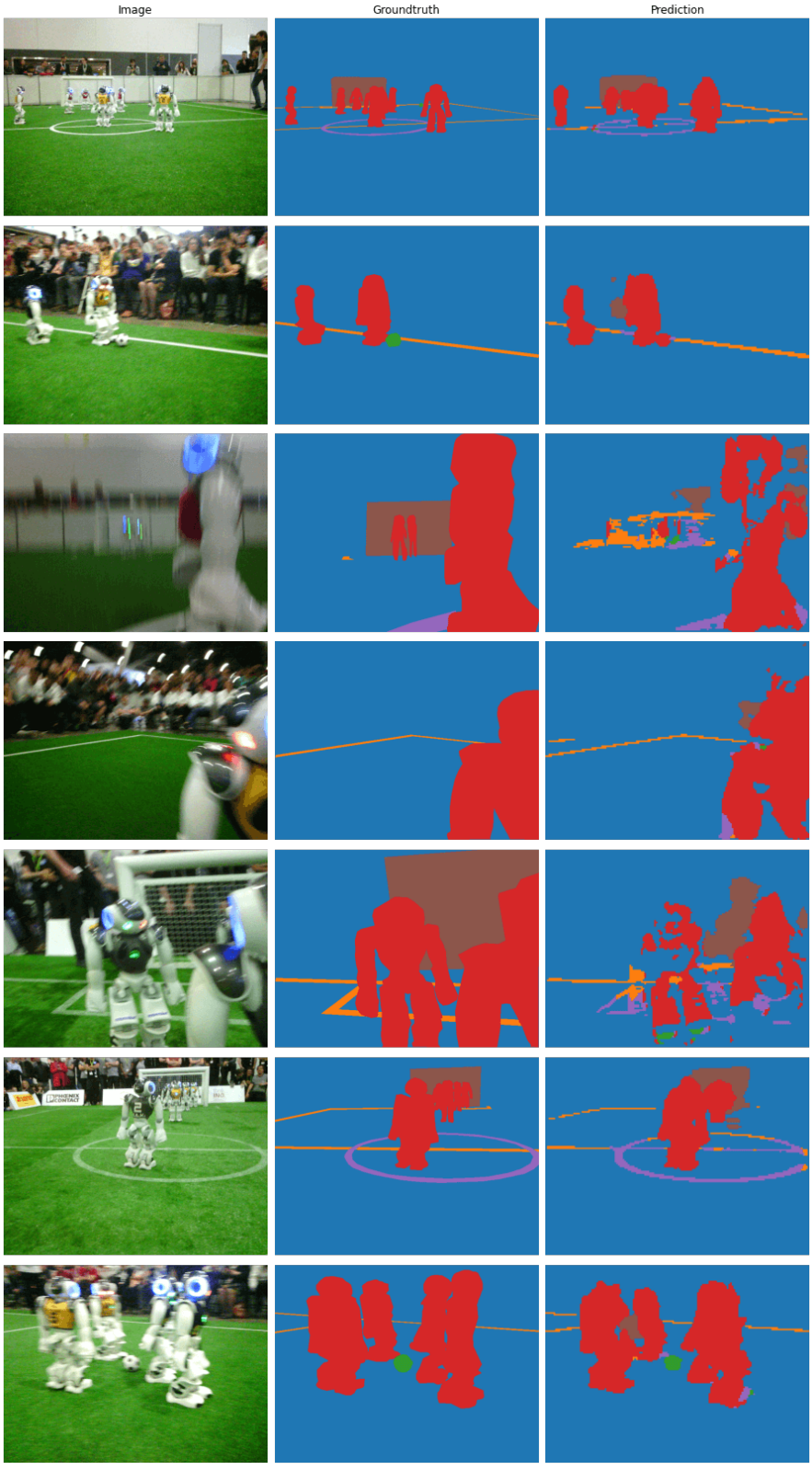


Abbildung A.5: Test Predictions von U-Net Model 1.

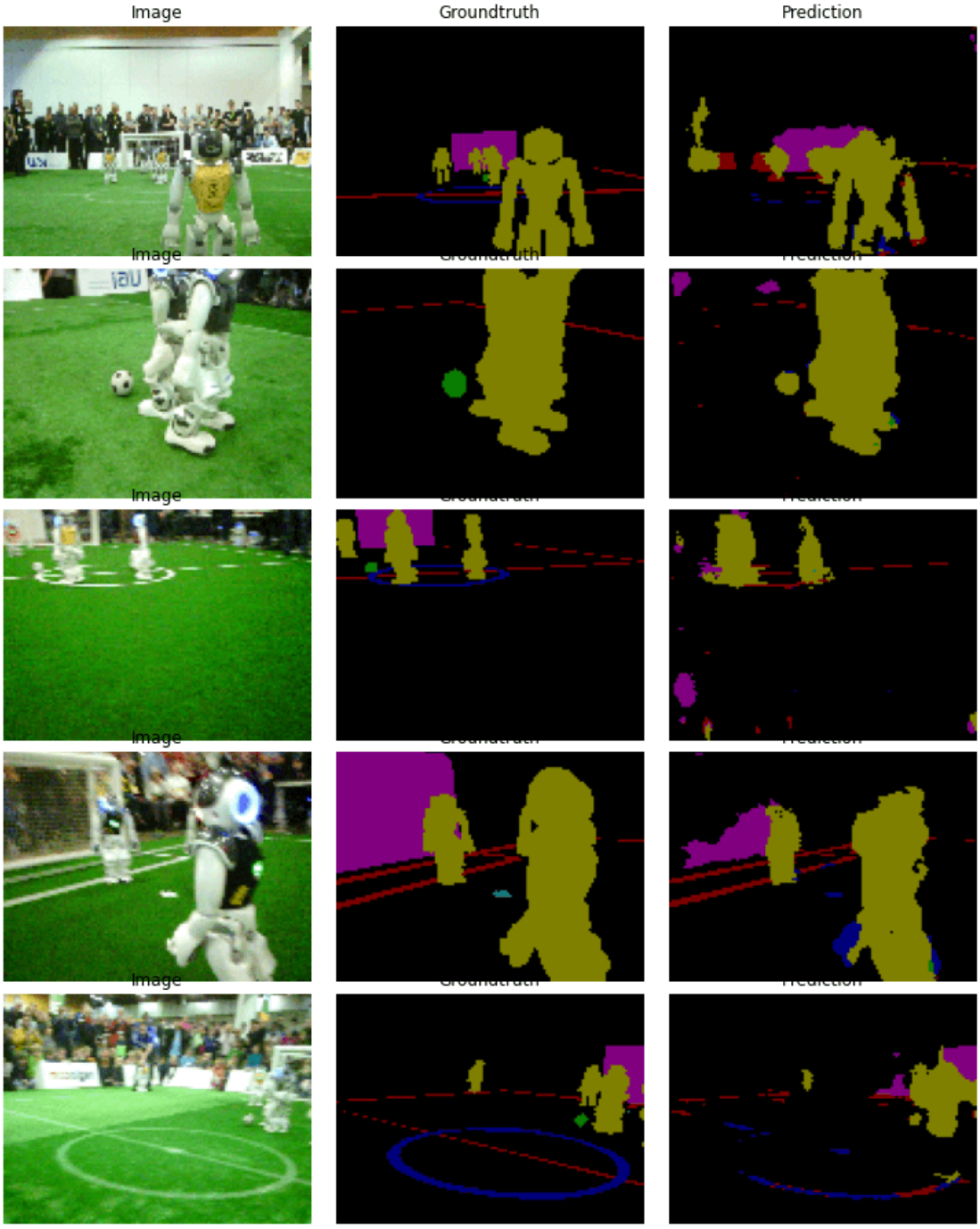


Abbildung A.6: Test Predictions von U-Net Model 2.

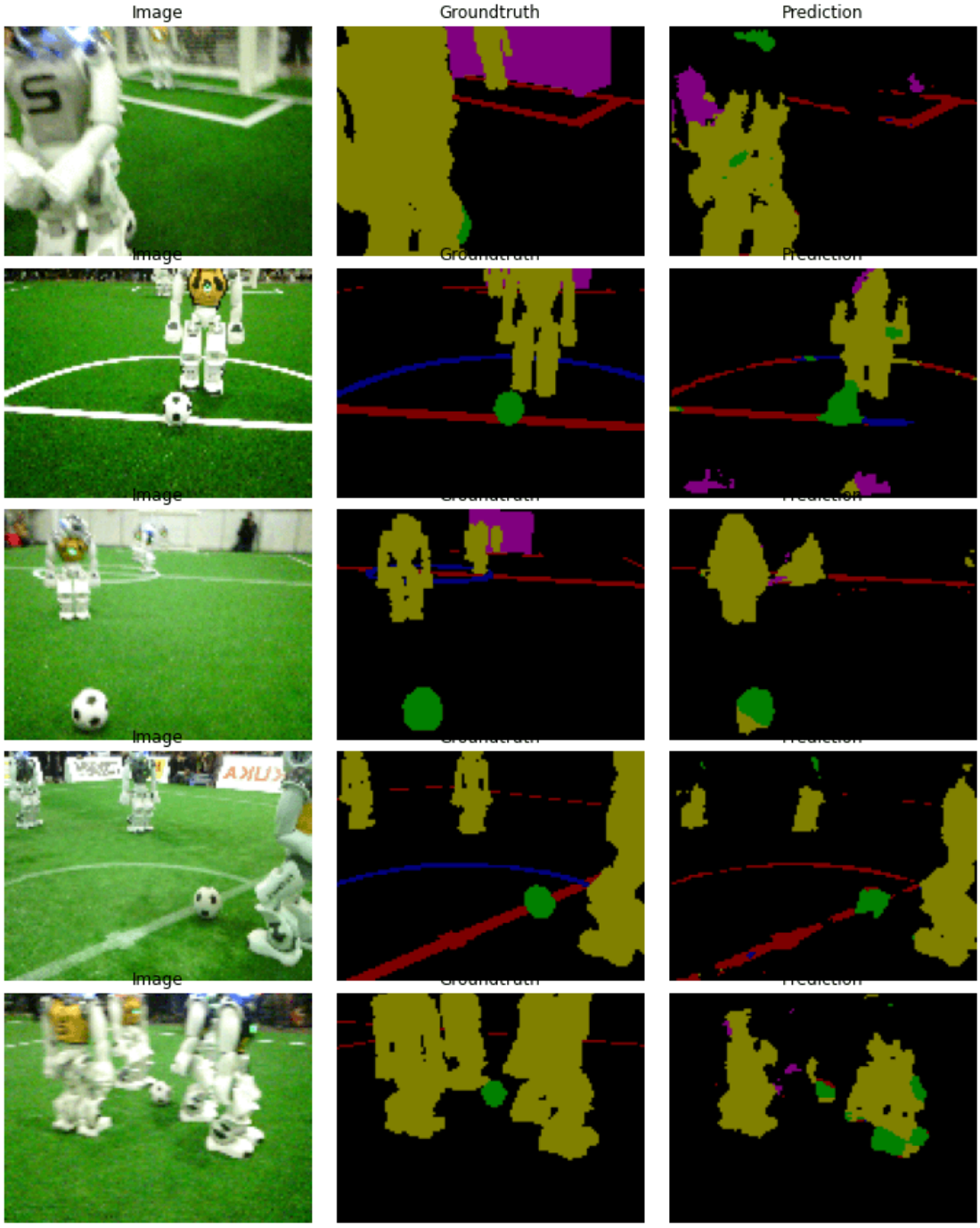


Abbildung A.7: Test Predictions von U-Net Model 2, trainiert mit zusätzlichen Bällen.

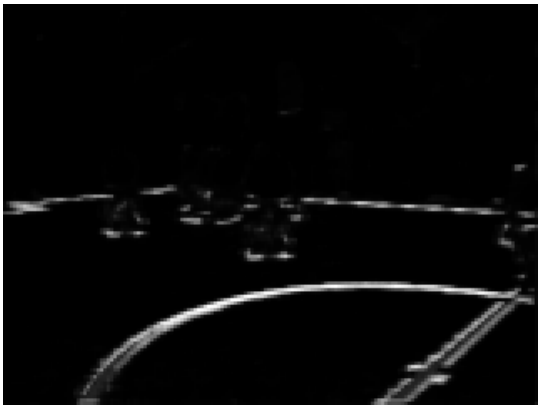
Linienerkennung bei möglichst hoher Bildauflösung

Im Folgenden sollen einige Inferenzen der in Abschnitt 2.3.5 auf Seite 30 vorgestellten Netz-Kandidaten gezeigt werden. Hierzu wurden vier beispielhafte Bilder gewählt, welche sich in den Testdaten befinden und somit nicht für das Training der Netz-Kandidaten verwendet wurden. Aufgrund der unterschiedlichen Ausgabegrößen der Netze wurden die Bilder auf eine einheitliche Größe skaliert.

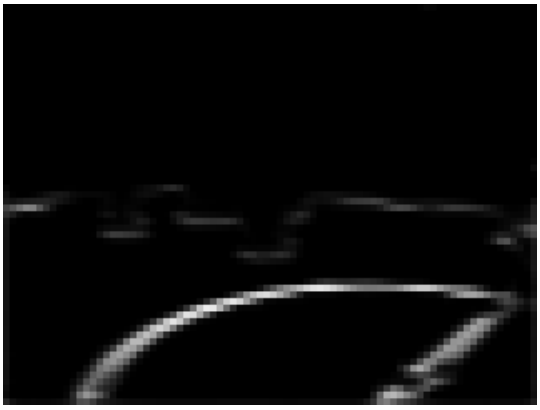
(a) Originalbild

(b) Ground truth

(c) Ergebnis NetCand01



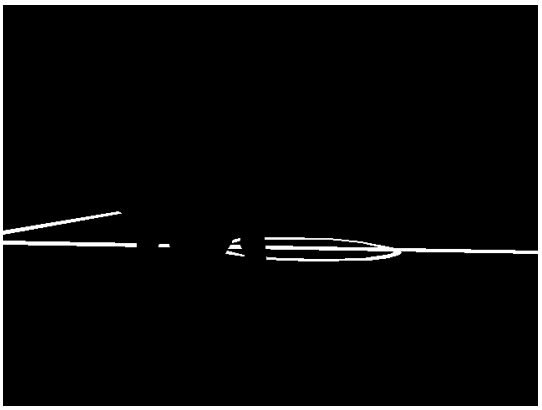
(d) Ergebnis NetCand02



(e) Ergebnis NetCand03

Abbildung A.8: Inferenz-Vergleich aller Test-Kandidaten für Bild mit starken Linien und stabilen Bildparametern.

(a) Originalbild



(b) Ground truth

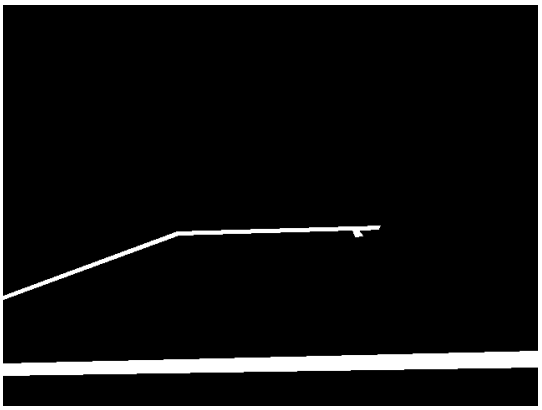
(c) Ergebnis NetCand01

(d) Ergebnis NetCand02

(e) Ergebnis NetCand03

Abbildung A.9: Inferenz-Vergleich aller Test-Kandidaten für Bild mit schwachen Linien und stabilen Bildparametern.

(a) Originalbild



(b) Ground truth

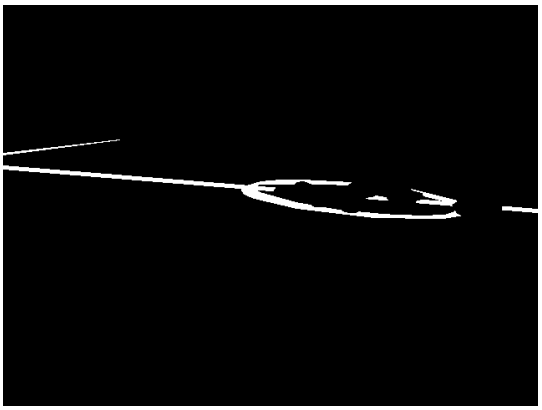
(c) Ergebnis NetCand01

(d) Ergebnis NetCand02

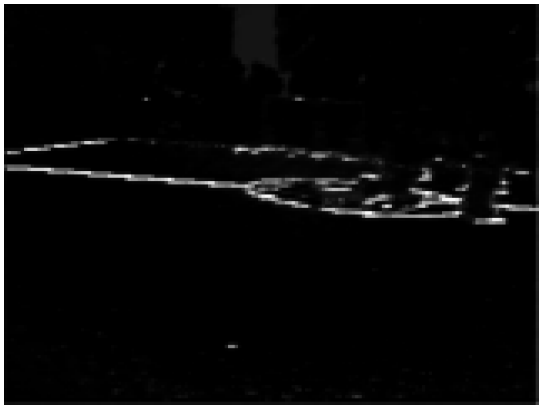
(e) Ergebnis NetCand03

Abbildung A.10: Inferenz-Vergleich aller Test-Kandidaten für Bild mit starken Linien und starker Bewegungsunschärfe.

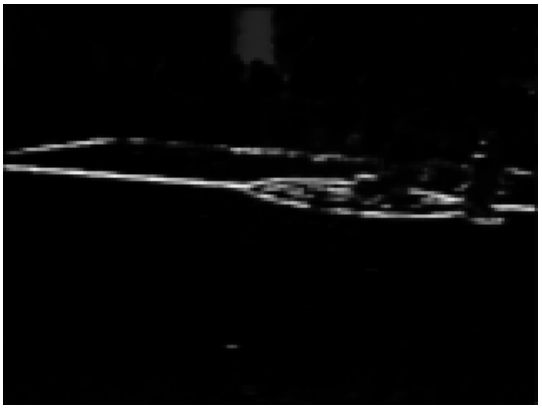
(a) Originalbild



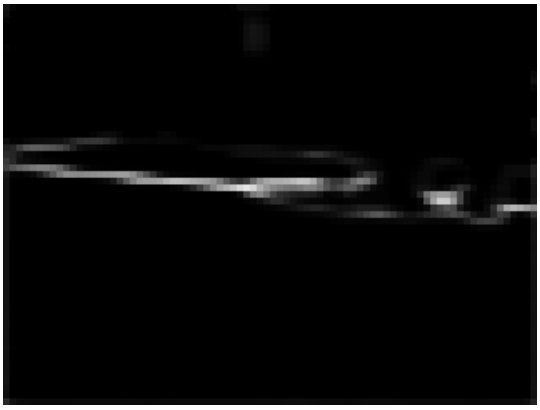
(b) Ground truth



(c) Ergebnis NetCand01



(d) Ergebnis NetCand02



(e) Ergebnis NetCand03

Abbildung A.11: Inferenz-Vergleich aller Test-Kandidaten für Bild mit starken Linien, starker Bewegungsunschärfe und mäßiges Bildrauschen.

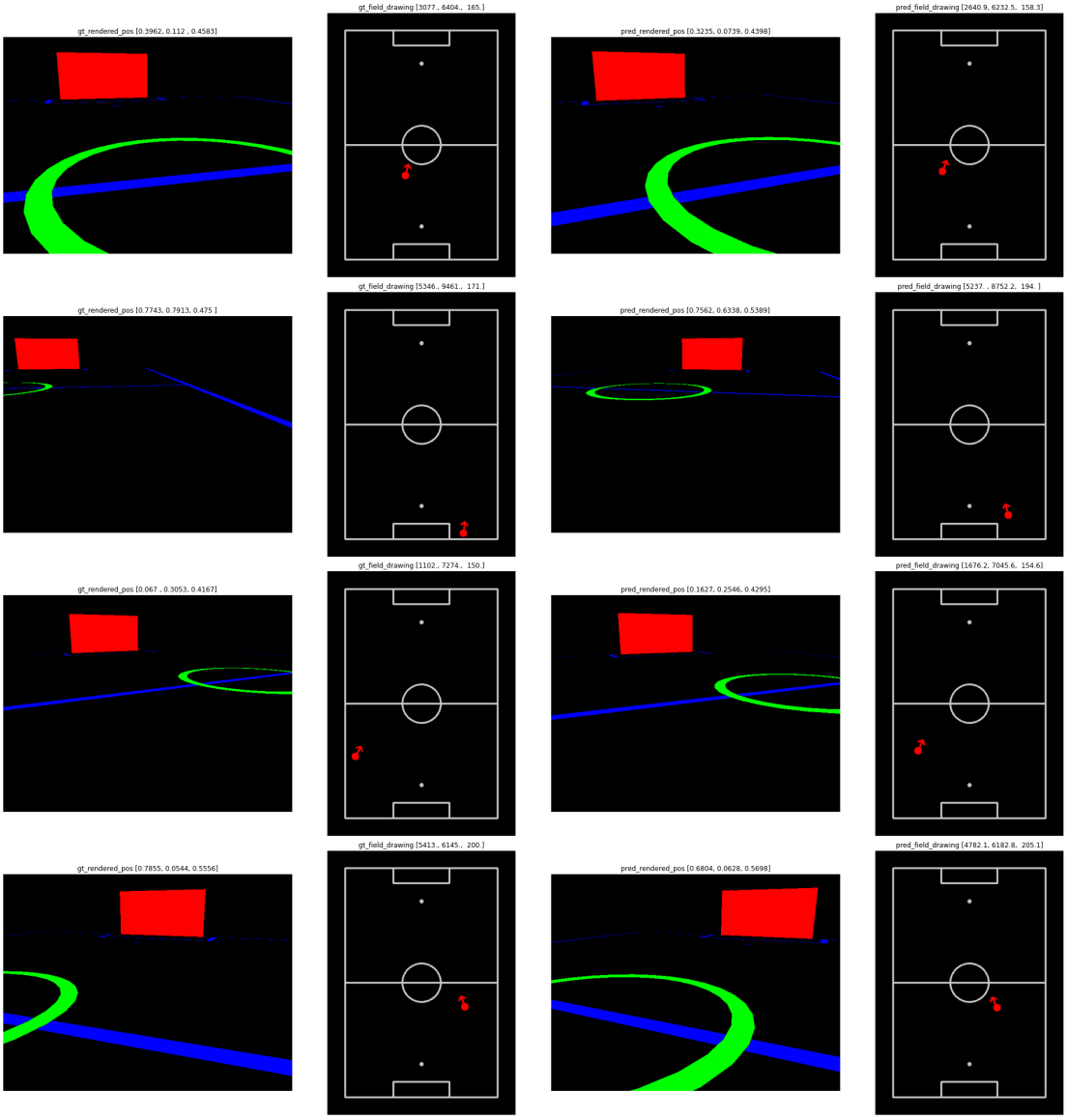


Abbildung A.12: Vier Positionsschätzungen des Modells aus dem Validierungsdatensatz. Von links nach rechts: die Sicht des Roboters aus der echten Position, die Position des Roboters, die Sicht des Roboters aus der geschätzten Position und die geschätzte Position.

B Tabellen

Ressourcenbeschränkte Objekterkennung

	Ebenentyp	Input Shape	Ebenenkonfiguration	Output Shape
1	Convolutional	320x240x3	filters: 8, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	320x240x8
2	Max Pooling	320x240x8	size: 2, stride: 2	160x120x8
3	Convolutional	160x120x8	filters: 10, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	160x120x10
4	Max Pooling	160x120x10	size: 2, stride: 2	80x60x10
5	Convolutional	80x60x10	filters: 14, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	80x60x14
6	Max Pooling	80x60x14	size: 2, stride: 2	40x30x14
7	Convolutional	40x30x14	filters: 18, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	40x30x18
8	Max Pooling	40x30x18	size: 2, stride: 2	20x15x18
9	Convolutional	20x15x18	filters: 22, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	20x15x22
10	Convolutional	20x15x22	filters: 26, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	20x15x26
11	Convolutional	20x15x26	filters: 22, size: 1x1, stride: 1, padding: 1, batchnormalize, leaky relu	20x15x22
12	Concatenate		Layers: 11, 9	20x15x44
13	Convolutional	20x15x44	filters: 24, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	20x15x24
14	Convolutional	20x15x24	filters: 21, size: 1x1, stride: 1, padding: 1, linear	20x15x21
Loss Function: Cross-Entropy				
Total MFLOPS: 91				

Tabelle B.1: Architektur von YOLOv3 320x240.

	Ebenentyp	Input Shape	Ebenenkonfiguration	Output Shape
1	Convolutional	240x180x3	filters: 8, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	240x180x8
2	Max Pooling	240x180x8	size: 2, stride: 2	120x90x8
3	Convolutional	120x90x8	filters: 12, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	120x90x12
4	Max Pooling	120x90x12	size: 2, stride: 2	60x45x12
5	Convolutional	60x45x12	filters: 16, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	60x45x16
6	Max Pooling	60x45x16	size: 2, stride: 2	30x23x16
7	Convolutional	30x23x16	filters: 20, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	30x23x20
8	Max Pooling	30x23x20	size: 2, stride: 2	15x12x20
9	Convolutional	15x12x20	filters: 24, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	15x12x24
10	Convolutional	15x12x24	filters: 28, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	15x12x28
11	Convolutional	15x12x28	filters: 32, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	15x12x32
12	Convolutional	15x12x32	filters: 20, size: 3x3. stride: 1, padding: 1, batchnormalize, leaky relu	15x12x20
13	Convolutional	15x12x20	filters: 24, size: 1x1, stride: 1, padding: 1, batchnormalize, leaky relu	15x12x24
14	Concatenate		Layers: 13, 9	15x12x48
15	Convolutional	20x15x48	filters: 24, size: 3x3, stride: 1, padding: 1, batchnormalize, leaky relu	15x12x24
16	Convolutional	20x15x24	filters: 21, size: 1x1, stride: 1, padding: 1, linear	15x12x21
Loss Function: Cross-Entropy				
Total MFLOPS: 64				

Tabelle B.2: Architektur von YOLOv3 240x180.

Layer	Output Shape
Input	(160, 120, 3)
3x3 2D conv. w/ ReLU, BatchNormalization	(160, 120, 32)
3x3 2D conv. w/ ReLU, BatchNormalization	(160, 120, 32)
| 2x2 Maxpooling 2D	(80, 60, 32)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(80, 60, 48)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(80, 60, 48)
| | 2x2 Maxpooling 2D	(40, 30, 48)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(40, 30, 64)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(40, 30, 64)
| | | 2x2 Maxpooling 2D	(20, 15, 64)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(20, 15, 80)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(20, 15, 80)
| | | | 2x2 Maxpooling 2D	(10, 7, 80)
| | | | 3x3 2D conv. w/ ReLU, BatchNormalization	(10, 7, 96)
| | | | 3x3 2D conv. w/ ReLU, BatchNormalization	(10, 7, 96)
| | | | 2x2 Upsampling 2D	(20, 15, 96)
| | | Concatenate	(20, 15, 176)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(20, 15, 96)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(20, 15, 96)
| | | 2x2 Upsampling 2D	(40, 30, 96)
| | Concatenate	(40, 30, 160)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(40, 30, 80)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(40, 30, 80)
| | 2x2 Upsampling 2D	(80, 60, 80)
| Concatenate	(80, 60, 128)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(80, 60, 64)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(80, 60, 64)
| 2x2 Upsampling 2D	(160, 120, 64)
Concatenate	(160, 120, 96)
3x3 2D conv. w/ ReLU, BatchNormalization	(160, 120, 48)
3x3 2D conv. w/ ReLU, BatchNormalization	(160, 120, 48)
1x1 2D conv. w/ Softmax	(160, 120, 7)
Loss function: Binary crossentropy Parameter 948279	

Tabelle B.3: U-Net Model 1.

Layer	Output Shape
Input	(128, 96, 3)
3x3 2D conv. w/ ReLU, BatchNormalization	(128, 96, 8)
3x3 2D conv. w/ ReLU, BatchNormalization	(128, 96, 8)
| 2x2 Maxpooling 2D	(64, 48, 8)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(64, 48, 12)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(64, 48, 12)
| | 2x2 Maxpooling 2D	(32, 24, 12)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(32, 24, 16)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(32, 24, 16)
| | | 2x2 Maxpooling 2D	(16, 12, 16)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(16, 12, 20)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(16, 12, 20)
| | | | 2x2 Maxpooling 2D	(8, 6, 20)
| | | | 3x3 2D conv. w/ ReLU, BatchNormalization	(8, 6, 24)
| | | | 3x3 2D conv. w/ ReLU, BatchNormalization	(8, 6, 20)
| | | | 2x2 Upsampling 2D	(16, 12, 20)
| | | Concatenate	(16, 12, 40)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(16, 12, 20)
| | | 3x3 2D conv. w/ ReLU, BatchNormalization	(16, 12, 16)
| | | 2x2 Upsampling 2D	(32, 24, 16)
| | Concatenate	(32, 24, 32)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(32, 24, 16)
| | 3x3 2D conv. w/ ReLU, BatchNormalization	(32, 24, 12)
| | 2x2 Upsampling 2D	(64, 48, 12)
| Concatenate	(64, 48, 24)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(64, 48, 12)
| 3x3 2D conv. w/ ReLU, BatchNormalization	(64, 48, 8)
| 2x2 Upsampling 2D	(128, 96, 8)
Concatenate	(128, 96, 16)
3x3 2D conv. w/ ReLU, BatchNormalization	(128, 96, 8)
3x3 2D conv. w/ ReLU, BatchNormalization	(128, 96, 8)
1x1 2D conv. w/ Softmax	(128, 96, 7)
Loss function: Binary crossentropy Parameter 8459	

Tabelle B.4: U-Net Model 2 mit Leaky ReLUs.

Layer	Output Shape	Optionen
Input	(640, 480, 3)	
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 12)	2x2 Strides
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 12)	2x2 Dilation rate
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	2x2 Strides
3x3 Depth. sep. 2D conv. w/ Hard sigmoid	(160, 120, 1)	
2x2 Upsampling 2D	(320, 240, 1)	
2x2 Upsampling 2D	(640, 480, 1)	
Loss function: Binary crossentropy 392 Parameter - 48,7 MFLOPS - Laufzeit: 60 ms		

Tabelle B.5: Netz-Kandidat NetCand01 zur Linienerkennung, ausgelegt auf Originalbilder mit 640x320 Pixel.

Layer	Output Shape	Optionen
Input	(320, 240, 3)	
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 12)	
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 12)	2x2 Dilation rate
3x3 Depth. sep. 2D conv. w/ ReLU	(320, 240, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	2x2 Strides
3x3 Depth. sep. 2D conv. w/ Hard sigmoid	(160, 120, 1)	
2x2 Upsampling 2D	(320, 240, 1)	
2x2 Upsampling 2D	(640, 480, 1)	
Loss function: Binary crossentropy 392 Parameter - 47,4 MFLOPS - Laufzeit: 53 ms		

Tabelle B.6: Netz-Kandidat NetCand02 zur Linienerkennung, ausgelegt auf Integralbilder mit 320x160 Pixel.

Layer	Output Shape	Optionen
Input	(160, 120, 3)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 12)	
1x1 Depth. sep. 2D conv. w/ ReLU	(160, 120, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 12)	
1x1 Depth. sep. 2D conv. w/ ReLU	(160, 120, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 16)	
3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	2x2 Strides
3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	
1x1 Depth. sep. 2D conv. w/ ReLU	(80, 60, 4)	
3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	
1x1 Depth. sep. 2D conv. w/ Hard sigmoid	(80, 60, 1)	
2x2 Upsampling 2D	(160, 120, 1)	
2x2 Upsampling 2D	(320, 240, 1)	
Loss function: Binary crossentropy 1.480 Parameter - 25,4 MFLOPS - Laufzeit: 23 ms		

Tabelle B.7: Netz-Kandidat NetCand03 zur Linienerkennung, ausgelegt auf 160x120 Pixel.

Layer	Output Shape	Optionen
Input	(160, 120, 3)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	
| 3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	
| 3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 8)	2x2 Strides
| | 3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	
| | 3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	
| | 3x3 Depth. sep. 2D conv. w/ ReLU	(40, 30, 16)	2x2 Strides
| | 3x3 Depth. sep. 2D conv. w/ ReLU	(40, 30, 24)	
| | 3x3 Depth. sep. 2D conv. w/ ReLU	(40, 30, 24)	
| | 3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 24)	
| | 2x2 Upsampling 2D	(80, 60, 24)	
| Concatenate	(80, 60, 32)	
| 3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	
| 3x3 Depth. sep. 2D conv. w/ ReLU	(80, 60, 16)	
| 2x2 Upsampling 2D	(160, 120, 16)	
Concatenate	(160, 120, 24)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	
3x3 Depth. sep. 2D conv. w/ ReLU	(160, 120, 8)	
1x1 Depth. sep. 2D conv. w/ Hard sigmoid	(160, 120, 1)	
Loss function: Binary crossentropy 4.780 Parameter - 60,4 MFLOPS		

Tabelle B.8: Netz-Kandidat NetCand04 zur Linienerkennung, reduzierte Version von [BBL19], ausgelegt auf Integralbilder mit 160x120 Pixel.

C Abkürzungsverzeichnis

AP Average Precision.

CNN Convolutional Neural Network.

COCO Common Objects in Context.

DCG Devils Code Generator.

DLA Deep Layer Aggregation.

DNN Deep Neural Network.

FPS Frames per second.

GEPP GoalEdgePointsPercept.

KI Künstliche Intelligenz.

mAP mean Average Precision.

ms Millisekunden.

NMS Non-Maximum Suppression.

NN neuronales Netz.

NPU Neural Processing Unit.

RoboCup Robot World Cup Initiative.

SPL Standard Platform League.

SSD Single Shot MultiBox Detector.

VOC Visual Object Classes.

YOLO You Only Look Once.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[BBL19] Jan Blumenkamp, Andreas Baude und Tim Laue. “Closing the Reality Gap with Unsupervised Sim-to-Real Image Translation for Semantic Segmentation in Robot Soccer”. In: (4. Nov. 2019). arXiv: 1911.01529v1 [cs.LG].

[Bro19] Justin Brooks. *Awesome-Dataset-Tools*. 2019. URL: https://github.com/jsbroks/awesome-dataset-tools (besucht am 31. 03. 2020).

[Bro20] Justin Brooks. *Coco-Annotator*. 2020. URL: https://github.com/jsbroks/coco-annotator (besucht am 31. 03. 2020).

[Bü+17] Yannick Bülter u. a. *B-Human: Team Report and Code Release 2017*. Techn. Ber. Deutsches Forschungszentrum für Künstliche Intelligenz, Universität Bremen, 6. Okt. 2017. URL: https://www.b-human.de/downloads/publications/2017/coderelease2017.pdf (besucht am 04. 04. 2020).

[Dai+17] Jifeng Dai u. a. “Deformable Convolutional Networks”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, S. 764–773.

[deG19] Max deGroot. *SSD: Single Shot MultiBox Object Detector, in PyTorch*. 2019. URL: https://github.com/amdegroot/ssd.pytorch (besucht am 28. 08. 2020).

[GBC16] Ian Goodfellow, Joshua Bengio und Aaron Courville. *Deep Learning*. MIT Press Ltd, 18. Nov. 2016. 800 S. ISBN: 0262035618. URL: https://www.ebook.de/de/product/26337726/ian_goodfellow_joshua_bengio_aaron_courville_deep_learning.html.

[Has18] Arne Hasselbring. *RoboCup SPL GameController - Update SPLStandardMessage for 2018*. 11. Feb. 2018. URL: https://github.com/bhuman/GameController/commit/9076b88c5d97a0336e9ffdf6af167e3dcc8b40c0 (besucht am 15. 08. 2020).

[HC18] Trent Houliston und Stephan K Chalup. “Visual mesh: Real-time object detection using constant sample density”. In: *Robot World Cup*. Springer. 2018, S. 45–56.

[Hes+17] Timm Hess u. a. “Large-scale Stochastic Scene Generation and Semantic Annotation for Deep Convolutional Neural Network Training in the RoboCup SPL”. In: *RoboCup 2017: Robot World Cup XXI, LNAI*. Springer, 2017.

[Liu+16] Wei Liu u. a. “SSD: Single Shot MultiBox Detector”. In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, S. 21–37. DOI: 10.1007/978-3-319-46448-0_2.

[Mat20] Matthew Matl. *pyrender*. 2020. URL: https://github.com/mmatl/pyrender (besucht am 21. 08. 2020).

[MC18] MaybeShewill-CV. *LaneNet-Lane-Detection*. 2018. URL: https://github.com/MaybeShewill-CV/lanenet-lane-detection (besucht am 31. 03. 2020).

[Moo20] Arne Moos. "Real time capable convolutional neural networks with recurrent structures and meta information for continuous object detection in the context of RoboCup". Masterarbeit. TU Dortmund, 2020.

[Naoa] *Nao Devils TU Dortmund by NaoDevils*. Robotics Research Institute at TU Dortmund University. URL: https://naodevils.de/ (besucht am 27. 08. 2020).

[Naob] *NAO the humanoid and programmable robot*. SoftBank Robotics. URL: https://www.softbankrobotics.com/emea/en/nao (besucht am 27. 08. 2020).

[Nev+18] Davy Neven u. a. "Towards End-to-End Lane Detection: an Instance Segmentation Approach". In: (15. Feb. 2018). arXiv: 1802.05591v1 [cs.CV].

[Red+15] Joseph Redmon u. a. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[RF18a] Joseph Redmon und Ali Farhadi. *YOLO: Real-Time Object Detection*. 2018. URL: https://pjreddie.com/darknet/yolo/ (besucht am 27. 08. 2020).

[RF18b] Joseph Redmon und Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018). arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767.

[RFB15] Olaf Ronneberger, Philipp Fischer und Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[Rob] *RoboCup Federation official website*. RoboCup Federation, Inc. URL: https://www.robocup.org/ (besucht am 27. 08. 2020).

[Rul] *RoboCup Standard Platform League (NAO) Rule Book*. RoboCup Technical Committee. URL: https://collaborating.tuhh.de/HULKs/robocup_tc_public/raw/master/SPL-Rules_2020.pdf (besucht am 28. 09. 2020).

[Sch+16] Ingmar Schwarz u. a. "A Robust And Calibration-Free Vision System for Humanoid Soccer Robots". In: *Proceedings RoboCup 2015 International Symposium*. Hefei, China, 2016.

[Sob14] Irwin Sobel. "An Isotropic 3x3 Image Gradient Operator". In: *Presentation at Stanford A.I. Project 1968* (Feb. 2014).

[Spl] *RoboCup Standard Platform League*. RoboCup Federation. URL: https://spl.robocup.org/ (besucht am 27. 08. 2020).

[Syn] Synced. *Huawei 7nm Kirin 810 Beats Snapdragon 855 and Kirin 980 on AI Benchmark Test*. URL: https://medium.com/syncedreview/huawei-7nm-kirin-810-beats-snapdragon-855-and-kirin-980-on-ai-benchmark-test-af31996fb10e (besucht am 27. 08. 2020).

[VJ] P. Viola und M. Jones. "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. IEEE Comput. Soc. DOI: 10.1109/cvpr.2001.990517.

[Wil09] Florian Wilmshöver. "Multi-Modell Kalman Filter zur Lokalisierung im Roboterfussball". In: *Technischer Report, Institut für Roboterforschung, TU Dortmund* (2009).

[Wu+19] Yuxin Wu u. a. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.

[xua19] xuannianz. *keras-CenterNet*. 2019. URL: https://github.com/xuannianz/keras-CenterNet (besucht am 28. 08. 2020).

[Yak19] Pavel Yakubovskiy. *Segmentation Models*. https://github.com/qubvel/segmentation_models. 2019.

[Yu+18] Fisher Yu u. a. "Deep Layer Aggregation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, S. 2403–2412.

[Zho19] Xingyi Zhou. *Objects as Points*. 2019. URL: https://github.com/xingyizhou/CenterNet (besucht am 28. 08. 2020).

[ZWK19] Xingyi Zhou, Dequan Wang und Philipp Krähenbühl. "Objects as Points". In: *arXiv preprint arXiv:1904.07850* (2019).