Universität Bremen

Bachelorarbeit

# Entwicklung eines Hidden-Markov-Models zur Rauschunterdrückung bei der Segmentierung vorklassifizierter Farbbilder

**Jannik Heyen**
Matrikelnummer: 4259088

10. Januar 2019

Informatik
Fachbereich 3
Universität Bremen

| | |
|---|---|
| Erstgutachter: | Dr. Tim Laue |
| Zweitgutachter: | Prof. Dr. Thomas Schneider |

# Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt und nicht anderweitig zu Prüfungszwecken vorgelegt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, 10. Januar 2019

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Jannik Heyen)

# Inhaltsverzeichnis


**1 Einleitung** **1**
- 1.1 RoboCup und Standard Platform League . . . 1
- 1.2 B-Human . . . 1
- 1.3 Der NAO . . . 2
- 1.4 Motivation und Ziele . . . 2

**2 Die Bildverarbeitung im B-Human-System** **3**
- 2.1 Scanlines . . . 3
- 2.2 Weitere Schritte in der Bildverarbeitung . . . 4

**3 Verwandte Arbeiten** **5**

**4 Grundlagen** **7**
- 4.1 Gaussian-Mixture-Model . . . 7
- 4.2 Hidden-Markov-Model . . . 7
- 4.3 Forward-Backward-Algorithmus . . . 9
  - 4.3.1 Forward . . . 9
  - 4.3.2 Backward . . . 10
  - 4.3.3 Gesamtergebnis . . . 10
  - 4.3.4 Optimierungen . . . 11
- 4.4 Baum-Welch-Algorithmus . . . 12

**5 Implementierung** **13**
- 5.1 Verwendung des Gaussian-Mixture-Models . . . 13
- 5.2 Forward-Backward-Algorithmus . . . 16
- 5.3 Baum-Welch-Algorithmus . . . 19

**6 Evaluation** **25**
- 6.1 Evaluation des Montréal-Logs . . . 27
- 6.2 Evaluation des Logs aus MZH 1400 . . . 28
- 6.3 Evaluation des Logs aus dem Foyer des MZH . . . 30
- 6.4 Evaluation der Zeit . . . 31

**7 Fazit** **33**

**8 Ausblick** **35**

**Anhang** **37**

**Literaturverzeichnis** **39**

# 1. Einleitung

## 1.1 RoboCup und Standard Platform League

Der RoboCup ist eine jährlich stattfindende Veranstaltung mit dem Ziel, die Forschung im Bereich künstliche Intelligenz, Bildverarbeitung und Robotik voranzutreiben. Dort treffen sich Teams aus aller Welt, um sich in Wettbewerben zu messen, die von ihnen programmierte oder entwickelte Roboter austragen. Die meisten Wettbewerbe haben einen starken Praxisbezug und somit müssen die Roboter mit Bedingungen aus der Realität umgehen können.

Ein sehr interessanter Wettbewerb des RoboCups ist die Standard Platform League (SPL). Dort spielen Mannschaften bestehend aus fünf kleinen humanoiden Robotern, wie sie in Abbildung 1.1 zu sehen sind, autonom Fußball und kämpfen um den Weltmeistertitel. Die Roboter für diesen Wettbewerb werden von den Teilnehmern bei dem Herrsteller SoftBank Robotics gekauft. Da jedes Team mit der gleichen Hardware spielt, geht der Weltmeistertitel an die Mannschaft, die die beste Software geschrieben hat und dadurch auch den erfolgreichsten Fußball gespielt hat.

Abbildung 1.1: Zwei NAOs kämpfen um den Ball (GermanOpen 2018).

Die RoboCup Federation, die den RoboCup organisiert, hat als Ziel ausgegeben, 2050 gegen den amtierenden menschlichen Fußballweltmeister spielen und gewinnen zu wollen. Diesem Ziel möchte man jedes Jahr einen Schritt näher kommen, sodass zu jedem RoboCup die offiziellen Regeln denen des menschlichen Fußballs angenähert werden. [RoboCup Federation, 2016] So wird zum Beispiel das Licht, welches ursprünglich in den Hallen konstant geschienen hat, schrittweise durch natürliches Licht ersetzt. Daher müssen die Roboter auch spielen können, wenn sich das Wetter und damit die Lichtverhältnisse während des Spiels schlagartig ändern.

## 1.2 B-Human

B-Human ist ein Projekt der Universität Bremen in Zusammenarbeit mit dem Deutschen Forschungszentrum für Künstliche Intelligenz. Das Team, hauptsächlich bestehend aus Informatikstudenten, ist seit 2008 Teilnehmer der SPL und nimmt seitdem jährlich an der deutschen Meisterschaft (GermanOpen) und der Weltmeisterschaft, dem RoboCup, teil. Mit sechs Weltmeistertiteln und achtmaligem Gewinn der GermanOpen ist B-Human eines der erfolgreichsten Teams der Welt in diesem Wettbewerb. [B-Human, 2018] Diese Bachelorarbeit findet im Kontext des Projekts B-Human statt und wird als Teil des Frameworks implementiert.

## 1.3 Der NAO

Der ca. 58cm große Roboter, mit dem alle Standard Platform League Teams spielen, heißt NAO und wird von Softbank Robotics hergestellt. Außerdem ist er neben diversen Gelenken und Motoren mit zwei Kameras ausgestattet, die in Echtzeit Informationen an den eingebauten 1,6 Ghz-taktenden Prozessor senden. Die eine befindet sich in der Stirn des NAO und ist minimal nach unten gerichtet (1,2°). Die zweite Kamera befindet sich dort, wo man den Mund des Roboters erwarten könnte. Sie ist um 39,7° nach unten ausgerichtet und zeichnet den Bereich unmittelbar vor dem NAO auf. Beide Kameras nehmen maximal mit einer Auflösung von 1280x960 bei 30 Bildern pro Sekunde auf. [SoftBank Robotics Europe, 2017]

Abbildung 1.2: Die neue Generation des NAO[1].

Der Prozessor in Kombination mit dem Arbeisspeicher von 1GB sorgen dafür, dass ein Programm, sollte es im Spiel laufen, sehr schnell und optimiert sein muss, um echtzeitfähig auf dem NAO zu sein. Die neueste Generation (V6) ist deutlich leistungsfähiger, wurde bisher jedoch noch auf keinem Wettbewerb eingesetzt.

## 1.4 Motivation und Ziele

Wenn ein Roboter Fußball spielen soll, muss er wichtige Merkmale des Spiels wie Spieler, Bälle und Linien erkennen können. Eine Kamera ermöglicht dies, doch während der Mensch zum Beispiel ein Fußballfeld problemlos als grün erkennt, kann ein Computerprogramm in Verbindung mit einer Kamera durchaus Schwierigkeiten haben. Wenn zum Beispiel Licht an einer kleinen Position des Feldes reflektiert wird, erscheinen einige Pixel in diesem Bereich auf dem Kamerabild sehr hell, mitunter sogar weiß aus. Der Grund für die unterschiedliche Wahrnehmung ist, dass der Mensch Vorkenntnisse hat und weiß, dass das Feld Grün ist. Außerdem kann das menschliche Augen kleine farbliche Unregelmäßigkeiten ignorieren. Damit eine Kamera ein solches Rauschen auch ignorieren kann, müssen passende Algorithmen gefunden werden, die diese Aufgabe erledigen. *Hidden-Markov-Models* (HMM) haben sich im Bereich Rauschunterdrückung bereits beweisen können und werden unter anderem für diesen Zweck in der Sprachverarbeitung verwendet (z.B. [Schuller u. a., 2003]). Der Einsatz dieser Technologie im Bereich Echtzeitbildverarbeitung soll mit dieser Bachelorarbeit anhand fußballspielender Roboter untersucht werden. Hierfür soll eine neue Klasse in der Bildverarbeitung im B-Human-System geschrieben werden, welche ein HMM nutzt und die Kamerabilder weitesgehend vom Bildrauschen befreit. Um auf dynamisches Licht reagieren zu können, soll als Eingabe Informationen über die Farbe der Pixel aus einem *Gaussian-Mixture-Model* (GMM) genutzt werden. Für einen Praxiseinsatz sollte das System weiterhin echtzeitfähig sein.

[1] https://www.softbankrobotics.com/emea/themes/custom/softbank/images/full-nao.png

# 2. Die Bildverarbeitung im B-Human-System

Die Grundlage der Bildverarbeitung sind die von den zwei Kamaras gestellten Bilder. Die obere Kamera löst die Bilder bei B-Human mit 640 x 480 Pixeln auf und liefert 30 Bilder pro Sekunde. Zwar kann die Kamera Bilder auch mit einer höheren Auflösung aufnehmen, dies würde aber Rechenzeit kosten. Da sich Dinge auch im Roboterfußball schnell ändern, hat man sich für erstere Variante entschieden. Die untere Kamera löst mit halber Auflösung ($320 \times 240$ Pixel) auf, da sie durch ihre Perspektive nur den Bereich direkt vor den Füßen des NAO aufnimmt. Dort stellen sich alle Objekte groß dar, sodass hier Rechenzeit gespart wird. Parallele Verarbeitung der Bilder ist technisch nicht möglich, sodass Bilder von oberer und unterer Kamera getrennt analysiert werden. Diese Bilder werden im nächsten Schritt der Bildverarbeitung mit Hilfe der sogenannten *Scanlines* analysiert. [Röfer u. a., 2017]

## 2.1 Scanlines

*Scanlines* bezeichnen im B-Human-System Reihen von Pixeln, die genau einen Pixel breit sind und sich nach einem vorgegebenen Muster horizontal oder vertikal durch das Bild ziehen. Dieses Muster sorgt dafür, dass die *Scanlines* im oberen Teil deutlich engmaschiger liegen, da hier aufgrund der Perspektive mehr Informationen auf kleinem Raum abgebildet werden. Dieses Muster wird in der sogenannten *Scangrid* beschrieben und ist, zusammen mit den wichtigsten *Scanlines*, in Abbildung 2.1 zu sehen. Unten im Bild sind Objekte tendenziell größer, sodass die *Scanlines* hier etwas mehr Abstand zueinander haben. Die *Scanlines*, die horizontal durch das Bild verlaufen heißen *ColorScanlinesHorizontal*. Vertikal verlaufende *Scanlines* werden *ColorScanlinesVertical* genannt. Eine *Scanline* ist in Regionen aufgeteilt, welche jeweils einer Farbklasse zugeordnet ist. B-Human kennt vier Farbklassen: Weiß, Grün, Schwarz und None, einer Restklasse, der alle restlichen Farben zugeordnet werden.

Die dritte Art von *Scanlines* sind die *ColorScanlinesVerticalClipped*. Sie sind wie die *ColorScanlinesVertical* ebenfalls vertikal angeordnet. Im oberen Teil des Bildes sind jedoch zusätzlich kleinere, kürzere *Scanlines*, wodurch dort deutlich mehr Fläche abgedeckt wird, als mit den normalen vertikalen *Scanlines*. Außerdem wird jede *ColorScanlineVerticalClipped* an der Grenze des Fußballfeldes abgetrennt. [Röfer u. a., 2017]

Dieses Vorgehen hat den Vorteil, dass das Bild grob analysiert werden kann, ohne wirklich jeden Pixel zu verwerten. Durch die Verteilung der *Scanlines* gehen trotzdem kaum Informationen über mögliche Objekte im Bild verloren, da sie meistens von mindestens einer *Scanline* abgedeckt werden.

In dieser Bachelorarbeit werde ich diese drei Arten von *Scanlines* berechnen. Genauer bedeutet dies, dass die Einteilung einer *Scanline* in Regionen mit einer zugehörigen

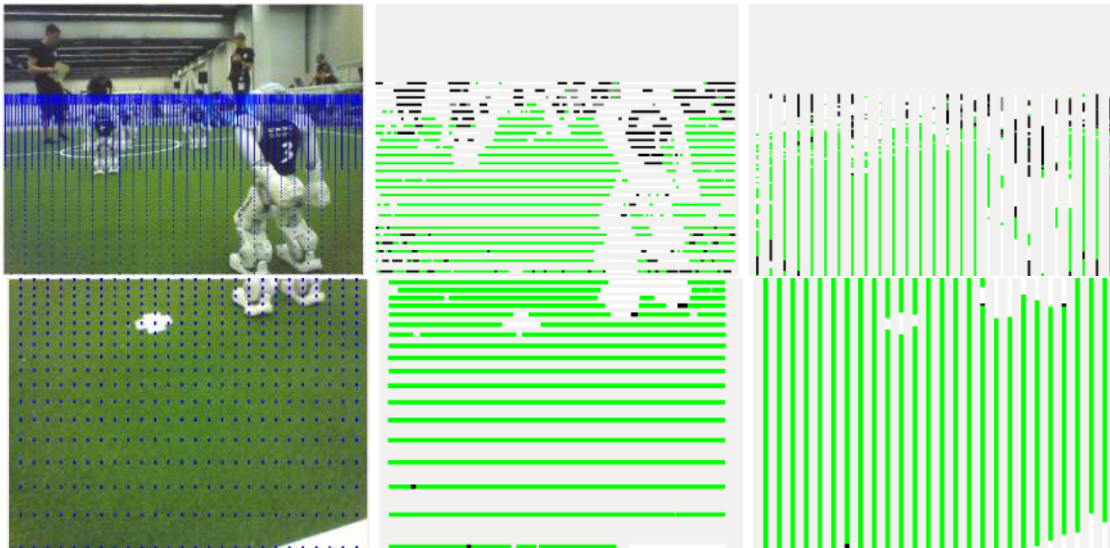


Abbildung 2.1: Eine Aufnahme aus einem Testspiel. Die blauen Punkte im linken Bild bilden die *Scangrid*. Die horizontalen *Scanlines* befinden sich in der Mitte, die vertikalen auf der rechten Seite. Die oberen drei Bilder stammen aus der oberen Kamera, die unteren drei aus der unteren.

Farbklasse von meinem *Modul* vorgenommen wird. Ein *Modul* ist im B-Human-Kontext eine C++ Klasse, die einen Datensatz, in diesem Fall die *Scanlines*, berechnet. Andere Module können diese Datensätze nutzen, um andere Daten zu berechnen (Position des Balls, der Feldlinien und vieles mehr).

## 2.2 Weitere Schritte in der Bildverarbeitung

Die berechneten *Scanlines* sind die Grundlage für fast alle Module in der B-Human-Bildverarbeitung. Sie werden verwendet, um interessante Punkte im Bild zu ermitteln, die zum Beispiel ein Fußball sein könnten. Diese Bereiche, auch *Spots* genannt, werden dann von weiteren Modulen verwendet, um detailliert zu prüfen, ob sie wirklich das darstellen, was zuvor vermutet wurde. Diese genauere Prüfung ist meistens zu rechenaufwendig, um sie auf dem gesamten Bild auszuführen.

Dieser Prozess begründet die Bedeutung der möglichst fehlerfreien *Scanlines*. Wird zu viel Rauschen in den *Scanlines* zugelassen, also gibt es zu viele farbig falsch klassifizierte Regionen, kann es vorkommen, dass *false Positives* entstehen, und etwas erkannt wird, was nicht vorhanden ist. Auch führen durch Rauschen eingeteilte, falsche Regionen dazu, dass Rechenzeit in Bereiche investiert wird, die eigentlich uninteressant sind. Weiterhin kann eine falsche und zu starke Rauschunterdrückung dazu führen, dass zu wenig Regionen gebildet werden und so z.B. die Farben des Fußballs als Rauschen identifiziert und so herausgefiltert würden. Die Konsequenz wäre in diesem Beispiel ein nicht erkannter Fußball.

# 3. Verwandte Arbeiten

Jeder Teilnehmer der Standard Platform League muss mit der selben Technologie arbeiten, denn der NAO ist in den SPL-Regeln vorgeschrieben (vgl. [RoboCup Technical Committee, 2017], Kapitel 2.1). Somit stehen die Teams alle vor den selben Problemen und Aufgaben, die nur durch Software gelöst werden dürfen. Ein Problem ist das in dieser Bachelorarbeit behandelte Thema der in Regionen eingeteilten *Scanlines*. Dadurch, dass die Roboter nach heutigen Maßstäben leistungsschwach sind, haben fast alle Teams sich dazu entschlossen ebenfalls *Scanlines* in ihre Perzeption einzuführen. Durch sie kann man im besten Falle ohne Informationsverlust das Bild nach interessanten Punkten und Objekten absuchen.

Der aktuell verwendete Ansatz von B-Human arbeitet ebenfalls mit Scanlines, wie sie in Abschnitt 2.1 beschrieben wurden.

Bei der Erzeugung der *Scanlines* steht bereits ein in die vier Farbklassen eingeteiltes Bild zur Verfügung. Diese Farbinformationen werde bei der Einteilung der *Scanlines* genutzt, um festzustellen, wann eine Region endet, bzw. anfängt. Der momentan genutzte Ansatz teilt die Regionen anschließend so ein, dass die Farbklassen sich dort widerspiegeln, wobei es eine Mindestlänge bei den horizonalen *Scanlines* gibt - eine Region muss mindestens vier Pixel lang sein.

Die Einteilung der vertikalen *Scanlines* nutzt einen interessanten Ansatz, der so nicht in dieser Bachelorarbeit verfolgt werden kann. Die *ScanGrid* bestimmt entlang den vertikalen *Scanlines* Punkte. Diese Punkte sind im oberen Teil des Bildes sehr eng gestaffelt, nach unten hin vergrößert sich ihr Abstand. Der Algorithmus zur Einteilung in die Regionen betrachtet nun, in welcher Farbklasse ein Punkt auf dieser *Scanline* ist. Bei der Betrachtung des darauf folgenden Punktes, wird diese Farbe nun mit dem vorherigen abgeglichen. Gibt es einen Unterschied, wird zwischen diesen Punkten der Farbübergang gesucht. Gibt es keinen, wird mit dem nächsten Punkt fortgefahren und der Bereich zwischen den Punkten der aktuellen Region zugeordnet. [Röfer u. a., 2017]

Dieses Vorgehen ist äußerst effizient und spart viel Rechenzeit. Auf die Hintergründe warum dieser Ansatz nicht ebenfalls verfolgt wurde, wird im späteren Verlauf eingegangen.

Auch die Nao Devils der Technischen Universität Dortmund nutzen das Konzept der *Scanlines*. Ebenfalls haben sie das Problem der dynamischen Lichtverhältnisse erkannt und sich dessen angenommen. Im Jahr 2015 haben Mitglieder des Teams ein Paper veröffentlich, in dem ein möglicher Algorithmus beschrieben wird. Ihr Ansatz ist ein gewichtetes Histogramm, welches die Farben in ihrer Häufigkeit betrachtet. Die Werte der Farbkanäle im *YCbCr*-Farbraum, die am häufigsten vorkommen, bilden die Feldfarbe (Grün). Die Werte in der Nähe des Maximums gehören wahrscheinlich ebenfalls zur Feldfarbe, weswegen eine Abweichung in der Breite des *Peaks* genommen wird, um die Spanne der Werte etwas zu erweitern.

Zusammen beschreiben Maximum und Breite des *Peaks* die Feldfarbe. Im Laufe eines Spiels kann die Feldfarbe sich verändern und das System berechnet mit jedem neuen Bild die Werte der Farbkanäle neu. Zu stark darf die neu berechnete Farbe allerdings nicht von der anfangs bestimmten Feldfarbe abweichen, da sonst ein Blick außerhalb des Feldes zu einer falschen Feldfarbe führen könnte. [Hofmann u. a., 2017]

*Hidden-Markov-Models* sind in der Praxis unter anderem für Spracherkennung eine häufig gewählte Lösung. So kann man beispielsweise mit Hilfe von *Hidden-Markov-Models* aus gesprochenen Sätzen auf die Emotion des Sprechenden schließen. Dies wurde in dem Paper „Hidden-Markov-Model-Based Speech Emotion Recognition“ von Schuller, Rigoll und Lange von der TU München gezeigt. Dort wurden mit Hilfe eines *Gaussian-Mixture-Models*(GMM) 20 Aspekte von gesprochener Sprache, wie z.B. durchschnittliche Tonhöhe, Position des höchsten und niedrigsten Tons, gemessen und analysiert. Das GMM lieferte dann eine Wahrscheinlichkeitsverteilung für die sieben Emotionen Wut, Ekel, Angst, Freude, Traurigkeit, Überraschung und die neutrale Emotion. Diese Wahrscheinlichkeitsverteilung wurde dann wiederum von einem *Hidden-Markov-Model* verwendet, um den zeitlichen Verlauf zu berechnen. [Schuller u. a., 2003]

Teile des *Hidden-Markov-Models* wurden mit dem *Baum-Welch-Algorithmus* gelernt, welcher auch in dieser Arbeit für eben diesen Zweck eingesetzt wird. Die Arbeit zeigt auch, dass es in der Praxis bereits Kombinationen von HMM und GMM gibt.

# 4. Grundlagen

## 4.1 Gaussian-Mixture-Model

Ein *Gaussian-Mixture-Model* (GMM) ist in diesem Anwendungsfall eine dreidimensionale Normalverteilung mit zehn Komponenten. Die drei Dimensionen sind die Farbkanäle des jeweilig betrachteten Pixel bzw. der betrachteten Farbklasse. Diese Farbkanäle sind die des *HSV*-Farbraums: Helligkeit, Farbwert und Sättigung.

Die zehn Komponenten sind die von GMM einzuteilende Farbklassen, die jeweils für sich eine dreidimensionale Normalverteilung ergeben. Außerdem hat das GMM für jede Farbklasse eine Gewichtung, die die momentane Häufigkeit der Klasse im Bild widerspiegelt. Je häufiger eine Farbe auftritt, desto höher ist das Gewicht. Die vom GMM erkannten zehn Farbklassen müssen im späteren Verlauf auf die vier von der Bildverarbeitung verwendeten Farbklassen projeziert werden.

Die Bildverarbeitung bei B-Human benötigt momentan in vielen Bereichen noch die vier verschiedenen Farbklassen. So auch das Modul *ColorScanlineRegionizer*, welches ebenfalls die Regionen der *Scanlines* einteilt. Es erhält das fertig farbsegmentierte Bild, auch *ECImage* genannt, als Eingabe. Dieses Bild hat die selbe Auflösung wie das ursprünglich aufgenommene Kamerabild, jedoch sind alle Pixel in eine der vier Farbklassen Weiß, Schwarz, Grün und None eingeteilt.

Unter anderem für diese Einteilung wurde das *Gaussian-Mixture-Model* bereits bei B-Human eingeführt, denn dieses GMM ist ein adaptives GMM und passt sich mit jedem neuen Bild an die neuen Gegebenheiten an. Wird z.B. das Feld durch die Sonne plötzlich heller beleuchtet als zuvor, soll das GMM die Farbklassen an die neue Helligkeit anpassen. Dies geschieht durch Stichproben, die aus den Scanlines entnommen werden. Diese Stichproben sind Teil einer Region und sind dadurch einer Farbklasse zugeordnet. Mit Hilfe eines *Erwartungs-Maximierungs-Algorithmus* aktualisiert sich das GMM automatisch. Dies betrifft die Mittelwerte, Standardabweichungen und Gewichte der zehn Farbklassen. Das *Gaussian-Mixture-Model* liefert als Ausgabe diese Werte in Form von 14 zehn-Wert-großen Vektoren. Diese zehn Werte stehen für die zehn Farbklassen des GMM. Die Mittelwerte und Standardabweichungen sind jeweils in sechs Vektoren unterteilt, denn jeder Farbkanal trägt seine Werte in eigenen Vektoren. Auch findet eine Unterteilung in die Verteilung der oberen Kamera und die Verteilung der unteren Kamera statt. Die restlichen beiden Vektoren sind die Vektoren der oberen und unteren Verteilung, die die Gewichte der Farbklassen hält. [Röfer u. a., unv ]

Durch die potenzielle Anpassung an dynamischem Licht, wurde der Ansatz des GMM weiter verfolgt und als Teil des *Hidden-Markov-Models* verwendet.

## 4.2 Hidden-Markov-Model

Ein *Hidden-Markov-Model* ist ein nach dem russischen Mathematiker Andrei Andrejewitsch Markow benanntes stochastisches Modell. Es besteht aus einer festen

Menge an Zuständen, welche die Wirklichkeit widerspiegeln. Sie sind jedoch, je nach Modell, aus unterschiedlichen Gründen nicht einsehbar, also verborgen (engl. *hidden*). Mit dem *Hidden-Markov-Model* wird auf stochastischem Wege versucht, die Wahrscheinlichkeiten der Zustände möglichst genau zu ermitteln. [Rabiner, 1989]

In dem Kontext dieser Bachelorarbeit gibt es die Zustände Weiß, Schwarz, Grün und None, die Klasse in der alle restlichen Farben enthalten sind. Für die weitere Bildverarbeitung ist die Einteilung aller Pixel in einer der vier Farbklassen wichtig. Theoretisch hat jeder Pixel genau einen Zustand. So ist ein Pixel, der einen Teil des Feldes abbildet, mit Sicherheit in der Farbklasse Grün und somit hat er auch den Zustand Grün. Durch unterschiedliche Helligkeiten und Ungenauigkeiten im Bild ist es einem Computer jedoch nicht möglich einen Pixel fehlerfrei in die richtige Farbklasse einzusortieren. Somit ist für das Computerprogramm die wirkliche Farbe, die hinter einem Pixel steht, verborgen. In diesem Anwendungsfall hat jeder Pixel vier Wahrscheinlichkeiten, die beschreiben, wie hoch die Chancen stehen, dass dieser Pixel jeweils diesen Zustand (einer der vier Farbklassen) hat. Durch Kontextwissen, welches in das *Hidden-Markov-Model* einfließt, wird versucht, die Wahrscheinlichkeiten der versteckten Zustände möglichst realitsätsnah auszurechnen und somit die Zustände aufzudecken.[Russell u. Norvig, 2012]

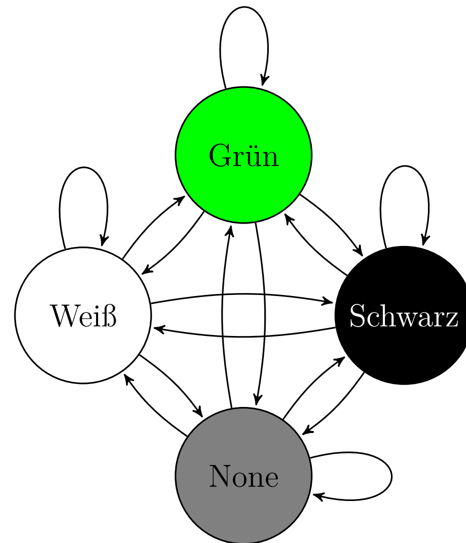



Abbildung 4.1: Die vollvermaschte Netz der Zustände ohne Übergangswahscheinlichkeiten.

Dafür gibt es in einem HMM unter anderem Übergänge und Übergangswahrscheinlichkeiten. Die *Scanlines* kann man als eine Kette von Pixeln beschreiben. Da die Verteilung der Zustände nicht gleichmäßig ist und z.B. Grün aufgrund der Fläche des Feldes sehr häufig auftritt, schwarz aber eher selten, ist die Chance, dass zwei Pixel dieser beiden Farben aufeinandertreffen deutlich kleiner als die Wahrscheinlichkeit, dass ein grüner Pixel auf einen anderen grünen Pixel folgt. Diese Tatsache wird mit der Übergangswahrscheinlichkeit beschrieben. Jedem Übergang wird ein Wert zugeordnet, sodass ein Netz mit vier Zuständen entsteht, welche alle miteinander verbunden sind. Dieses Netz sieht wie in Abbildung 4.1 aus.

Ein weiterer Bestandteil des HMM ist das Sensormodell. Es beschreibt die Wahrscheinlichkeit eines Pixel, basierend auf den Informationen aus den Farbkanälen, zu einer Farbklasse zu gehören. Hierbei ist zu beachten, dass dies nicht die Wahrscheinlichkeit für die Zugehörigkeit in einem Zustand ist, diese wird erst später berechnet. In der Literatur wird dazu häufig ein Sensormodell verwendet, welches ähnlich wie das Übergangsmodell in einer quadratischen Matrix notiert ist und den Fehler in der Messung repräsentiert. [Russell u. Norvig, 2012] In meinem Modul werden die Wahrscheinlichkeiten für die Farben über das verwendete *Gaussian-Mixture-Model* berechnet, welches die Farbklassen dynamisch konfiguriert und somit das beste Resulat bringen sollte.

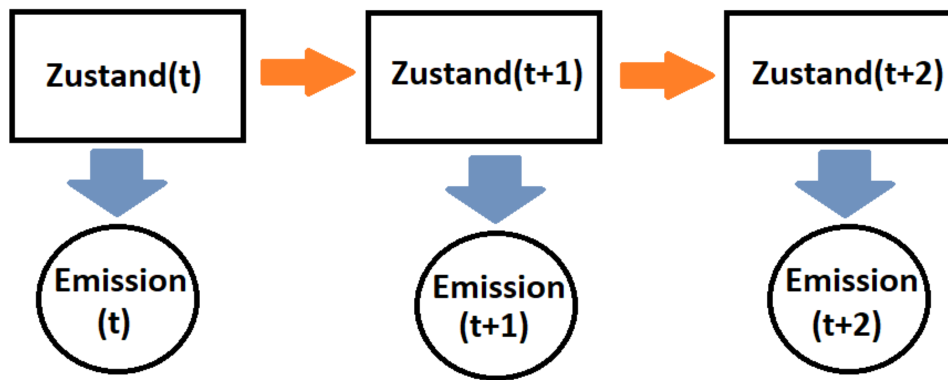



Abbildung 4.2: Eine Kette von drei Zuständen. Jeder emitiert Hinweise auf sich selbt (Emission, blaue Pfeile). Die orangenen Pfeile verdeutlichen die Übergänge.

Den Zusammenhang von Übergangs- und Sensormodell verdeutlicht Abbildung 4.2. Man kann die Zustände nicht direkt einsehen, jedoch erhält man Informationen über sie, da sie für sich spezifische Emmissionen liefern. In diesem Anwendungsfall sind die Emissionen die Werte in den Farbkanälen, denn jede Farbe ist dort anders vertreten. Die Emmissionen werden ausgewertet und später in Form des Sensormodells nutzbar gemacht. Diese Aufgabe übernimmt das bereits erklärte *Gaussian-Mixture-Model*.

Die Nachbarschaft eines Pixels hat ebenfalls ihren Einfluss auf die Wahrscheinlichkeit eines Zustandes. Dies wird mit den orangenen Pfeilen verdeutlicht, die aus den Zuständen eine Kette machen. Die Wahrscheinlichkeit einer Nachbarschaft zweier Pixel wird im Übergangsmodell eingetragen, welches später gelernt wird.

## 4.3 Forward-Backward-Algorithmus

Die Berechnung der Wahrscheinlichkeiten liefert die verlässlichsten Daten, wenn man bei der Betrachtung von Pixel $A_t$, der an Position t liegt, die vorherigen Pixel $A_1$ bis $A_{t-1}$ ebenfalls betrachtet und ihre Wahrscheinlichkeitsverteilung bezüglich der Zustände kennt. Der im folgenden beschriebene *Forward-Backward-Algorithmus* stammt aus dem Buch „Künstliche Intelligenz - ein moderner Ansatz“ von Russel und Norvig (siehe [Russell u. Norvig, 2012]).

### 4.3.1 Forward

Der *Forward*-Algorithmus macht sich dieses Wissen zu Nutze und bezieht dieses Wissen mit in die Berechnung der Wahrscheinlichkeiten für Pixel $A_t$ ein. Die Definition für den *Forward*-Algorithmus ist wie folgt:

$$forward(Z, e_{1:t}) = P(Z_t|e_{1:t}) = \alpha * P(e_t|Z_t) * \sum_{z_{t-1}} (P(Z_t|z_{t-1}) * P(z_{t-1}|e_{1:t-1}))$$

Diese Formel berechnet die Wahrscheinlichkeit für den Zustand Z an der Position t, gegeben den Evidenzen aller vorherigen Pixel der Scanline. Die Wahrscheinlichkeit $P(e_t|Z_t)$ erhält man direkt aus dem Sensormodell. Durch die Summierung aller

möglichen Wege zu Zustand $Z_t$ erhält man die gesamte Wahrscheinlichkeit des Zustandes an Position t. Der erste Term hinter dem Summenzeichen beschreibt die Wahrscheinlichkeit, dass von einem Zustand $z_{t-1}$ zu einem Zustand $Z_t$ gewechselt wird. Sie stammt direkt aus dem Übergangsmodell. Der rekursive Aufruf über alle Positionen t bis 1 der Scanline geschieht im letzten Term $P(z_{t-1}|e_{1:t-1})$. Die Rekursion endet mit der Anfangswahrscheinlichkeit $P(Z_0)$. Diese kann entweder berechnet werden, oder standardmäßig eine Gleichverteilung haben. Je nach dem wie schnell die Zustände wechseln, macht dies kaum einen bemerkbaren Unterschied. Das $\alpha$ zu Beginn der Gleichung normalisiert dieses Ergebnis so, dass die Summe der Wahrscheinlichkeiten aller Zustände eins ergibt.

### 4.3.2 Backward

In dem bekannten Anwendungsfall liegt die *Scanline* bereits zu Beginn der Berechnung vollständig vor. Dies ermöglicht es, während jeder Pixel sequentiell betrachtet wird, in die „Zukunft“, also einige Pixel nach vorne zu blicken. So erhält man zusätzlich zu den Informationen über die Pixel „vor“ dem betrachteten Pixel auch die Informationen „danach“. Die Formel die einen solchen Nachrichtentransport in rückwärtiger Richtung (also *backwards*) beschreibt, sieht wie folgt aus:

$$backward(Z, e_{k+1:t}) = P(e_{k+1:t|X_k}) = \alpha * \sum_{x_{k+1}} (P(e_{k+1}|x_{k+1}) * P(x_{k+1}|X_k) * P(e_{k+2:t}|x_{k+1}))$$

Es wird die Wahrscheinlichkeit für die nachfolgende Sequenz von Evidenzen k+1 bis t berechnet, gegeben dem Zustand X an Position k. Der erste Term beschreibt die Sensorwahrscheinlichkeit des Pixels an der nächsten Position. Dieser wird mit der Übergangswahrscheinlichkeit für den Übergang von $X_k$ zu $x_k$ multipliziert. Zum Schluss wird der rekursive Aufruf für die nächsten Pixel gestartet. Er terminiert, sobald die letzte betrachtbare Evidenz verwendet wurde. Diese ist entweder das Ende der Sequenz oder die Evidenz die an Position t liegt. Ist das Ende erreicht, wird der Wert 1 verrechnet. Der feste Abstand zwischen k und t variiert je nach Konfiguration und kann beliebig gewählt werden. Die Summierung hat den Hintergrund, dass alle möglichen Pfade betrachtet werden, die ihren Ursprung im an Position k liegenden Pixel mit Zustand X haben. Im Verlauf der Bachelorarbeit habe ich bemerkt, dass es sinnvoll ist auch, die *Backward*-Wahrscheinlichkeit zu normalisieren, sodass die Summe aller Zustandswahrscheinlichkeiten Eins ergibt. Grund hierfür ist, dass durch die Multiplikation sehr geringer Wahrscheinlichkeiten über längere Distanzen hinweg häufig alle *Backward*-Wahrscheinlichkeiten Null ergaben. Deshalb habe ich in die Formel die Normalisierungskonstante $\alpha$ eingefügt.

### 4.3.3 Gesamtergebnis

Beide Funktionen kombiniert ergeben den *Forward-Backward-Algorithmus*. Die Wahrscheinlichkeit für den Zustand Z an Position k ist die Multiplikation der *Forward-Wahrscheinlichkeit* mit der *Backward*-Wahrscheinlichkeit:

$$P(Z_k|e_{1:t}) = \alpha * forward(Z, e_{1:k}) * backward(Z, e_{k+1:t})$$

### 4.3.4 Optimierungen

Im Kontext der *Scanlines* müssen die Wahrscheinlichkeiten für alle vier Zustände (Weiß, Schwarz, Grün, None) für jeden Pixel berechnet werden. Um nicht jede Formel vier mal verwenden und die Ergebnisse in vier Variablen speichern zu müssen, kann man die Wahrscheinlichkeiten auch als Vektor darstellen und die Formeln in Vektorrechnungen umwandeln. Dies sorgt für eine starke Vereinfachung des Quellcodes und zu einer besseren Lesbarkeit. Das verwendete Sensormodell ist ebenfalls als Vektor darstellbar. Das Übergangsmodell lässt sich leicht als Übergangs- oder Transitionsmatrix, wie in Tabelle 4.1, verstehen.

| | | | |
|---|---|---|---|
| $\text{Weiß}_t \rightarrow \text{Weiß}_{t+1}$ | $\text{Weiß}_t \rightarrow \text{Schwarz}_{t+1}$ | $\text{Weiß}_t \rightarrow \text{Grün}_{t+1}$ | $\text{Weiß}_t \rightarrow \text{None}_{t+1}$ |
| $\text{Schwarz}_t \rightarrow \text{Weiß}_{t+1}$ | $\text{Schwarz}_t \rightarrow \text{Schwarz}_{t+1}$ | $\text{Schwarz}_t \rightarrow \text{Grün}_{t+1}$ | $\text{Schwarz}_t \rightarrow \text{None}_{t+1}$ |
| $\text{Grün}_t \rightarrow \text{Weiß}_{t+1}$ | $\text{Grün}_t \rightarrow \text{Schwarz}_{t+1}$ | $\text{Grün}_t \rightarrow \text{Grün}_{t+1}$ | $\text{Grün}_t \rightarrow \text{None}_{t+1}$ |
| $\text{None}_t \rightarrow \text{Weiß}_{t+1}$ | $\text{None}_t \rightarrow \text{Schwarz}_{t+1}$ | $\text{None}_t \rightarrow \text{Grün}_{t+1}$ | $\text{None}_t \rightarrow \text{None}_{t+1}$ |

Tabelle 4.1: Die Bedeutung der Felder in der Transitionsmatrix T.

In diesem Zustand ist der *Forward-Backward-Algorithmus* nicht verwendbar, denn viele Berechnungen werden mehrfach durchgeführt. Da die *Scanline* nur einmal durchlaufen werden soll und dabei für jeden Pixel entschieden werden soll, welchen Zustand er am wahrscheinlichsten hat, kann man die *Forward*-Wahrscheinlichkeit wiederverwenden. Sie wird mit jedem weiteren Pixel weiterberechnet und „trägt“ somit das Ergebnis der vorherigen mit. Vereinigt man den Ansatz der Vektorrechnung mit der neuen Formel für die *Forward*-Wahrscheinlichkeit, sieht die Berechnung wie folgt aus:

$$forward_t(S_t, forward_{t-1}) = \alpha * diag(S_t) * T^T * forward_{t-1}$$

Hierbei ist $S_t$ das Sensormodell als Vektor, welches pro Pixel durch das GMM berechnet wird. Es wird zur Berechnung als Diagonalmatrix verwendet. $T^T$ ist die transponierte Transitionsmatrix, welche im gesamten Prozess konstant bleibt. Der *Forward*-Vektor $forward_{t-1}$ ist das Ergebnis selber Rechnung bei vorherigem Pixel. Wie bisher ist $forward_0$ die apriori-Wahrscheinlichkeit, in der jeder Wert des Vektors 0,25 ergibt.

Notiert als Vektorfunktion, sieht die Berechnung der *Backward*-Wahrscheinlichkeit so aus:

$$backward_t(S_{t+1}, backward_{t+1}) = \alpha * T * S_{t+1} * backward_{t+1}$$

Die Transitionsmatrix ist die selbe wie im *forward*-Kontext, jedoch muss sie in dieser Formel nicht transponiert werden, da die Pixel in der *backward*-Richtung liegen, also nach dem betrachteten Pixel in der Scanline kommen. Deshalb wird die Matrix von der anderen Seite genutzt und muss nicht transponiert werden. Die letzte *Vektor* der Scanline ist wie bisher an jeder Position 1.

## 4.4 Baum-Welch-Algorithmus

Die Übergangsmatrix muss für einen optimalen Einsatz im *Hidden-Markov-Model* so an die Umgebung angepasst werden, dass sie die Realität möglichst präzise widerspiegelt. Im Falle des Roboterfußballs ist zum Beispiel der Übergang von einem grünen zu einem grünen Pixel aufgrund der Farbe des Felds sehr wahrscheinlich, während andere Übergänge deutlich unwahrscheinlicher sind. Für die optimale Adaption einer Übergangsmatrix wird der *Baum-Welch-Algorithmus*, wie er in „A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models“ von Jeff A. Bilmes beschrieben wurde (siehe [Bilmes, 1998]), ausgewählt. Dieser zählt die Häufigkeiten der Übergänge und verrechnet sie mit der jeweiligen Wahrscheinlichkeit aus dem Sensormodell. Je häufiger der Algorithmus einen Übergang zählt, desto höher ist später der Wert in der Übergangsmatrix. Die Zählung der Häufigkeit sieht als Formel so aus:

$$\xi_{ij}(t) = \frac{\gamma_i(t) * a_{ij} * s_j(t+1) * \beta_j(t+1)}{\beta_i(t)}$$

Im nächsten Schritt wird die Häufigkeit eines Übergangs von Zustand i zu Zustand j in Relation zu den wahrscheinlichen Übergängen weg von Zustand i gesetzt. Dadurch ergibt die Summe jeder Zeile in der Transitionsmatrix eins. Die zuvor genutzte Normalisierungskonstante $\alpha$ ist bereits in dieser Gleichung enthalten.

$$\tilde{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

| Symbol | Bedeutung |
|---|---|
| $\xi_{ij}$ | Häufigkeit des Übergangs von Zustand i zu Zustand j. |
| $\gamma_i(t)$ | Ergebnis des *Forward-Backward-Algorithmus* für Position t. |
| $a_{ij}$ | Übergangswahrscheinlichkeit für den Übergang von Zustand i zu Zustand j. Wird nicht im Verlauf des Algorithmus verändert. |
| $s_j(t+1)$ | Sensorwahrscheinlichkeit für den Zustand j an der Position t+1. |
| $\beta_j(t)$ | *Backward*-Wahrscheinlichkeit für den Zustand j an Position t. |
| $\tilde{a}_{ij}$ | Durch den Algorithmus optimierte Übergangswahrscheinlichkeit von Zustand i zu Zustand j. |
| T | Die Position des letzten Pixels in der *Scanline* |

Tabelle 4.2: Legende für im *Baum-Welch-Algorithmus* verwendete Symbole

Der *Baum-Welch-Algorithmus* benötigt zu Beginn eine Übergangsmatrix als Ausgangspunkt. Diese muss geschätzt werden, sollte jedoch möglichst nahe an der späteren Übergangsmatrix liegen, denn der Algorithmus findet nur ein lokales Maximum. Mit dieser Matrix wird dann die gesamte Laufzeit des Algorithmus gerechnet, bis der Nutzer sich entscheidet ihn abzubrechen oder die Werte sich nicht mehr verändern. Die neu berechnete Transitionsmatrix sollte den Zusammenhang, den sie repäsentieren soll, verbessert darstellen.

# 5. Implementierung

Die Algorithmen wurde in den Klassen *HMMColorScanlineRegionizer.cpp /-.h* und *HMMHiResColorScanlineRegionizer.cpp / -.h* implementiert. Es gibt zwei Module, da es wechselseitige Abhängigkeiten in der weiteren Bildverarbeitung gibt, die nur durch das Aufteilen in zwei Module gelöst werden konnten. Diese Abhängigkeiten entstehen in der Berechnung der Feldgrenzen, wofür *ColorScanlineVertical* benötigt werden. Die Feldgrenzen wiederum werden für die Berechnung der *ColorScanlineVerticalClipped* benötigt. Da zu Beginn der Berechnung der *Scanlines* alle Daten vorhanden sein müssen, funktioniert die Berechnung beider Scanline-Varianten in einem Modul nicht. Die Lösung ist eine Teilung des Moduls. Die horizontalen und vertikalen Scanlines können gemeinsam im *HMMColorScanlineRegionizer* berechnet werden.

## 5.1 Verwendung des Gaussian-Mixture-Models

Das *Gaussian-Mixture-Model* war bereits im B-Human System vorhanden. Es musste jedoch so angepasst werden, dass es für jeden Pixel die Wahrscheinlichkeiten für die vier Farbklassen bereitstellt. Für diesen Zweck konnten Standardabweichung, Mittelwert und Gewichte verwendet werden, die bereits pro Bild ausgegeben werden. Diese Werte bestimmen, wie die jeweilige Farbklasse aussieht. Die Standardabweichung und der Mittelwert beziehen sich auf jeden Farbkanal einer jeden Farbklasse, das Gewicht wird pro Farbklasse ausgeben und zeigt an, wie häufig eine Farbe vorkommt. Dies ergibt pro Farbe sieben Informationen. Das GMM kennt insgesamt zehn Farbverteilungen. Drei davon sind die Farbklassen Weiß, Schwarz und Grün. Die anderen sieben Farbklassen bilden zusammen die Restklasse, die auch als „None“ bezeichnet wird. In ihr werden alle restlichen Farben zusammengefasst und gespeichert. Mit diesen Informationen wird vor den anderen im HMM-Modul verwendeten Algorithmen eine dreidimensionale Normalverteilung erzeugt. Die drei Dimensionen bilden hierbei die Farbkanäle der jeweiligen Farbklasse. Die zehn Farbklassen sind die Komponenten.

Roboter des Typen NAO haben zwei Kameras, die in einem unterschiedlichen Winkel am Kopf angebracht sind. Die obere Kamera sieht das gesamte Feld, während die untere lediglich die Umgebung unmittelbar vor dem NAO aufnimmt. Infolgedessen werden den Kameras auch unterschiedliche *Gaussian-Mixture-Models* zugeordnet. Daraus folgt auch die Unterscheidung zwischen der oberen Verteilung der Farben und der unteren. Beide Normalverteilungen werden im HMM-Modul selbst gespeichert und mit jedem Bild neu berechnet bzw. aktualisiert.

Die Funktion der gewichteten Wahrscheinlichkeitsberechnung war bereits im GMM-Modul gegeben [Röfer u. a., unv ]. Für diese Arbeit wurde sie ein wenig umstrukturiert, um die Rechenzeit zu minimieren. Als Ergebnis sieht die optimierte Methode wie

folgt aus:

```
Function Gewichtete-Wahrscheinlichkeits-Berechnung
  Data:   Vektor pixel; //3x1 Vektor
          Vektor mittelwerte; // 3x10 Vektor
          Vektor kovarianzInvers; // 3x10 Vektor
          Vektor gewichte; // 1x10 Vektor
          Vektor ersterTerm; // 1x10 Vektor
  Result: Wahrscheinlichkeitsverteilung für 10 Farbklassen basierend auf
          Farbinformationen des aktuellen Pixels
  begin
    Vektor wahrscheinlichkeiten; // 10x1 Vektor
    for index = 0; index < 10; index++ do
      Vektor differenz = pixel - mittelwerte[index] ;
      Vektor kovarianzInvers = kovarianzenInvers[index];
      float zweiterTerm = -0,5 * differenz.transponiert() *
        kovarianzInvers.alsDiagonalMatrix() * differenz;
      wahrscheinlichkeiten[index] = ersterTerm[index] * e^zweiterTerm;
    end
    return wahrscheinlichkeiten * gewichte;
  end
```

Die Vektoren *kovarianzenInvers* und *mittelwerte* sind dreidimensional und enthalten drei mal zehn Werte. Jede Reihe steht für eine Farbklasse, jeder Wert in dieser Reihe für den jeweiligen Wert des Farbkanals, wie in Tabelle 5.1 einsehbar. Diese Werte werden bei der Aktualisierung der Verteilung mit jedem Bild aktualisiert und abgespeichert. Sie stammen aus den Mittelwerten und Standardabweichungen, die das GMM mit jedem Bild für jede der 10 Farbklassen und deren Farbkanäle berechnet.

| Position | Farbklasse | Helligkeit | Farbwert | Sättigung |
|---|---|---|---|---|
| 0 | Weiß | | | |
| 1 | Schwarz | | | |
| 2 | Grün | | | |
| 3 | None | | | |
| 4 | None | | | |
| 5 | None | | | |
| 6 | None | | | |
| 7 | None | | | |
| 8 | None | | | |
| 9 | None | | | |

Tabelle 5.1: Aufbau der Vektoren *kovarianzenInvers, mittelwerte* und *standardabweichung* als Tabelle. Die leeren Felder sind die entsprechenden Werte im jeweiligen Vektor.

Ebenfalls bei der Aktualisierung wird der eindimensionale Vektor *ersterTerm* befüllt. Dieser enthält das Ergebnis $e$ der Formel, die für jede der 10 Farbklassen ausgeführt

und abgespeichert wird.

$$e_k = \frac{1}{\sqrt{2 * \pi^3}} * \sum_{i=1}^{3} \frac{1}{s_k^2(i)}$$

In der Gleichung steht k für die betrachtete Farbklasse. Die Standardabweichung der Klasse wird als $s_k$ dargestellt und $i$ ist der jeweilige Farbkanal. Ursprünglich wurde diese Formel mit jedem Pixel berechnet [Röfer u. a., unv ], sie wurde jedoch ausgelagert, um nur noch einmal pro Bild berechnet zu werden. Dies ist möglich, da sich Mittelwert und Standardabweichung nur maximal einmal pro Bild ändern. Somit unterscheidet sich das Ergebnis durch die Umstellung nicht von dem der vorherigen Version. Das Produkt der Kovarianz wird gebildet, da in der ursprünglichen, implementierten Funktion die Kovarianz als Diagonalmatrix dargestellt wurde, deren Determinante berechnet wird. Diese ist in diesem Falle stets das Produkt der Diagonalen.

Der Rest der ursprünglichen Gleichung muss pro Pixel berechnet werden, da die für die Berechnung benötigte Differenz zwischen Mittelwerten und aktuellen Werten des Pixels für jeden Pixel anders ist.

Die berechneten Wahrscheinlichkeiten werden nun in der Methode *Hole-Sensormodell-Pro-Pixel* verarbeitet, da im momentanen Zustand zehn, statt der geforderten vier Wahrscheinlichkeiten, vorhanden sind. Auch kommt es häufig vor, dass mehrere Farbklassen die Wahrscheinlichkeiten 0 haben, während eine Farbklasse zu 100% wahrscheinlich ist. Diese Sicherheit ist trügerisch, da es niemals eine vollkommene Gewissheit bei der Bestimmung des Zustands des Pixels geben kann. Eine 0 in der Wahrscheinlichkeitsverteilung wirkt sich im weiteren Verlauf des Programms verfälschend auf das Ergebnis aus, denn durch den *Forward-Backward-Algorithmus* werden die Wahrscheinlichkeitsverteilungen einer Farbklasse miteinander multipliziert. Eine Null in der Verteilung würde dazu führen, dass eine oder mehrere Farbklassen für keinen Pixel einer Scanline wahrscheinlich ist. Deshalb muss das Ergebnis der Funktion aufbereitet werden.

Die Methode *Hole-Sensormodell-Pro-Pixel*, die die *Gewichtete-Wahrscheinlichkeits-*

*Berechnung-Methode* aufruft und deren Ergebnis verarbeitet, sieht wie folgt aus:

```
Function Hole-SensorModell-Pro-Pixel
    Data: Position des Pixels (x, y);
    Result: Sensormodell als Vektor für Pixel (x, y)
    begin
        Vektor pixel = Farbkanäle an Pos(x, y); //1x3 Vector
        Vektor wktn; // 1x10 Vektor
        float summe;
        if Pixel aus der oberen Kamera then
            wktn = oberesGMM.Gewichtete-Wahrscheinlichkeits-Berechnung(x, y);
        else
            wktn = unteresGMM.Gewichtete-Wahrscheinlichkeits-Berechnung(x, y);
        end
        Vektor ergebnis; // 4x1 Vektor
        ergebnis[0] = max(wktn[0], 0.05f);
        ergebnis[1] = max(wktn[1], 0.05f);
        ergebnis[2] = max(wktn[2], 0.05f);
        ergebnis[3] = max(1 - (wktn[0] + wktn[1] + wktn[2]), 0.05f);
        ergebnis *= 1 / ergebnis.Summe();
    end
    return ergebnis
```

Der Vektor *wktn* aus dem Algorithmus wird mit der *Gewichtete-Wahrscheinlichkeiten-Berechnung* des GMM berechnet. Der Vektor der Wahrscheinlichkeiten, die final zurückgegeben werden, besteht an Position 0 aus der Wahrscheinlichkeit für die Farbe Weiß, an Position 1 für Schwarz und an Position 3 für die Farbe Grün. Die Restklasse, die im folgenden Verlauf als eine Klasse mit einer Wahrscheinlichkeit verstanden werden soll, wird in dieser Funktion aus den Wahrscheinlichkeiten der restlichen sieben Farbverteilungen zusammengesetzt. Außerdem wird jede Wahrscheinlichkeit auf mindestens 5% gesetzt, um einen realistischen Fehler darzustellen. Zum Schluss wird der Vektor normalisiert, sodass die Summe des Vektors 1 bzw. 100% ergibt.

## 5.2 Forward-Backward-Algorithmus

Der *Forward-Backward-Algorithmus* musste, wie die beiden anderen Methoden, ebenfalls in beiden Modulen implementiert werden, inhaltlich unterscheidet er sich jedoch nicht. Außerdem wurde er in eine eigene Methode gekapselt, damit sowohl horizontale als auch vertikale *Scanlines* mit der selben Methode eingeteilt werden und Codeduplikate verhindert werden. Der *Forward-Backward-Algorithmus*, der hier

implementiert wurde, funktioniert wie folgt:

```
Function Forward-Backward-Algorithmus
  Data: int distanz,
        Transitionsmatrix T;
        Position des Pixels (x, y);
        Buffer forwardBuffer;
        Buffer backwardBuffer;
        Vektor forward;
        Vektor backward;
  Result: Wahrscheinlichkeitsverteilung als Vektor für Pixel (x, y)
  begin
    Vektor s = Hole-Sensormodell-Pro-Pixel(x, y);
    forwardBuffer.pushFront(s) ;
    Vektor c = T * s;
    c *= 1 / c.Summe(); //normalisieren;
    backwardBuffer.pushFront(c);
    if Position des Pixels in der Scanline >= distanz then
      forward = forward * forwardBuffer.popEnd();
      forward *= 1 / forward.Summe(); //normalisieren
      foreach Vektor v in backwardBufffer do
        backward = backward * v;
      end
    end
    return Normalisiertes Ergebnis von forward * backward;
  end
```

Der hier implementierte Algorithmus ist in der Struktur stark an den *Fixed-Lag-Smoothing-Algorithmus* aus [Russell u. Norvig, 2012] angelehnt. Dieser arbeitet grundsätzlich wie der *Forward-Backward-Algorithmus*, wobei die Distanz, die der *Backward*-Teil des Algorithmus nach vorne „blickt“, einen festen Wert hat. Diesen Grundsatz verdeutlicht Abbildung 5.1.

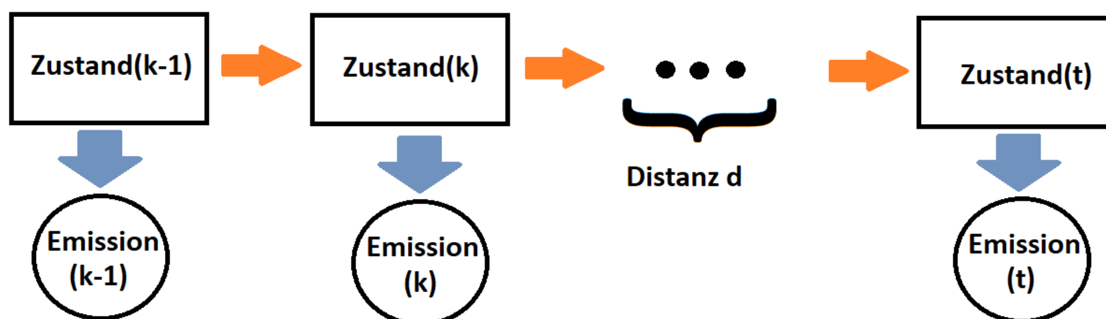



Abbildung 5.1: Bei der Bestimmung der Wahrscheinlichkeitsverteilung für den Pixel an Position k, „blickt“ der *Backward-Algorithmus* $d$ Pixel voraus und kann so die Evidenzen der Zustände an den Positionen $k+1$ bis $t$ mit einbeziehen. Hierbei ist $t = k + d$. Die Distanz $d$ ist konstant und wird im voraus bestimmt. In diesem Kontext wurde eine Distanz von 2 gewählt, um das Bild nicht zu stark zu glätten.

Unterschiede befinden sich in der Berechnung der *Backward*-Wahrscheinlichkeit. Der

*Fixed-Lag-Smoothing-Algorithmus* basiert auf der Idee, dass die Distanz zwischen dem gerade evaluierten Pixel und dem „vordersten“, für die *Backward*-Wahrscheinlichkeit benötigten Pixel keinen Einfluss auf den Aufwand haben soll. Hierfür wird eine *Backward*-Nachricht als Matrix erstellt, die alle in der gewählten Reichweite liegenden *Backward*-Nachrichten verschlüsselt. Sie wird mit jedem neuen Schritt aktualisiert. Das bedeutet, dass die älteste Nachricht durch Multiplikation mit der inversen Matrix entfernt wird und die neue hinzugefügt wird. Diese geschieht so:

$$B_{t-d+2:t+1} = O_{t-d+1}^{-1} T^{-1} B_{t-d+1:t} * T * O$$

In dieser Formel ist $T$ die bekannte Transitionsmatrix, $B$ die iterativ berechnete *Backward*-Matrix, die die *Backward*-Nachrichten von $d$ Positionen kodiert. Das Sensormodell wird als Diagonalmatrix $O$ dargestellt. $O^{-1}$ ist dementsprechend die inverse Matrix von $O$.

Durch diese Kodierung bleibt der Rechenaufwand konstant und unabhängig von der Reichweite der *Backward*-Nachricht. Diese Methode wurde auch in dieser Bachelorarbeit versucht anzuwenden, jedoch traten hier schon sehr früh Fehler auf. Durch die Verrechnung der invertierten Matrizen kam es bereits nach wenigen Durchläufen zu dem Phänomen, dass die in der Matrix enthaltene *Backward*-Nachricht negative Werte enthielt. Da die Werte Wahrscheinlichkeiten repräsentieren, ist dies falsch. Eine Korrektur ist nicht möglich, da die invertierten Matrizen perfekt auf die alten Nachrichten passen müssen, um sie zu egalisieren. Der Grund für das Scheitern liegt in der Ungenauigkeit der Multiplikation mit dem Inversen, denn nicht immer entsteht daraus genau die Einheitsmatrix, sondern oft weicht das Ergebnis leicht von ihr ab. Durch diese Rundungsfehler verstärkt sich in der nächsten Rechnung die Ungenauigkeit, was im Endeffekt zu negativen Wahrscheinlichkeiten führt.

Als Ersatz für die *Backward*-Matrix, wurde ein Buffer genutzt, der mit jedem neuen Pixel $\alpha * T * S$ hinzufügt und einen alten Wert dafür herauslöscht. Wird nun die *Backward*-Nachricht benötigt, werden alle Vektoren des Buffers multipliziert. Dies ist zwar aufwändiger, wenn viel Distanz größer wird, in diesem Anwendungsfall darf sie jedoch nicht zu hoch ausfallen, da sonst Feldlinien als Rauschen identifiziert und „eliminiert“ werden. Das beste Ergebnis wurde mit einer Distanz von zwei erzielt.

Das implementierte Modul orientiert sich in der Grundform an den *ColorScanlineRegionizer*, der bisher für die Regionenbildung bei B-Human genutzt wird. So wurde die Struktur, wie eine *Scanline* aus der *ScanGrid* entnommen wird und welche Pixel verarbeitet werden, übernommen. Dies hat auch den Vorteil, dass die *Scanlines* der beiden Module sehr gut vergleichbar sind, denn sie unterscheiden sich nur im Algorithmus, der die Regionen einteilt. Der grundsätziche Ablauf der Regioneneinteilung

in dem neu erstellten Modul veranschaulicht folgender Pseudocode:

```
Function Regionize-Scanline
  Data: int d; //distanz
        Scanline l;
        Scangrid g;
        Transitionsmatrix T;
  Result: In Regionen eingeteilte Scanline
  begin
    Aktualisiere Gaussian-Mixture-Model;
    Buffer forwardBuffer;
    Buffer backwardBuffer;
    foreach Pixel p in aktuelle Scanline der ScanGrid do
      Forward-Backward-Algorithmus(x, y, forwardBuffer, backwardBuffer,
        forward, backward);
      if Position des Pixels in der Scanline >= d then
        if wahrscheinlichster Zustand gewechselt then
          bisherige Region wird gespeichert;
        end
      end
      if Pixel ist letzter in der Scanline then
        bisherige Region wird gespeichert;
      end
    end
  end
```

Diese Funktion wird für jede Scanline eines Bildes verwendet.

## 5.3 Baum-Welch-Algorithmus

Der *Baum-Welch-Algorithmus* wird in der Implementierung verwendet, um vor der Einteilung der *Scanlines* eine Transitionsmatrix zu berechnen, die die Verteilung der Farben des Spiels bestmöglich widerspiegelt. Wie oft sie berechnet werden muss, hängt davon ab, wie stark sich Umgebungen unterscheiden. Es sollte in den meisten Fällen reichen, sie einmal zu generieren und dann auf jedem Wettbewerb nutzen zu können, da die Farbverteilung sich nicht stark unterscheiden sollte. Es wurde auf eine Unterscheidung zwischen unterer und oberer Kamera verzichtet, da das Lernen und auch die spätere Anwendung einer eigenen Übergangsmatrix für die untere Kamera als nicht sinnvoll erachtet wurde. Die untere Kamera nimmt im Verlauf des Spiels fast nur die Farbklasse Grün auf und dadurch würde die Wahrscheinlichkeitsverteilung für die anderen Farbklassen zu klein geraten. Dies könnte u.a. dazu führen, dass ein Ball nicht erkannt wird. Um dies zu vermeiden, wurde eine Übergangsmatrix mit der oberen Kamera trainiert, die später für Aufnahmen beider Kameras genutzt wurde.

Zu Beginn des Algorithmus muss eine Übergangsmatrix angegeben werden, die bereits gut die Zustandsübergänge zeigt, da der *Baum-Welch-Algorithmus* nur das lokale Maximum finden kann.

Um diese Matrix zu schätzen, wurden die Werte in der Diagonalen von links oben nach rechts unten der Matrix erhöht, da häufig Pixel mit der selben Farbe nebeneinander liegen. Es wäre auch möglich die Werte für alle Pixel in Kontakt mit einem grünen

Pixel zu erhöhen, dies ergab jedoch, dass sich diese Werte weiter erhöhten und alle anderen sich 0 anäherten.

Der erhöhte Wert sollte nicht zu hoch gewählt werden, da auch dies das Ergebnis zu stark beeinflussen könnte. Deshalb wurde der Wert 0,4 gewählt, der zum einen eine erhöhte Wahrscheinlichkeit widerspiegelt, zum anderen werden dadurch die anderen Werte außerhalb der Diagonalen nicht zu klein. Der Wert für die anderen Wahrscheinlichkeiten außerhalb der Diagonalen ist somit 0,2 bzw. 20%.

| 0,4 | 0,2 | 0,2 | 0,2 |
|---|---|---|---|
| 0,2 | 0,4 | 0,2 | 0,2 |
| 0,2 | 0,2 | 0,4 | 0,2 |
| 0,2 | 0,2 | 0,2 | 0,4 |

Tabelle 5.2: Die Übergangsmatrix zu Beginn des *Baum-Welch-Algorithmus*.

Mit der Matrix in Tabelle 5.2 wurde der Algorithmus gestartet. Die daraus resultierende optimierte Matrix wurde später zur Einteilung des *Scanlines* verwendet.

Der *Baum-Welch-Algorithmus* in seiner Grundstruktur wurde aus der Literatur entnommen [Bilmes, 1998] und in Matrix bzw. Vektorschreibweise umgeschrieben. In dieser Schreibweise lässt sich der Algorithmus wie folgt darstellen:

$$\xi(t) = \frac{\begin{pmatrix} \gamma_w & \gamma_w & \gamma_w & \gamma_w \\ \gamma_s & \gamma_s & \gamma_s & \gamma_s \\ \gamma_g & \gamma_g & \gamma_g & \gamma_g \\ \gamma_n & \gamma_n & \gamma_n & \gamma_n \end{pmatrix}_t \circ \begin{pmatrix} ww & ws & wg & wn \\ sw & ss & sg & sn \\ gw & gs & gg & gn \\ nw & ns & ng & nn \end{pmatrix} \circ \begin{pmatrix} \lambda_w & \lambda_s & \lambda_g & \lambda_n \\ \lambda_w & \lambda_s & \lambda_g & \lambda_n \\ \lambda_w & \lambda_s & \lambda_g & \lambda_n \\ \lambda_w & \lambda_s & \lambda_g & \lambda_n \end{pmatrix}_{t+1} \circ \begin{pmatrix} \beta_w & \beta_s & \beta_g & \beta_n \\ \beta_w & \beta_s & \beta_g & \beta_n \\ \beta_w & \beta_s & \beta_g & \beta_n \\ \beta_w & \beta_s & \beta_g & \beta_n \end{pmatrix}_{t+1}}{\begin{pmatrix} \beta_w & \beta_w & \beta_w & \beta_w \\ \beta_s & \beta_s & \beta_s & \beta_s \\ \beta_g & \beta_g & \beta_g & \beta_g \\ \beta_n & \beta_n & \beta_n & \beta_n \end{pmatrix}_t}$$

*Hinweis zur Notation:* *Die normale Matrixmultiplikation wird mit dem Symbol $*$, die elementweise Multiplikation mit $\circ$ und die später verwendete elementweise Division mit $\oslash$ gekennzeichnet. Auch der normale Bruchstrich symbolisiert die elementweise Division bei Matrizen.*

In diesem Kontext steht das w für Weiß, s für Schwarz, g für Grün und n für None. Die erste Matrix enthält die Wahrscheinlichkeiten für die Position $t$ aus dem *Forward-Backward-Algorithmus*, symbolisiert durch $\gamma$. Die Transitionsmatrix befindet sich an zweiter Stelle, die Bedeutung des Inhaltes entspricht der zuvor genannten Transitionsmatrix in Tabelle 4.1. Die Kürzel stehen jeweils für ein Tupel zweier Zustände die aufeinander treffen. Nach der Transitionsmatrix folgt der Vektor, der die Sensorwahrscheinlichkeit $\lambda$ für den Pixel an Position $t+1$ enthält. Zum Schluss folgt die *Backward*-Wahrscheinlichkeit an Position $t+1$. Die Matritzen im Zähler des Bruches werden elementweise multipliziert.

Das Ergebnis dieser Rechnung ist eine Matrix die mittels *Hadamard-Division*, also ebenfalls elementweise, durch die *Backward*-Matrix von Position $t$ geteilt. Bei der Transformation in die Matrix/ Vektor-Schreibweise war es nötig, den *Gamma*- und die *Backward*-Vektoren in eine Matrix zu formatieren, damit eine einfache elementweise Multiplikation möglich ist. Hierfür muss beachtet werden, dass die *Gamma*-Matrix

aus vier *Gamma*-Vektoren besteht, die spaltenweise eingefügt wurde. Dies steht im Gegensatz zu der *Backward*-Matrix im Zähler der Gleichung, denn sie besteht aus *Backward*-Vektoren, die zeilenweise eingefügt wurden. Die *Backward*-Matrix im Nenner ist wiederum wie die *Gamma*-Matrix angeordnet. Dies ist nur so möglich, da sonst die originale Formel verfälscht würde.

Diese 1:1 Übertragung in Vektor-/ Matrixschreibweise hat den Nachteil, dass viele Informationen doppelt in den Matrizen enthalten sind. Auch müssen die meisten dieser Matrizen erstellt und befüllt werden, was natürlich später negativ auf die Echtzeitfähigkeit des Programmes auswirkt, schließlich wird diese Formel für jeden Pixel einer *Scanline* ausgewertet. Zwar wird der Algorithmus nur in der Lernphase eingesetzt, also nicht während eines Wettkampfspiels, schnell muss er trotzdem sein, um auf dem NAO laufen zu können. Deshalb wurde die Gleichung umgebaut, mit dem Ziel möglichst viele Vektoren nutzen zu können, die schon existieren und möglichst wenig Matrizen erzeugen zu müssen. Im ersten Schritt wurde hierfür die Matrix im Nenner mit der *Gamma*-Matrix aus dem Bruch gezogen, sodass folgende Gleichung entsteht:

$$\xi(t) = \frac{\begin{pmatrix} \gamma_w & \gamma_w & \gamma_w & \gamma_w \\ \gamma_s & \gamma_s & \gamma_s & \gamma_s \\ \gamma_g & \gamma_g & \gamma_g & \gamma_g \\ \gamma_n & \gamma_n & \gamma_n & \gamma_n \end{pmatrix}_t}{\begin{pmatrix} \beta_w & \beta_w & \beta_w & \beta_w \\ \beta_s & \beta_s & \beta_s & \beta_s \\ \beta_g & \beta_g & \beta_g & \beta_g \\ \beta_n & \beta_n & \beta_n & \beta_n \end{pmatrix}_t} \circ \begin{pmatrix} ww & ws & wg & wn \\ sw & ss & sg & sn \\ gw & gs & gg & gn \\ nw & ns & ng & nn \end{pmatrix} \circ \begin{pmatrix} \lambda_w & \lambda_s & \lambda_g & \lambda_n \\ \lambda_w & \lambda_s & \lambda_g & \lambda_n \\ \lambda_w & \lambda_s & \lambda_g & \lambda_n \\ \lambda_w & \lambda_s & \lambda_g & \lambda_n \end{pmatrix}_{t+1} \circ \begin{pmatrix} \beta_w & \beta_s & \beta_g & \beta_n \\ \beta_w & \beta_s & \beta_g & \beta_n \\ \beta_w & \beta_s & \beta_g & \beta_n \\ \beta_w & \beta_s & \beta_g & \beta_n \end{pmatrix}_{t+1}$$

Es ist leicht zu sehen, dass der Bruch nun viele redundante Berechnungen enthält. Dies wird verhindert, indem eine Matrix V definiert wird, die den Bruch auf vier elementweise Divisionen verkürzt und die Ergebnisse in der Diagonalen speichert. Da im B-Human-Umfeld mit der Mathematikbibliothek Eigen gearbeitet wird, lässt sich dies als ein normaler Vektor speichern, der als später als Diagonalmatrix verwendet werden kann.

Die gleiche Art der Optimierung bietet sich auch bei den letzten zwei Matrizen an. Sie werden in der Diagonalmatrix W gespeichert.

$$V(t) = \begin{pmatrix} \frac{\gamma_w}{\beta_w} & 0 & 0 & 0 \\ 0 & \frac{\gamma_s}{\beta_s} & 0 & 0 \\ 0 & 0 & \frac{\gamma_g}{\beta_g} & 0 \\ 0 & 0 & 0 & \frac{\gamma_n}{\beta_n} \end{pmatrix}_t \qquad W(t{+}1) = \begin{pmatrix} \lambda_w * \beta_w & 0 & 0 & 0 \\ 0 & \lambda_s * \beta_s & 0 & 0 \\ 0 & 0 & \lambda_g * \beta_g & 0 \\ 0 & 0 & 0 & \lambda_n * \beta_n \end{pmatrix}_{t+1}$$

Bildet man nun eine Formel mit der Transitionsmatrix muss nun die gängige Matrixmultiplikation verwendet werden, um das selbe Ergebnis zu haben. Die finale Berechnung, die auch in der Implentierung so verwendet wurde, sieht wie folgt aus:

$$\xi(t) = V(t) * T * W(t+1)$$

$$\xi(t) = \begin{pmatrix} \frac{\gamma_w}{\beta_w} & 0 & 0 & 0 \\ 0 & \frac{\gamma_s}{\beta_s} & 0 & 0 \\ 0 & 0 & \frac{\gamma_g}{\beta_g} & 0 \\ 0 & 0 & 0 & \frac{\gamma_n}{\beta_n} \end{pmatrix}_t * \begin{pmatrix} ww & ws & wg & wn \\ sw & ss & sg & sn \\ gw & gs & gg & gn \\ nw & ns & ng & nn \end{pmatrix} * \begin{pmatrix} \lambda_w * \beta_w & 0 & 0 & 0 \\ 0 & \lambda_s * \beta_s & 0 & 0 \\ 0 & 0 & \lambda_g * \beta_g & 0 \\ 0 & 0 & 0 & \lambda_n * \beta_n \end{pmatrix}_{t+1}$$

Diese Konstruktion bietet den Vorteil, dass die Daten, die bisher bereits als Vektor vorliegen, ohne Formatierung im *Baum-Welch-Algorithmus* nutzbar sind. So wird eine klare und einfache Implementierung erreicht, die sowohl lesbar ist, als auch Fehler in der Verwendung des Algorithmus verhindert. Es ist leicht zu sehen, dass zwar die Notation geändert wurde, die Ergebnis der ursprünglichen Formel jedoch nicht.

Bisher wird nur das $\xi$ an einer Position berechnet. Möchte man später die Wahrscheinlichkeiten über eine bestimmte Zeitperiode haben, muss man die relative Häufigkeit eines Überganges bestimmen. Die geschieht in dem man die Summe aller $\xi$ ($\xi_{sum}$) und $\gamma$ ($\gamma_{sum}$) bestimmt und sie am Ende mit elementweiser Division verrechnet. Jede Spalte von $\xi_{sum}$ wird mittels Hadamard-Divison elementweise durch den $\gamma_{sum}$-Vektor geteilt.

Nachdem ein ganzes Bild mit Scanlines analysiert wurde, wird die verbesserte Transitionsmatrix $\tilde{T}$ erstellt. Auch dies wird wieder in Matrix bzw. Vektorschreibweise umgewandelt.

$$\tilde{T} = \begin{pmatrix} \tilde{ww} & \tilde{ws} & \tilde{wg} & \tilde{wn} \\ \tilde{sw} & \tilde{ss} & \tilde{sg} & \tilde{sn} \\ \tilde{gw} & \tilde{gs} & \tilde{gg} & \tilde{gn} \\ \tilde{nw} & \tilde{ns} & \tilde{ng} & \tilde{nn} \end{pmatrix} = \frac{\begin{pmatrix} \xi_{ww} & \xi_{ws} & \xi_{wg} & \xi_{wn} \\ \xi_{sw} & \xi_{ss} & \xi_{sg} & \xi_{sn} \\ \xi_{gw} & \xi_{gs} & \xi_{gg} & \xi_{gn} \\ \xi_{nw} & \xi_{ns} & \xi_{ng} & \xi_{nn} \end{pmatrix}}{\begin{pmatrix} \tilde{\gamma_w} & \tilde{\gamma_w} & \tilde{\gamma_w} & \tilde{\gamma_w} \\ \tilde{\gamma_s} & \tilde{\gamma_s} & \tilde{\gamma_s} & \tilde{\gamma_s} \\ \tilde{\gamma_g} & \tilde{\gamma_g} & \tilde{\gamma_g} & \tilde{\gamma_g} \\ \tilde{\gamma_n} & \tilde{\gamma_n} & \tilde{\gamma_n} & \tilde{\gamma_n} \end{pmatrix}}$$

Da mehrere Szenen und Spielsituationen einbezogen werden sollten, bietet es sich an, den *Baum-Welch-Algorithmus* über mehrere Bilder hinweg laufen zu lassen. Hierfür wird eine Transitionsmatrix $\hat{\mathrm{T}}$ berechnet, die mit jedem Bild aktualisiert wird. Dies geschieht einfach über das arithmethisches Mittel pro Feld der Matrix, unter Berücksichtigung der Anzahl der bisherigen analysierten Bilder p. Die Rechnung wird durch diese Gleichung ausgedrückt:

$$\hat{T}(p+1) = \frac{\hat{T}(p) * p + \tilde{T}(p)}{p+1}$$

Die Implementierung des Algorithmus befindet sich im selben Modul, wie der *Forward-Backward-Algorithmus.* Begründet ist dies durch die Nutzung dessen und der Berech-

nung des Sensormodells durch den *Baum-Welch-Algorithmus*.

```
Function Baum-Welch-Algorithmus
    Data: Distanz d;
          Scanline l;
          Scangrid g;
          Transitionsmatrix T;
    Result: Verbesserte Transitionsmatrix
    begin
        Aktualisiere Gaussian-Mixture-Model;
        Buffer forwardBuffer;
        Buffer backwardBuffer;
        Vektor gammaSum;
        Matrix xiSum;
        Vektor forward, backward, gamma;
        foreach Pixel p in aktuelle Scanline der ScanGrid do
            Vektor prevBackward = backward;
            Vektor prevGamma = gamma;
            gamma = Forward-Backward-Algorithmus(x, y, forwardBuffer,
             backwardBuffer, forward, backward);
            if Position des Pixels in der Scanline >= d then
                Vektor firstTerm = prevGamma ⊘ prevBackward;
                Vektor lastTerm = forwardBuffer.back() ∘ backward;
                xiSum += firstTerm.asDiagonal() ∗ T ∗ lastTerm.asDiagonal();
                gammaSum +=gamma
            end
        end
        Matrix improvedMatrix;
        for int i = 0; i < 4; i++ do
            improvedMatrix.Spalte(i) = xiSum.Spalte(i) ⊘ gammaSum;
        end
        improvedMatrix.Normalisiere();
        if Durchläufe = 1 then
            newMatrix = T;
        end
        newMatrix = (newMatrix * Durchläufe) + improvedMatrix) / (Durchläufe
         +1);
    end
```

Eine Anleitung wie sich im dieser *Erwartungs-Maximierungs-Algorithmus* im B-Human-System verwenden lässt, befindet sich in einer *ReadMe*-Datei, die dem Quelltext beigefügt ist.

Die für den im weiteren Verlauf der Bachelorarbeit verwendete Transitionsmatrix wurde auf Bildern trainiert, die die Licht- und Feldverhältnisse unter Wettbewerbsbedingungen repräsentieren. Dieses Log wurde bei der Weltmeisterschaft in Montréal, Kanada in einem Testspiel aufgezeichnet und lieferte folgende Matrix:

$$TransitionMatrix = \begin{pmatrix} 0.624912 & 0.0600562 & 0.266327 & 0.0507025 \\ 0.112595 & 0.368461 & 0.4586 & 0.0623429 \\ 0.0529769 & 0.0504067 & 0.852235 & 0.0463799 \\ 0.136641 & 0.0923846 & 0.578245 & 0.194729 \end{pmatrix}$$

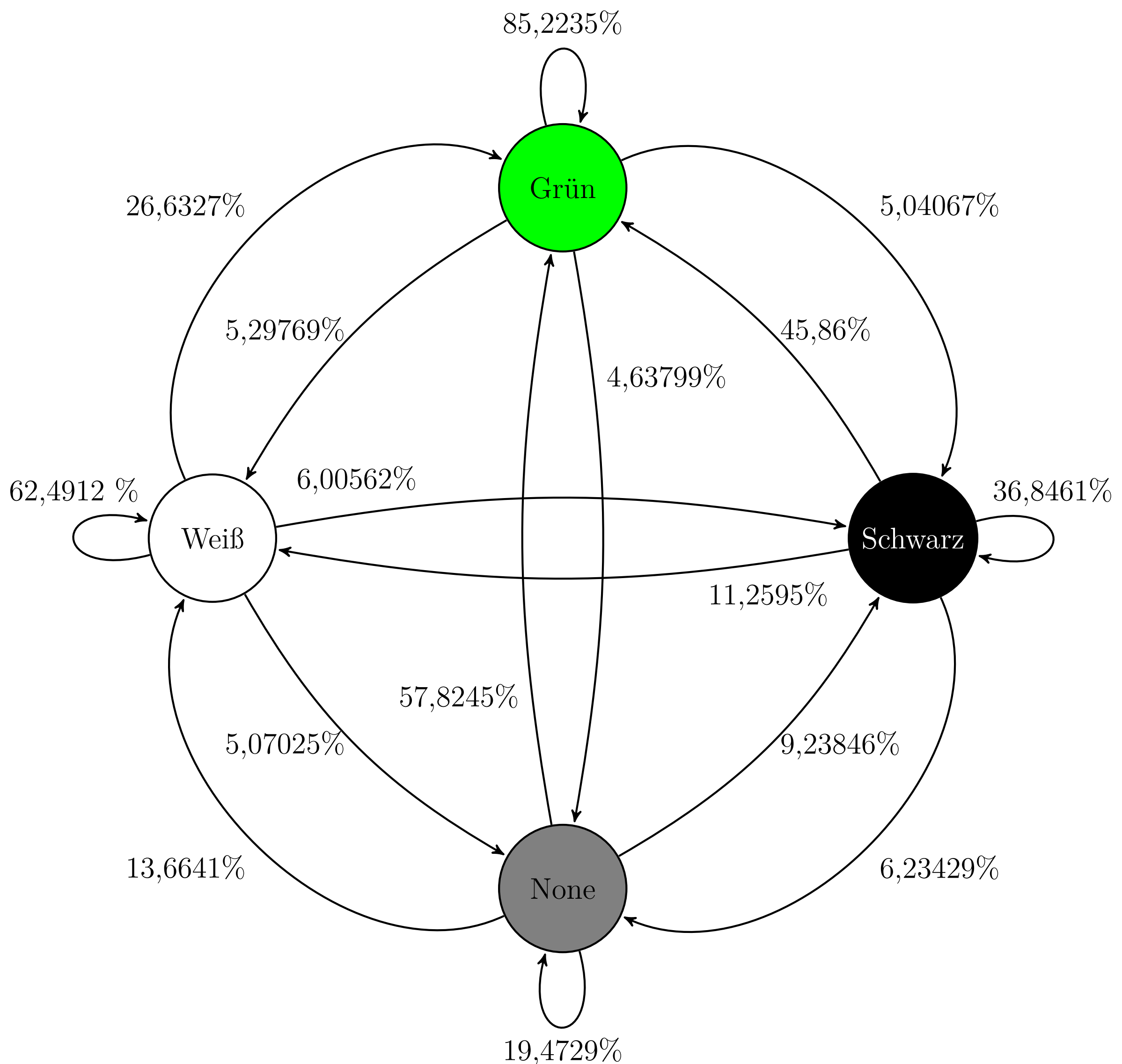



Abbildung 5.2: Das vollvermaschte Netz der Zustände mit den berechnenten Übergangswahscheinlichkeiten, die im weiteren Verlauf verwendet wurde.

Der *Baum-Welch-Algorithmus* findet nur das lokale Maximum, weswegen darauf zu achten ist, ob diese Werte sinnvoll sind. In dieser Matrix weisen die Übergange zu Grün eine hohe Wahrscheinlichkeit auf, da Grün aufgrund der Feldfarbe der mit Abstand häufigste Zustand ist. Ebenfalls auffallend ist die erhöhte Wahrscheinlichkeit bei Übergängen, wo sich der Zustand nicht ändert (Weiß → Weiß 62%, Schwarz → Schwarz 36%). All dies führte zu der Bewertung, dass sich diese Matrix gut eignet, um als Transitionsmatrix für den *Forward-Backward-Algorithmus* verwendet zu werden.

# 6. Evaluation

Die Bewertung der Qualität der Einteilung in Regionen ist nicht einfach, da Fehler nicht maschinell erkannt werden können. Die Zustände des *Hidden-Markov-Models* sind auch in der Nachbetrachtung versteckt und können nur durch andere Techniken, wie maschinelles Lernen oder anderes aufgedeckt werden, die jedoch den Rahmen dieser Bachelorarbeit übersteigen würden. Eine weitere Möglichkeit ist es, das menschliche Auge zu nutzen und Fehler händisch zu bestimmen. Dieser Ansatz ist der Ausgangspunkt für die Evaluation der Bachelorarbeit. Um einen guten Vergleich zu haben und zu bestimmen wie viele Fehler gut, bzw. schlecht sind, wurde ein kleines Modul geschrieben, welches mögliche Fehler auf Bildern markiert. Diese „Fehler“ sind anfangs die Unterschiede zwischen dem momentan eingesetzen Modul zur *Scanline*-Einteilung und dem für diese Bachelorarbeit erstellten *HMM-ColorScanlineRegionizers*. Es zeigt für jede *Scanline* die *Scanline*-Regionen beider Module nebeneinander an und markiert die Unterschiede mit einem kleinen roten Strich. Das Evaluationsmodul kann im Simulator auf Logs oder Simulations-Szenen angewandt werden. Eine Anleitung zur Verwendung des Moduls befindet sich in einer *ReadMe*-Datei, welche dem Quelltext beiliegt.

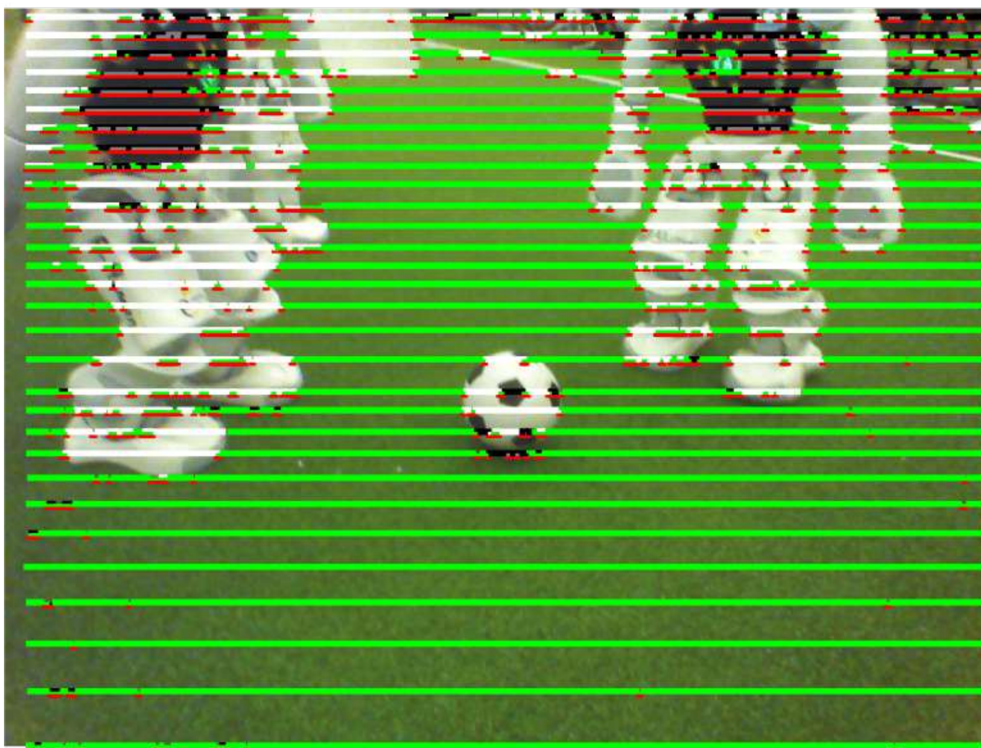


Abbildung 6.1: Unterschiede der *Scanlines*. Spielszene aus Montréal mit mehreren Robotern und dem Ball. Aufgenommen mit der oberen Kamera.

In Abbildung 6.1 sieht man ein beispielhaftes Bild aus dem in Montréal aufgenommenen Log. Zu sehen ist eine Spielszene, die mehrere Roboter und den Ball zeigt. Auch eine Feldlinie kann man im Hintergrund rechts erkennen. In diesem Fall können mit diesem Bild die horizontalen *Scanlines* des neuen mit denen des bisherigen Moduls verglichen werden. In der Darstellung ist jede *Scanline* zweigeteilt: Die obere Hälfte ist die Scanline des bestehenden Moduls, die untere Hälfte die des HMM-Moduls. Bei der Betrachtung vertikaler *Scanlines* befinden sich die Regionen des alten Moduls auf der linken Seite, die des HMM-Modul rechts daneben.

Die Unterschiede wurden im nächsten Schritt händisch durchgesehen und nach folgender Skala bewertet:

| **Minimaler Unterschied** | Kleine Abweichung bei Beginn/ Ende einer Feldlinie oder eines Objekts | 0 Fehlerpunkte |
|---|---|---|
| **Kleiner Fehler** | Einige, wenige Pixel nebeneinander falsch | 1 Fehlerpunkt |
| **Mittlerer Fehler** | Deutlicher Unterschied durch falsche Pixel | 2 Fehlerpunkte |
| **Großer Fehler** | Längere Strecke offensichtlich falsch eingeteilt | 3 Fehlerpunkte |
| **Freie Pixel** | Nicht eingeteilte und dadurch freigelassene Pixel eingeteilt in einer der obigen Kategorie. | Hälfte der Fehlerpunkte der oberen Kategorien. |

Für eine möglichst realisische Einschätzung wurde die Evaluation auf drei Logs durchgeführt, die eine möglichst gute Breite in der Farbgebung haben sollten. Das erste Log stammt aus Montréal, von einem Platz auf dem 2018 die RoboCup Weltmeisterschaft ausgetragen wurde. Auf diesem Log wurde bereits die Übergangsmatrix gelernt. Es stellt optimale Licht- und Feldverhältnisse dar. Als zweites Los wurde eine Aufnahme eines Testspiels von B-Human in einem verhältnismäßig dunklem Raum gewählt. Das dritte Log stammt ebenfalls von einem Testspiel, welches jedoch nahe einer großen Fensterfläche ausgetragen wurde. So kam sehr viel direktes Sonnenlicht auf das Feld und dadurch sehr helle Flächen und viele Schatten. Mit diesen drei Logs lassen sich die möglichen Extremumgebungen in einem Robocupspiel simulieren.

Aus jedem Log wurden 10 Spielszenen ausgewertet, für jede Spielszene wurden sowohl horizontale als auch vertikale Scanlines evaluiert. So kommen pro Log 20 Bilder zusammen, die händisch bewertet wurden. Von diesen 20 Bildern stammten 14 aus der oberen Kamera und 6 aus der unteren. Der Grund für die geringere Anzahl an Bildern aus der unteren Kamera besteht in den deutlich weniger vielfältigen Situationen, da aufgrund der Perspektive sehr häufig das selbe zu sehen ist.

Es wurde versucht die Spielszenen möglichst repräsentativ für das jeweilige Log auszuwählen. Auch die Situationen sollen über alle drei Logs hinweg ähnlich sein. Es wurde versucht Bilder mit möglichst hoher Reichweite, Zweikämpfen, Feldlinien und mit Ball auszuwählen. Die Auswahl ist aufgrund der Vorgehensweise subjektiv. Auch die Bewertung ist subjektiv und kann nicht als eine zuverlässige Quelle für einen quantitativen Vergleich gesehen werden. Dieser Teil der Evaluation dient nur zu einer groben Einschätzung der implementieren Lösung.

## 6.1 Evaluation des Montréal-Logs

In Abbildung 6.4 sieht man das Ergebnis der Evaluation des Logs, welches Wettbewerbsbedingungen widerspiegelt (vgl. Abbildung 6.2). Man kann erkennen, dass für die obere Kamera die Fehlerzahl beider Module in etwa gleichauf liegen. Trotz vergleichsweiser hoher Fehlerzahl, hat das HMM-Modul einige hundert Fehlerpunkte weniger gesammelt. Ähnliches Ergebnis zeigt auch die Fehlerzahl bei den Bildern

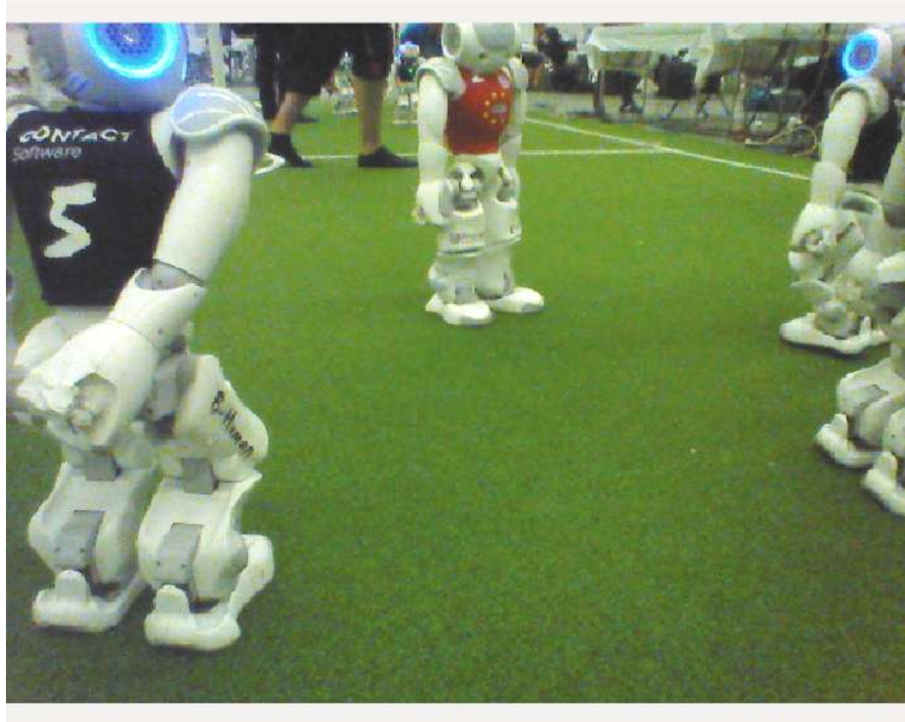



Abbildung 6.2: Beispielhafte Spielszene aus Montréal.

der unteren Kamera. Dort ist im direkten Vergleich das neu entwickelte Modul der Gewinner. Gleichwohl muss man beachten, dass hier die Fehleranzahl nur im Vergleich zu dem bisherigen Modul gering erscheint.

Viele der, vor allem in den Bildern der oberen Kamera auftretenden Fehler, zeigen ein grundsätzliches Problem des HMM-Moduls. Die Einteilung in die Farbklasse Schwarz geschieht viel zu selten. Häufig werden Pixel, die eigentlich schwarz sind, in die Restklasse eingeordnet. Dies führt zu einer erhöhten Fehlerzahl, da auf vielen Bildern B-Human-Roboter in schwarzen Trikots zu sehen sind. Auch ist die Einteilung weißer Pixel in die Restklasse None ein Problem, wie in Abbildung 6.3 zu sehen ist. Tatsächlich machen die fehlerhaften Pixel der Trikots und der Linien das Gros der Fehlerpunkte des HMM-Moduls aus, sodass das Rauschen des Feldes nur für einen Bruchteil der tatsächlichen Fehler verantwortlich ist.

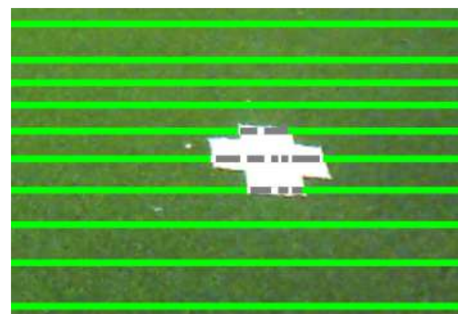


Abbildung 6.3: Der Elfmeterpunkt wird für einen Moment zum Großteil in die Restklasse None eingeteilt (dargestellt durch die graue Farbe).

Nimmt man die später vorgestellten Ergebnisse hinzu, sieht man, dass das alte Modul scheinbar Probleme auf diesem Log hatte und möglicherweise die Farben nicht richtig kalibriert wurden. Sichtbar wird dies bei Betrachtung der Bilder. An den äußeren Rändern vieler Bilder lässt die Sättigung der Farbe nach. Dies führt dazu, dass Pixel häufiger falsch in die Farbklasse Schwarz eingeordnet werden und weniger in die grüne Farbklasse. Das neue Modul hat dieses Problem in weitaus weniger starken Ausmaßen, was auch an der möglichen Adaptierung des *Gaussian-Mixture-Models* an die Farbgebung liegt. Da dieses Log in einem Testspiel aufgenommen wurde, kann es ebenfalls sein, dass die Farben noch nicht perfekt kalibriert wurden und die richtige Farbgebung erst im späteren Verlauf gefunden wurde.

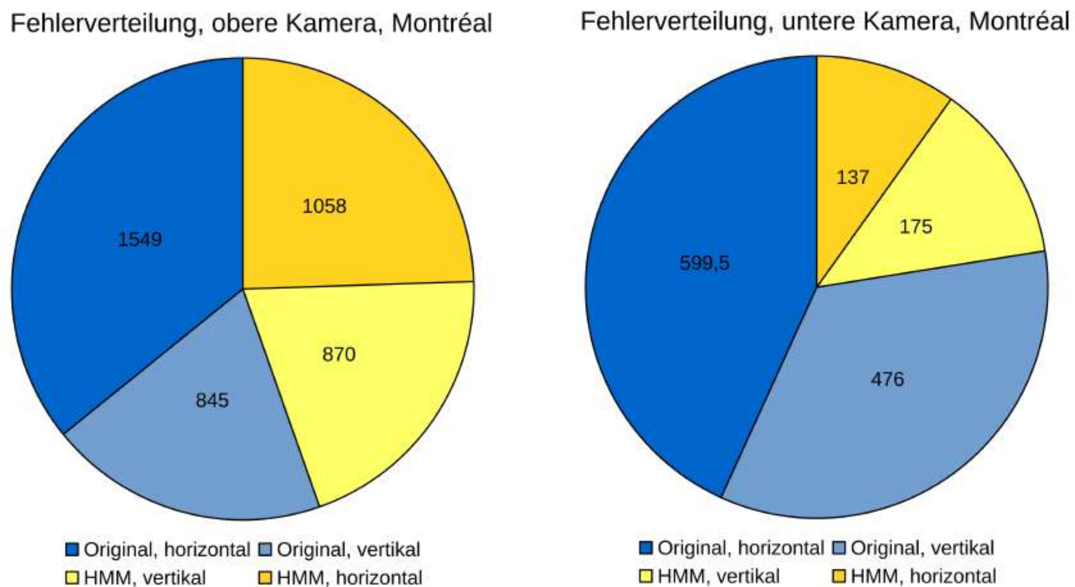



Abbildung 6.4: Ergebnis der Evaluation des Logs mit Wettbewerbsbedingungen.

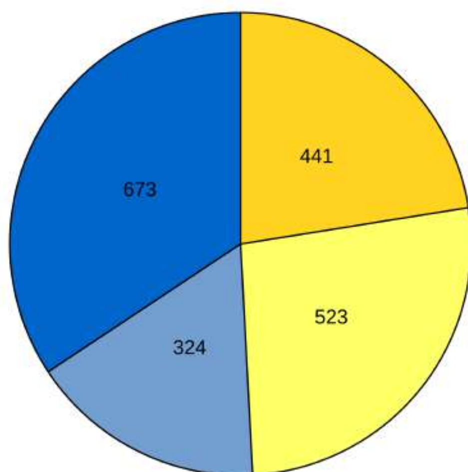
## 6.2 Evaluation des Logs aus MZH 1400

Dieses Log hat sich durch seine dunkle Farbgebung, beispielhaft zu sehen in Abbildung 6.5, ausgezeichnet und wurde deswegen für die Evaluation gewählt und bewertet. Ein häufiges Phänomen war die falsche Zuordnung von Feldbereichen in die Farbklasse Schwarz. Grund hierfür dürften die schlechten Lichtverhältnisse sein, die es auch dem menschlichen Betrachter ohne Vorwissen schwer machen würde, die Pixel in die Farbklassen einzuteilen. Dies ist ein Problem beider Module und kann momentan nur durch richtige Kalibrierung der Farbklassen behoben werden. Da das HMM-Modul nicht kalibriert wird, sondern sich selbst mit Hilfe des *Gaussian-Mixture-Models* aktualisiert, ist dieses Log eine Probe, inwieweit sich das GMM an eine dunkle Umgebung adaptieren kann. Das Ergebnis is in Abbildung 6.6 dargestellt.

Während in der oberen Kamera die Fehlerzahl nahezu perfekt ausgeglichen ist, unterscheiden sich die Module bei der Qualität der Einteilung auf den Bildern der unteren Kamera massiv. Besonders die vertikalen Scanlines des HMM-Moduls erweisen sich im Vergleich scheinbar als sehr fehleranfällig. Betrachtet man das Log genauer, fällt früh die Ursache auf, die auch in anderen Logs auftritt:

Abbildung 6.5: Beispielhafte Spielszene aus dem MZH 1400.

Das HMM-Modul schwankt häufig bei der Einteilung in der Restklasse. Hier wird sehr häufig für einige Bilder sehr viel falsch eingeteilt, bis dies automatisch korrigiert wird. So ist in diesem Log mehrmals die Farbklasse Weiß komplett in der Restklasse None verschwunden, wodurch alle Feldlinien für ein paar Sekunden massiv falsch klassifiziert wurden. Wie auch bei der Evaluation des Logs aus Montréal bereits erwähnt, geschieht auch auf diesem Log die falsche Einteilung in die Restklasse.



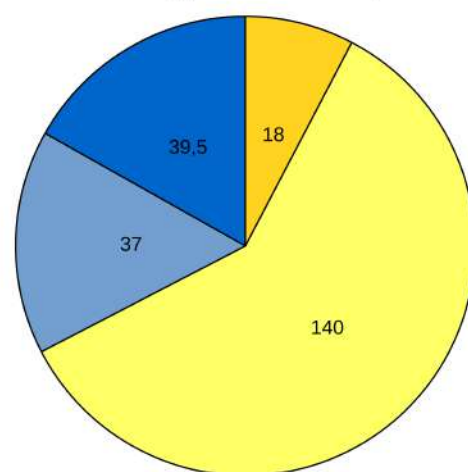




Abbildung 6.6: Ergebnis der Evaluation des Logs einer dunklen Umgebung.

Ein weiterer Umstand, der dem HMM-Modul in diesem Log Probleme bereitet hat, ist der Untergrund des Raumes, in dem gespielt wurde und auf dem der Kunstrasen des Feldes lag. Während bei Wettbewerben wie in Montréal zumeist ein kontrastreicher Fußbodenbelag vorliegt, liegt im Raum dieses Logs ein holzfarbiger Pakettbelag. Dieser wurde vom GMM zumeist als Grün klassifiziert, wodurch eine klare Unterscheidung zwischen Fußboden und Feld nicht mehr möglich ist. Dies würde im Spiel Probleme bereiten, da die Feldgrenze, also die Grenze zwischen Feld und außerhalb des Feldes, nicht mehr richtig berechnet werden kann.

## 6.3 Evaluation des Logs aus dem Foyer des MZH

Das dritte Log stammt aus einem Foyer mit einer großen Fensterfront (siehe Abbildung 6.7) und wurde wegen des dynamischen Tageslichts ausgewählt. Die vielen Schatten machten weitaus weniger Probleme als anfangs vermutet, obwohl sie einen massiven Kontrast zum hellen Boden bildeten. Das größte Problem war das direkte Sonnenlicht, welches das Feld in einem ungünstigen Winkel anschien und dadurch manche Stellen weiß wirkten.

Abbildung 6.7: Beispielhafte Spielszene aus dem MZH Foyer.

Das Ergebnis, zu sehen in Abbildung 6.8, ist ähnlich dem des vorherigen Logs. Der Unterschied bei der oberen Kamera ist relativ ausgeglichen, wenn auch diesmal das bisherige Modul eindeutig weniger Fehler gemacht hat, als das neu implementierte Modul. Das Ergebnis der unteren Kamera unterscheidet sich zwar stark, darf aber nicht überbewertet werden. Auf dem ersten Blick ist das neue Modul klar schlechter, aber die Höhe der Fehlerpunkte (36 und 34 Fehlerpunkte) ist verglichen mit den Ergebnissen der anderen Logs (137 und 175, bzw. 18 und 140 Fehlerpunkte) auf einem sehr guten Niveau.

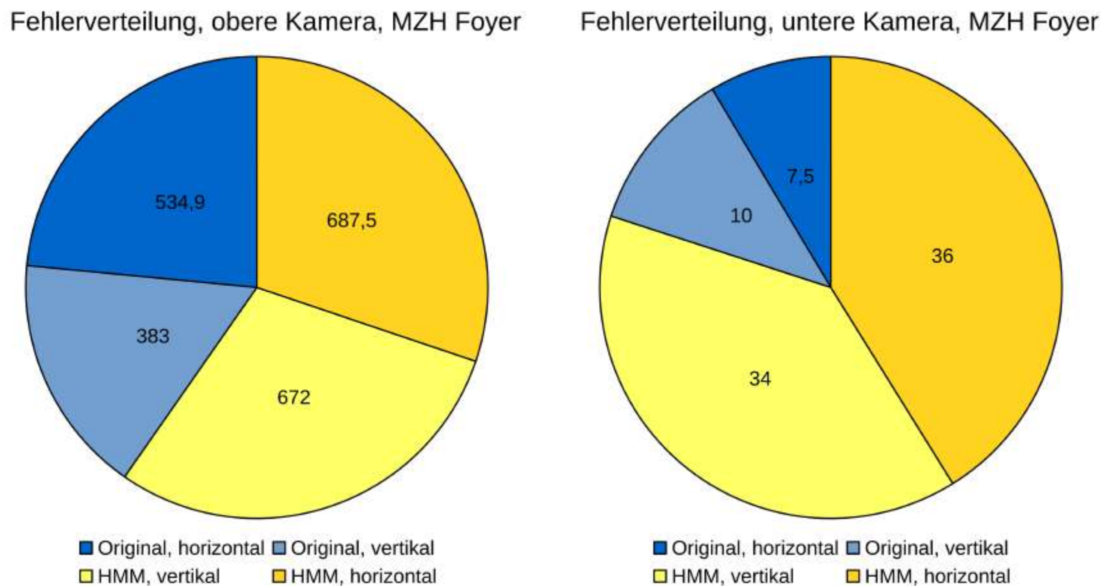



Abbildung 6.8: Ergebnis der Evaluation eines Logs aus eine Umgebung mit Tageslicht

## 6.4 Evaluation der Zeit

Die Laufzeit der Implementierung ist im Kontext des Roboterfußballs von essenzieller Bedeutung, da ein Modul, welches nicht echtzeitfähig ist, auch nicht auf den Robotern während eines Spiels laufen kann. Deshalb war es ein Ziel dieser Bachelorarbeit, das Modul so schnell wie möglich zu gestalten. Das Ergebnis lässt sich nur überprüfen, wenn auf einem NAO die Software ausgeführt wird. Dies wurde zuerst auf der momentan auf Wettbewerben gängigen NAO V5, also dem NAO der fünften Generation, getestet. Das Ergebnis zeigt, dass die Ausführung leider nicht echtzeitfähig und somit in diesem Zustand auch nicht einsatzfähig ist.

Filter:

| Stopwatch | Min | Max | Avg |
|---|---|---|---|
| Cognition | 13.725 | 690.332 | 300.164 |
| HMMCSR:getSensorModelForPixel | 3.534 | 463.606 | 199.456 |
| ColorScanlineRegionsHorizontal | 3.998 | 335.866 | 140.859 |
| ColorScanlineRegionsVertical | 0.849 | 250.262 | 117.031 |
| ColorScanlineRegionsVerticalClipped | 0.088 | 78.931 | 13.1269 |
| CameraImage | 2.988 | 27.201 | 11.4895 |

Abbildung 6.9: Laufzeiten auf einem NAO der fünften Generation.

In 6.9 werden die Zeiten aufgeführt, die der *HMMColorScanlineRegionizer* für die horizonalen *Scanlines ColorScanlineRegionsHorizontal* und für das vertikale Pendant *ColorScanlineRegionsVertical* benötigt. Auch die Zeiten für die Berechnung der *ColorScanlineVerticalClipped* durch das zweite Modul *HiResHMMColorScanlinesRegionizer* werden aufgezeigt.

Im Vergleich zu dem alten Modul, welches für die *Scanlines* im Maximum stets unter einer Millisekunde liegt, benötigt das neue Modul mit bis zu 336 ms für die horizontalen *Scanlines* deutlich zu lange für die Bilder. Ein Grund dafür ist die Berechnung der

vier Wahrscheinlichkeiten durch den *Forward-Backward-Algorithmus*. Zwar wurde dieser bereits auf möglichst wenige Rechenoperationen optimiert, trotzdem sind noch viele Multiplikationen vorhanden. Es ist zu beachten, dass all dies pro Pixel passiert, was sich immens auf die Laufzeit auswirkt. Im Vergleich dazu ist die Implementierung im alten Modul wesentlich schmaler und kostet so auch weniger Laufzeit.

Weiterhin hat die Berechnung der Sensorwahrscheinlichkeit eine massive Verlangsamung der Software zur Folge. Dieser Bereich kostet allein ein knappes Drittel der Rechenzeit, wie in 6.9 unter dem zweiten Punkt *HMMCSR:getSensorModelForPixel* abzulesen ist. Die gesamte Kognition inkl. Scanlinekalkulation dauert mit neuem Modul ca. 690 ms maximal, davon benötigt das Sensormodell nur für die normalen horizontalen und vertikalen *Scanlines* 463 ms.

Eine weitere Eigenart des Vorgehens, welche zur erhöhten Laufzeit führt, ist die nicht optimale Ausnutzung der *Scangrid*. Während das alte Modul Bilder durch die *Scangrid* an einzelnen Punkten betrachtet und bei Unterschied der Pixel den Bereich dazwischen genauer analysiert, kann das neue Modul dieses effiziente Vorgehen nicht verwenden. Das *Hidden-Markov-Model* benötigt für die Einteilung eines Pixels den Kontext vor und nach dem Pixel, dadurch kann man keinen Pixel einzeln ohne seine „Nachbarschaft“ in eine Farbklasse einteilen. Deshalb kann die *Scangrid* nicht voll ausgenutzt werden und es muss jeder Pixel einer *Scanline* betrachtet werden. Dies kostet natürlich auch einiges an Rechenzeit.

# 7. Fazit

Die Ziele der Bachelorarbeit wurden nur zum Teil erreicht. Es wurde ein Modul geschaffen, welches Bilder mittels *Hidden-Markov-Model* farbsegmentieren und *Scanlines* mit einer akzeptablen Quote richtig in Regionen einteilen kann. Diese Segmentierung birgt jedoch noch einige Fehler und ist vor allem in der jetzigen Form nicht auf dem NAO anwendbar, da es viel zu langsam ist. Eine wichtige Ursache für die trotz höherer Laufzeit auch etwas höhere Fehleranfälligkeit, ist das genutzte *Gaussian-Mixture-Model*, welches für die Sensorwahrscheinlichkeit verwendet wird.

Das GMM in seiner ursprünglichen Form hat mit einigen Problemen zu kämpfen, wie etwa die fehlerhafte Einteilung der Farbe Schwarz. Dies zeigt sich in dem bereits beschriebenen Phänomen, dass schwarze Trikots durch das GMM in der Farbklasse None eingeordnet werden. Dies passiert auch zum Teil bei weißen Feldlinien und den weißen Flächen des Roboters.

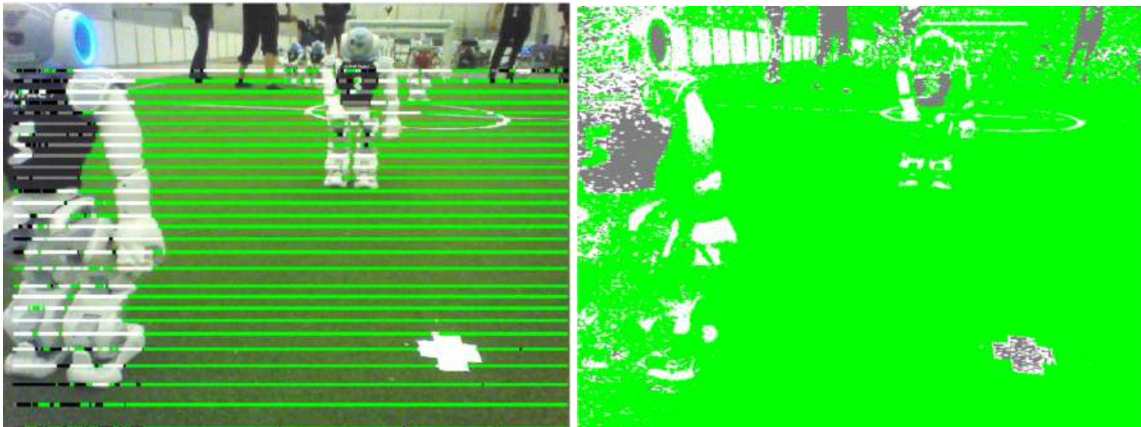


Abbildung 7.1: Die mit HMM eingeteilten horizontalen Scanlines (links) im Vergleich zur Farbklassifizierung nur durch das GMM (rechts).

Ein weiterer häufiger Fehler ist die fehlerhafte Einteilung der Farbe Schwarz in die Farbklasse Grün, sodass ganze Trikots im Feld „verschwinden“. Dieses Problem ist in der vom HMM verwendeten Version weitaus weniger vorhanden. Im Gegenzug hat das GMM bei der Klassifizierung des Bildes ein weitaus glatteres und rauschfreieres Bild vom grünen Feld, während es in den vom HMM-Modul generierten *Scanlines* vorkommt, dass einzelne Pixel einer falschen Farbklasse zugeordnet werden. Dies ist direkt unter dem Roboter am vorderen linken Rand in Abbildung 7.1 zu sehen. Außerdem ist dort der Elfmeterpunkt in den *Scanlines* weiß eingeteilt, während das GMM den Punkt in die Restklasse einordnet. Das zeigt, dass es, obwohl das *Hidden-Markov-Model* Informationen des GMM nutzt, zum Teil große Unterschiede in der Farbklassifizierung gibt.

Dieses Problem liegt in der Machart des GMM. In der reinen Form wird die Einteilung vorgenommen, indem eine Hierarchie beachtet wird. Zuerst wird ein Pixel auf die

Farbklasse Grün geprüft. Erreicht er einen Schwellwert in der Farbe nicht, wird er auf Weiß geprüft. Dann auf Schwarz und später auf None. Dies hat zur Folge, dass z.B. die Restklasse None weitaus mehr auffangen kann und das Vorwissen nutzt, dass ein Pixel keiner der vorherigen Farbklasse zugehörig ist. Bei der Wahrscheinlichkeitsberechnung geht dies jedoch nicht, sodass die höchste Wahrscheinlichkeit für einen Zustand nicht zwangläufig der Zustand ist, den das GMM ausgewählt hätte.

Dieser Fehler kann behoben werden, indem die Nutzung des GMM auch diese Hierarchie bedenkt und in die Wahrscheinlichkeiten einfließen lässt. Dieser Lösungsansatz verschiebt das Problem jedoch in die Problematik des *Gaussian-Mixture-Models* an sich, welches die Farbe schwarz häufig ganz fehlen lässt und so Informationen verschwinden.

Eine Möglichkeit zur Verbessung der Qualität ist es, auch das *Gaussian-Mixture-Model* zu Beginn zu kalibrieren, da es in der Anwendung mit dem HMM bei Extrembedingungen noch zu Problemen kommen kann. Kalibrieren würde man wie bisher vor einem Spiel als Anpassung an die Umgebung und das GMM geht dann, in dem vorherig festgelegten Rahmen auf die veränderten Lichtsituationen während des Spiels ein.

Die Laufzeit könnte vermutlich deutlich optimiert werden, wenn die Sensorwahrscheinlichkeiten direkt in dem GMM berechnet würden, statt wie bisher die Mittelwerte, Standardabweichungen und Gewichte im GMM zu berechnen und später für das HMM-Modul ein anderes GMM zu erstellen, welches wiederum mit den eben genannten Informationen die Wahrscheinlichkeiten pro Pixel berechnet.

Diese Möglichkeiten zeigen, dass in den erarbeiteten Modulen noch Verbesserungspotential steckt und sie in diesem Zustand noch keine Alternative zu den bisher verwendeten *Scanline*-Segmentierung-Modulen ist. Das Ergebnis würde sich signifikant verbessern, wenn das verwendete *Gaussian-Mixture-Model* bessere Resultate liefern würde und wenn noch an einigen Stellen Optimierungen gemacht werden, die die Laufzeit konkurrenzfähig machen könnte. Die Arbeit hat aber auch gezeigt, dass es gut möglich ist, für diesen Anwendungszweck ein *Hidden-Markov-Model* einzusetzen.

# 8. Ausblick

Das momentane Verständnis des Begriffs „Echtzeitfähigkeit“ bezieht sich auf den NAO V5, welcher bereits aus heutiger Sicht leistungsarme Technik verwendt und somit auch von der Rechenleistung nicht mehr State-Of-The-Art ist. Der neue NAO der sechsten Generation, der aller Voraussicht nach 2019 seine Premiere bei den RoboCup-Wettbewerben feiern wird, ist technisch schon deutlich weiter fortgeschritten und liefert dementsprechend auch mit diesem Modul bessere Laufzeiten. So benötigte er mit maximal 87,9 ms für die horizontalen *Scanlines* ca. 248 ms weniger, als eine vergleichbare Messung auf dem NAO V5. Auch der Unterschied bei den vertikalen *Scanlines* ist mit 147 ms beachtlich (V5: 250 ms, V6: 103 ms). Dieser Trend zeigt deutlich, dass in einigen Jahren und mit ein paar Optimierungen, ein *Hidden-Markov-Model* sehr gut in der Lage ist, die *Scanlines* zu segmentieren. Der große Vorteil, dass direkt mit den Wahrscheinlichkeiten des *Gaussian-Mixture-Models* gerechnet werden kann, erweist sich momentan noch als Problem, da die Berechnung des Sensormodels auch auf dem V6 noch 133 ms (V5: 263 ms) beträgt und durch die falsche Einteilung in die None-Farbklasse einen Großteil der Fehler verursacht. Es bietet aber zukünftig die Chance, sich auf dynamisches Licht einzustellen und dadurch auch weiterhin die Farben der Umgebung wahrzunehmen.

Ein immer wieder aufkommenes Thema ist der komplette Verzicht auf die Farben bei der Verarbeitung der Bilder. Während in der SPL der Ball bis 2016 orange und die Tore bis 2012 blau bzw. gelb und danach beide gelb waren, wurden durch Regeländerungen diese Elemente dem menschlichen Fußball angeglichen und schwarz-weiß bzw. weiß (vgl. [RoboCup Technical Committee, 2012] und [RoboCup Technical Committee, 2016]). Dadurch haben Farben in der Bildverarbeitung im RoboCup Jahr für Jahr an Bedeutung verloren. Fehlen die Farben komplett und betrachtet man ein Graustufenbild wie Abbildung 8.1, fällt auf, dass für den menschlichen Betrachter weiterhin fast alle wichtigen Informationen ohne Probleme ersichtlich sind. Lediglich bei der Unterscheidung der Teams könnten einem Menschen Fehler unterlaufen, aber die Linien, der Ball und die Roboter sind klar erkennbar.

Die fehlende Bestimmung der Farbklassen und den weiteren Verzicht dieser Informationen bei der Bildverarbeitung haben den Vorteil, dass Fehler bei der Farbbestimmung verhindert werden können. Auch fällt die momentan benötigte und zeitaufwendige Farbkalibrierung, die vor fast jedem Spiel vollzogen wird, um das System möglichst genau auf die Spielumgebung anzupassen, weg. Durch das dynamische Licht wird eine richtige händische Kalibrierung immer schwieriger, sodass diese in der Zukunft möglichst verhindert werden soll.

Als eine Alternative zu *Gaussian-Mixture-Model* und dem Ansatz der Nao Devils aus Dortmund mit gewichteten Histogrammen [Hofmann u. a., 2017], verfolgt unter anderem das UChile Robotics Team der Universidad de Chile den Ansatz der farblosen Bildverarbeitung in der SPL. In ihrem auf dem RoboCup 2018 in Montréal vorgestelltem Paper präsentierten sie jeweils einen Ball-, Roboter-, Feldfeature- und

Abbildung 8.1: Eine Spielszene als Graustufenbild, aufgenommen von der oberen Kamera.

Roboterorientierungserkenner. Alle Module arbeiten auf Graubildern wie Abbildung 8.1 und liefern dabei eine höhere Erkennungsrate, als vergleichbare Module von B-Human. Die vorgestellten Module basieren zum Teil auf äquivalente B-Human-Module. So verläuft die Erkennung von Feldlinien und deren Kreuzungen wie bei B-Human - nur dass die Vorverarbeitung auf farblosen Bildern stattfindet und mit Kontrasten statt Farbübergängen arbeitet. [Leiva u. a., 2018]

Die Resultate zeigen, dass es viele Möglichkeiten in der Zukunft gibt, die Roboter im Fußball auf dynamische Lichtverhältnisse eingehen zu lassen. Farbinformationen bieten jedoch auch weiterhin Vorteile, die durch eine farblose Bildverarbeitung verloren gehen könnten. Die Trikotfarben sind nur bedingt auf Graustufenbildern auseinanderzuhalten, sodass eine spätere Unterscheidung zwischen Mitspieler und Gegenspieler nicht immer möglich ist. Da der Roboterfußball sich jedoch immer näher dem menschlichen Fußball annähern wird, ist dies ein Problem, welches bedacht werden muss, um in naher Zukunft guten Fußball spielen zu können.

# Anhang

| Datei/ Ordner | Beschreibung |
|---|---|
| ReadMe.txt | Anleitung zur Verwendung aller implementierten Module. |
| Evaluation | Ordner mit allen evaluierten Bildern, den entsprechenden Logs, sowie die Excel-Tabelle, die die Fehlerpunkte den Scanlines und Bildern zuordnet. |
| B-Human | Stand der B-Human-Software, auf dem entwickelt und evaluiert wurde. |
| Bachelorarbeit.pdf | Digitale Version dieser Bachelorarbeit |

# Literaturverzeichnis


[B-Human 2018] B-Human: *About B-Human*. https://www.b-human.de/index.html, 2018. – zuletzt aufgerufen am 5. Januar 2019

[Bilmes 1998] Bilmes, Jeff A.: *A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models*. http://melodi.ee.washington.edu/people/bilmes/mypapers/em.pdf, 1998. – zuletzt abgerufen am 3. Januar 2019

[Hofmann u. a. 2017] Hofmann, Matthias ; Schwarz, Ingmar ; Urbann, Oliver ; Larisch, Aaron: *Nao Devils Dortmund - Team Report 2017*. https://github.com/NaoDevils/CodeRelease/blob/master/TeamReport2017.pdf, 2017. – zuletzt aufgerufen am 5. Januar 2019

[Leiva u. a. 2018] Leiva, Francisco ; Cruz, Nicolás ; Bugueño, Ignacio ; Solar, Javier R.: *Playing Soccer without Colors in the SPL: A Convolutional Neural Network Approach*. https://arxiv.org/pdf/1811.12493.pdf, 2018. – zuletzt aufgerufen am 5. Januar 2019

[Rabiner 1989] Rabiner, Lawrence R.: A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognitions. In: *Proceedings of the IEEE* Bd. 77 Ausg. 2, IEEE, 1989

[RoboCup Federation 2016] RoboCup Federation: *Objective*. https://www.robocup.org/objective, 2016. – zuletzt abgerufen am 5. Januar 2019

[RoboCup Technical Committee 2012] RoboCup Technical Committee: *RoboCup Standard Platform League (NAO) Rule Book*. https://spl.robocup.org/wp-content/uploads/downloads/Rules2012.pdf, 2012. – zuletzt abgerufen am 5. Januar 2019

[RoboCup Technical Committee 2016] RoboCup Technical Committee: *RoboCup Standard Platform League (NAO) Rule Book*. https://spl.robocup.org/wp-content/uploads/downloads/Rules2016.pdf, 2016. – zuletzt abgerufen am 5. Januar 2019

[RoboCup Technical Committee 2017] RoboCup Technical Committee: *RoboCup Standard Platform League (NAO) Rule Book*. https://spl.robocup.org/wp-content/uploads/downloads/Rules2017.pdf, 2017. – zuletzt abgerufen am 5. Januar 2019

[Röfer u. a. unv ] Röfer, Thomas ; Laue, Tim ; Baude, Andreas ; Felsch, Gerrit ; Hasselbring, Arne ; Poppinga, Bernd ; Thielke, Felix ; Urban, Timo:

B-Human Team Description for RoboCup 2018. In: *RoboCup 2018: Robot World Cup XXII Preproceedings*. Montreal, Canada : RoboCup Federation, unv. (noch unveröffentlicht)

[Röfer u. a. 2017] Röfer, Thomas ; Laue, Tim ; Bülter, Yannick ; Krause, Daniel ; Kuball, Jonas ; Mühlenbrock, Andre ; Poppinga, Bernd ; Prinzler, Markus ; Post, Lukas ; Roehrig, Enno ; Schröder, René ; Thielke, Felix: *B-Human Team Report and Code Release 2017*. Nur online verfügbar: http://www.b-human.de/downloads/publications/2017/coderelease2017.pdf, 2017. – zuletzt aufgerufen am 4. Januar 2019

[Russell u. Norvig 2012] Russell, Stuart ; Norvig, Peter: *Künstliche Intelligenz - ein moderner Ansatz*. 3. Auflage. München : Pearson Deutschland, 2012

[Schuller u. a. 2003] Schuller, Björn ; Rigoll, Gerhard ; Lang, Manfred: *Hidden Markov Model-Based Speech Emotion Recognition*. https://mediatum.ub.tum.de/doc/1138359/file.pdf, 2003. – zuletzt aufgerufen am 5. Januar 2019

[SoftBank Robotics Europe 2017] SoftBank Robotics Europe: *NAO Documentation*. http://doc.aldebaran.com/2-1/family/robots/index_robots.html#all-robots, 2017. – zuletzt abgerufen am 5. Januar 2019