Wettbewerbsfähigkeit autonom erlernter Verhaltensweisen im simulierten Roboterfußball: Optimierung eines Dribbelverhaltens zur Einsatzbereitschaft bei RoboCup-Turnieren

Tobias Raschke
1017209

Bachelorthesis
Zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

30. März 2017

Frankfurt Univeristy of Applied Sciences
Fachbereich 2 -Informatik und Ingenieurswissenschaften
Studiengang Informatik (B. Sc.)

Betreuer: Prof. Dr. Thomas Gabel
Korreferent: Prof. Dr. Christian Baun

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Hausarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Die Versicherung bezieht sich auch auf in der Arbeit gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

____

Datum, Unterschrift

Zusammenfassung

Die RoboCup-Mannschaft FRA-UNIted der Frankfurt University of Applied Sciences setzt sich als Ziel, verstärkendes Lernen in möglichst vielen Entscheidungsprozessen zu verwenden. Ziel dieser Bachelorarbeit ist es, ein Teilverhalten zu entwickeln, welches auf Basis des verstärkenden Lernens effizient dribbelt. In diesem Zusammenhang ist Dribbeln definiert, als Bewegung eines Spielers in eine Zielrichtung, während jener den Ball im eigenen Ballkontrollbereich hält. Effizient meint eine möglichst große Distanz innerhalb einer geringen Anzahl von Zeitschritten zurückzulegen. Um dieses Ziel zu erreichen, wird das Problem des Dribbelns als Markov-Entscheidungsprozess modelliert. Die Aktionswahl findet auf Basis des Werteiterationsverfahrens statt, wobei ein künstliches neuronales Netz die zu iterierenden Werte approximiert.

abstract

The goal of the RoboCup-Team FRA-UNIted of the Frankfurt University of Applied Sciences is to support their decision processes with Reinforcement Learning wherever possible. The goal of this bachelor thesis is to develop a sub-behavior, which shall use Reeinforcement Learning to dribble efficiently. In this context dribbling is defined as player movement towards a target direction while the ball is within kick range. It is considered to be efficient if it manages to move a large distance in a short time. To reach this goal the dribble problem is described as a Markov-Decision-Process. Actions are chosen based on value iteration, where an artificial neural network approximates the values to iterate.

Inhaltsverzeichnis

1 Einleitung **7**

2 Theoretische Grundlagen **8**
2.1 RoboCup: 2D-Fußball-Simulation . . . 8
2.2 FRA-UNIted . . . 10
2.3 Entscheidungsprozesse . . . 11
2.3.1 Zeitpunkte . . . 11
2.3.2 Zustände . . . 11
2.3.3 Aktionen . . . 11
2.3.4 Zustandsübergänge . . . 11
2.3.5 Belohnungen oder Kosten . . . 12
2.3.6 Strategien . . . 12
2.3.7 Markov-Eigenschaft . . . 12
2.3.8 Modell . . . 12
2.3.9 Wertefunktion . . . 13
2.3.10 Politikiteration . . . 13
2.3.11 Werteiteration . . . 14
2.4 Maschinelles Lernen . . . 15
2.5 Exploration und Exploitation . . . 15
2.6 Neuronale Netze . . . 15
2.6.1 Struktur . . . 16
2.6.2 Neuronen . . . 16
2.6.3 Lernen . . . 17

3 Implementierung **19**
3.1 Modellierung des Dribbelproblems als Markov-Entscheidungsprozess . . . 19
3.1.1 Zustände . . . 19
3.1.2 Aktionen . . . 20
3.1.3 Zustandsübergänge . . . 20
3.1.4 Kosten . . . 20
3.1.5 Diskontierungsfaktor . . . 21
3.2 Neuronales Netz . . . 21
3.3 Aufbau des Agenten . . . 22
3.3.1 NeuroDribble2017 . . . 22
3.3.2 CommandProvider . . . 23
3.3.3 SituationAssessment . . . 23
3.3.4 CommandAnalysis . . . 24

3.3.5 DribbleNetControl 25
3.4 Lernvorgang 26
3.4.1 Trainer 26
3.4.2 Steuerndes Verhalten 27
3.4.3 Wertefunktion 27

4 Lernszenarien **28**
4.1 Lernszenario 1: Vereinfachte Situation 28
4.2 Lernszenario 2: Turnierbedingungen 29
4.3 Lernszenario 3: Erweiterung der Aktionsmenge 29

5 Lernergebnisse **30**
5.1 Lernszenario 1: Vereinfachte Situation 31
5.1.1 Lernverhalten 31
5.1.2 Evaluation 32
5.2 Lernszenario 2: Turnierbedingungen 35
5.2.1 Lernverhalten 35
5.2.2 Evaluation 36
5.3 Lernszenario 3: Erweiterung der Aktionsmenge 38
5.3.1 Lernverhalten 38
5.3.2 Evaluation 39

6 Fazit **41**

7 Ausblick **42**

1 Einleitung

Die künstliche Intelligenz gewinnt immer mehr an Bedeutung. Industrie und Wirtschaft sparen durch die Verwendung von Experten Systemen Millionen [7]. Auch im privaten Gebrauch finden sich Systeme, die künstliche Intelligenz verwenden, wie Einparkhilfen. Es gibt fortschreitende Entwicklungen im Bereich selbst fahrender Autos. In einigen Anwendungsbereichen, wie der medizinischen Diagnose oder dem Beweisen mathematischer Theoreme, steht einer künstlichen Intelligenz ein vergleichsweise großes Zeitfenster zum Erreichen der Zielsetzung zur Verfügung. In anderen, wie dem Steuern eines Fahrzeugs in belebtem Verkehr, ist dieses Zeitfenster wesentlich kleiner. Entscheidungen müssen innerhalb von Sekundenbruchteilen getroffen werden. Dabei ist die Lösungsfindung bei realen Problemen nicht zwangsläufig trivial. Beim Fahren im Straßenverkehr gilt es die Aktionen anderer zu beachten.

Fußball erfreut sich weltweit großer Beliebtheit. Darüber hinaus bietet diese Sportart eine nicht-triviale Umgebung, denn es gibt viele Faktoren zu beachten und die Anzahl der unterschiedlichen Zustände und Aktionen ist unendlich, die zur Verfügung stehende Zeit ist beschränkt, denn Entscheidungen müssen innerhalb von Millisekunden gefällt werden. Außerdem ist die Koordination mit anderen Spielern von zentraler Bedeutung.

Seit 1997 veranstaltet die Robot World Cup Initiative (kurz: RoboCup) jährlich Wettkämpfe, in denen sowohl Agenten in simulierten Umgebungen, wie auch Roboter in der echten Welt in unterschiedlichen Disziplinen gegeneinander antreten. Der RoboCup hat es sich als Ziel gesetzt, Forscher zu animieren, Systeme zu entwickeln, die eben jene zeitkritischen und komplexen Probleme lösen können. Auch sollen junge Menschen für die Forschung begeistert werden. Aus den genannten Gründen ist der Fußball eine tragende Säule des RoboCups. Ziel des Wettbewerbs ist es, bis 2050 eine Roboter-Fußball-Mannschaft zu entwickeln, die den dann amtierenden Fußball Weltmeister schlagen kann. Um dieses Ziel zu erreichen existieren mehrere Ligen. Eine davon ist die 2D-Simulationsliga. Die Frankfurt University of Applied Sciences entsandte 2016 erstmals die eigene Mannschaft FRA-UNIted, um in der Simulationsliga anzutreten, wo sie den 11. Platz belegte. Ziel der Mannschaft ist es, möglichst viele Entscheidungen mithilfe des verstärkenden Lernens *(engl. Reinforcement Learning)* zu treffen [2]. Für das Dribbeln wird bisher ein nicht erlerntes Verhalten verwendet. In diesem Zusammenhang bedeutet Dribbeln, dass der Spieler möglichst schnell in eine Richtung läuft und dabei den Ball in seinem Ballkontrollbereich behält. Ziel dieser Arbeit ist es, ein erlerntes Verhalten zur Verfügung zu stellen, welches effizienter ist, als das bisher Verwendete. Dazu wird die in [8] vorgestellte Lösung aufgegriffen und erkannte Fehler behoben. Das hier vorgestellte Verhalten verwendet ein mehrschichtiges, vorwärts gerichtetes neuronales Netz, welches eine auf Basis des Werteiterationsverfahrens ermittelte Wertefunktion für alle Zustände approximieren soll.

2 Theoretische Grundlagen

Im Folgenden werden die Grundlagen, auf denen Handeln und Lernen des Agenten basieren, erläutert. Außerdem wird die Umgebung, in der er agiert, vorgestellt.

2.1 RoboCup: 2D-Fußball-Simulation

Das in dieser Arbeit vorgestellte Dribbelverhalten ist zur Benutzung durch einen Agenten der 2D-Simulationsliga bestimmt, daher wird an dieser Stelle die Simulationsumgebung beschrieben.

Es treten zwei Mannschaften mit je zwölf Agenten gegeneinander an. Pro Mannschaft interagieren dabei je 11 Agenten mit der Umgebung, wobei je einer als Torwart besonderen Regeln untersteht. Der zwölfte Agent ist der Trainer *(engl. Coach)*. Die Agenten verbinden sich mit einem Server, der die Umgebung stellt. Diese ist ein simuliertes Fußballfeld, das 105 Längeneinheiten lang und 68 Längeneinheiten breit ist. Ein Spiel ist in zwei Halbzeiten unterteilt, von denen jede 3000 Zeitschritte lang ist. Standardmäßig ist ein Zeitschritt 100 Millisekunden lang. Der Server wertet die Befehle, die von den Spielern gesendet werden, aus und berechnet den Folgezustand. Anschließend übermittelt er diesen an die Spieler. Dabei erhält jeder Spieler nur Informationen, die er auch wahrnehmen kann. Diese sind Rufe in seinem Hörradius, visuelle Informationen aus seinem Sichtkegel, sowie Informationen aus seinem unmittelbaren Umfeld, die er erfühlen kann. Der Informationsgehalt, wie auch die Präzision, variieren je nach Sensor und Distanz. Beispielsweise kann bei einem weit entfernten Spieler nicht erkannt werden, welche Rückennummer er trägt. Darüber hinaus belegt der Server jede Information mit einem Rauschen. Auch jede Aktion, die ein Spieler durchführt wird leicht verrauscht. Auf diese Art werden Umwelteinflüsse und Sensorungenauigkeiten simuliert.

Die Spieler haben Werte, die ihre körperlichen Attribute widerspiegeln. Ein Spieler hat einen gewissen Aktionsradius, der seine Beinlänge darstellen soll. Auch hat jeder Spieler zwei Ausdauer Reserven. Der Spieler verbraucht die erste Reserve, um sich zu bewegen. Wenn diese Ausdauer Reserve einen gewissen Wert unterschreitet, kann sich der Spieler nur noch mit verringerter Geschwindigkeit beschleunigen. Die zweite Ausdauer Reserve repräsentiert seine absolute Ausdauer. Es erfolgt eine Aufladung der ersten Reserve aus der zweiten. Dabei kann nur ein begrenzter Wert pro Zeitschritt transferiert werden.

Der Server erzeugt bei Spielbeginn eine Auswahl von heterogenen Spielertypen mit unterschiedlichen Attributen. Die Trainer wählen dann Typen für ihre Spieler aus. Dabei gibt es einen besonderen Typen, der nicht neu generiert wird. Dieser Standard-Spieler steht immer zur Verfügung und dient in dieser Arbeit auch als Ausgangspunkt.

Alle Spieler unterstehen Regeln, die den Regeln des echten Fußballs entsprechen. So kann ein Spieler eine Karte bekommen, wenn er einen Gegenspieler foult. Darüber hinaus

gibt es besondere Regeln, die Beispielsweise verhindern, dass eine Gruppe von Spielern den Ball mit ihren Körpern vor den Gegenspielern abschirmen.

Das Spiel befindet sich immer in einem Modus, der die aktuelle Situation beschreibt. Es gibt unter anderem das normale Spiel, den Zeitraum des Anstoßes, 4 Eck-Situationen (eine für jede Ecke) und 2 Freistoß-Situationen (eine für jede Mannschaft).

Für den Trainer stehen 2 verschiedene Modi zur Verfügung. Als Online-Trainer kann er Nachrichten beliebiger Länge an die Spieler schicken, wobei diese Nachrichten mit einer Verzögerung von 50 Zeitschritten ankommen. Des weiteren kann er eine begrenzte Anzahl von Einwechselungen anfordern, die dann bei der nächsten Spielunterbrechung durchgeführt werden. Als Offline-Trainer kann er darüber hinaus Spieler und Ball beliebig platzieren und besondere Anweisungen geben, wie die Ausdauer Reserven eines Spielers komplett aufzuladen. Auch andere Schiedsrichter spezifischen Aufgaben, wie das setzen des Spielmodus übernimmt der Offline-Trainer.

Der Online-Trainer kommt unter Turnierbedingungen zum Einsatz, während der Offline-Trainer entwickelt wurde, um beim maschinellen Lernen zu assistieren.

Ein Spieler hat eine Vielzahl von Möglichkeiten mit der Umgebung zu interagieren. Er kann eine kurze Nachricht beliebigen Inhalts an seine Umgebung senden, die Blickrichtung sowie Breite des Sichtfeldes ändern und eine Aktion durchführen. Diese Aktionen sind unter anderem:

Dash Die Beschleunigung in eine von acht Richtungen. Die Effektivität dieser Aktion ist am höchsten, wenn der Spieler sie aus eigener Sicht nach vorne durchführt.

Turn Das Drehen des Körpers in eine beliebige Richtung. Je schneller sich der Spieler bewegt, desto weniger weit kann er sich in einem Zeitschritt drehen.

Kick Das Treten des Balles in eine beliebige Richtung. Die maximale Stärke des Tritts ist abhängig von der Position des Balles im Ballkontrollbereich des Spielers.

Tackle Das Durchführen einer Grätsche. Dadurch kann Kraft auf den Ball ausgeübt werden, wenn er nicht im Ballkontrollbereich liegt. Diese Aktion ist mit einer Erfolgschance verbunden, deren Wert mit steigender Distanz zum Ball abnimmt. Die Grätsche kann als Foul durchgeführt werden, wodurch sich die Erfolgschancen erhöhen, allerdings ahndet der Schiedsrichter die Aktion mit einer gewissen Wahrscheinlichkeit als Foul, wenn ein Gegenspieler im Ballbesitz ist. Nach einer Grätsche ist der Spieler für zehn Zeitschritte bewegungsunfähig.

Catch Das Fangen des Balles. Diese Aktion ist dem Torwart vorbehalten und gestattet es ihm den Ball sofort unter Kontrolle zu bringen. Dadurch ändert sich der Spielmodus auf Abschlag. Diese Aktion kann nur durchgeführt werden, wenn der sich der Torwart im Strafraum aufhält.

2.2 FRA-UNIted

Bei den Brainstormers handelt es sich um eine 1998 gegründete Mannschaft der 2D-Fußball-Simulationsliga. 2010 findet die Arbeit an der Mannschaft ein Ende und wird seit 2014 an der Frankfurt University of Applied Sciences unter dem Namen FRA-UNited fortgeführt [2]. Ziel der Mannschaft ist es möglichst viele Entscheidung unter Verwendung des verstärkenden Lernens zu treffen, sofern dies sinnvoll ist.

Die Module der FRA-UNIted Agenten sind in C++ verfasst. Da in der 2D-Simulations-Liga eine Zeitbegrenzung pro Zeitschritt vorliegt, handelt es sich bei den Agenten, die in dieser Liga zum Einsatz kommen, um Echtzeitsysteme. Die Sprache C++ eignet sich für den Bau von Echtzeitsystemen, da Programmierer die Kontrolle über eine Reihe von Operationen, wie dem Zuweisen und Freigeben von Speicher und der Konstruktion und Destruktion von Objekten, innehaben. Im Gegensatz dazu stehen Java und C#, die in ihrer nativen Form keine Kontrolle über die Destruktion bieten, da der dafür verantwortliche Garbage Collector eigenständig operiert. Selbst bei gezielten Anweisungen an diesen kann es zu unvorhergesehenem Verhalten kommen, welches unter Umständen unerwartet viel Zeit einnehmen kann.

Die im Kontext dieser Arbeit relevanten Eigenheiten der FRA-UNIted Mannschaft (im Folgenden auch FRA-UNIted Framework genannt), sind die Darstellung des Weltmodelles mit Markov-Eigenschaft und die Unterteilung des Entscheidungsmoduls in Untermodule.

Jedes Modul, welches eine Entscheidung treffen kann, erbt von der *BaseBehavior* Klasse. Die wichtigste zu überschreibende Methode ist get_cmd, die eine Referenz auf ein *Cmd* Objekt als Argument erwartet und einen Boolean zurückgibt. Alle aufgerufenen Verhalten innerhalb des Frameworks haben über get_cmd() die Gelegenheit einen Befehl zu setzen. Das als Argument übergebene *Cmd* Objekt dient dem Transport des gesetzten Befehls. Das zurückgegebene Boolean beschreibt, ob das aufgerufene Verhalten das *Cmd* Objekt manipuliert hat.

Das *Cmd* Objekt besteht aus mehreren Teilen, die unterschiedliche zu setzende Befehle beschreibt, denn es können pro Zeitschritt mehrere unterschiedliche Befehle abgegeben werden. So kann der Agent gleichzeitig den Ball treten und seinen Kopf drehen. In dieser Arbeit ist Cmd_Main von Interesse, das für den Transport der in 2.1 beschriebenen Aktionen verantwortlich ist.

Jedes Verhalten kann eine beliebige Anzahl anderer Verhalten nutzen. Dies ist von Bedeutung, da ein Verhalten nicht nur eine Fähighkeit, wie das Dribbeln, sondern auch eine Taktik oder Strategie abbilden kann. Ein strategisches Verhalten kann aus einer Reihe von taktischen Verhalten das für das aktuelle Spielgeschehen am besten Geeignete wählen. Das taktische Verhalten wiederum wählt für den aktuellen Zustand eine geeignete Fähigkeit, wie den Torschuss, aus. Dieses für das Steuern einer Fähigkeit verantwortliche Verhalten entscheidet sich dann für eine konkrete Aktion. So führt der Aufruf eines Verhaltens, das strategische Entscheidungen trifft, zu einem konkreten Befehl.

2.3 Entscheidungsprozesse

Damit ein Agent Aktionen in seiner Umwelt wählen kann, benötigt er einige Elemente. Dazu gehören eine Beschreibung der Umgebung und die Menge aller zur Verfügung stehenden Aktionen [9]. Diese Elemente sollen im Folgenden beschrieben werden.

2.3.1 Zeitpunkte

Um einen Entscheidungsprozess beschreiben zu können besteht die Notwendigkeit, diskrete Zeitpunkte darzustellen[9]. Es sei T die Menge aller Zeitpunkte. Im Folgenden ist n die Anzahl aller Elemente in T. $t_n \in T$ ist der terminale Zustand, welcher den vorgegebenen Abbruchzeitpunkt beschreibt. Dieser ist beispielsweise erreicht, wenn ein Agent seine Zielstellung erfüllt, muss aber nicht zwangläufig existieren. Bei der Problemstellung des Dribbelns existiert kein terminaler Zustand, da es sich um einen unendlich anhaltenden Vorgang handelt. In einem solchen Fall spricht man von einem unendlichen Horizont.

2.3.2 Zustände

Damit ein Agent in der Lage ist mit seiner Umwelt zu interagieren, muss diese in maschinenverständlicher Form beschrieben sein [9]. Die Beschreibung der Umgebung zu einem beliebigen Zeitpunkt $t \in T$ heißt Zustand $s_t \in S$, wobei S die Menge aller Zustände und T die Menge aller Zeitpunkte ist, die angenommen werden können. In der 2D-Simulationsliga ist die Menge der möglichen Zustände S unendlich und kontinuierlich, die Menge der Zeitpunkte T hingegen endlich und diskret. Idealerweise bildet ein Zustand wenigstens alle für die Entscheidungsfindung relevanten Informationen ab. Jede einzelne Information, die für die Beschreibung eines Zustandes hinzugezogen wird, ist als eigene Dimension beschrieben. Wenn also ein Zustand $s_t \in S$ mit zwei Informationen beschrieben wird, heißt der Zustandsraum S zweidimensional.

2.3.3 Aktionen

Ein Agent manipuliert seine Umgebung zum Zeitpunkt $t \in T$, indem er eine Aktion $a_t \in A(s_t)$ ausführt. Die Menge der zur Verfügung stehenden Aktionen $A(s_t)$ ist abhängig von dem Zustand s_t, in dem sich der Agent zum Zeitpunkt t befindet. Beispielsweise ist die Aktion Catch nicht Teil der Aktionsmenge, wenn sich der Agent im Zustand s_t nicht im Strafraum aufhält. Diese Aktion steht außerdem nicht zur Verfügung, wenn der Agent kein Torwart ist.

2.3.4 Zustandsübergänge

Wenn ein Agent im Zustand $s_t \in S$ eine Aktion $a_t \in A(s_t)$ ausführt, folgt daraus ein neuer, sogenannter Folgezustand $s_{t+1} \in S$ oder $s' \in S$. Um diesen zu ermitteln, wird die Zustandsübergangsfunktion $f(s_t, a_t) = s'$ verwendet. Sollte die Umgebung nicht deterministisch sein, wird stattdessen die bedingte Wahrscheinlichkeitsverteilung $P(s' = j | s_t = i, a_t = a) p_{ij}(a)$ verwendet.

2.3.5 Belohnungen oder Kosten

Beim verstärkten Lernen spielen Belohnungen beziehungsweise Kosten eine fundamentale Rolle. Dabei können Kosten als negative Belohnungen und Belohnungen als negative Kosten aufgefasst werden. Im Folgenden ist von Kosten und Belohnungen als negativen Kosten auszugehen.

Die Kostenfunktion $c(s_t), s_t \in S$ gibt dabei den negativen Nutzen eines Zustandes zur Erfüllung eines festgelegten Zieles als reale Zahl $\mathbb{R}$ zurück. Alternativ kann die Kostenfunktion ein Zustands-Aktions-Paar bewerten: $c(s_t, a_t), s_t \in S, a_t \in A(s_t)$.

2.3.6 Strategien

Bei einer Strategie $\hat{\pi}$ handelt es sich um eine Menge von n Auswahlfunktionen $\pi_t(s_t) = a_t$, wobei n gleich der Anzahl aller Zeitschritte $t \in T$ ist. Eine Auswahlfunktion, auch Politik genannt, wählt bei einem Zustand $s_t \in S$ eine Aktion $a_t \in A(s_t)$ aus. Eine Strategie heißt stationär, wenn nur eine Politik existiert, die für alle Zeitpunkte t eine Aktion wählt. In diesem Fall gilt $\hat{\pi} = \pi$.

2.3.7 Markov-Eigenschaft

Die Markov-Eigenschaft verlangt, dass die Folgezustände eines Entscheidungsprozesses nicht abhängig von vorherigen Zeitpunkten sind [9]. Als Beispiel diene der Zustand des Balles in der 2D-Simulationsliga. Der Zustand $s_t \in S$ des Balles ist beschrieben durch seine Absolute Position auf dem Spielfeld, sowie seiner Beschleunigung. Beides wird als zweidimensionaler Vektor der Form $(\mathbf{x}, \mathbf{y})$ dargestellt. Die zukünftigen Zustände sind nur vom aktuellen Zustand s_t, sowie dem Rauschen des Server bestimmt. Darüber hinaus beeinflussen etwaige Aktionen von Agenten die Folgezustände des Balles. Zustände und Aktionen, die in der Vergangenheit liegen, werden keinen weiteren Einfluss auf zukünftige Zustände nehmen.

Ein Entscheidungsprozess, bei dem alle Elemente die Markov-Eigenschaft erfüllen heißt Markov-Entscheidungsprozess *(engl. Markov Decision Process, kurz MDP)*. Dieser ist durch das 6-Tupel (T, S, A, f, c, γ) dargestellt, wobei es sich bei γ um den sogenannten Diskontierungsfaktor handelt.

2.3.8 Modell

Das Modell eines Agenten gestattet es ihm den Folgezustand $s' \in S$ zu bestimmen, wenn er sich in einem Zustand $s_t \in S$ befindet und die Aktion $a_t \in A(s_t)$ durchführt. Anhand dieser Vorhersage lässt sich auch das Ergebnis der entsprechenden Kostenfunktion berechnen. Das FRA-UNIted Framework besitzt ein Modell, welches in der Lage ist, die für das Dribbeln benötigten Vorhersagen zu treffen. Im Zusammenhang der Entscheidungsprozesse dient das Modell der agenteninternen Berechnung der Übergangsfunktion f. Auch der aktuelle Zustand des Agenten kann zum Teil aus Berechnungen des Modells stammen. Bei einer nicht vollständig beobachtbaren Umgebung, wie dem Roboterfußball, zieht der Agent anhand des Modells logische Schlüsse über vorher beobachtete Teile der

Umgebung. Selbst wenn der Agent den Ball für wenige Zeitschritte nicht beobachtet, kann er am Modell berechnen, wo der Ball sich aufgrund seiner vorherigen Geschwindigkeit nun Aufhalten müsste, unter der Annahme, dass kein anderer Agent den Ball beeinflusst hat. Somit kann die unbeobachtete Position des Balls teil des Zustandes sein.

2.3.9 Wertefunktion

Damit einer Politik π eine Grundlage für die Auswahl einer „guten“ Aktion zur Verfügung steht, benötigt sie eine Möglichkeit, die Güte dieser Aktion zu bewerten. Dabei entspricht die Güte dem Wert eines Zustandes, der als die Summe der Kosten des Zustandes, sowie aller folgenden Kosten, definiert ist. Die Wertefunktion V einer Politik π für den Zustand $s_t \in S$ ist dabei wie folgt definiert:

$$V^\pi(s_t) = \sum_{i=0}^{n-t} c(s_{t+i}, \pi(s_{t+i})), s_{t+1} = f(s_t, a_t) \tag{2.1}$$

Folgerichtig gilt eine Politik als Optimal π^*, wenn die Wertefunktion minimale Werte zurückliefert. Die Wertefunktion lässt sich allerdings nur dann berechnen, wenn ein terminaler Zustand $t_n \in T$ existiert. Andernfalls konvergiert die oben beschriebene Formel gegen unendlich. Der in 2.3.7 erwähnte Diskontierungsfaktor $0 < \gamma < 1$ findet hier Anwendung. Dieser wird mit den zu erwartenden Kosten multipliziert, um deren Einfluss zu verringern, bis das Produkt gegen 0 konvergiert. Daraus ergibt sich bei unendlichem Horizont die Formel:

$$V^\pi(s_t) = \sum_{i=0}^{n-t} \gamma^i * c(s_{t+i}, \pi(s_{t+i})), s_{t+1} = f(s_t, a_t) \tag{2.2}$$

Der Diskontierungsfaktor γ entscheidet demnach darüber, wie stark zukünftige Zustände den Wert des betrachteten Zustandes beeinflussen. Ein niedriger Wert führt zu einer stärkeren Gewichtung unmittelbarer Kosten, wohingegen ein großer Wert zu einer starken Gewichtung zukünftiger Kosten führt.

2.3.10 Politikiteration

Mithilfe der Politikiteration lässt sich eine optimale Politik ermitteln [3]. Eine zufällige Politik π dient dabei als Ausgangspunkt. Es wird die Wertefunktion für einen Ausgangszustand $s \in S$ gefunden:

$$V^\pi(s) = c(s, \pi(s)) + \gamma * V^\pi(s') \tag{2.3}$$

Sowie diese Wertefunktion bekannt ist, lässt sich überprüfen, ob eine Änderung der Politik eine Verbesserung bewirkt, indem in diesem Startzustand eine andere Aktion gewählt wird. Wenn eine Änderung eine Verbesserung bewirkt, findet die neue Politik Anwendung und der Vorgang wird wiederholt. Falls keine Verbesserung gefunden werden kann, ist die aktuelle Politik die optimale Politik π^*.

Der folgende Pseudocode soll dieses Vorgehen darstellen:

Output: π^*
Erzeuge zufällige Politik π'
while $\pi' \neq \pi$ **do**
  $\pi \leftarrow \pi'$
  Bestimme $V^{\pi}(s) = c(s, \pi(s)) + \gamma * V^{\pi}(s')$
  Verbessere die Politik: $\pi'(s) \leftarrow \max_a c(s,a) + \gamma * V^{\pi}(s')$
end

Die Politikiteration setzt eine Überprüfung aller Zustände und Aktionen voraus. Im Roboterfußball lässt sich streng genommen keine Überprüfung aller Aktionen vornehmen, da dies unendlich viele sind. Selbst wenn die Aktionsmenge begrenzt wäre, ließe sich die Politikiteration nicht durchführen, da unendlich viele Zustände existieren.

2.3.11 Werteiteration

Mithilfe der Wertiteration wird eine minimale und damit optimale Wertefunktion ermittelt [9]. Dazu wird das Bellmansche-Optimalitätsprinzip angewandt. Es sei V^* eine optimale Wertefunktion, dann gilt:

$$V^*(s_t) = \min_{a_t \in A(s_t)} (c(s_t, a_t) + \gamma * V^*(f(s_t, a_t)), s_{t+1} = f(s_t, a_t), a_t \in A(s_t)) \tag{2.4}$$

Es sei $i \in S$ und j ein beliebiger Folgezustand von i. Innerhalb eines nicht deterministischen MDPs gilt hingegen:

$$V^*(i) = \min_{a_t \in A(i)} (c(i, a_t) + \gamma * \sum_j p_{ij}(a_t) * V^*(j, a_t)) \tag{2.5}$$

Der Werteiteration-Algorithmus bestimmt eine beliebige Wertefunktion V_0 und führt folgende Zuweisung aus:

$$V_k(s_t) = min_{a_t \in A(s_t)}(c(s_t, a_t) + \gamma * V_{k-1}(f(s_t, a_t)), k \in \mathbb{N} \tag{2.6}$$

Der Algorithmus hat die optimale Wertefunktion ermittelt, wenn für alle Zustände $s \in S$ gilt: $V_k(s) = V_{k-1}(s)$. Diese Bedingung kann bei einer unendlichen Zustandsmenge S allerdings niemals erreicht werden. Ein ähnliches Problem findet sich auch bei der Politikiteration wieder, allerdings kann die Werteiteration über eine Teilmenge aller Zustände approximiert werden.

2.4 Maschinelles Lernen

Das maschinelle Lernen wird auf eine von drei Arten durchgeführt: Das überwachte, das unüberwachte, sowie das verstärkende Lernen. Beim überwachten Lernen stehen bekannte Lösungen zur Verfügung und ein Agent entwickelt anhand derer eine Strategie. Es steht ein Experte als eine Art Mentor oder Lehrer zur Verfügung, der die zu erlernende Lösung vorgibt und den Lernvorgang steuert und gegebenenfalls korrigiert. Beim unüberwachten Lernen existiert eine Datenmenge, die es gilt zu Kategorisieren. Im Gegensatz zum ersten Vorgehen steht hier kein Lehrer zur Verfügung. Beim verstärkenden Lernen steht ähnlich wie beim unüberwachten Lernen kein Lehrer zur Verfügung. Allerdings steht auch keine Datenmenge zur Verfügung. Der Gedanke hier ist, dass der Agent Aktionen durchführt und daraufhin eine Strafe erfährt, die er zu minimieren versucht. Mit dieser Methode können Roboter das Laufen erlernen. Je weiter sie laufen, ohne zu fallen, desto niedriger fallen ihre Kosten aus. Der Agent hat dabei kein Vorwissen über eine mögliche Strategie und führt Aktionen in seiner Umgebung durch. Dadurch sammelt er Zustände und Folgezustände, die sich aus Aktionen ergeben, an und benutzt diese, um sein Vorgehen zu optimieren.

2.5 Exploration und Exploitation

Da beim verstärkenden Lernen Zustände besucht werden müssen, um deren Kosten zu erfahren, stellt sich die Frage, wie sich bei einer großen Zustandsmenge der Lernerfolg maximieren lässt. Um die Gesamtkosten zu minimieren, müssen zum einen Kostengünstige Zustände besucht werden, zum anderen müssen neue Zustände getestet werden. Wenn ein Agent bekannte Zustands-Aktions-Paare weiterhin nutzt, nennt man dies Exploitation oder Ausbeutung. Das besuchen neuer Zustands-Aktions-Paare wird Exploration genannt. Es gibt eine Reihe von Algorithmen, die einen Ausgleich zwischen Exploration und Exploitation suchen, allerdings ist die simpelste auch die am häufigsten genutzte [7]: Mit einer beliebig gewählten konstanten Wahrscheinlichkeit wird eine zufällige Aktion ausgeführt.

2.6 Neuronale Netze

In [4] stellen McCulloch und Pitts ein von realen neuronalen Netzen inspiriertes künstliches neuronales Netz *(engl. artificial Neural Network)* vor. Dieses basiert auf Annahmen, die zu einem vereinfachten neuronalen Netz führen. Einige dieser Annahmen sind:

- Die Aktivierung eines Neuronen ist ein „alles oder nichts“-Prozess.
- Eine gewisse Anzahl von Synapsen müssen angeregt werden, um ein Neuron anzuregen. Diese Anzahl ist unabhängig von vorhergehenden Aktivierungen und von der Position am Neuron.
- Die Struktur eines Netzes ändert sich nicht mit der Zeit.

Neuronen, auch Knoten oder Einheiten *(engl. Units)* genannt, entscheiden anhand eingehender Werte, ob sie aktiviert werden. Damit ein Neuron ermitteln kann, ob es aktiviert ist, oder nicht, gibt es heute eine Anzahl an Funktionen. Auf eine Auswahl davon wird in 2.6.2 eingegangen. Ein Neuron gibt bei Aktivierung Signale über Synapsen an andere Neuronen weiter, die diese Werte für ihren eigenen Aktivierungsprozess verwenden. Diese Synapsen geben die Signale in von der Synapse abhängiger Intensität weiter. In künstlichen neuronalen Netzen sind Synapsen als gerichtete und gewichtete Kanten realisiert.

2.6.1 Struktur

Neuronen liegen innerhalb eines künstlichen neuronalen Netzes *(kurz kNN)* in mehreren sogenannten Schichten. Dabei gibt es immer eine Eingabe- und eine Ausgabeschicht. Die Struktur eines kNN lässt sich also mit den Folgenden Komponenten beschreiben:

- Die Anzahl der Schichten.
- Die Anzahl der Neuronen pro Schicht.
- Die Synapsen mit Start und Ziel Neuron.
- Die Gewichte jeder einzelnen Synapse.

Die Struktur eines kNN unterliegt dabei einigen wenigen Einschränkungen. Es gibt immer genau eine Eingabe- sowie eine Ausgabeschicht. Zwischen diesen Schichten darf eine beliebige Anzahl von versteckten Schichten liegen. Ein kNN mit wenigstens einer versteckten Schicht wird auch mehrschichtiges Netz genannt.

Das bisher beschriebene kNN wird rekurrent genannt, da Verbindungen in die eigene Schicht des Neurons, sowie in beliebige Richtungen gestattet sind. Wenn ein Neuron eine Verbindung mit sich selbst hat, wird dies zyklisch genannt.

Ein Netz, in dem Neuronen nur Verbindungen in eine Richtung haben, wird vorwärts gerichtet *(engl. Feed-Forward)* genannt. In diesem Fall sind Verbindungen in die eigene Schicht, und damit auch zyklische Verbindungen, untersagt.

Ein mehrschichtiges kNN hat die Fähigkeit mithilfe der Neuronen, die in der versteckten Schicht liegen, eine interne Repräsentation aufzubauen [6], weswegen sowohl die Anzahl der versteckten Schichten wie auch die Anzahl der Neuronen in diesen Einfluss auf die Effektivität des Netzes hat.

2.6.2 Neuronen

Ein Neuron gibt, sofern eine Ausgabe getätigt wird, wenigstens ein ausgehendes Signal ab. Es kann eine beliebige Menge eingehender Signale empfangen. Bei einem Neuron ohne Eingabe spricht man auch von einem *Bias*. Dieser wird so behandelt, als ob er einen Netto-Eingang von 1 hätte, unterliegt aber sonst den selben Regeln, wie jedes Neuron. Es wird zwischen Eingabe und Netto-Eingabe *(engl. Netinput)* unterschieden. Es sei a_j das Signal der Quelle j. Es sei i das Neuron i. Es sei w_{ij} das Gewicht der Eingabe j für das Neuron i, wobei im Regelfall für Neuronen der Eingabeschicht gilt: $w_{ij} = 1$. Es sei

die Eingabe des Neurons i aus der Quelle j $in_{ij} = a_j * w_{ij}$ Dann ist die Netto-Eingabe des Neurons i $netin_i = \sum_j in_{ij}$, also $netin_i = \sum_j a_j * w_{ij}$. Oder um es in Worte zu fassen: Die Netto-Eingabe eines Neurons ist die Summe der gewichteten Eingaben.

Im Gegensatz zu dem in Kapitel 2.6.1 vorgestellten kNN, ist das ausgehende Signal moderner kNNs nicht zwangsläufig binärer Natur. Normalerweise geht man von einer Gleitkommazahl aus und binäre Ausgaben werden als 0 (oder -1) und 1 dargestellt. Um zu ermitteln welche Ausgabe getätigt werden soll, gibt es eine Reihe unterschiedlicher Aktivierungsfunktionen, von denen die für diese Arbeit relevanten vorgestellt werden sollen.

Lineare Aktivierungsfunktion

Die lineare Aktivierungsfunktion ist die, mit den wenigsten Einschränkungen hinsichtlich möglicher Ausgaben. Die Ausgabe des Neurons i out_i verhält sich hier linear zur Netto-Eingabe. Durch das Setzen eines Schwellwertes kann dieser Zusammenhang verschoben werden.

Sigmoide Aktivierungsfunktion

Die Sigmoide Aktivierungsfunktion beschränkt den Ausgabewert auf 0 (oder -1) bis 1. Diese Funktion passt die Ausgabe je nach Wert unterschiedlich stark an. Werte, deren Betrag groß ist, werden stark in Richtung des entsprechenden möglichen Ausgabewertes angepasst. Negative Werte dabei in Richtung des kleinsten möglichen Ausgabewertes und positive in Richtung des größten möglichen Ausgabewertes. Bei moderaten Werten erfolgt eine geringe bis keine Anpassung. Für Sigmoide Aktivierungsfunktionen werden für gewöhnlich logistische oder Tangens-Hyberbolicus-Funktionen verwendet.

2.6.3 Lernen

Die Struktur eines Netzes bleibt nach dem Aufbau zum Großteil unverändert. Einzig die Gewichte werden noch angepasst. Durch eine Reihe von Anpassungen passt die Ausgabe des Netzes sich dem Zielwert iterativ an. Eine verbreitete Möglichkeit zur Anpassung ist der Backpropagation-Algorithmus [6, 5, 1]. Dieser ist in drei Schritte unterteilt:

- Für einen Eingabevektor wird ein entsprechender Ausgabevektor berechnet.
- Für jeden Wert im Ausgabevektor wird die Differenz zwischen diesem und dem erwarteten Wert berechnet und durch das entsprechende Ausgabeneuron zurückpropagiert.
- Anhand des Fehlersignals berechnet jedes Gewicht seinen Fehler und passt sich unter Verwendung des Gradientenabstiegsverfahrens an, um diesen Fehler zu minimieren.

Der in dieser Arbeit verwendete Resilient Propagation-Algorithmus stellt eine Modifikation dieses Algorithmus dar und soll im Folgenden erkärt werden.

Resilient Propagation

Wie auch bei dem unmodifizierten Backpropagation-Algorithmus wird bei dem Resilient Propagation-Algorithmus *(kurz Rprop)* die Differenz zwischen dem Zielwert und der tatsächlichen Ausgabe durch das Netz zurückpropagiert. Allerdings ist die Anzahl der Faktoren, die über die Größe der Gewichtsaktualisierung *(kurz Δ_{ij})* entscheiden, bei dem Rprop-Algorithmus geringer und leichter einstellbar [5]. Der initiale sowie maximale Wert von Δ_{ij} treten als einzige einstellbare Variablen auf, während der Backpropagation-Algorithmus eine einstellbare Lernrate anbietet. Die Wahl der Lernrate selbst ist dabei von enormer Wichtigkeit, da sie darüber entscheidet wie effektiv gelernt werden kann und ob überhaupt gelernt werden kann. Zu große oder zu kleine Werte führen zu langsamen Lernerfolgen. Braun und Riedmiller konnten zeigen, dass die Einstellung der Variablen bei Rprop leichter ist, weil eine große Menge von Einstellungen zu sehr ähnlichen Ergebnissen führt.

Die Berechnung des Einflusses eines Gewichts anhand einer Fehlerfunktion ist für beide Algorithmen nötig. Bei beiden ist Δ_{ij} von der Ableitung dieser Einflussfunktion abhängig. Bei dem Backpropagation-Algorithmus wird jedoch die Lernrate mit dieser Ableitung verrechnet, während der Rprop-Algorithmus Δ_{ij} sowie dessen Anwendung anhand des Vorzeichens dieser Ableitung berechnet. Wenn sich das Vorzeichen bei der letzten Anpassung nicht verändert hat, wird Δ_{ij} um den Faktor $1,2$ erhöht. Falls das Vorzeichen sich geändert hat, wird Δ_{ij} um den Faktor $0,5$ verkleinert. Falls die Ableitung den Wert 0 hat, wird Δ_{ij} nicht angepasst.

Die Ermittlung der Fehler der Gewichte findet unter Verwendung mehrerer Trainingsexemplare statt. Die Anzahl dieser ist frei wählbar. Wenn alle Differenzen zwischen den Exemplaren und den entsprechenden erwarteten Werten zurückpropagiert wurden, findet die Gewichtsanpassung statt. Dies nennt man eine Epoche. Für den Rprop-Algorithmus werden mehrere dieser Epochen durchgeführt.

Nach der Anpassung von Δ_{ij} erfolgt die Anpassung des Gewichtes w_{ij}. Sollte das Vorzeichen positiv sein, wird Δ_{ij} von w_{ij} abgezogen. Ist das Vorzeichen negativ wird Δ_{ij} auf w_{ij} addiert. Es existiert eine Ausnahme dieser Regel. Sollte die letzte Anpassung zu einer Änderung des Vorzeichens geführt haben, macht der Rprop-Algorithmus diese ungeschehen. Außerdem setzt er den Wert der Ableitung auf 0.

3 Implementierung

Im Folgenden werden grundlegende Entscheidungen des Aufbaus des Verhaltens erläutert.

3.1 Modellierung des Dribbelproblems als Markov-Entscheidungsprozess

Wie in 2.3.7 bereits erläutert, bildet der Markov-Entscheidungsprozess die theoretische Grundlage für das Lernen des hier beschriebenen Verhaltens. Daher soll die Umsetzung dessen im Folgenden vermittelt werden.

3.1.1 Zustände

Das FRA-UNIted Framework stellt die gesamte bekannte Umgebung mithilfe von absoluten Vektoren und Winkeln dar. Dabei beginnen alle Positionsvektoren im Mittelpunkt des Spielfeldes. Die absolute Position auf dem Spielfeld ist für das Lösen des Dribbelproblems allerdings eher hinderlich, da Zustände, die für die Entscheidungswahl identisch sein sollten, es so nicht sind. Wichtig für eine Entscheidung sind die relativen Position und Geschwindigkeiten. Beim Dribbeln ist es egal, ob der Spieler zusammen mit dem Ball auf der linken oder rechten Flanke ist, oder ob er gerade den Mittelkreis passiert. Diese Informationen sind auf taktisch-strategischer Ebene relevant. Das hier entwickelte Dribbelverhalten soll allerdings nur die Fähigkeit in eine Richtung zu dribbeln zur Verfügung stellen. Die Beschreibung eines Zustandes beinhaltet daher diese Informationen:

- Die Geschwindigkeit des Spielers.
- Die Geschwindigkeit des Balls.
- Die Position des Balls relativ zum Spieler.
- Die Richtung des Zielwinkels.

Zweidimensionale Vektoren beschreiben alle Informationen. Es erfolgt eine Drehung entgegen des absoluten Winkels des Spielers, damit nicht nur Positionen sondern auch Richtungen relativ abgebildet sind. Die Richtung des Zielwinkels findet in der ersten Versuchsphase noch keine Anwendung. Daher ist der Zustandsraum S entweder sechs- oder achtdimensional.

Die Informationen, aus denen die Ermittelung des relativen Zustands erfolgt, sind in einer Klasse namens Situation gekapselt. Dadurch lassen sich bereits besuchte Zustände erneut abrufen. Auch der interne Datenfluss ist dadurch gewährleistet. Erst wenn ein Situation-Objekt durch ein Netz propagiert werden soll, findet die Berechnung des relativen Zustandes statt. Dadurch bleibt die Relativierung unter der Kontrolle des Netzes.

3.1.2 Aktionen

Im Roboterfußball ist die Aktionsmenge unendlich groß. In dem hier präsentiertem Fall ist dies jedoch hinderlich. Daher wird die Aktionsmenge limitiert. Die verwendeten Aktionsgruppen beschränken sich auf Ballbeschleunigungen (Kick), Spielerbeschleunigungen (Dash) und Spielerdrehungen (Turn). Die Aktionsmenge innerhalb jeder Aktionsgruppe ist über zwei oder im Falle von Turn einem Parameter zu beschreiben. Der Parameter von Turn drückt dabei den Drehwinkel aus. Der erste Parameter von Kick und Dash beschreibt dabei die Stärke von 0 bis 100. Der Zweite Parameter ist der Winkel relativ zum Spieler, in dessen Richtung die Aktion durchgeführt werden soll. Ein Dash(75, 90) beschleunigt den Spieler mit 75 prozentiger Stärke von ihm aus gesehen nach Links. Bei den Dash-Aktionen sei noch einmal darauf hingewiesen, dass nur acht Richtungen unterstützt werden und dass alle Aktionen, deren Winkel nicht 0 entspricht, abgeschwächt sind. Die Stärke-Parameter der qualifizierten Aktionen werden in diskrete 5er Schritte aufgeteilt. Das selbe Verfahren findet auf den Winkel-Parametern von Kick und Turn Anwendung. Einzig der Winkel-Parameter von Dash wird aus offensichtlichen Gründen in 45er Schritte unterteilt. Der Wert 0 wird für alle Stärke-Parameter übersprungen, da dessen Verwendung dem Durchführen keiner Aktion entspricht. Auch Turn(0) ist zu verwerfen. Daher ergeben sich 684 Kick-, 71 Turn- sowie 152 Dash-Aktionen.

3.1.3 Zustandsübergänge

Da die 2D-Simulationsliga eine nicht deterministische Umgebung darstellt, müsste eine Wahrscheinlichkeitsverteilung verwendet werden. Es gibt eine unendliche Menge von möglichen Folgezuständen, die allerdings alle sehr eng beieinander liegen. Aufgrund der Ähnlichkeit der Folgezustände und der Annahme, dass das neuronale Netz diese Verteilung von selbst erlernt, wird anstelle der Wahrscheinlichkeitsverteilung von einer Übergangsfunktion $f(s_t, a_t) = s_{t+1}$ ausgegangen. Der empirische Beweis folgt in der Versuchsauswertung.

3.1.4 Kosten

Die Kosten für ein Zustands-Aktions-Paar errechnen sich anhand der Differenz der Distanz zwischen einem in Zielrichtung liegenden Punkt, sowie der Spielerposition. Es sei p_t die Position des Spielers zum Zeitpunkt t, p' der in Zielrichtung liegende Punkt und $d(p_t, p_{t+1})$ eine Funktion, die die Distanz zweier Punkte berechnet. Dann lautet die konkrete Formel:

$$c(s_t, a_t) = 10 * (d(p_{t+1}, p') - d(p_t, p')), s_t \in S, a_t \in A(s_t) \tag{3.1}$$

In der Umsetzung erfolgt eine Substitution der Aktion durch den Folgezustand, um die Implementierung zu vereinfachen.

3.1.5 Diskontierungsfaktor

In Szenario 1 und 2 hat der Diskontierungsfaktor γ den Wert $0,5$. Eine Reihe von Tests ergibt, dass mit diesem Wert unter den jeweiligen Bedingungen die besten Ergebnisse erzielt werden. In Szenario 3 hat γ den Wert $0,7$. Dieser ergibt sich, wie auch in den anderen Szenarien, aus einer Reihe von Tests, in denen mit $0,7$ eindeutig bessere Werte erreicht werden.

3.2 Neuronales Netz

Wie schon in 1 erwähnt erfolgt der Aufbau des kNN (künstliches neuronales Netz) durch den n++ Simulator. Dieser findet innerhalb des FRA-UNIted Frameworks mehrfach Verwendung und kann als Standardlösung für den Aufbau sowie die Kontrolle von kNNs innerhalb des Frameworks betrachtet werden. Die Vorteile der Verwendung von n++ liegen zum einen in der Fülle an Beispielen für die Implementierung, wodurch die Arbeit mit n++ erleichtert wird. Zum anderen erleichtert es Mitgliedern des FRA-UNIted Teams die Wartung und Weiterentwicklung des Verhaltens, da auf innerhalb des Frameworks bekannte Klassen zurückgegriffen wird. Ein weiterer Vorteil liegt in der Möglichkeit den n++ Quelltext zu warten, da dieser vorliegt.

Bei dem kNN selbst handelt es sich um ein mehrschichtiges, vorwärtsgerichtetes Netz mit einer versteckten Schicht. Dabei finden je nach Szenario sechs oder acht Eingabe Neuronen Verwendung. Bei sechs Eingabe Neuronen befinden sich vierzehn Neuronen in der versteckten Schicht, ansonsten zweiundzwanzig. Bei allen Konstellationen existiert ein Ausgabe Neuron. Die Anzahl der Neuronen in der versteckten Schicht ergibt sich aus einer Reihe von Versuchen mit unterschiedlicher Anzahl. Die gewählten Strukturen erweisen sich dabei in den Versuchen als am effektivsten.

Die Aktivierungsfunktion des Ausgabe Neurons ist eine lineare Funktion, während die Neuronen der versteckten Schicht auf eine logistische Funktion zurückgreifen. Die Notwendigkeit dazu ergibt sich aus der Verwendung des in 2.6.3 vorgestelltem Rprop-Algorithmus.

Alle Neuronen einer Schicht sind mit allen Neuronen der nachfolgenden Schicht verbunden. Dies ist das Standardvorgehen des n++ Simulators. Alle Gewichte haben bei Initialisierung Werte im Bereich von $-0,5$ bis $+0,5$, wobei der Wert 0 nicht verwendet wird, da dieser nicht mit Backpropagation-Algorithmen kompatibel ist.

3.3 Aufbau des Agenten

Der hier vorgestellte Agent ist Modular aufgebaut. Dieser Aufbau soll im Folgenden erläutert werden.

3.3.1 NeuroDribble2017

Das Dribbelverhalten *NeuroDribble2017* dient als Koordinationsstelle, die alle Elemente des Verhaltens bündelt und steuert. Diese Klasse erbt von *BaseBehavior* und qualifiziert sich damit als Untermodul des Entscheidungsmoduls des FRA-UNIted Frameworks. Die get_cmd ist in Zusammenhang mit der set_target Methode von zentraler Bedeutung. Die set_target Methode sollte dabei vor jedem Aufruf der get_cmd Methode Verwendung finden, da der hier übergebene Winkel die Zielrichtung beschreibt, ohne die nur das in 4.1 beschriebene Verhalten arbeiten kann.

Über die Methoden start_training, pause_training und execute_training lässt sich der Lernvorgang steuern. start_training aktiviert den Lernvorgang. Ein mehrfaches Aufrufen löst keinen Fehler aus. pause_training pausiert den Lernvorgang, damit das Verhalten nur relevante Situationen zur Trainingsmenge hinzufügt. Mit dem Aufruf von execute_training verlässt das Verhalten den Lernzustand, wie es dies auch bei pause_training tut. Bei execute_training findet allerdings der eigentliche Lernvorgang statt, da hier das neuronale Netz eine Veränderung erfährt.

NeuroDribble2017 verwendet in der get_cmd Methode eine Instanz der Klasse *CommandAnalysis*, um die bestmögliche Aktion zu wählen. Im Lernzustand übergibt *NeuroDribble2017* eine Zustandsbeschreibung an ein Objekt der Klasse *TrainControl*, die das Training des Netzes steuert. Außerdem wählt *NeuroDribble2017* im Lernzustand mit einer fünfprozentigen Wahrscheinlichkeit eine Zufallsaktion aus. Die Auswahl dieser Zufallsaktion geschieht unter Verwendung von *CommandAnalysis*.

In beiden Fällen erfolgt der Aufruf von *CommandAnalysis* für jede Aktions-Gruppe. *NeuroDribble2017* setzt die Aktion mit der besten Bewertung in das übergebene *Cmd* Objekt ein.

NeuroDribble2017 führt keine weitere Überprüfung durch. Die gewählte Aktion muss also nicht zwanglsäufig sinnvoll sein. Falls ein anderes Verhalten dieses Modul aufruft, obwohl der Ball außerhalb des Einflussbereiches des Spielers liegt, kann keine als sinnvoll, geschweige denn als optimal zu erachtende Aktion gewählt werden. Das Training des verwendeten Netzes findet ausschließlich unter der Annahme statt, dass der Spieler den Ball beeinflussen kann. Diese Voraussetzung ist zu stellen, da es sich bei dem Abfangen des Balles um eine gänzlich unterschiedliche Problemstellung handelt, die nicht von einem Dribbelverhalten übernommen werden sollte. Ein etwaiges aufrufendes Verhalten hat dies zu berücksichtigen

3.3.2 CommandProvider

CommandProvider ist eine Schnittstelle, welche die einheitliche Iteration durch eine Aktions-Gruppe, wie den Spielerdrehungen, ermöglicht. Folgende Methoden müssen von einer umsetzenden Klasse implementiert werden:

- provide(Cmd &cmd, int iteration) : void
- get_max_iterations() : int
- is_of_type(Cmd &cmd) : bool
- copy(Cmd ©_target, Cmd ©_source) : void
- to_string(Cmd &cmd) : string

Die Methode provide stellt dabei die zentrale Methode der Schnittstelle dar. Nach Vereinbarung füllt die Methode das übergebene Objekt mit der n-ten Aktion aus, wobei n durch iteration+1 gegeben ist. Als Beispiel diene folgender Fall. Es sei *DashProvider* der Art *CommandProvider*. *DashProvider* solle eine Auswahl der Spielerbeschleunigungen (Dash) bereitstellen. Die Auswahl sei definiert durch Start- und Endstärke (s_s, s_e), Start- und Endwinkel (w_s, w_e), sowie Schrittgröße der jeweiligen Elemente (Δ_s, Δ_w). Diese seien als Tupel $(s_s, \Delta_s, s_e, w_s, \Delta_w, w_e)$ dargestellt. *DashProvider* gehe dabei so vor, dass für jeden möglichen Winkel alle Stärke Einstellungen iteriert werden. Es seien die Auswahleinstellungen $(5, 5, 100, 0, 10, 180)$. Dann ist die zehnte Aktion, gegeben durch iteration$= 9$ ein Dash mit Stärke 50 und Winkel 0.

Die Methode get_max_iterations gibt die Anzahl der durch das jeweilige Objekt zur Verfügung gestellten Aktionen zurück.

is_of_type gibt zurück, ob ein gesetztes *Cmd* Objekt mit einer Aktion der jeweiligen implementierenden Klasse ausgefüllt ist.

Die copy Methode extrahiert die gesetzten Parameter aus copy_source und setzt die entsprechende Aktion mit diesen Parametern in copy_target. Dies geschieht unter der Annahme, dass in copy_source die Aktion, die durch die implementierende Klasse repräsentiert wird, gesetzt ist.

Die Methode to_string gibt den Aktionsnamen der jeweiligen Klasse, sowie die im *Cmd* Objekt gesetzten Parameter in Form eines *String* Objekts zurück.

3.3.3 SituationAssessment

Bei *SituationAssessment* handelt es sich um eine Schnittstelle, die zwei Methoden bereitstellt. Beide geben dabei eine Gleitkommazahl mit doppelter Präzision *(double)* zurück, die einer Bewertung entspricht. Es gilt: Je höher der Rückgabewert, desto besser die Bewertung. Die Methode assess bewertet ein einzelnes *Situation* Objekt, transition zwei. Für *NeuroDribble2017* sind zwei implementierende Klassen wichtig: *DribbleNetAssessment* und *RandomDribbleAssessment*.

RandomDribbleAssessment gibt eine Zufällige Bewertung zurück, es sei denn der bewertete Zustand führe zu einem Abbruch der aktuellen Episode durch den Trainer. Dann

gibt assess einen negativen Wert mit großem Betrag zurück. Die Methode transition gibt 0 zurück, es sei denn die Differenz der Ausrichtungen in beiden Situation überschreitet 15. In diesem Fall gibt die Methode einen negativen Wert mit großem Betrag zurück. Durch diese Rückgabewerte lassen sich frühzeitige Abbrüche durch äußerst schlechte Entscheidungen einschränken, auch wenn so noch immer Situationen entstehen, in denen der Spieler zwanglsäufig den Ball verlieren muss.

DribbleNetAssessment gibt bei Aufruf von assess den negativen Rückgabewert der forward Methode von *DribbleNetControl* zurück. Bei transition findet das selbe Vorgehen mit der calculate_cost Methode Anwendung.

3.3.4 CommandAnalysis

Die Klasse *CommandAnalysis* besitzt genau eine Methode mit Namen get_best_command. Diese Methode liefert einen *double* zurück, welcher der Bewertung der gewählten Aktion entspricht. Die Methode erwartet ein Objekt der Klasse *Situation*, eine *Cmd* Referenz, eine *CommandProvider* Referenz, sowie eine *SituationAssessment* Referenz als Parameter.

Die übergebene Situation dient als Ausgangspunkt der Entscheidung. Es erfolgt eine Simulation des Folgezustandes für jede Aktion, die das *CommandProvider* Objekt zur Verfügung stellt. *SituationAssessment* bewertet jeden dieser Folgezustände und *CommandAnalysis* übergibt die Aktion mit dem am besten bewerteten Folgezustand an das *Cmd* Objekt. *CommandAnalysis* übergibt die erste qualifizierte Aktion, falls mehrere in Frage kommen.

Der folgende Pseudocode soll das Vorgehen verdeutlichen. Dabei ist $f(s_t, a_t)$ die Simulation des Folgezustandes s_{t+1}.

Input: situation, cmd, provider, situation_assessment
Output: assessment
$i \leftarrow 0$
$assessment \leftarrow -1000$
$best_iteration \leftarrow 0$
while $i < provider.get_max_iterations()$ **do**
  $provider.provide(pseudo_cmd, i)$
  $next_situation \leftarrow f(situation, pseudo_cmd)$
  $current_assessment \leftarrow situation_assessment.assess(next_situation)$
  if $current_assessment > assessment$ **then**
    $assessment \leftarrow current_assessment$
    $best_iteration \leftarrow i$
  end
  $i \leftarrow i + 1$
end
$provider.provide(cmd, best_iteration)$

3.3.5 DribbleNetControl

Die Klasse *DribbleNetControl* dient der Steuerung eines für das Dribbeln ausgelegten Netzes. Der Konstruktor dieser Klasse erwartet ein *String* Objekt, die den Pfad zu einer für den n++ Simulator lesbaren Datei repräsentiert. Falls kein solcher Pfad gesetzt ist, findet der Standardwert „data\nets_neuro_dribble2017\dribble2017.net“ Anwendung. Falls keine Datei am Ziel vorliegt, oder diese nicht interpretierbar ist, erstellt *DribbleNetControl* ein neues Netz, welches den in 3.2 vorgestellten Spezifikationen entspricht.

DribbleNetControl bietet Methoden zur Verwendung eines neuronalen Netzes. Dazu zählt das Vor- und Zurückpropagieren durch das Netz. Die entsprechenden Methoden erwarten ein *Situation* Objekt, welches sie in die entsprechenden Eingabewerte für das Netz übersetzen. Auch eine Methode zur Kostenberechnung gehört dazu. Diese erwartet anstelle eines Zustand-Aktions-Paares ein Zustand-Folgezustand-Paar. Dies entspricht nicht den Vorgaben des Markov-Entscheidungsprozesses, hat aber den selben Effekt: Die Berechnung der Kosten für einen Zustandsübergang. Diese Alternative ist in der Benutzung leichter durchzuführen. Außerdem ist die Performanz verbessert, da so doppelte Simulationsprozesse vermieden werden.

3.4 Lernvorgang

Das kNN erlernt das Approximieren der Wertefunktion, indem Zustände mit ihren jeweiligen Zielwerten präsentiert werden und die Differenz aus der aktuellen Ausgabe des kNN und diesem Zielwert durch das Netz zurückpropagiert werden. Das Sammeln der Zustände erfolgt dabei unter Verwendung des Soccer-Servers. Der Agent spielt dabei mit seiner Aktuellen Konfiguration und dribbelt für einhundert Zeitschritte.

3.4.1 Trainer

Um den Lernvorgang zu vereinfachen findet der sogenannte Offline-Trainer Verwendung. Dieser hat dieselbe Kontrolle, wie der Schiedsrichter, kann also den Spielzustand setzen und Spieler, sowie den Ball mit beliebigen Geschwindigkeiten und Positionen versehen. Auch die Ausdauer kann so aufgefüllt werden.

Der Trainer setzt den Spieler und den Ball auf eine Startposition und setzt den Spielzustand auf „normales Spiel“ Der Trainer erzeugt dabei die Startzustände zu Beginn einer Trainingspartie zufällig. Die zu erzeugenden Startzustände unterliegen dabei folgenden Bedingungen:

- Der Spieler befindet sich auf Position $\mathbf{p(-50/0)}$.
- Der Spieler beginnt mit der Geschwindigkeit $\mathbf{v(0/0)}$.
- Der Betrag der Differenz des X-Wertes der Position des Balles und des X-Wertes der Position des Spielers muss zwischen $0,3$ und $0,2$ liegen. Diese Regel gilt entsprechend für Y-Werte.
- Der Betrag des X- sowie Y-Wertes der Geschwindigkeit des Balles ist $< 0,3$

Durch diese Regeln wird gewährleistet, dass der Ball im Einflussbereich des Spielers startet und keine Situation erzeugt wird, in dem der Spieler die Ballkontrolle nicht halten kann.

Der Zeitraum vom Startzustand bis zum nächsten Startzustand heißt Episode. Einhundert Episoden bilden eine Sequenz. Am Ende einer Sequenz erfolgt das Lernen des Netzes.

Eine Episode endet mit dem Ballverlust des Spielers oder nach einhundert Zeitschritten. Ballverlust ist dabei wie folgt definiert: Der Ball befindet sich nicht mehr im Ballkontrollbereich des Spielers. Am Ende einer Episode setzt der Trainer den Spielzustand auf „Freistoß für die Linke Mannschaft“. Damit kann der Agent das Ende einer Episode feststellen, ohne entsprechende eigene Mechanismen zu verwenden.

3.4.2 Steuerndes Verhalten

Das steuernde Verhalten *DribbleOverride* erkennt das Ende einer Episode anhand des gesetzten Spielzustandes und ermittelt das Ende einer Sequenz. Dieses Hauptverhalten wird in jedem Zeitschritt aufgerufen und gibt entsprechende Befehle an das Dribbelverhalten *NeuroDribble2017* weiter. *DribbleOverride* informiert *NeuroDribble2017* dabei über das Ende einer Episode und einer Sequenz. Außerdem steuert *DribbleOverride*, ob *NeuroDribble2017* sich aktuell im Lernmodus befindet. Ansonsten ruft *DribbleOverride* *NeuroDribble2017*s set_target(Angle angle), sowie get_cmd(Cmd &cmd) Methoden auf.

3.4.3 Wertefunktion

Der Agent speichert jeden Zustand s, den er während seines Trainings besucht, in die speziell dafür vorgesehene Klasse *TrainingsData*. Dabei weist diese Klasse jedem Zustand s den entsprechenden Folgezustand s' zu. Sobald der Folgezustand s' zur Verfügung steht, lassen sich die Kosten $c(s_t, a_t), a_t \in A(s_t)$ berechnen. Damit erfolgt die Zuweisung der Wertefunktion $V_k(s)$, die lautet:

$$V_k(s) = c(s, a) + \gamma * V_{k-1}(s') \tag{3.2}$$

Dabei propagiert *DribbleNetControl* den Zustand s' durch das Netz, um $V_{k-1}(s')$ zu bestimmen. Der Wert von $V_k(s)$ dient dann als neuer Zielwert des Netzes bei Zustand s.

TrainingsData führt dabei zwei Listen von Einträgen, eine für neue Einträge und eine für alte Einträge. Alte Einträge benötigen eine erneute Simulation, da der Agent andere Entscheidungen fällen würde. Dies ist notwendig, da die Zuweisung der Wertefunktion die Auswahl der Aktion mit der besten Bewertung erfordert, um Optimalität zu erreichen.

Aus der *TrainingsData* Instanz gewinnt die Klasse, die das Lernen steuert, dann 20 000 Zustände, deren Werte im Rprop-Algorithmus Verwendung finden. Circa 10 000 dieser Zustände sind jene, die der Agent in der letzten Sequenz sammeln konnte. Die restlichen Zustände sind zufällig gewählt. Auf diese Art kann der Agent sowohl den Umgang mit verrauschten Aktionen erlernen, wie auch von bereits gewonnenen Erfahrungen profitieren.

Die dann ausgewählten Zustände werden 200 mal zurückpropagiert. Eine Reihe von Tests ergibt, dass der Agent auf diese Art die besten Ergebnisse erreicht.

4 Lernszenarien

Im Folgenden soll der Aufbau der einzelnen Lernszenarien erläutert werden.

4.1 Lernszenario 1: Vereinfachte Situation

Dieses erste Szenario dient als Test- und Referenzszenario. Der Spieler ist schon in Zielrichtung ausgerichtet. Daher besteht keine Notwendigkeit die Zielrichtung in die Eingabe des neuronalen Netzes aufzunehmen. Es ergeben sich folgende sechs Eingaben für das Netz:

- Der X-Wert der Geschwindigkeit des Spielers
- Der Y-Wert der Geschwindigkeit des Spielers
- Der X-Wert der relativen Ballposition
- Der Y-Wert der relativen Ballposition
- Der X-Wert der Geschwindigkeit des Balles
- Der Y-Wert der Geschwindigkeit des Balles

Relativ bedeutet relativ zum Spieler. Alle Vektoren sind entgegen der Spielerausrichtung gedreht.

Um die Erfolgreiche Anpassung des Netzes an die verrauschte Umgebung zu demonstrieren, findet dieses Lernszenario sowohl in einer deterministischen sowie einer stochastischen Umgebung statt.

Da der Spieler bereits In Zielrichtung gedreht ist, steht die Dreh-Aktion nicht zur Verfügung. Des weiteren kann er keine Spielerbeschleunigung mit einem anderen Winkel als 0 durchführen. Er kann also nur nach Vorne beschleunigen.

4.2 Lernszenario 2: Turnierbedingungen

In diesem Szenario findet sowohl eine Erweiterung der Eingabe Werte, als auch der zur Verfügung stehenden Aktionen statt.

Der Spieler ist in diesem Szenario nicht mehr in die korrekte Richtung gedreht.

Zu den in 4.1 beschriebenen Eingabe Werten kommen zwei hinzu:

- Der X-Wert der Zielrichtung
- Der Y-Wert der Zielrichtung

Daraus ergibt sich eine Gesamtmenge von acht Eingaben. Die Zielrichtung ist ein Vektor, der sich aus dem Zielwinkel ableitet. Dieser ist auf die Länge 1 normalisiert und so wie alle Vektoren entgegen der Spielerausrichtung gedreht.

Um die Zielrichtung zu erreichen, steht dem Agenten die Aktionsgruppe der Drehbewegungen zur Verfügung.

4.3 Lernszenario 3: Erweiterung der Aktionsmenge

Dieses Szenario stellt die geringste Erweiterung der bisherigen Szenarion dar. Die in 4.2 beschriebenen Eingaben und Aktionen finden Anwendung. Darüber hinaus findet eine Erweiterung der Aktionsmenge um die ausbleibenden Mitglieder der Spielerbeschleunigungsaktionen statt. Der Spieler kann nun nicht mehr nur nach Vorne beschleunigen, sondern in alle acht zur Verfügung stehenden Richtungen.

5 Lernergebnisse

Die Lernfortschritte des Agenten werden im folgenden Kapitel vorgestellt und mit den Ergebnissen bereits bestehender Dribbelverhalten verglichen. Der Agent erlernt das Dribbeln mithilfe eines simplen Explorationsmechanismus, der mit einer Wahrscheinlichkeit von 5% eine zufällige Aktion auslöst.

Die Erhebung der Daten der Lernvorgänge findet unter Verwendung von 100 unterschiedlichen Episoden pro Sequenz statt, wobei sich die Episoden in ihren Startzuständen unterscheiden. Sowohl die Evaluations- wie auch Lernepisoden sind bis zu 100 Zeitschritte lang, werden aber bei Ballverlust beendet. Abgesehen von dem ersten Teil des ersten Szenarios, finden alle Lernvorgänge und Evaluationen in einer verrauschten Umgebung statt.

Die genauen Regeln der Szenarien sind Kapitel 4 zu entnehmen. Die Evaluationen unterliegen den selben Regeln, wie die Trainingsszenarien. Die Startzustände sind für die Evaluationen neu generiert.

Für Liniendiagramme gilt: Die Nummer der Lernepisode ist auf der x-Achse abgebildet. Die zurückgelegte Distanz ist auf der y-Achse abgebildet. Während in Szenario 1 nur von der Distanz ausgegangen wird, müssen in Szenario 2 und 3 die zurückgelegte Distanz in Richtung Ziel und die insgesamt gelaufene Strecke miteinander in Relation gebracht werden. Durch diesen Vergleich ergibt sich, wie weit der Agent von seiner Zielrichtung gestreut ist.

Die Ergebnisse der Evaluation werden mit dem Verhalten TingleTangle verglichen, da es das schnellste Dribbelverhalten des FRA-UNIted Frameworks ist, welches den Anforderungen entspricht. Dies ergibt sich aus einer Reihe von Tests. Die Vergleichswerte selbst ergeben sich aus wiederholten Versuchen mit TingleTangle, wobei stets der beste Wert als Referenz verwendet wird.

Während die Relation zwischen der Anzahl der verstrichenen Zeitschritte und der zurückgelegten Distanz die Effizienz des Dribbelverhaltens bestimmt, ergibt sich die Performanz aus der realen Zeit, die der Agent pro Zeitschritt für seine Berechnungen benötigt. Diese Angaben sind hochgradig relevant, da die Aktion innerhalb von 100 ms an den Server gesendet werden muss. Es ist zu erwarten, dass einige andere Berechnungen stattfinden, bis ein etwaiges Verhalten das hier vorgestellte NeuroDribble2017 aufruft. Falls zu viele Überprüfungen im Vorfeld stattfinden, ist die Verwendung von wenig performanten Verhalten mit großer Vorsicht durchzuführen, da sonst innerhalb eines Zeitschrittes möglicherweise kein Befehl abgesetzt wird, wodurch die Durchführung einer Aktion ausbliebe.

Die Angaben zurückgelegter Distanzen beziehen sich auf Durchschnittswerte, falls nicht anders angegeben. Maximum bezieht sich auf den besten Wert in der jeweiligen Episode. Die Evaluation des Agenten findet alle 10 Lernepisoden statt. Das Verhalten NeuroDribble2017 wird in den Grafiken mit ND17 abgekürzt.

Alle Tests finden auf einer Maschine mit 8 GB RAM sowie einer 4-Kern CPU a 3,2 GHz statt.

5.1 Lernszenario 1: Vereinfachte Situation

5.1.1 Lernverhalten

Die Abbildung 5.1 zeigt die Lernkurve für das Szenario 1. Die Kurven für das Szenario mit abgeschaltetem und eingeschaltetem Rauschen sind hier abgebildet.

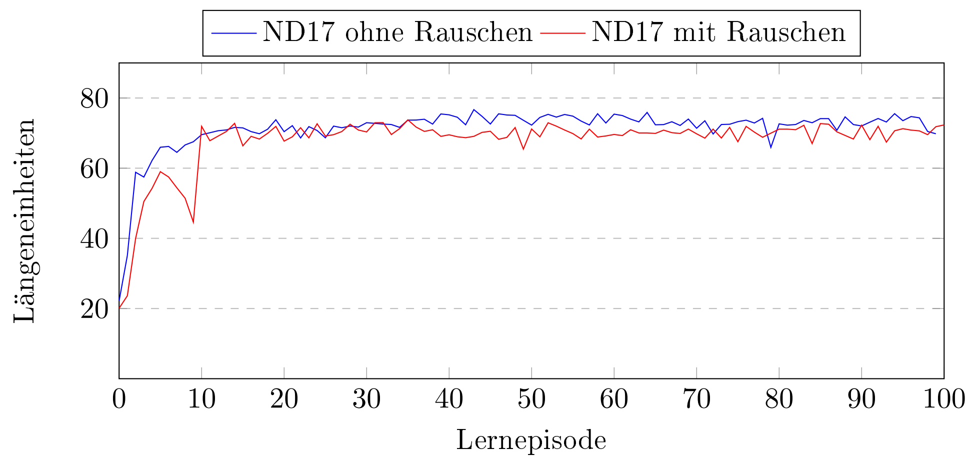

Abbildung 5.1: Lernkurve Szenario 1.1 und 1.2

Ohne Rauschen legt der Agent nach 11 Lernepisoden durchschnittlich 70 Längeneinheiten *(LE)* zurück. Nach 35 Episoden pendelt der Wert um 73 LE.

Der Agent schafft nach 10 Episoden 72 LE mit eingeschaltetem Rauschen, pendelt aber später um 70 LE. Die Verschlechterung im Vergleich zu Szenario 1.1 ist auf das Rauschen zurückzuführen, da die Ballkontrolle erschwert ist.

5.1.2 Evaluation

Die Abbildung 5.2 zeigt die Lernergebnisse aus Szenario 1. Um die Behauptung aus Kapitel 3.1.3, die deterministische Zustandsübegangsfunktion sei ausreichend, zu belegen, sind die Evaluationen der Szenarien 1.1 und 1.2 zu vergleichen.

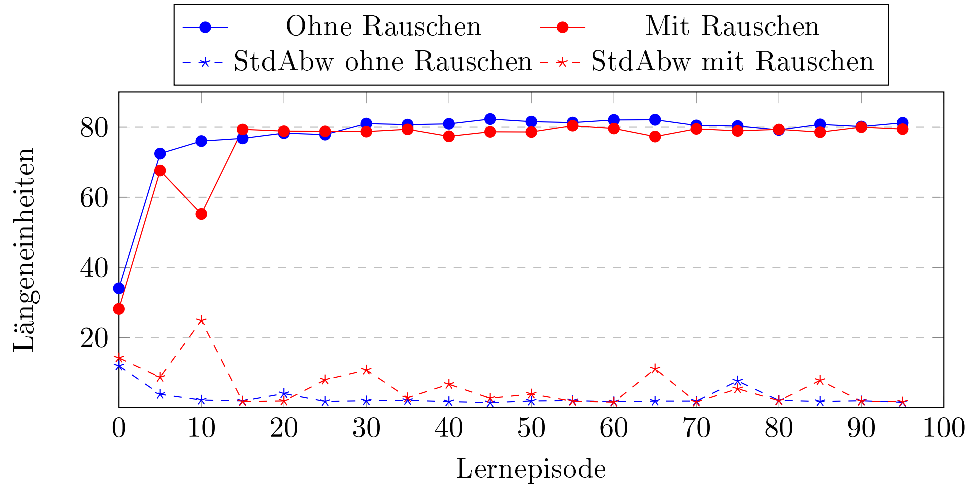

Abbildung 5.2: Lernerfolg Szenario 1.1 und 1.2

In Szenario 1.1 erreicht der Agent nach 30 Episoden eine durchschnittliche Geschwindigkeit von 81 LE pro 100 Zeitschritte und pendelt um 80 LE pro 100 Zeitschritte. Er legt in Episode 45, 60 und 65 durchschnittlich mehr als 82 LE zurück. Die Standardabweichung pendelt zwischen 1 und 2 LE, mit Ausreißern in Episode 0 und 75.

Wie erwartet schneidet Szenario 1.2 nicht ganz so gut ab. Die durchschnittlich zurückgelegte Distanz liegt nach 15 Episoden bei 79 LE und pendelt um diesen Wert. Die Verschlechterung der Durchschnittsgeschwindigkeit ist mit 2 LE gering, doch die Standardabweichung weist mehr Ausreißer auf, wobei der Wert zwischen 1 und 3 LE pendelt. Der Unterschied ist auf das Rauschen selbst zurückzuführen.

Die Abbildung 5.3 zeigt den Vergleich des Szenarios 1.1 mit den Ergebnissen von TingleTangle.

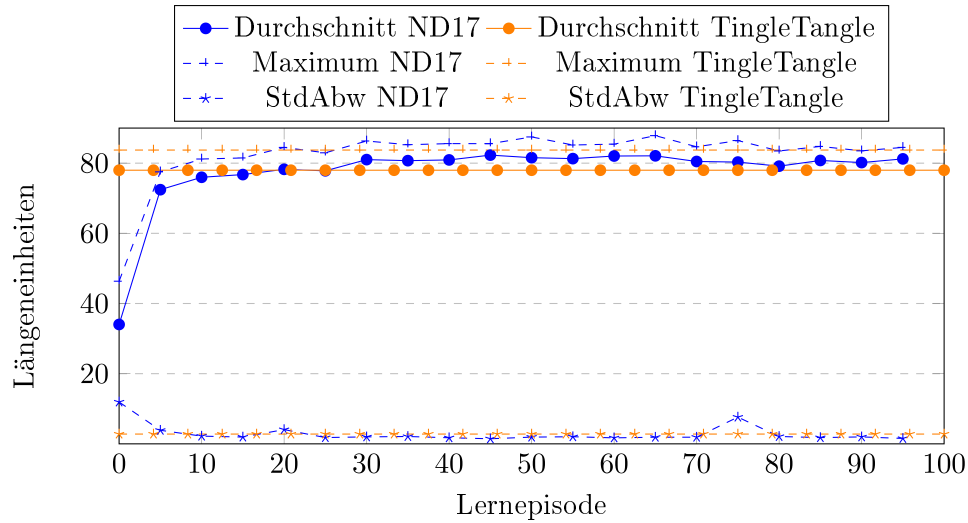

Abbildung 5.3: Lernerfolg Szenario 1.1

TingleTangle legt im Durchschnitt eine Distanz von 77,9909 LE innerhalb von 100 Zeitschritten zurück, wobei die Standardabweichung bei 2,78768 LE liegt. TingleTangle erreicht bis zu 83,7358 LE.

Es ist zu erkennen, dass NeuroDribble2017 ab Episode 30 durchweg besser ist als TingleTangle. Mit der Ausnahme von Episode 75 ist die Standardabweichung von NeuroDribble2017 geringer. Die durchschnittlich zurückgelegte Distanz von NeuroDribble2017 ist größer als die von TingleTangle. Darüber hinaus ist der beste Wert von NeuroDribble2017 besser, als der von TingleTangle.

Die Abbildung 5.4 zeigt den Vergleich des Szenarios 1.2 mit den Ergebnissen von TingleTangle.

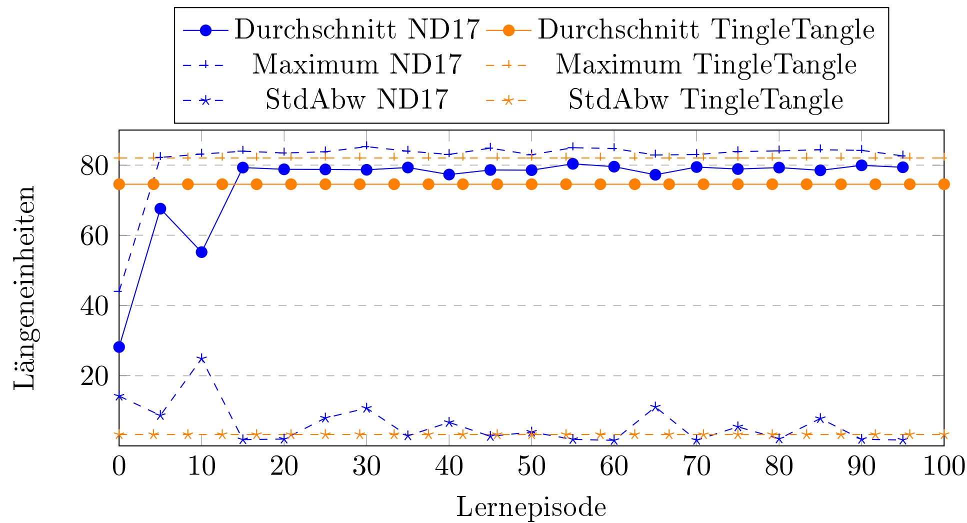

Abbildung 5.4: Lernerfolg Szenario 1.2

Auch TingleTangles Ergebnisse verschlechtern sich Aufgrund des Rauschens. Die durchschnittliche Distanz liegt bei 74,5649 LE und hat damit 3,426 LE im Vergleich zum abgeschalteten Rauschen eingebüst. Die Standardabweichung steigt auf 3,227172 LE.

NeuroDribble2017 ist nach 15 Lernepisoden besser als TingleTangle. Die Durchschnittsdistanz ist um rund 5 LE größer. NeuroDribble2017 übertrifft TingleTangles Maximalwert regelmäßig, wobei die größte Differenz nach 30 Episoden bei 3,24 LE liegt.

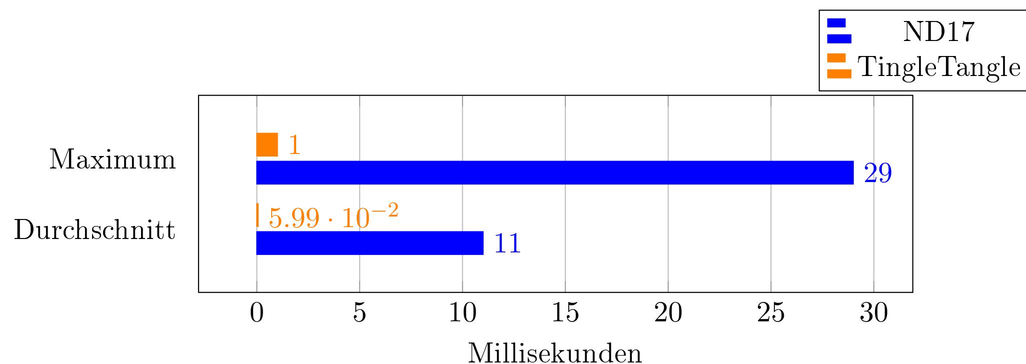

Abbildung 5.5: Performanz Szenario 1

Wie aus Abbildung 5.5 zu entnehmen ist, liegt die Rechenzeit von NeuroDribble2017 deutlich über der von TingleTangle. Letzteres benötigt bis zu 1 ms, bis alle Berechnungen abgeschlossen sind, wohingegen NeuroDribble2017 bis zu 29 ms benötigt. Die durchschnittliche Rechenzeit von NeuroDribble2017 liegt bei 11 ms, während TingleTangle nach durchschnittlich 0,059902 ms bereits ein Ergebnis liefert.

5.2 Lernszenario 2: Turnierbedingungen

5.2.1 Lernverhalten

Die Abbildung 5.6 zeigt die Distanz, die Der Agent während des Lernens zurückgelegt hat. Diese Distanz ist mit der Distanz, die er in die Zielrichtung zurückgelegt hat zu vergleichen.

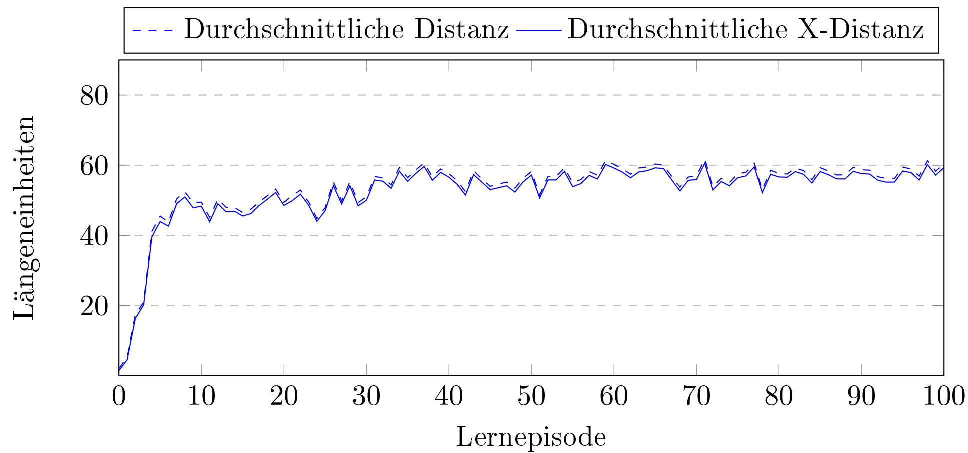

Abbildung 5.6: Lernkurve Szenario 2

Es zeigt sich, dass der Agent mit einer sehr geringen Abweichung zwischen der Distanz in Zielrichtung und der gesamten Distanz arbeitet. Die Differenz in allen Episoden beträgt in etwa 2 LE. Daraus lässt sich schließen, dass der Agent mit hoher Zuverlässigkeit in die angegebene Richtung läuft. Die durchschnittlich zurückgelegte Distanz pendelt ab Episode 31 um 57,5 LE.

5.2.2 Evaluation

Abbildung 5.7 stellt die Distanzen, die insgesamt zurückgelegt wurden, sowie die in Zielrichtung zurückgelegten Distanzen dar. Sie bildet sowohl TingleTangle als auch NeuroDribble2017 ab.

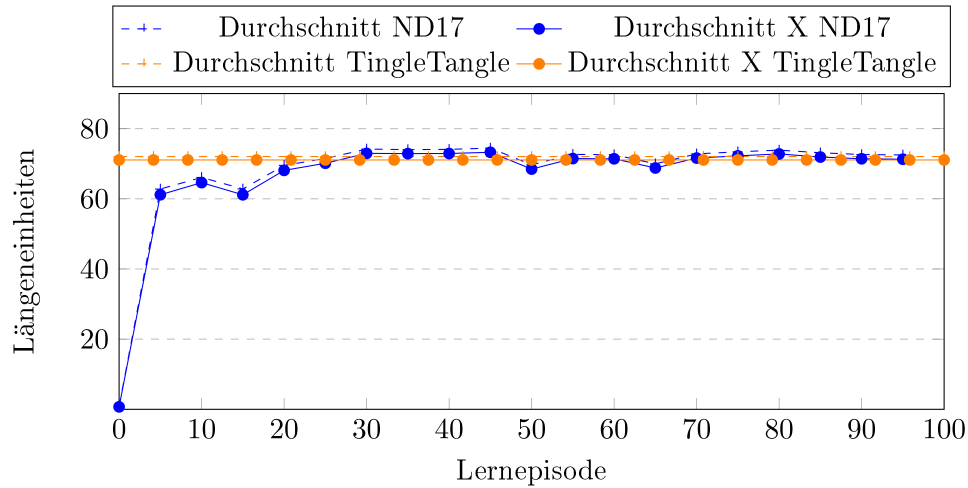

Abbildung 5.7: Lernerfolg Szenario 2: Streuung

TingleTangle legt durchschnittlich 71,07 LE in Zielrichtung zurück. Bei einer gesamten Durchschnittsdistanz von 72,0465 ergibt sich eine Abweichung von rund 1 LE. Daher lässt sich folgern, dass TingleTangle sehr zuverlässig in die Zielrichtung bewegt.

Diese Differenz ist mit 1,2 LE bei NeuroDribble2017 nur geringfügig größer, als bei TingleTangle, damit ist es als unzuverlässiger anzusehen.

Abbildung 5.8 setzt die erreichten Distanzen in Zielrichtung von NeuroDribble2017 und TingleTangle in Relation zueinander.

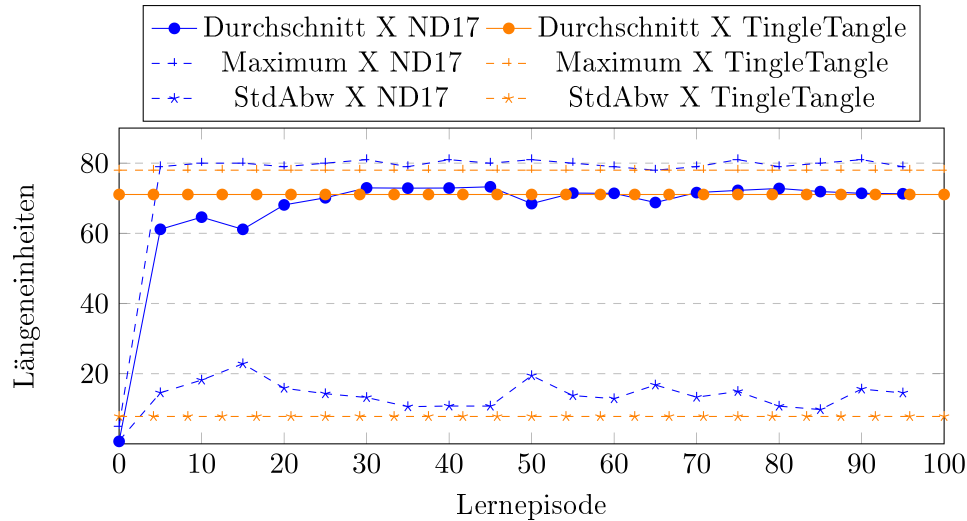

Abbildung 5.8: Lernerfolg Szenario 2

Schon ab Lernepisode 5 schafft es NeuroDribble2017 eine größere Maximaldistanz als TingleTangle zurückzulegen. Ab Episode 30 gelingt es NeuroDribble2017 auch die Durchschnittsdistanz zu überbieten. Es legt etwa 3 LE mehr zurück als TingleTangle. Allerdings ist die Standardabweichung von NeuroDribble2017 höher, als die von TingleTangle. Während TingleTangle eine Standardabweichung von 7,80545 LE aufweist, liegt die Standardabweichung von NeuroDribble2017in Episode 35 bei 10,5 LE. Dieser Wert hat sein Maximum mit 22,8 LE in Episode 15.

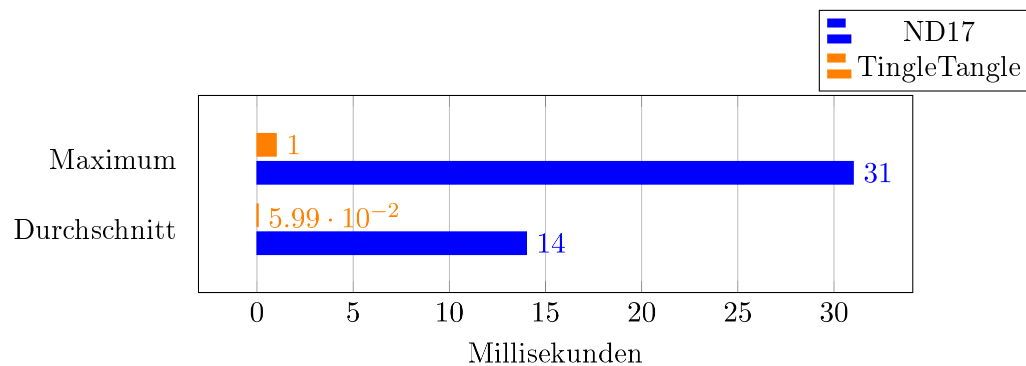

Abbildung 5.9: Performanz Szenario 2

Durch die vergößerte Aktionsmenge benötigt NeuroDribble2017 in Szenario 2 mehr Rechenzeit. Wie aus dem Diagramm 5.9 hervorgeht, steigt die Durchschnittszeit im Vergleich zu Szenario 1 um 3 ms auf 14 ms an und die Maximalzeit um 2 ms auf 31 ms.

5.3 Lernszenario 3: Erweiterung der Aktionsmenge

5.3.1 Lernverhalten

Die Grafik 5.10 zeigt die Lernkurve des Agenten in Szenario 3. Wie auch in Szenario 2 findet der Vergleich zwischen der gesamten und der in Zielrichtung zurückgelegten Distanz statt.

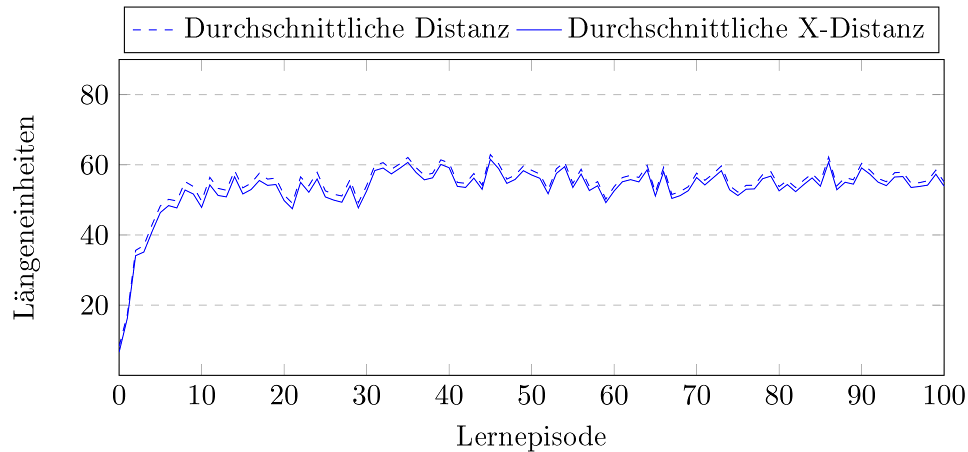

Abbildung 5.10: Lernkurve Szenario 3

Die dargestellte Lernkurve verhält sich ähnlich zu der in 5.6. Genau wie im Vorgänger Szenario liegt die Differenz zwischen den Distanzen bei rund 2 LE. Ab Episode 32 pendelt der Wert um 58 LE, allerdings sind die Abweichungen von Episode zu Episode größer als in Szenario 2. Das kann daran liegen, dass etwaige zufällige Seitwärtsschritte die zurückgelegte Distanz stärker negativ beeinflussen, als andere Zufallsaktionen.

Die Abbildung 5.11 zeigt die Streuung von NeuroDribble2017 und TingleTangle in Szenario 3. Dabei fällt auf, dass im Vergleich zu Szenario 2 keine signifikanten Änderungen der Streuung zu finden sind.

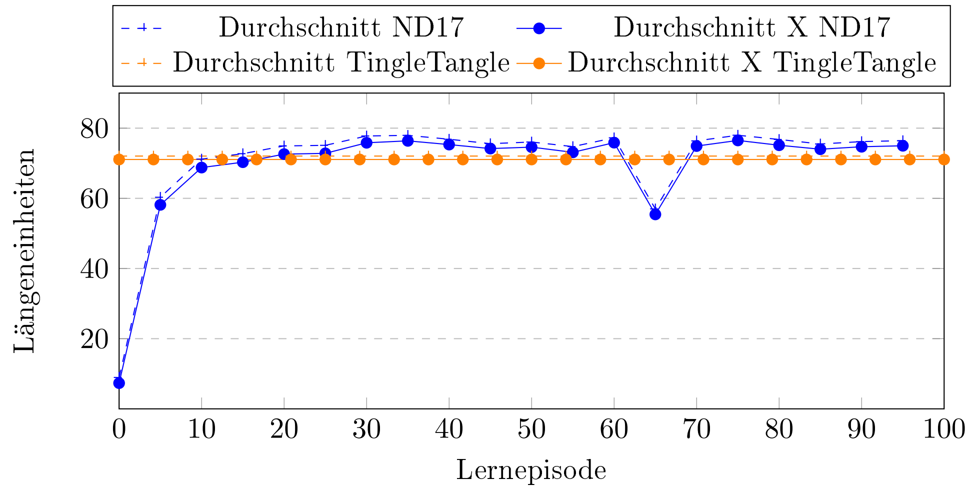

Abbildung 5.11: Lernerfolg Szenario 3: Streuung

5.3.2 Evaluation

In der Grafik 5.12 ist der Vergleich zwischen TingleTangle und NeuroDribble2017 zu sehen.

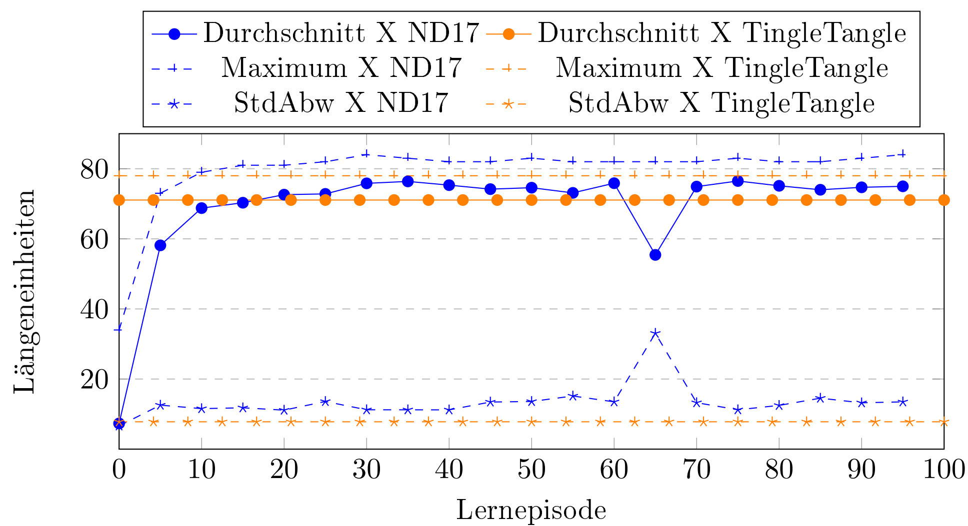

Abbildung 5.12: Lernerfolg Szenario 3

Ab Episode 10 legt NeuroDribble2017 eine größere, maximale Distanz in Zielrichtung zurück als TingleTangle. Mit Ausnahme von Episode 65 ist auch die Durchschnittsgeschwindigkeit von NeuroDribble2017 größer, als die von TingleTangle. In Episode 35 liegt die Differenz zwischen den beiden Durchschnittsgeschwindigkeiten bei 5,3 LE. Aber auch NeuroDribble2017s Standardabweichung ist größer. Diese schwankt zwischen 20 LE und

11 LE, wobei sie in Episode 65 auf 33 LE ansteigt. Im besten Fall liegt die Differenz zwischen der Standardabweichung von TingleTangle und NeuroDribble2017 also bei 3,2 LE. TingleTangle bietet daher zwar stabilere Ergebnisse, doch NeuroDribble2017 schafft es im Regelfall eine größere Distanz zurückzulegen.

Das Diagramm 5.13 vergleicht die Rechenzeiten von NeuroDribble2017 und TingleTangle. In Szenario 3 ist die Rechenzeit von NeuroDribble2017 weiter angestiegen. Sie ist durchschnittlich 6 ms länger als in Szenario 2 und der Maximalwert ist um 10 ms gestiegen.

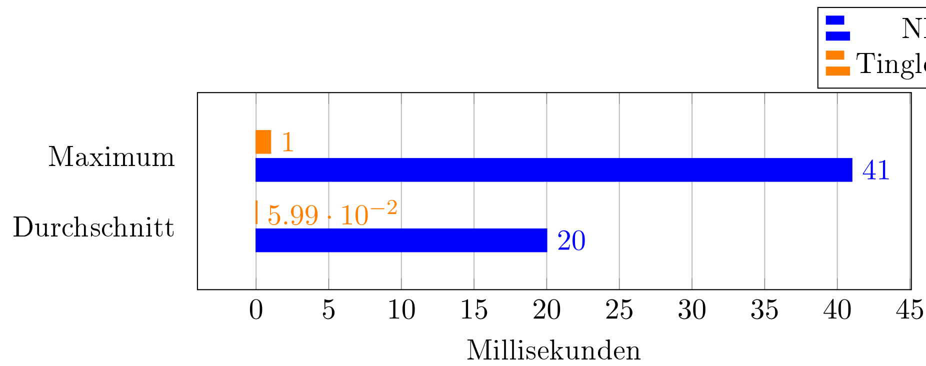

Abbildung 5.13: Performanz Szenario 3

6 Fazit

Im Folgenden sollen die Ergebnisse aus Kapitel 5 reflektiert werden und mit der Zielsetzung in Verbindung gebracht werden. Die Zielsetzung dieser Arbeit lautet ein Dribbelverhalten zu erstellen, welches unter Turnierbedingungen effizienter agiert, als das bisher verwendete. Das erste Lernszenario dient der Feststellung der Robustheit des Agenten in einer verrauschten Umgebung. Die Erfolge des Agenten sprechen für diese Robustheit. Unter den Bedingungen des zweiten Teils von Szenario 1 gelingt es dem Agenten eindeutig bessere Ergebnisse zu erreichen, als das TingleTangle-Verhalten. Sowohl NeuroDribble2017s Durchschnitts- und Maximalwerte als auch die Standardabweichung sind besser, als TingleTangles Werte. Allerdings ist NeuroDribble2017 rechenintensiver als TingleTangle. In Szenario 2 soll die Fähigkeit des Agenten sich in eine Zielrichtung zu bewegen ermittelt werden. Das erlernte liegt nahezu gleichauf mit dem handgeschriebenen Verhalten. Doch TingleTangles Standardabweichung wie auch Rechenzeit bieten bessere Werte, als NeuroDribble2017s Ergebnisse. Das Szenario 3 erweitert die Aktionsmenge des Agenten um Seitwärtsschritte. Durch diese gelingt es ihm sowohl Maximal- wie auch Durchschnittsdistanz im Vergleich zu Szenario 2 zu erhöhen. Diese Werte liegen damit über denen von TingleTangle. Allerdings weist das erlernte Verhalten eine größere Standardabweichung und Rechenzeit auf. Aus Zeitgründen konnte das Ziel nicht ganz erreicht werden. Es fehlt das Training mit heterogenen Spielertypen, die unter Turnierbedingungen zum Einsatz kommen. Da die Ergebnisse in Szenario 3 jedoch die von TingleTangle übertreffen, ist davon auszugehen, dass dieser letzte Trainingsschritt zu keinen großen Einbusen führen wird. Die größten Nachteile des erlernten Verhaltens sind die höhere Standardabweichung und die viel längere Rechenzeit. Die Standardabweichung ist in Szenario 3 jedoch tendenziell geringer als die dazu gewonnene Durchschnittsdistanz, weswegen dies als effizienter angesehen werden kann. Die längere Rechenzeit stellt ein nicht unerhebliches Problem dar. Inwieweit sie wirklich problematisch ist, kann allerdings erst festgestellt werden, wenn dieses Verhalten innerhalb des FRA-UNIted Frameworks Verwendung findet. Es obliegt den implementierenden Programmierern die Vor- und Nachteile abzuwägen.

7 Ausblick

Dieses Kapitel beschäftigt sich mit ausgebliebenen Aspekten dieser Arbeit, sowie möglicher Folgearbeiten. Eigenschaften wie der Radius des Ballkontrollbereichs oder die maximale Geschwindigkeit unterscheiden sich je nach Spielertyp. In dieser Arbeit ist nur mit dem Standardspielertyp trainiert worden. Daher ist von einer Überspezialisierung auszugehen. Eine mögliche Lösung besteht in einem Training mit vielen verschiedenen Spielertypen. Auf diese Art kann das Netz neue Abschätzungen ausgeben, die den heterogenen Spielertypen gerecht wird. Ein anderes Verhalten muss das hier Entwickelte aufrufen, falls es Verwendung finden soll. Es bietet sich an, ein taktisches oder strategisches Verhalten zu entwickeln, welches anhand der Positionen aller Spieler entscheidet, in welche Richtung das Dribbeln durchgeführt werden soll. Darüber hinaus kann dieses Verhalten bestimmen, wie viel Zeit in diesem Zeitschritt verbleibt, um eine Entscheidung zu fällen. Da TingleTangle ähnliche Möglichkeiten wie das hier entwickelte Verhalten bietet, kann das aufrufende Verhalten darauf zurückgreifen, falls nicht mehr genug Rechenzeit bleibt.

Literatur

[1] URL: http://www.neuronalesnetz.de/ (besucht am 12. 03. 2017).

[2] Thomas Gabel und Constantin Roser. „FRA-UNIted — Team Description 2016“ (2016).

[3] Leslie Pack Kaelbling und Michael L. Littman. „Reinforcement Learning: A Survey“ (1996).

[4] Warren S. McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“ (1943).

[5] M. Riedmiller und H. Braun. „A direct adaptive method for faster backpropagation learning: the RPROP algorithm“ (1993).

[6] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. „Learning representations by back-propagating errors“ (1986).

[7] Stuart Russell und Peter Norvig. *Künstliche Intelligenz*. Pearson, 2012.

[8] Fabian Sommer. „Maschinelles Lernen im simulierten Roboterfußball: Entwicklung eines neuronalen Dribbelverhaltens mit Methoden des Reinforcement Learnings“. Frankfurt Univerity of Applied Sciences, 2016.

[9] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

Abbildungsverzeichnis

5.1 Lernkurve Szenario 1.1 und 1.2 31
5.2 Lernerfolg Szenario 1.1 und 1.2 32
5.3 Lernerfolg Szenario 1.1 33
5.4 Lernerfolg Szenario 1.2 34
5.5 Performanz Szenario 1 34
5.6 Lernkurve Szenario 2 35
5.7 Lernerfolg Szenario 2: Streuung 36
5.8 Lernerfolg Szenario 2 37
5.9 Performanz Szenario 2 37
5.10 Lernkurve Szenario 3 38
5.11 Lernerfolg Szenario 3: Streuung 39
5.12 Lernerfolg Szenario 3 39
5.13 Performanz Szenario 3 40