# Anwendung von Reinforcement Learning in der RoboCup Soccer Simulation League

Von
Cand. Ing. Arne Usadel

EINGEREICHT ZUR ERLANGUNG DES AKADEMISCHEN GRADES
DIPLOM INGENIEUR(FH)
AN DER
HOCHSCHULE RAVENSBURG-WEINGARTEN
WEINGARTEN, BADEN-WÜRTTEMBERG
DEZEMBER 2005

© Copyright by Arne Usadel, 2005

# HOCHSCHULE RAVENSBURG-WEINGARTEN
# TECHNIK | WIRTSCHAFT | SOZIALWESEN

| | |
|---|---|
| Datum: | **Dezember 2005** |
| Autor: | **Arne Usadel** |
| Titel: | **Anwendung von Reinforcement Learning in der Robocup Soccer Simulation League** |
| Betreuer: | **Prof. Dr. rer. nat. Wolfgang Ertel** |
| 2. Betreuer: | **Prof. Dr. rer. nat. Martin Zeller** |
| Abteilung: | **Angewandte Informatik** |
| Tag der Anmeldung: | **16.08.2005** |
| Tag der Abgabe: | **15.12.2005** |

Der Hochschule Ravensburg-Weingarten werden sämtliche Veröffentlichungen im Ganzen oder als Auszug aus dieser Arbeit ohne weitere Genehmigung des Verfassers zugestanden. Jede weitere Veröffentlichung bedarf der ausdrücklichen schriftlichen Genehmigung des Autors.

DER AUTOR BESTÄTIGT MIT SEINER UNTERSCHRIFT, DASS ER DIE VORLIEGENDE DIPLOMARBEIT OHNE UNZULÄSSIGE FREMDE HILFE VERFASST UND KEINE ANDEREN ALS DIE ANGEGEBEN QUELLEN UND HILFSMITTEL BENUTZT HAT.

IN ALLEN FÄLLEN IN DENEN EINE EINVERSTÄNDNISERKLÄRUNG DES RECHTSINHABERS BENÖTIGT WURDE, WURDE DIESE EINGEHOLT ODER LAG SCHON VOR. DER AUTOR VERSICHERT DAS MIT DIESER ARBEIT KEINE COPYRIGHT-VERLETZUNGEN BEGANGEN WURDEN.

_____

*Unterschrift des Verfassers*

# Inhaltsverzeichnis

**1 Einleitung** **1**
1.1 Zielsetzung der Arbeit . . . . . 1
1.2 Gliederung der Arbeit . . . . . 2

**2 Theoretische Grundlagen** **3**
2.1 Reinforcement Learning . . . . . 3
2.1.1 Das Agenten-Environment Interface . . . . . 4
2.1.2 Goals und Rewards . . . . . 4
2.1.3 Returnwerte . . . . . 5
2.1.4 Die Markovsche Eigenschaft . . . . . 6
2.1.5 Der Markovsche Entscheidungsprozess . . . . . 7
2.1.6 Value Functions . . . . . 8
2.2 Observation based Reinforcement Learning . . . . . 8
2.2.1 Generalisierung beim Reinforcement Learning . . . . 8
2.2.2 Konsequenzen der Generalisierung . . . . . 9
2.2.3 Observation based Reinforcement Learning Verfahren 10
2.2.4 Elemente eines Observation based RFL Algorithmuses 10
2.2.5 Der Ablauf des Observation based Learning . . . . . 13

**3 Die RoboCup Umgebung** **14**
3.1 Die Simulationsumgebung . . . . . 14
3.2 Der RoboCup Monitor . . . . . 14
3.2.1 Einführung . . . . . 15
3.2.2 Kommunikation zwischen Server und Monitor . . . . 15
3.3 Der RoboCup Soccerserver . . . . . 15
3.3.1 Klient- Server Kommunikation . . . . . 16
3.3.2 Die Serverkonfiguration . . . . . 17

**4 Das Lernverfahren** **19**
4.1 Anwendung des Observation based RFL beim Agenten . . . 20
4.1.1 Daten des Environments . . . . . 20
4.1.2 Die Aktionen des Agenten . . . . . 21
4.2 Das UvA Trilearn Team . . . . . 22
4.2.1 Der Trainer . . . . . 23
4.2.2 Das Weltmodell des Spielers . . . . . 24
4.2.3 Wichtige Datentypen des Weltmodells . . . . . 25

4.2.4 Vom Lernagenten verwendete Methoden des Weltmodells . . . 26
4.2.5 Die Problematik der Partikelfilter . . . 27
4.3 Entwicklung des finalen Agenten . . . 28
4.3.1 Grober Ablauf der Lernfunktion . . . 28
4.3.2 Aufbau der Datenstrukturen . . . 29
4.3.3 Abbildung der Observationen . . . 31
4.3.4 Speicher- und Einlesefunktionen . . . 32
4.3.5 Der Trainer dieses Trainingsszenarios . . . 32
4.3.6 Realisierungsprobleme . . . 33

**5 Lernverhalten und Leistungsfähigkeit 36**
5.1 Szenario- und Auswertungsbeschreibung . . . 36
5.1.1 Parametersets . . . 36
5.1.2 Lerneffektivität . . . 39
5.1.3 Performance . . . 40
5.1.4 Verteilungsmessung . . . 41
5.2 Die Trainingsszenarien . . . 41
5.2.1 Die 12x36 Welt - Teil 1 . . . 41
5.2.2 Die 12x36 Welt - Teil 2 . . . 51
5.2.3 Die 24x72 Welt - Teil 1 . . . 55
5.2.4 Die 24x72 Welt - Teil 2 . . . 59
5.2.5 Die 4x4 Welt - Ein Vergleich mit dem Q-Learning . . . 62
5.2.6 Die 24x72 Welt - Ein Vergleich mit Q-Learning . . . 69

**6 Auswertung und Ausblick 75**
6.1 Auswertung der Testergebnisse . . . 75
6.2 Verbesserungen am Lernverfahren . . . 76
6.3 Ausblick . . . 78

**A Sourcecode - rli.h 79**

**B Sourcecode - rli.cpp 84**

**C Sourcecode - PlayerTeams.cpp 96**

**D Sourcecode - BasicCoach.cpp 106**

# Abbildungsverzeichnis


1 SoccerServer Informationsaustausch . . . . . 16
2 Veranschaulichung des Ballwinkels . . . . . 21
3 Veranschaulichung des Körperwinkels . . . . . 22
4 Veranschaulichung des Bewegungsrichtungswinkels . . . . . 23
5 Problematik Lerneffektivität . . . . . 40
6 Referenzparamter der 12x36 Welt - Teil 1 . . . . . 42
7 Lernperformance der 12x36 Welt - Policy 1 . . . . . 43
8 Lernperformance der 12x36 Welt - Policy 2 . . . . . 44
9 Lernperformance 12x36 Welt - Ein Vergleich . . . . . 45
10 Verteilungsmessung der 12x36 Welt - Policy 1 . . . . . 46
11 Verteilungsmessung der 12x36 Welt - Policy 2 . . . . . 47
12 Verteilungsmessung der 12x36 Welt - Ein Vergleich . . . . . 48
13 Verteilung der Startzustände, max 0,8 Meter rund um den Agenten . . . . . 49
14 Verteilung der Startzustände, max 1,085 Meter rund um den Agenten . . . . . 50
15 Referenzparameter der 12x36 Welt - Teil 2 . . . . . 51
16 Lernperformance der 12x36 Welt - Policy 3 . . . . . 52
17 Lernperformance der 12x36 Welt - Ein Vergleich . . . . . 54
18 Lerneffektivität der 12x36 Welt - Ein Vergleich . . . . . 54
19 Referenzparameter der 24x72 Welt . . . . . 56
20 Lernperformance der 24x72 Welt . . . . . 57
21 Lernereffektivität der 24x72 Welt . . . . . 57
22 Die Referenzparameter der 24x72 Welt - Teil 2 . . . . . 59
23 Lernperformance der 24x72 Welt - Teil 2 . . . . . 60
24 Lernperformance der 24x72 Welt - Ein Vergleich . . . . . 60
25 Lerneffektivität der 24x72 Welt - Teil 2 . . . . . 61
26 Lerneffektivität der 24x72 Welt - Ein Vergleich . . . . . 62
27 Die Referenzparameter der 4x4 Welt . . . . . 63
28 Lernperformance der 4x4 Welt - Mit Observation based RFL 64
29 Lernperformance der 4x4 Welt - Mit Q-Learning . . . . . 65
30 Lerneffektivität der 4x4 Welt - Mit Observation based RFL . 66
31 Lerneffektivität der 4x4 Welt - Mit Q-Learning . . . . . 67
32 Verteilungsmessung der 4x4 Welt - Mit Observation based RFL 68
33 Verteilungsmessung der 4x4 Welt - Mit Q-Learning . . . . . 69
34 Die Referenzparameter der 24x72 Welt . . . . . 70

35 Lernperformance der 24x72 Welt - Mit Observation based RFL 71
36 Lernperformance der 24x72 Welt - Mit Q-Learning . . . . . 72
37 Lerneffektivität der 24x74 Welt - Mit Observation based RFL 73
38 Lerneffektivität der 24x74 Welt - Mit Q-Learning . . . . . 74
39 Verteilungsmessung der 24x74 Welt - Mit observation based RFL . . . . . 74
40 Neue Generalisierungsidee . . . . . 77

# 1 Einleitung

Diese Arbeit entstand im Rahmen der Bemühungen der HS Ravensburg-Weingarten ein eigenes Team in der RoboCup Simulation League aufzustellen. Es ist eine der ersten Arbeiten an der Hochschule, welche sich mit der Machbarkeit von Lernmethoden in der RoboCup Domäne beschäftigt. Angestrebt wird dabei die Beantwortung der Frage, welches Lernverfahren die notwendige Performance besitzt, um in einem später entwickelten RoboCup Soccer Team, zum Einsatz zu kommen. Die Beantwortung der Frage sollte bald erfolgen, da ein Team mit Grundfähigkeiten bereits zur Verfügung steht, dieses jedoch aufgrund des Mangels an intelligenten Fähigkeiten noch nicht eingesetzt werden kann. In Folge dieser und weiterer Arbeiten soll entschieden werden, welches Lernverfahren schlußendlich in dem RoboCup Team zum Einsatz kommen wird. Das ausgewählte Lernverfahren muß dabei die nötige Leistungsfähigkeit haben - sollte aber auch weiterhin so einfach bleiben, daß das Lernen von vielen verschiedenen Fähigkeiten in einem angemessenen Zeitrahmen möglich bleibt.

## 1.1 Zielsetzung der Arbeit

Diese Arbeit hat zum Ziel, das sogenannte „Observation based Reinforcement Learning“ auf Leistungsfähigkeit hin zu testen. Die Leistungsfähigkeit wird, im Rahmen einer RoboCup Soccer Simulation stattfinden, da dies später auch das Einsatzgebiet des Lernverfahrens sein wird. Um das Lernverfahren zu testen, soll ein simulierter Fußballspieler dazu gebracht werden, einen ruhenden Ball, welcher zufällig um den Spieler positioniert wird, in ein vorgegebenes Tor zu schießen. Zu diesem Zweck wird der sogenannte „RoboCup Soccer Server“ eingesetzt, welcher für den Spieler eine physikalisch korrekt simulierte Umwelt zur Verfügung stellt. Der Spieler wird während seiner Lernarbeit nicht beaufsichtig. Er weiß wohin er sich bewegen muß und welche Aktionen er ausführen darf. Anhand dieser Informationen muß er in der Lage sein, selbständig eine Problemlösung zu erarbeiten. Zur Beurteilung der Leistung des Spielers wird herangezogen, wie schnell er lernen kann und welche Güte die von ihm gelernten Lösungen haben. Die Leistung soll mit der des sogenannten „Q-Learnings“ verglichen werden.

## 1.2 Gliederung der Arbeit

Im Einzelnen findet der Leser dieser Diplomarbeit die folgenden Kapitel:

Kapitel 2: *Theoretische Grundlagen*, stellt allgemein das Reinforcement Learning vor und die Markovsche Eigenschaft. Im Speziellen wird auf das später angewandte „Observation based Reinforcement Learning“eingegangen.

Kapitel 3: *Die RoboCup Umgebung*, verschafft dem Leser einen Überblick über die eingesetzten Programme und stellt diese und die Konfiguration des RoboCup Servers kurz vor.

Kapitel 4: *Das Lernverfahren*. Dieses Kapitel verschafft einen Überblick darüber, wie das eingesetzte Lernverfahren angewendet wird. Es wird hier der ursprüngliche Agent beschrieben, welcher für die Diplomarbeit weiterentwickelt wurde. Ferner wird der Aufbau der kompletten Lernsimulation beschrieben sowie die zum Einsatz gekommenen Teile der Simulation. Ein weiterer wichtiger Punkt ist hier, wie das Lernverfahren im Agenten implementiert wurde und welche Fehler dabei auftraten.

Kapitel 5: *Lernverhalten und Leistungsfähigkeit*. Dies stellt den Schwerpunkt der Arbeit dar. Hier werden die vom Agenten gefundenen Lösungsstrategien bewertet. Um herauszufinden, welche Parameter einen Einfluss auf die Leistungsfähigkeit des Agenten haben, werden an dieser Stelle mehrere unterschiedliche „Trainingsszenarien“ miteinander verglichen. Weiterhin wird hier die Leistungsfähigkeit des eingesetzten Lerverfahrens mit der Leistungsfähigkeit des sogenannten „Q-Learnings“ verglichen.

Kapitel 6: *Auswertung und Ausblick*. Hier sollen Verbesserungsvorschläge für das eingesetzte Lernverfahren vorgestellt werden und es soll aufgezeigt werden wie es in der Praxis zur Anwedung kommen könnte.

# 2 Theoretische Grundlagen

In diesem Kapitel sollen die für die Arbeit relevanten Grundlagen aus dem Bereich des Reinforcement Learnings und des Observation based Reinforcement Lernings im Speziellen vorgestellt werden.

## 2.1 Reinforcement Learning

Reinforcement Learning (im weiteren RFL genannt) ist eine Lernmethode, welche ermittelt, welche Aktionen bei einem gegebenen Zustand ausgeführt werden müssen, um ein in Zahlen ausgedrücktes sogenanntes „reward signal“ zu maximieren. RFL verwendet sogenannte Lernagenten, die ein vorgegebenes Ziel besitzen. Sie können die Eingenschaften ihrer Umgebung wahrnehmen und Aktionen auswählen, welche die Umgebung verändern. Dem Lernagenten werden keine Aktionen vorgegeben, stattdessen muß er die Aktionen, welche den größten Reward für ihn liefern, durch Ausprobieren herausfinden. Bei einer Vielzahl der komplexeren Anwendungen von Reinforcement Learning kann es dazu kommen, daß eine Aktion nicht nur den sogenannten „immediate Reward“ beeinflußt, sondern alle nachfolgenden Rewards. Diese beiden Charakteristiken, nämlich die Trial-and-Error Suche nach geeigneten Aktionen und der „delayed Reward“, sind die beiden markantesten Merkmale des Reinforcement Learnings. Reinforcement Learning beschreibt keine Lernmethoden, sondern Lernprobleme. Jedwede Methode welche dazu genutzt werden kann, um solch ein Problem zu lösen, ist eine Reinforcement Learning Methode (dazu später noch mehr).
Das RFL teilt sich in zwei Teile: Zum einen gibt es die sogenannte „Exploitation“, welche dafür sorgt, daß bereits bekannte Aktionen genutzt werden. Zum anderen gibt es die „Exploration“, welche dafür verantwortlich ist, daß neue Aktionen erforscht werden. Eine der wichtigsten Aufgaben des RFL besteht darin, ein gutes Verhältnis zwischen „Exploration“ und „Exploitation“ zu finden. Ein Fehlverhältnis dieser beiden Faktoren kann ein vernünftiges Lernen scheitern lassen.
In den nun folgenden Unterkapiteln soll das RFL näher erklärt werden.

### 2.1.1 Das Agenten-Environment Interface

Grob umrissen verwendet RFL Methoden, um über Interaktionen ein Ziel („Goal“) zu erreichen. Der Lernende, welcher die Entscheidungen trifft, wird hierbei als (Lern-)Agent bezeichnet. Der Agent selber interagiert mit seiner Umgebung (dem „Environment“), welches alle Elemente, die nicht zum Agenten selber gehören, umfasst. Der Agent und das Environment interagieren kontinuierlich miteinander: Der Agent wählt Aktionen aus und das Environment reagiert auf diese Aktionen und liefert einen neuen Zustand an den Agenten zurück.
Der Agent und das Environment interagieren im Zuge eines jeden diskreten Zeitschrittes der Folge $t = 0, 1, 2, 3, ..., n$. Zu jedem Zeitpunkt $t$, bekommt der Agent eine Information über den Zustand des Environments $s_t \in S$, wobei $S$ die Menge der möglichen Zustände ist und $s_t$ der Zustand s in einem Zeitpunkt $t$. Anhand dieser Informationen wählt der Agent eine Aktion $a_t \in A(s_t)$ aus, wobei $A(s_t)$ die möglichen Aktionen im momentanen Zustand $s_t$ sind. Einen Zustand später, als Folge der ausgeführten Aktion, bekommt der Agent einen numerischen „Reward“ $r_{t+1} \in R$ und erreicht einen neuen Zustand $s_{t+1}$. Der Reward $r_{t+1}$ wird vergeben, wenn in Zustand $s_t$ die Aktion $a_t$ ausgeführt wird. Es gilt, daß $R$ die Menge aller möglichen Rewards ist. Zu jedem Zeitpunkt versucht der Agent, einem Zustand eine Aktion zuzuweisen. Die Vorschrift für die Zuweisung wir „Policy“ des Agenten genannt und wird mit $\pi_t$ bezeichnet. Es gilt, daß $\pi_t(s, a)$ die Wahrscheinlichkeit für das eintreten von $s_t = s$ unter der Vorraussetzung von $a_t = a$ ist. Ein RFL Verfahren beschreibt also wie der Agent, aufgrund seiner gemachten Erfahrungen, seine Policy ändert. Das Ziel dabei ist, den Gesamtbetrag des Rewards zu maximieren.

### 2.1.2 Goals und Rewards

Beim RFL wird der Agent in seinem Bestreben sein Ziel („Goal“) zu erreichen gesteuert. Zu diesem Zweck bekommt der Agent bei jedem Zustandsübergang vom Environment einen Reward. Zu jedem Zeitpunkt ist der Reward stets $r_t \in R$. Man kann also sagen, daß das Ziel des Agenten ist, den Reward zu maximieren. Gemeint ist jedoch nicht der Reward zu jedem Zeitpunkt $t$ („immediate reward“), sondern der Reward über einen langen Zeitraum, der sogenannte „delayed reward“.

### 2.1.3 Returnwerte

Bisher wurde recht ungenau formuliert und es wurde gesagt, daß das Ziel des Agenten ist, den Reward über lange Sicht hin zu maximieren. Aber wie läßt sich diese Absicht formalisieren? Was soll maximiert werden, wenn die Folge der Rewards nach jedem Zeitschritt $t$ mit $r_{t+1}, r_{t+2}, r_{t+3}, ..., r_{t+n}$ angegeben wird? Normalerweise wird versucht, den erwarteten Returnwert zu maximieren, wobei dieser Returnwert $R_t$ durch eine Funktion der Rewardsequenzen angegeben wird. Im einfachsten Falle kann diese Funktion einfach die Summe alle Rewards sein:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{1}$$

Wobei $T$ dem letzten Zeitschritt entspricht. Dieser Ansatz ist sinnvoll bei allen Anwendungen, bei denen es ein vorhersehbares Ende gibt. Hierbei wird das Lernen in Sequenzen aufgeteilt, welche man Episoden nennt. Als Beispiel kann man sich z.B. eine Serie von Labyrinth-Druchläufen vorstellen, wobei jeder Durchlauf durch das Labyrinth einer Episode entspricht (es wird später auch von Lernepisoden gesprochen). Jede Episode endet in einem Endzustand (auch „Terminal State“ genannt), und es folgt eine Zurücksetzung zu einem wohldefinierten Startzustand. Dieser Startzustand wird aus einer Menge von möglichen Startzuständen ausgewählt. Bei episodischen Lernaufgaben, wie hier beschrieben, wird zwischen nicht terminalen Zuständen $S$ oder auch $S^-$ und terminalen Zuständen $S^+$ (positiven Endzuständen) unterschieden.

Neben den episodischen Aufgaben gibt es die nicht episodischen Aufgaben. Hier läßt sich das Lernen nicht in einzelne Episoden aufteilen, sondern es gestaltet sich als eine fortlaufende Aufgabe. Solche Aufgaben bezeichnet man als „continuing tasks“. Die Returnfunktion welche unter (1) angegeben wurde, ist in solchen Fällen problematisch, da hier gilt: $T = \infty$. Dies führt jedoch dazu, daß der Return, welcher eigentlich maximiert werden sollte, ins Unendliche laufen kann.

Um diesem Problem aus dem Wege zu gehen, benutzt man das sogenannte „discounting“. Ein Agent, der das Discounting anwendet, versucht seine Aktionen so auszuwählen, daß die Summe der abgeschwächten Rewards, die er in der Zukunft erhalten wird, maximiert wird. Die Formel zur

Berechnung des abgeschwächten Returns lautet:

$$R_r = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (2)$$

Der Parameter $\gamma$, der den Bereich $0 \leq \gamma \leq 1$ einnehmen kann, wird „Discount Factor“ genannt. Der „Discount Factor“ bestimmt den momentanen Wert der zukünftigen Rewards. Ein Reward, welcher $k$ Zeitschritte in der Zukunft erhalten wird, wird mit $\gamma^{k-1}$ seines eigentlichen Wertes bewertet. Wenn $\gamma < 1$ ist, ist die Summe der Returns endlich, solange die Folge der Rewards $r_k$ begrenzt ist. Sollte hingegen $\gamma = 0$ sein, so wird der Agent nur die unmittelbaren Rewards maximieren. In diesem Fall besitzt der Agent die Aufgabe, die Aktion $a_t$ so zu wählen, daß nur $r_{t+1}$ maximiert wird. Wenn die Aktion des Agenten nur den unmittelbaren Reward beeinflusst, kann der Return maximiert werden, indem jeder unmittelbare Reward maximiert wird.

### 2.1.4 Die Markovsche Eigenschaft

Es soll nun eine Eigenschaft des Zustandes des Environments beschrieben werden, welche die Markov Eigenschaft genannt wird. Was dazu benötigt wird ist idealerweise ein Zustandssignal, welches alle vergangenen Zustände in einer Art und Weise zusammenfaßt, daß keine relevanten Informationen verloren gehen. Dies erfordert meistens mehr als nur die momentane Wahrnehmung, aber niemals mehr als den Verlauf aller bisherigen Wahrnehmungen. Ein Zustandssignal, welches dem Anspruch genügt, alle relevanten Informationen zu beinhalten, hat die sogenannte Markovsche Eigenschaft (oder ist ein Markov Zustand). Als Beispiel kann man sich ein Damespiel denken: Die aktuelle Position der Spielfiguren wäre ein Markov Zustand, da sie alle relevanten Informationen über die Zustände beinhaltet, die zum aktuellen Zustand geführt haben. Es sind zwar viele Informationen über den Zustandsverlauf verloren gegangen, jedoch sind alle wichtigen Informationen erhalten geblieben.
Um die Markovsche Eigenschaft mathematisch möglichst einfach zu beschreiben, geht man davon aus, daß es eine begrenzte Anzahl von Zuständen und Rewardwerten gibt. Man stelle sich vor, wie das Environment sich zu einem Zeitpunkt $t + 1$ verhält, wenn zu einem früheren Zeitpunkt $t$ die Aktion $a_t$ ausgeführt wurde. Im normalen kausalen Fall wird die Aktion

ausgewählt unter der Betrachtung aller bisheriger Ereignisse. In diesem Fall kann die Systemdynamik nur anhand einer kompletten Wahrscheinlichkeitsverteilung angegeben werden:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\} \quad (3)$$

für alle $s', r$ und alle möglichen Werte der vergangenen Ereignisse: $s_t, a_t, r_t, ..., r_1, s_0, a_0$. Hat das Zustandssignal jedoch die Markovsche Eigenschaft, hängt die Auswirkung im Environment im Zeitschritt $t + 1$ nur vom Zustand und der ausgewählten Aktion in $t$ ab. In diesem Fall kann die Systemdynamik angegeben werden mit:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} \quad (4)$$

für alle $s', r, s_t$ und $a_t$. Ein Zustandssignal besitzt die Markovsche Eigenschaft, wenn gilt (3) gleich (4) für alle $s', r$ und für die vorherigen $s_t, a_t, r_t, ..., r_1, s_0, a_0$ ist. In diesem Fall hat sowohl das Environment als auch die Lernaufgabe selber eine Markovsche Eigenschaft.

### 2.1.5 Der Markovsche Entscheidungsprozess

Eine RFL Aufgabe, die die Markovsche Eigenschaft besitzt, kann durch einen „Markov Decision process“ (Markovscher Entscheidungsprozess, MDP) gelöst werden. Wenn zusätzlich Zustands- und Aktionsraum endlich sind, so handelt es sich um einen „finite Markov decision process“ (finiter Markovscher Entscheidungsprozess, finite MDP).
Ein bestimmter finiter MDP wird definiert durch seinen Zustand, seine Menge der Aktionen und die „ein Schritt Dynamik“ des Environments. Die Wahrscheinlichkeit eines jeden nächsten Zustandes $s'$ ist mit gegebener Aktion $a$ und gegebenem Zustand $s$:

$$P^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \quad (5)$$

Diese Mengen werden „transition probabilites“ (Übergangswahrscheinlichkeiten) genannt. Gleichermaßen gilt für eine gegebene Aktion $a$, gegebenem Zustand $s$ und mit dem gegebenen nächsten Zustand $s'$ für den Wert des erwarteten Rewards:

$$R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \quad (6)$$

Diese beiden Mengen, $P^a_{ss'}$ und $R^a_{ss'}$ geben die wichtigsten Aspekte einer finiten MDP Dynamik an. $E$ ist eine Entscheidungsmethode des Environments, welche den Reward unter den gegebenen Vorraussetzungen bestimmt.
Quelle: Frei nach [SBarto]

### 2.1.6 Value Functions

Da Value Functions nicht für das Lernverfahren gebraucht werden, welches später eingesetzt wird, sei der interessierte Leser an dieser Stelle an [Harmon] verwiesen.

## 2.2 Observation based Reinforcement Learning

Es soll nun herausgearbeitet werden, wo die Probleme des RFLs liegen und wie sie durch das sogenannte Observation based Reinforcement Learning teilweise beseitigt werden können.

### 2.2.1 Generalisierung beim Reinforcement Learning

In vielen RFL Lernaufgaben gibt es oft das Problem, daß es einen sehr großen Zustandsraum gibt. Da die großen Zustandsräume oft zu Rechenzeitproblemen führen, läßt dies die Forderung nach einer Generalisierung laut werden. Der Vorteil einer Generalisierung liegt jedoch nicht nur in einer möglichen Verkleinerung des Zustandraumes, sondern vielmehr in der Tatsache, daß eine gelernte Policy nicht nur in der Trainingsumgebung einsetzbar ist, sondern optimalerweise in allen ähnlichen Umgebungen. Um diesen Sachverhalt zu verdeutlichen, wird jetzt folgendes Beispiel betrachtet:
Gegeben sei ein Array, welches sortiert werden soll. Dieses Array hat die Länge $l$ mit Integerzahlen aus dem Bereich $I := \{1, ..., n\}$. Ein Zustand ist ein l-Tupel von Elementen aus $I$. Die Aufgabe besteht darin, das Array in aufsteigender Reihenfolge zu sortieren. Die möglichen Aktionen beinhalten das Tauschen bestimmter Elemente. Mit festem $l$ und $n$ ist diese Lernaufgabe ein MDP. Der Reward wird auf einen Betrag $\geq 0$ festgelegt, wenn das Array sortiert ist, andernfalls ist der Reward $-1$. Die optimale Policy wird also versuchen, mit möglichst wenigen Zügen, das Array zu sortieren.
Ein klassisches tabellengestütztes RFL Verfahren würde zu jedem Zustand

eine Aktion speichern und würde somit jede mögliche Vertauschung im Array selber speichern. Allem Anschein nach ist dies jedoch keine gute Strategie, da diese Policy nur in der Lage wäre, Arrays mit gleichem $l$ und $n$ zu sortieren. Würde beispielweise $l$ um eins erhöht, würde die Policy versagen, da sich der Zustandsraum vergrößert hätte, die Policy jedoch auf den vorherigen Wert von $l$ spezialisiert wäre.
Um zu zeigen, welche Vorteile die Generalisierung bietet, sollte nicht nur ein spezielles Trainingsbeispiel betrachtet werden, sondern stattdessen eine ganze Klasse von MDPs. Für das angegebene Sortierungsbeispiel, wäre solch eine Klasse z.B. alle möglichen Arrays aller Größen mit allen möglichen Integerzahlen zu sortieren.
Es gibt noch einen anderen Generalisierungsansatz im RFL, die sogenannte interne Generalisierung. Es beschreibt den Wunsch, daß ein Agent sich in ähnlichen Situationen ähnlich verhält, z.B. soll der Agent im Stande sein, aus einer einzigen gelernten Dribble Policy unabhängig von der Position überall dribbeln zu können. Es soll also keine Unterscheidung geben, ob der Agent an der einen Stelle dribbelt oder an einer anderen. Die Motivation hinter so einer Generalisierung ist, daß das Lernen beschleunigt und verbessert wird.
Obwohl die Idee der internen Generalisierung durchaus Bewandnis in der Praxis hat, wollen wir im weiteren mit der Idee der Generalisierung über MDP Klassen weiterarbeiten.

### 2.2.2 Konsequenzen der Generalisierung

Wenn man eine einzige Policy als Modell für eine ganze Klasse von MDPs benutzen will, müssen die MDPs bestimmte Randbedingungen erfüllen, um ein konsistentes Interface zwischen der Problembeschreibung und der Policy zu ermöglichen. Um ein solches Interface zu ermöglichen, braucht man zum einen eine Menge an Aktionen, welche für alle möglichen Elemente der Problemklasse qualifiziert sind und zum anderen eine passende Darstellung des momentanen Zustandes. Da der Zustandsraum für die Policies verborgen bleiben sollte, wird zu diesem Zweck eine Menge von „Observationen“ eingeführt, welche für die gesamte Problemklasse gültig sind. Zu diesem Zweck braucht man eine Abbildungsfunktion, welche Zustände auf Observationen abbildet.
In obigen Beispielprogramm muß die Struktur des Problems in einer Art und Weise geändert werden, so daß durch die Generalisierung die Größen-

angabe $l$ und die Anzahl der möglichen Einträge $n$ wegfallen. Dies kann erreicht werden, indem man einen virtuellen Zeiger einführt, der auf einen Eintrag des Arrays zeigt. Die möglichen Aktionen wären, den Zeiger nach links oder rechts zu bewegen, oder zum linken oder rechten Rand des Arrays. Weiterhin sollte es möglich sein, die Zahl, auf welche der Zeiger zeigt, mit einem der beiden benachbarten Felder zu tauschen. Wie sich hieran unschwer erkennen läßt, sind die Observationen nur auf die Zeigerposition bezogen und somit völlig unabhängig von der Arraygröße. Als Observation wird folgendes definiert: Der Eintrag, auf welchen der Zeiger zeigt, ist größer oder kleiner als einer der benachbarten. Weiterhin ist es wichtig zu wissen, ob der Zeiger sich am linken oder rechten Rand befindet. Durch diese Einteilung wird erreicht, daß sowohl die Aktionen als auch die Observationen unabhängig von der Größe des Arrays sind und somit bekommt man eine Policy für die gesamte Problemklasse. Die so erhaltene Policy gilt nicht nur auf dem ursprünglichen Trainingsszenario sondern bei einer breiten Auswahl an Szenarien.
Ein Vorteil dieser Generalisierung ist ebenso, daß das Lernen mit diesem Algorithmus schneller von statten geht, da die Anzahl der Parameter minimiert wurde. Durch die Möglichkeit, daß man die Policies in kleinen Domänen (Trainingsszenarien) lernen kann und sie danach auf große Domänen übertragen kann, entsteht ein weiterer sehr großer Vorteil. Die Lernkomplexität nimmt ab.

### 2.2.3 Observation based Reinforcement Learning Verfahren

In diesem Abschnitt soll erläutert werden, aus welchen Teilen das Observation based Reinforcement Learning besteht und wie dieses Lernverfahren abläuft. Der Ablauf selber wird an dieser Stelle zunächst theoretisch erklärt und später - wenn er dann zum Einsatz kommt - wird er noch einmal ausführlich besprochen.

### 2.2.4 Elemente eines Observation based RFL Algorithmuses

In diesem Abschnitt soll untersucht werden, welche Möglichkeiten es für das Observation based RFL gibt, um zu einer brauchbaren Policy zu konvergieren. Es wird nach einem Algorithmus gesucht, welcher eine Policy in einer konstruktiven Art und Weise erstellt, jedoch keine blinde Suche ausführt und welcher nach einer falschen Entscheidung nicht hängen bleibt.

Zu diesem Zwecke werden beide Extreme betrachtet und am Ende eine mögliche Mischung dieser beiden vorgestellt. Die folgende Analyse ist nur gültig unter der Prämisse, daß es sich bei den untersuchten Lernproblemen um deterministische MDPs mit einem finiten Zustands- und Aktionsraum handelt. Dem Agenten sei die Fähigkeit gegeben, zwischen Observations zu unterscheiden, welche einen Zielzustand repräsentieren und solchen, die es nicht tun.

**2.2.4.1 Blind Search approach** Eine Art und Weise eine korrekte Policy zu ermitteln, wäre es, den kompletten Zustandsraum abzusuchen. Dies kann erreicht werden, indem z.B. eine „Monte Carlo“ Methode angewandt wird. Diese Methode würde zufällig erzeugte Policies ausprobieren und entscheiden, ob es sich bei ihnen um eine korrekte Policy handelt oder nicht. Jedoch muß man ein modellfreies Lernen annehmen (d.h. die Übergangsfunktion der Zustände ist unbekannt) und daher kann man eine Policy auf theoretischer Basis nicht auf Plausibilität prüfen. Es steht jedoch die Möglichkeit offen, die Policy durch Testen auf Plausibilität zu prüfen. Dies kann erreicht werden, indem man die Policy anwendet und darauf wartet, ob ein Zielzustand erreicht wird oder nicht. Ist die Policy nicht korrekt, gibt es mindestens einen Startzustand von dem aus die Policy nicht in der Lage ist in einen Endzustand zu führen. In diesem Fall werden einige Zustände innerhalb der Policy zyklisch und die Trainingsepisode kann nicht mehr beendet werden (Beispiel: Zustand 1 führt und Zustand 2 und dieser zurück in Zustand 1).
Man kann folgende Methode zum Aufdecken einer nicht korrekten Policy verwenden: $N$ sei die Anzahl der Zustände. Es wird eine Trainingsepisode gestartet und die Anzahl der Zustandsübergänge gezählt. Wenn nach $N$ Zustandsübergängen kein Endzustand erreicht wurde, ist bekannt, daß es sich bei der aktuellen Policy um eine nicht korrekte Policy handelt. Wenn dies der Fall ist, wird die nicht korrekte Policy durch eine neue zufällige Policy ersetzt. Es wird ein zufälliger Startzustand ausgewählt und der Test beginnt von vorne. Sobald eine korrekte Policy gefunden wurde, wird diese nicht mehr ersetzt.
Wenn auch der „blind search algorithm“ dazu in der Lage ist ab und zu eine korrekte Policy zu finden, so bleibt dennoch das Problem, daß dies unter Umständen lange dauern kann. Ein weiteres Problem kann auch sein,

daß keinerlei Informationen über eine gelöschte Policy beibehalten werden, somit kann es vorkommen, daß gleiche Policies mehrere Male getestet werden.

**2.2.4.2 Dynamic programming approach** Das zweite Extrem eines Algorithmus ist, daß niemals Informationen vergessen werden und die Policy in einer durchschaubaren Art und Weise gelernt wird.
Wenn jede Observation genau einem Zustand zugeordnet ist, kann man eine korrekte Policy mit folgendem Algorithmus berechnen: $\omega$ bezeichnet eine Observation und $P$ eine Policy. $P(\omega) = *$ besagt, daß noch keine Aktion ausgewählt wurde. Die Anfangswerte sind also:

$$P(\omega) \leftarrow \begin{cases} * & falls\ \omega\ kein\ Zielzustand\ ist \\ a & falls\ \omega\ ein\ Zielzustand\ ist \end{cases}$$

wobei $a$ frei wählbar ist. Nach einem Zustandsübergang mit der ausgeführten Aktion $a_t$ von $\omega_t$ nach $\omega_{t+1}$ wird die Policy aktualisiert:

$$P(\omega_t) \leftarrow \begin{cases} * & falls\ P(\omega_{t+1}) = * \\ a_t & anderenfalls \end{cases}$$

Dieser Algorithmus benutzt eine „dynamic programming“ Technik, um einen Informationsfluss vom Endzustand aus zu initiieren. Das Verfahren konvergiert bei deterministischen MDPs, bei denen alle Observationen nicht mehr als einen Zustand beschreiben.
Unglücklicherweise kann der Standardalgorithmus fehlschlagen, wenn eine Observation mehr als einen Zustand beschreibt. Der Algorithmus kann eine bestimmte Aktion in Zustand $A$ auswählen und kann sie nicht mehr ändern, auch wenn sie in einem Zustand $B$, welcher durch die gleiche Observation repräsentiert wird, fehlschlägt.
Obwohl der „dynamic programming“ Ansatz dem „blind search algorithm“ bei weitem überlegen ist, in der Hinsicht, daß er bereits gemachte Erfahrung nicht verwirft, läuft er dennoch Gefahr, in nicht korrekten Policies hängen zu bleiben. Es wird also eine Möglichkeit benötigt die Informationen zurückzuverfolgen, was dem Algorithmus ermöglicht, bestimmte bereits gesammelte Informationen zu vergessen.

### 2.2.5 Der Ablauf des Observation based Learning

Durch das Verbinden der beiden oben genannten Verfahren, kann man ein exemplarisches Observation based Verfahren realisieren.
Der Algorithmus besteht aus einer momentanen Policyfunktion $P$ wie in dem „dynamic programming“ (DP) Algorithmus, einem Schrittzählen $c$, einer Markierungsfunktion $Z$, die jeden Zustand als bereits besucht markieren kann und einer Konstanten $L$, welche größer als die Anzahl der Zustände sein sollte. Diese Konstante muß gegeben sein (bis ein guter Wert gefunden wird, ist diese Konstante meist willkürlich gesetzt). Die Initialwerte sind:

- $P$ wird initialisiert wie im DP Algorithmus
- Alle Observations in $Z$ werden als „nicht besucht“ initialisiert.
- Initialisiere $c \leftarrow 0$

In einem Zielzustand wird ein neuer Startzustand zufällig ausgewählt, der Zähler $c$ wird auf $0$ zurückgesetzt und die Markierungsfunktion $Z$ wird auf „nicht besucht“ gesetzt.
In nicht Endzuständen wird der Nachfolgezustand berechnet und die nachfolgende Observation $\omega'$ zurückgegeben. Es können dabei 2 Fälle eintreten:

- 1. $c + 1 \geq L$ (reset branch): $P$ wird zurückgesetzt auf $*$ für alle Zustände, welche durch $Z$ markiert sind. Des weiteren werden alle $Z$ Werte auf „nicht besucht“ zurückgesetzt und $c \leftarrow 0$
- 2. $c + 1 < L$ (constructive branch): Die Policy wird gemäss des DP-Verfahrens aktualisiert, $c$ wird um eins erhöht und der Zustand wird als besucht markiert.

Der Algorithmus verbindet die Vorteile vom DP Verfahren und des Suchverfahrens. Das hier vorgestellte Verfahren, lernt eine Policy ohne eine „cost-to-go“ [Cost] Funktion zu berechnen. Unter bestimmten Voraussetzungen zur Auswahl der Aktionen und der Startzustände, ist es möglich, die Konvergenz zu einer korrekten Policy, sollte diese existieren, zu beweisen. Dieser Beweis soll jedoch nicht an dieser Stelle geführt werden [Rmiller] / [Rein].

# 3 Die RoboCup Umgebung

Die 1995 ins Leben gerufene RoboCup-Initiative hat sich zur Aufgabe gemacht, die Forschung in verschiedenen Bereichen der Robotik und künstlichen Intelligenz durch die Formulierung eines ehrgeizigen Ziels voranzutreiben: Bis zum Jahr 2050 sollen Roboter gegen Menschen Fußball spielen können. Die Herausforderung, welche in dieser Aussage liegt, hat viele Wissenschaftler und Studenten dazu animiert, sich in RoboCup einzubringen und das Projekt voranzutreiben. Auch nach außen hin wurde durch den RoboCup starkes Interesse geweckt, da Robotik immer noch ein außergewöhnliches Thema ist und Menschen auf eine Art und Weise begeistert, wie es vielen anderen Wissenschaften nicht möglich ist. Die Ziele des RoboCup können an folgendem Zitat sehr gut abgelesen werden:

> By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA against the winner of the most recent World cup [Robo01]

## 3.1 Die Simulationsumgebung

Die RoboCup Simulation League basiert auf einem RoboCup Simulator, welcher sich RoboCup Server (oder auch Soccer Server) nennt. Es handelt sich hierbei um ein System, welches die physikalischen Eigenschaften einer Fußballsimulation emuliert [RoCu01]. Die Spiele werden durch den RoboCup Monitor visualisiert, welcher später noch genauer beschrieben wird. Der Soccer Server wurde entwickelt, um eine virtuelle Umgebung zur Verfügung zu stellen, in der virtuelle Spieleragenten gegeneinander antreten können. Zu den Spieleragenten sei gesagt, daß diese jeweils eigenständige Programme sein müssen, d.h. sie dürfen nicht über gemeinsame Daten oder ähnliches verfügen. Sie müssen autonom arbeiten.

## 3.2 Der RoboCup Monitor

Es soll nun sehr kurz der RoboCup Monitor vorgestellt werden.

### 3.2.1 Einführung

Der RoboCup Monitor bietet ein visuelles Interface, welches erlaubt, die laufenden Spiele anzuschauen und den Fortlauf der Spiele zu kontrollieren. In Verbindung mit dem sogenannten „Logplayer“ ist der Monitor sogar dazu in der Lage, bereits beendete Spiele zu wiederholen. Dies bietet eine gute Möglichkeit, um das Spielverhalten der Spieler zu analysieren und gegebenenfalls zu debuggen.

### 3.2.2 Kommunikation zwischen Server und Monitor

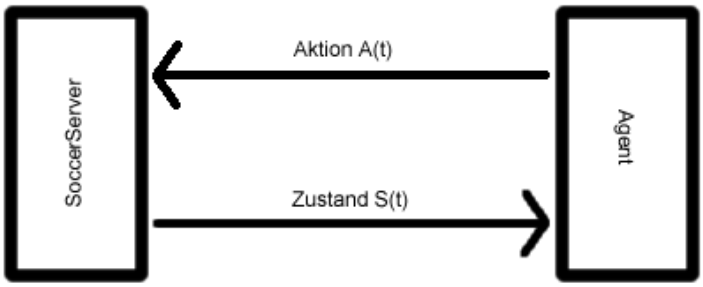
Der Soccerserver und der Monitor kommunizieren mit Hilfe von UDP über Port 6000. Wenn sich der Monitor zum Server verbindet, sendet der Server jeden Zyklus Informationen seines Environments zum Monitor. Der Server unterscheidet normalerweise zwischen zwei verwendeten Formaten (version 1 und 2). Der Server entscheidet anhand der Kommunikationsinitialisierung zwischen Server und Monitor, welches Format zu benutzen ist.

## 3.3 Der RoboCup Soccerserver

Der Soccer Server ist ein System, welches es mehreren Teams erlaubt, gegeneinander im Fußball anzutreten. Es gibt keine Regelung darüber, wie die Teams zu programmieren sind. Die einzige Vorgabe ist, daß die Agenten über UDP mit dem Server kommunizieren können. Dies ist der Fall, da jegliche Kommunikation zwischen dem Server, allen Klienten und dem Monitor über UDP Sockets stattfindet. Jeder Spieler ist ein eigenständiger Prozess und kommuniziert mit dem Server über eine eigene UDP Verbindung. Nachdem ein Spieler einmal eine Verbindung zum Server aufgebaut hat, werden alle Informationen über diesen Port gesendet. Ein Team kann aus bis zu 12 Teilnehmer bestehen, davon 11 Spieler (10 Feldspieler und 1 Torwart) und 1 Trainer. Die Spieler senden Anfragen an den Server mit Informationen über die Aktionen die sie ausführen wollen (z.B. den Ball schießen, sich selber drehen, laufen etc.). Der Server empfängt diese Nachrichten, arbeitet die Anfragen ab und aktualisiert sein Environment anhand dieser Anfragen. Zusätzlich versieht der Server die Klienten mit sensorischen Informationen (visuellle Daten bezogen auf die Position des Agenten auf dem Spielfeld, welche Objekte er von dort aus wahrnehmen kann und

Daten über die Ressourcen des Agenten wie z.B. Stamina (Ausdauer), Geschwindigkeit oder ähnliches). Es ist wichtig zu erwähnen, daß der Server

Abbildung 1: SoccerServer Informationsaustausch




als Echtzeitsystem, mit diskreten Zeitintervallen (Zyklen), realisiert ist. Jeder Zyklus hat eine wohldefinierte Zeitdauer. Aktionen, welche in einem bestimmten Zyklus ausgeführt werden sollen ( Aktionen eines Agenten ), müssen beim Server im richtigen Intervall ankommen. Folglich hat eine schlechte Performance von Spielern, wodurch sie eine Aktion verpassen können, einen entscheidenen Einfluß auf die Leistung der Mannschaft. Als Beispiel kann man sich vorstellen: Spieler 1, welcher den Ball führt, will den Ball an Spieler 2 passen, da dieser zu 100% ein Tor schiessen könnte. Spieler 1 passt zu Spieler 2 - dieser führt jedoch den Torschuss, aufgrund seiner schlechten Performance nicht aus, stattdessen wird ihm der Ball von einem gegnerischen Spieler abgenommen.

### 3.3.1 Klient- Server Kommunikation

Wie bereits erläutert, werden die Informationen zwischen Klienten und dem Server über UDP ausgetauscht. Die Befehle werden im Klartext übermittelt. Die wichtigen Befehle für das benutzte Trainingsszenario sind zum

einen der Turnbefehl und zum anderen der Kickbefehl. Es ist zu beachten, daß die Befehle genau in dem folgend angegebenen Format an den Server zu schicken sind. Sollten sie dieser Form nicht entsprechen, wird vom Server eine Fehlermeldung zurückgegeben.

- $(turn < Winkel\ \delta >)$: Der Winkel liegt hierbei zwischen $-180$ und $+180$
- $(kick < Power >< Winkel\ \epsilon >)$: Die Power liegt zwischen $-100$ und $+100$. Der Winkel liegt zwischen $-180$ und $+180$.

Die oben genannten Maximal- und Minimalwerte können in der Serverkonfiguration geändert werden, jedoch wäre eine Änderung nicht angebracht, da der Lernagent, welcher später vorgestellt wird, genau auf diesen Einstellungen beruht. Es sollte bedacht werden, daß diese Variablen stetig sind und somit eine Vergößerung oder eine Verkleinerung nur wenig Nutzen haben würde.

### 3.3.2 Die Serverkonfiguration

Die Serverkonfiguration findet über eine Konfigurationsdatei statt, welche pro Benutzer unter */.rcssserver/server.conf* abgelegt wird. Aus diesen Konfigurationsdateien sollen die wichtigsten Punke für das benutzte Trainingsszenario herausgegriffen werden.

- *server::coach_port = 6001* Gibt an, auf welchem UDP Port sich ein möglicher Coach verbindet
- *server::olcoach_port = 6002* Gibt an, auf welchem UDP Port sich ein möglicher Onlinecoach verbindet.
- *server::recv_step = 10* Gibt an, in welchen Zeitabständen der Server Befehle von den Agenten entgegen nimmt. Die Zeiteinheit wird in Millisekunden [ms] angegeben.
- *server::send_step = 10* Gibt an, in welchen Zeitabständen der Server visuelle Informationen an die Klienten sendet. Die Zeiteinheit wird in Millisekunden [ms] angegeben.
- *server::send_vi_step = 10* Gibt an, in welchen Zeitabständen der Server Informationen an den Coach sendet. Die Zeiteinheit wird in Millisekunden [ms] angegeben.

- *server::simulator_step = 100* Zeitintervall der Serverzyklen, wird in Millisekunden angegeben [ms].
- *server::coach = 1* Gibt an, ob ein Coach oder ein Trainer vorhanden ist - die genaue Beschreibung der Unterschiede erfolgt später.
- *server::fullstate_l = 1* Beschreibt, ob der Server genaue Daten über alle Objekte der linken Seite (oder an das linke Team - genaue Informationen hierüber sind leider nicht vorhanden) an die Agenten senden soll.
- *server::fullstate_r = 1* Analog wie oben nur bezogen auf die rechte Seite.
- *server::wind_random = 0* Gibt an, ob es einen zufälligen Windfaktor gibt.
- *server::ball_rand = 0* Gibt an, ob es eine zufällige Ballbewegung gibt.
- *server::kick_rand = 0* Gibt an, ob es eine zufällige Bewegung beim Schiessen des Balles gibt.
- *server::player_rand = 0* Gibt an, ob der Spieler selbst zufälligen Bewegungen unterworfen ist.
- *server::visible_distance = 100* Gibt die Sichtweite der Spieler an. Wird in Metern angegeben.
- *server::visible_angle = 360* Gibt den Sichtwinkel der Spieler an.

# 4 Das Lernverfahren

In diesem Kapitel soll das benutzte Trainingsszenario beschrieben werden und wie das Lernverfahren beim Agenten implementiert wurde. Das Trainingsszenario selber besteht aus 4 Teilen. Es sind dies der Server, der Monitor, der Lernagent und der Trainer. Für Server und Monitor wurden bereits die wichtigen Aspekte beschrieben und so soll an dieser Stelle von einer nochmaligen näheren Erläuterung abgesehen werden. Die beiden wichtigsten Elemente für das Lernen sind der Agent und der Trainer. Der Agent wird für das komplette Lernverfahren benötigt - mit ihm können theoretisch alle denkbaren Spielzüge gelernt werden. Er wird dabei durch den Trainer unterstützt indem der Trainer ein Trainingsszenario immer und immer wiederholt und dementsprechend die Objekte auf dem Spielfeld hin und her verschiebt. Zunächst sollen hier die Ziele des Agenten formuliert werden und später die Aufgaben des Trainers, um den Agenten bei seiner Aufgabe zu unterstützen.

Ziel ist es, daß der Agent mit Hilfe des Observation based RLFs lernt, einen Ball zu kicken. Dabei steht der Agent immer 40 Meter rechts des Spielfeldmittelpunktes in Richtung des rechten Tores. Er hat keine Ausgangsgeschwindigkeit, nur seine Körperrichtung wird am Anfang einer jeden Trainingsepisode zufällig gesetzt. Der Ball wird zufällig in der sogenannten „kickable area“ des Agenten positioniert - dieses Gebiet ist ein Kreis um den Agenten mit dem Radius von $1,085m$. Um eine Kollision mit dem Agenten zu vermeiden, was eine Verschiebung des Agenten oder des Balles von Seiten des Servers zur Folge hätte, ist es nicht gestattet, den Ball näher als $0,368m$ an den Agenten zu positionieren. Diese Grenze von $0,386m$ wird auch „collision area“ genannt. Aufgabe des Agenten ist es nun mit Hilfe einer vorgegebenen Menge von Aktionen die nötigen Aktionen auszuwählen, welche den Agenten von einem zufällig ausgewählten Startzustand in einen (oder auch mehrere) definierten Endzustand bringt.

Die validen Startzustände sind alle Zustände in denen der Agent auf seiner festen Position steht (wie oben bereits beschrieben) und der Ball sich in dem Raum zwischen der äußersten Grenze der „kickable area“ und der äußersten Grenze der „collision area“ befindet. Der Ball darf dabei eine Geschwindigkeit von $0$ oder höher haben. Die eingesetzten Trainingsszenarien wurden darauf beschränkt, daß sich der Ball am Anfang nicht bewegt.

Gültige Endzustände, sind alle Zustände, in denen der Ball sich mit einer bestimmten Endgeschwindigkeit (die Minimalgeschwindigkeit beträgt hier $0,5m/s$. Diese Minimalgeschwindigkeit ist allen Trainingsszenarien gemein.) in Richtung eines bestimmten Objektes bewegt. In diesen Trainingsszenarien ist das Objekt der Tormittelpunkt. Der Ball darf dabei 7 Meter (Standardeinstellung, jedoch gibt es Szenarien, in denen dieser Parameter anders gesetzt ist) in beide Richtungen vom Tormittelpunkt abweichen.

## 4.1 Anwendung des Observation based RFL beim Agenten

Es wird nun das bereits beschriebene Observation based RFL auf das Ziel des Lernagenten übertragen. Hauptsächlich müssen also beim Environment die Observationen und beim Agenten die Aktionen beschrieben werden.

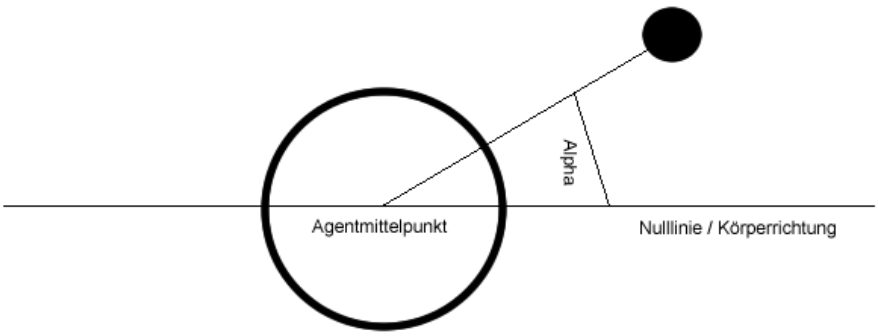
### 4.1.1 Daten des Environments

Ins Environment müssen alle für das Lernproblem relevanten Daten einfließen, um eine korrekte Policy zu erhalten. Wie bereits erwähnt soll gelernt werden, einen Ball ins Tor zu schießen. Auf Seiten des Agenten sollte festgehalten werden, daß der Agent eine Körperrichtung und eine Blickrichtung hat - die Blickrichtung ist für unser Lernproblem aber nicht relevant. Es sind folgende Informationen für das Lernproblem relevant:

- Der Winkel $\alpha$ zwischen der Körperrichtung des Agenten und dem Ball.
- Der Winkel $\beta$ zwischen Körper und Zielwinkel (Zielwinkel ist in diesem Fall der Winkel zum anzuspielenden Objekt, welches hier der Tormittelpunkt ist)
- Der Winkel $\gamma$, welcher die Bewegungsrichtung des Balles angibt.
- Die Ballgeschwindigkeit $v_{ball}$

Mit diesen 4 Werten läßt sich der Aufenthaltsort des Balles relativ zum Spieler und zum Objekt, was angespielt werden soll, exakt beschreiben - mehr Informationen benötigt der Agent nicht, um einen Torschuss auszuführen. Da diese Werte jedoch alle stetig sind, müssen sie diskretisiert werden. Es

Abbildung 2: Veranschaulichung des Ballwinkels

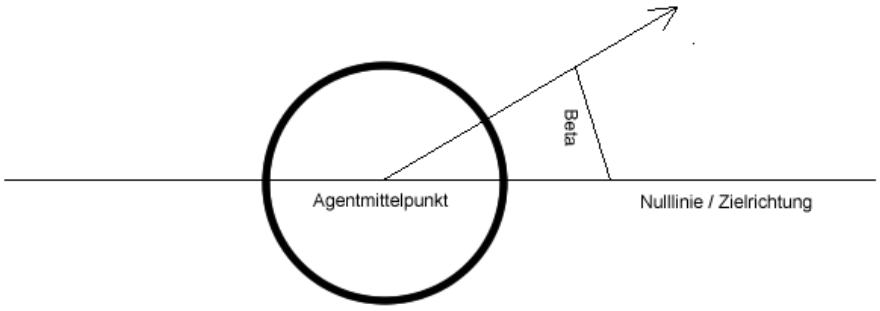



werden somit Observationen eingeführt. Um herauszufinden, wie die einzelnen Werte diskretisiert werden müssen damit man eine gute Performance beim Lernen erzielt, kann über Probieren oder aber über mathematische Berechnung herausgefunden werden. Es hat sich bei den durchgeführten Tests herausgestellt, daß eine Diskretisierung von 15 Grad pro Winkel zu einer sehr guten Performance führt. Desweiteren wird die Ballgeschwindigkeit in 10 Teilbereiche aufgeteilt. Es muß immer abgewogen werden, wie fein man diskretisiert, da die Diskretisierungen die bestimmenden Faktoren für den Rechenaufwand und für die Performance des Lernagenten sind. Festzuhalten gilt also: Die Winkel $\alpha, \beta$ und $\gamma$ werden in 24 Teilbereiche zu je 15 Grad aufgeteilt. Die Geschwindigkeit wird in 10 Bereiche aufgeteilt: $v_{ball} < 0,5 \mid 0,5 \leq v_{ball} < 1,0 \mid 1,0 \leq v_{ball} < 1,5 \mid 1,5 \leq v_{ball} < 2.0 \mid 2.0 \leq v_{ball} < 2,1 \mid 2,1 \leq v_{ball} < 2,2 \mid 2,2 \leq v_{ball} < 2,3 \mid 2,3 \leq v_{ball} < 2,4 \mid v_{ball} \geq 2,4$. Die Winkel werden von $0$ bis $360$ Grad jeweils in $15$ Grad Schritten eingeteilt.

### 4.1.2 Die Aktionen des Agenten

Um auf die eingeteilten Observationen des Environments gut zu reagieren, muß man auch bei den Aktionen eine gute Einteilung entweder durch Probieren oder aber durch mathematische Berechnung herausfinden. Die folgende Einteilung wurde zum Designzeitpunkt des Lernverfahrens und

Abbildung 3: Veranschaulichung des Körperwinkels



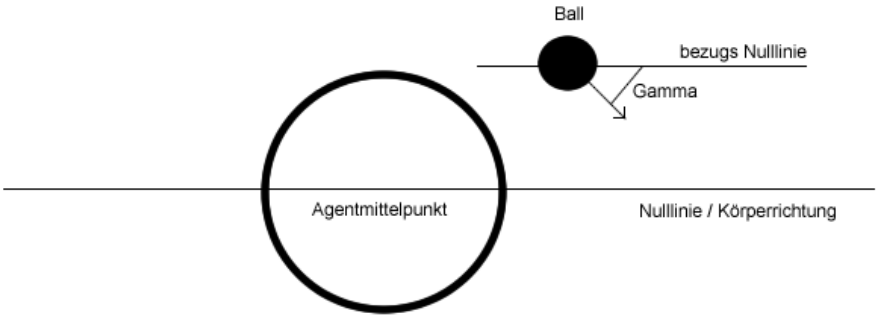

des Agenten festgelegt - Sie wurde jedoch später bei den Trainingsszenarien wieder geändert. Es wird hier zwischen zwei verschiedenen Arten von Aktionen unterschieden: Zum einen gibt es sogenannte „Turn“ Befehle mit denen sich der Agent mit seinem Körper drehen kann (d.h. er kann seinen Körperwinkel ändern) und zum anderen die „Kick“ Befehle, mit welchen sich der Ball, wenn er kickbar ist, in eine bestimmte Richtung schießen läßt. Der Aufbau der Befehle ist wie folgt:

- *Turn-Befehle*: Sie haben ein Argument und zwar einen Winkel $\delta$, um den sich der Agent drehen soll. Dieser wurde in $5$ Grad Schritte eingeteilt.
- *Kick-Befehle*: Diese Befehle haben 2 Argumente. Zum einen einen Winkel $\epsilon$, welcher auch hier in $5$ Grad Schritte aufgeteilt ist und zum anderen ein Power Argument, welches einen Bereich von $0 - 100$ einnehmen kann und in 10 Bereiche aufgeteilt wird.

## 4.2 Das UvA Trilearn Team

Der eingesetzte Lernagent wurde aus dem sogenannten „UvA Trilearn“ Team weiterentwickelt, welches das RoboCup Soccer Simulation Team der „Intelligent Autonomous Systems Group (IAS)“ [IAS2004] der Universität

Abbildung 4: Veranschaulichung des Bewegungsrichtungswinkels




Amsterdam bildet. Das eigentliche UvA Team hat schon viele gute Ergebnisse und Platzierungen bei RoboCup Wettbewerben erreicht. Im Jahre 2003 wurde der Basiscode des UvA Lernagenten zum Download [UvA2003] freigegeben. Dieser wurde in seiner Intelligenz beschnitten und enthält nur sogenannte „Low-Level Skills“. Diese umfassen hauptsächlich einfache Aufgaben, wie z.B. den Ball in eine bestimmte Richtung zu schießen. Der Basiscode bietet eine gute Grundlage für einen Lernagenten, da er ein gut ausgebautes Weltmodell hat, die Kommunikation zwischen Server und Klienten komplett abdeckt und eine einfache Schnittstelle zur Verfügung stellt. Das UvA Trilearn Team enthält auch ein Trainerpaket, welches essentiell wichtig für das Trainingsszenario ist. Es soll zunächst auf den Trainer eingegangen werden.

### 4.2.1 Der Trainer

Zunächst muß eine Unterscheidung vorgenommen werden, da es serverseitig zwei verschiedene Arten von Trainern gibt. Dies sind zum einen die sogenannten „Online-Coaches“ und zum anderen die Trainer (wobei diese auch Offline-Coaches genannt werden). Der Unterschied besteht darin, daß der Online-Coach bei richtigen Spielen eingesetzt wird - er darf den Agenten nur Anweisungen geben, nicht aber ins Spielgeschehen eingreifen. Der Trainer hingegen wird dazu benutzt, die Agenten bei jedweden Lern-

aufgaben zu unterstützen. Er hat die Möglichkeit aktiv ins Spielgeschehen einzugreifen mit folgenden Aktionen [SoServer]:

- Er kann den Spielmodus setzen (Spielmodi sind z.B. Elfmeter, Eckstoß, Halbzeit, ..)
- Er kann Nachrichten an die Agenten verschicken
- Er bekommt rauschfreie Informationen vom Server über alle beweglichen Objekte auf dem Spielfeld
- Er kann jedes bewegbare Objekt auf jede beliebige Position auf dem Spielfeld bewegen

Der Trainer des UvA-Teams ist bereits so weit realisiert, daß man direkt den für das Trainingsszenario wichtigen Code einbauen kann. Es müssen für ein Trainingsszenario dabei also nur noch die Befehle für den Server implementiert werden, welche die Objekte auf dem Spielfeld gemäss des Trainingsszenarien richtig setzen.

### 4.2.2 Das Weltmodell des Spielers

Das Weltmodell des Agenten beinhaltet alle Informationen über die Objekte, welche es auf dem Spielfeld gibt und stellt Methoden bereit, mit denen sich Informationen über diese Objekte abrufen lassen oder sich die Objekte manipulieren lassen. Die Informationen, welche im Weltmodell selber angeboten werden, lassen sich in 4 Gruppen einteilen.

- *Environmental Information*: Allgemeine Informationen über den Soccer Server
- *Match Informationen*: Infos über den Spielstatus
- *Objekt Informationen*: Informationen über alle Objekte auf dem Spielfeld, dazu zählen z.B. Spieler, der Ball, aber auch Flaggenobjekte. Flaggenobjekte werden dazu benutzt, um z.B. das Spielfeld und die Tore einzugrenzen, sie bilden also die Eckpunkte für bestimmte Objekte.
- *Actions Informations*: Informationen über die Aktionen, welche vom Agenten ausgeführt wurden.

Methoden, welche im Weltmodell zur Verfügung gestellt werden, können wie folgt klassifiziert werden:

- *Informationen über ein Objekt holen*: Diese Methoden sind definiert in der Datei „WorldModel.c“. Für den Lernagenten sind dies die wichtigsten Methoden.
- *Update Methoden*: Diese befinden sich in „WorldModelUpdate.c“ und verarbeiten die sensorischen Informationen - diese Methoden interessieren zunächst nicht, da sie vom Agenten selber aufgerufen werden und das Weltmodell stetig auf dem neuesten Stand halten.
- *Vorhersage Methoden*: Die Vorhersage Methoden sind allesamt in der Datei „WorldModelPredict.c“ definiert. Mit ihnen kann der Agent berechnen, welche Auswirkung eine bestimmte Aktion auf das Weltmodell haben würde.
- *High-Level-Methoden*: Definiert in „WorldModelHighLevel.c“. Beinhaltet Methoden, welche z.B. berechnen, ob der Ball für einen Torwart fangbar ist oder nicht. Es gibt im Basiscode nur sehr wenige „High-Level-Methoden“, da wie bereits früher beschrieben, die meisten aus dem Basiscode entfernt wurden.

### 4.2.3 Wichtige Datentypen des Weltmodells

Hier werden die wichtigsten Datentypen des Weltmodells vorgestellt und beschrieben.

- *VecPosition*: Gibt die Position eines Objektes an. Diese kann je nach Bedarf karthesisch oder polar dargestellt werden. Die Klasse VecPosition hat viele mathematische Methoden implementiert um Operationen wie +, -, * etc richtig auszuführen.
- *AngDeg*: Typen Definition für Winkel in Grad.
- *AngRad*: Typen Definition für Winkel in Rad.
- *Line*: Diese Klasse repräsentiert eine Gerade und hat Methoden um z.B. Schnittpunkte zu berechnen usw.

### 4.2.4 Vom Lernagenten verwendete Methoden des Weltmodells

Hier werden die verwendeten Methoden des Weltmodelles näher vorgestellt, welche vom Lernagenten genutzt werden. Wofür die einzelnen Methoden genutzt werden, soll an dieser Stelle noch nicht erläutert werden.

- *getCurrentCylce()*: Liefert den aktuellen Serverzyklus der Simulation zurück.
- *getAgentGlobalBodyAngle()*: Liefert den Körperwinkel des Agenten zurück. Dieser ist auf die Nulllinie des Spielfeldes bezogen.
- *getRelativeAngle(<Object o>, <bool b>)*: Liefert den Winkel zwischen dem Agenten und dem Object o zurück. Ist bool b auf 0 gesetzt wird der Winkel zwischen dem Objekt und dem Nacken des Agenten berechnet, andernfalls zwischen dem Körperwinkel des Agenten und dem Objekt.
- *getBallDirection()*: Liefert die globale Bewegungsrichtung des Balles zurück. Auch hier ist der Winkel auf die Nulllinie bezogen.
- *getBallSpeed()*: Liefert die Geschwindigkeit des Balles zurück.
- *getTimeLastRecvSenseMessage*: Liefert den Zyklus zurück, in welchem der Agent das letzte mal eine sensorische Information vom Server erhalten hat.
- *getConfidence()*: Liefert den Wahrscheinlichkeitswert eines Objektes zurück. Wenn ein Objekt sich an einer Stelle befindet, sollte dieser Wert $1,0$ betragen. Für jeden Zyklus in dem keine neue Information über ein Objekt geliefert wird, sinkt dieser Wert. Da die Position eines Objektes über Partikelfilter bestimmt wird, sinkt die Genauigkeit der Objektposition mit jedem Zyklus, in dem keine neue Information eingeht und damit auch der Confidencewert.
- *getBallPosition()*: Liefert die Vektorposition des Balles zurück.
- *getGlobalPosition(Object o)*: Liefert die Vektorposition eines Objektes o zurück.
- *getX()*: Liefert X-Koordinate einer Vektorposition.

- *getY()*: Liefert Y-Koordinate einer Vektorposition.
- *getIntersection(Linie l)*: Berechnet den Schnittpunkt zwischen Gerade l und der Geraden, über welche diese Methode aufgerufen wird. Wert wird als Vektorposition zurückgeliefert.
- *makeLineFromPositionAndAngle(VecPosition p, AngDeg w)*: Erstellt eine Gerade aus dem Punkt p und dem Winkel w.
- *makeLinefromTwoPoints(VecPosition p1, VecPosition p2)*: Erstellt eine Gerade durch die Punkte p1 und p2.
- *isBallKickable()*: Prüft, ob der Ball für den Agenten tretbar ist oder nicht.

Wozu die weitere Methoden zu gebrauchen sind, ist [Schleginski] zu entnehmen.

### 4.2.5 Die Problematik der Partikelfilter

Beim Agenten kann ein großer Störfaktor das Lernen beeinflussen. Durch eine Mißkonfiguration am Server kann es dazu kommen, daß der Trilearnplayer zur Bestimmung der Objekte in seinem Weltmodell Partikelfilter einsetzt. Wenn in der Server Konfiguration die beiden Optionen fullstate_l=1 und fulllstate_r=1 gesetzt sind, kommen keine Partikelfilter zum Einsatz (Standardeinstellung ist hier $0$, also kommen sie standardmäßig zum Einsatz). Werden Partikelfilter eingesetzt, verursacht der Trainer folgendes Problem: Beim Start einer Trainingsepisode wird vom Trainer schlagartig die Postion vom Ball geändert. Da dies stellenweise eine große Änderung der Position darstellt und da diese darüber hinaus noch entgegen des eigentlichen Bewegungstrendes des Balles gesetzt werden kann, braucht der Partikelfilter mehrere Zyklen, um die neue Position wieder einigermassen bestimmen zu können. In dem Moment jedoch, wo der Ball versetzt wird, erhält der Agent durch den Partikelfilter eine fehlerhafte Vorstellung von der Position des Balles. Die Fehlerabweichung kann zwar sehr gering sein, jedoch kann es auch dann ausreichen, um die Vorstellung des Agenten, in welcher Observation er sich befindet, durcheinander bringen. Aus diesem Grund sind Partikelfilter beim Lernprozess auf alle Fälle auszuschalten!

## 4.3 Entwicklung des finalen Agenten

Hier soll dargestellt werden, was zusätzlich zum Basiscode entwickelt werden mußte, um das RFL in den Agenten zu integrieren. Bisher wurden schon die benötigten Methoden und die Datenstrukturen beschrieben, welche für diese Weiterentwicklung nötig waren. Nun folgt eine Beschreibung der RFL Implementierung im UvA Basiscode. Um eine Weiterentwicklung und eine Implementierung des RFLs möglich zu machen, muß im vorhandenen Basiscode eine Methode gesucht werden, welche es ermöglicht, in jedem Serverzyklus eine Aktion auszuführen. Dies begründet sich in der Tatsache, daß der Agent in jedem Serverzyklus neue sensorische Informationen vom Server bekommt. Der Agent wiederum muß für diese neuen Daten innerhlab eines Zykluses eine passende Aktion finden. Glücklicherweise gibt es eine Methode, die genau diesen Ansprüchen genügt. Es ist die Methode „Player::deMeer5()“ welche in der Quelldatei „PlayerTeams.cpp“ zu finden ist. Die zu dem Basiscode hinzugefügten Funktionen stehen zu etwa 90 Prozent in den Dateien rli.h, rli.cpp, PlayerTeams.cpp und PlayerTeams.h. In ihnen findet sich der Quellcode für das komplette Lernverfahren.

### 4.3.1 Grober Ablauf der Lernfunktion

Nachdem der Einsprungpunkt für den Lerncode bereits bestimmt wurde („Player::deMeer5()“), wird nun der allgemeine Ablauf des Lernvorganges beschrieben. Es sollte klar sein, daß dieser Einsprungpunkt in jedem Zyklus (gemeint ist hier der Serverzyklus) mindestens einmal aufgerufen wird. Beim ersten Durchlauf werden die für den Lernvorgang wichtigen Datenstrukturen und Semaphore erstellt und initialisiert. Durch Erstellen des Agentenobjektes (beinhaltet später die Policy etc), wird sowohl die Diskretisierung der Winkel festgelegt als auch die Entscheidung, ob eine früher bereits generierte Policy geladen oder ob eine neue erzeugt werden soll. An dieser Stelle wird ein Semaphor erzeugt und auf den Wert 1 initialisiert, welcher später dazu dient, den Agenten und den Trainer zu synchronisieren.

Sobald alle Objekte erzeugt sind und der erste reguläre Methodendurchlauf gestartet wird, belegt der Agent den Semaphor und gibt ihn erst frei, sobald die Methode verlassen wird. Dies verhindert, daß der Trainer Objekte umsetzen kann, solange sich der Agent im Lernvorgang befindet. Diese

Maßnahme zur Synchronisierung des Agenten und des Trainiers ist unerlässlich, da ein zu frühes Einsetzen der Traineraktivität die Daten für den Agenten so sehr verfälscht, daß dieser nicht mehr in der Lage ist, eine korrekte Policy zu lernen.
Ist der Semaphor belegt, beginnt der eigentliche Lernvorgang im Agenten, welcher sich wie folgt darstellt: Wie weiter oben schon erwähnt kann diese Methode, in welcher gearbeitet wird, pro Serverzyklus mehrfach durchlaufen werden. Da dies regelmäßig der Fall ist, müssen alle zusätzlichen Durchläufe abgefangen werden. Es ist hierbei enorm wichtig, daß die *ersten* Durchläufe abgefangen werden, da bei diesen die Informationen, die der Server sendet, stellenweise sehr ungenau sind und somit unbrauchbar für den Agenten wären. In der Policy, welche bei der Agenteninitialisierung erzeugt wurde, sind alle möglichen Observationen vermerkt, welche für den Agenten von Wichtigkeit sind - es sind noch keine Aktionen zugeordnet und auch gilt noch keine Observation als bereits besucht. Ist eine Trainingsepisode beendet (zur Erinnerung: eine Trainingsepisode bezeichnet die Anzahl der Zyklen, welche maximal zur Verfügung stehen, oder auch den „L“-Wert) werden, sofern die Episode in einem positiven Endzustand endete, die bisher getroffenen Aktionen beibehalten und alle Observationen wieder als „nicht besucht“ markiert. Sollte die Episode in einen negativen Endzustand kommen, werden die getroffenen Akionen wieder gelöscht und auch hier alle Observationen als nicht besichtigt markiert.
Ist die Trainingsepisode noch nicht beendet, wird in jedem Zyklus geprüft, ob es für die aktuelle Observation bereits eine gespeicherte Aktion gibt - ist dies der Fall, wird diese Aktion ausgeführt und die Observation als besucht markiert. Ist es nicht der Fall, wird eine zufällige Aktion ausgewählt (aus dem Vektor der zulässigen Aktionen), diese in der Policy in Verbindung mit der aktuellen Observation gespeichert und die Observation selber wird als besucht markiert.

### 4.3.2 Aufbau der Datenstrukturen

Um den Lerncode besser verwalten zu können und um ihn besser vom restlichen Basiscode abgrenzen zu können, werden alle wichtigen Daten über die Klasse „Agent“ verwaltet. Hier sind Informationen über die Diskretisierungen der Winkel, welche die Observationen ausmachen, gespeichert. Die Policy, welche essentieller Bestandteil des kompletten Lernens ist, wird

durch Objekte der Klasse „PolicyElement“ widergespiegelt. Dabei enthält ein „PolicyElement“ folgende Teile:

- *Observation omega*: Die Observation, auf die sich die zusätzlichen Informationen beziehen.
- *SoccerCommand taken*: Ein SoccerCommand Objekt, welches die Aktion beinhaltet, welche für die Observation gespeichert wurde. Hier ist nur ein Objekt hinterlegt, sofern überhaupt eine Aktion gespeichert ist, anderfalls ist dieser Wert auf „NULL“ gesetzt.
- *bool z*: Gibt an, ob in diesem Trainingszyklus diese Observation bereits einmal besucht worden ist. Ist dies der Fall so ist $z = 1$, ansonsten ist $z = 0$
- *bool d*: Bezeichnet, ob die Aktion in diesem Zustand deterministisch ist und ob sie auch über das Ende der Trainingsepisode hinweg gespeichert bleibt. Dieser Wert wird dazu benötigt, um beim Lernprozess Zwischenergebnisse ausgeben zu können.

Wie zu sehen ist, enthält ein PolicyElement genau eine Observation (und die ihr zugehörigen Daten) - in ihr werden die wichtigen Daten des Weltmodelles gespeichert (welche Daten wichtig sind, wurde an früherer Stelle schon einmal erklärt und wird hier nicht nochmal aufgegriffen). Die Observationen, welche in den PolicyElement Objekten abgelegt werden, sind Elemente der Klasse „Observation“. Sie bestehen aus 4 vorzeichenlosen Ganzzahlen, welche da sind:

- *angleBall*: Enthält den bereits diskretisierten, vorzeichenlosen Wert des Winkels zwischen der Nullinie und dem Ball.
- *anlgeBody*: Enthält den bereits diskretisierten, vorzeichenlosen Wert des Winkels zwischen der Nullinie und dem Agentenkörper
- *angleVelocity*: Enthält den bereits diskretisierten, vorzeichenlosen Wert des Winkels zwischen der Nulllinie und der Bewegungsrichtung des Balles.
- *velocity*: Enthält den bereits diskretisierten, vorzeichenlosen Wert der Geschwindigkeit des Balles.

Die Tatsache, daß in den Observationen bereits diskretisierte Werte und somit nicht mehr die wirklichen Winkelwerte (und der Geschwindigkeitswert) stehen, hat den Vorteil, daß hierdurch die Observation in ihrer Größe sehr eingeschränkt bleibt. Jedoch macht diese Tatsache es auch von Nöten, eine Abbildungsfunktion zu schreiben, welche sich die Daten aus dem Weltmodell des Spielers holt und diese in eine Observation umrechnet.

Im Agenten selber, werden die „PolicyElement“ Objekte in einem Vektor zu der kompletten Policy des Agenten zusammengefaßt. Der Vorteil eines Vektors ist, daß er eine dynamische Größe besitzt, es aber trotzdem möglich ist, durch eine Arrayindizierung auf die einzelnen Elemente zuzugreifen. Alle möglichen Aktionen, die der Agent ausführen kann, werden in einem zweiten Vektor zusammengefaßt. Beim Starten des Lernvorganges müssen diese beiden Vektoren initialisiert werden. Dies kann entweder durch eine neue Initialisierung erfolgen, oder durch das Laden einer alten Policy aus einer Datei werden die Vektoren rekonstruiert.

### 4.3.3 Abbildung der Observationen

Wie im letzten Abschnitt bereits angedeutet, wird eine Abbildungsmethode benötigt, welche die Weltdaten auf die im Agenten festgelegten Observationen abbildet. Für diesen Zweck gibt es die Methode „int Agent::getCurrentObservation( WorldModel *WoMo )“, welche sich die wichtigen Daten aus dem an die Methode übergebenen Weltmodell holt. Die Daten der Winkel und der Geschwindigkeit, welche für die Observation von Wichtigkeit sind, werden anhand der im Agenten festgelegten Diskretisierung (was die Standarddiskretisierung ist, wurde früher bereits beschrieben) auf eine vorzeichenlose Ganzzahl abgebildet. Die Winkel werden im Weltmodell von $-180\ bis\ +180$ Grad dargestellt - als erstes wird diese Darstellung transformiert in eine Darstellung in den Bereich von $0\ bis\ 360$ Grad. Danach wird eine Ganzzahlendivision durch die Diskretisierung durchgeführt, was zu dem endgültigen Wert führt. Beispiel: Gegeben sei die Diskretisierung von 20 Grad pro Observation und der Winkel $\gamma$ sei $235$ Grad. So ist der abgebildete Wert von $\gamma$ nämlich $A(\gamma) = 235/20 = 11$. Das Symbol / repräsentiert in diesem Fall die Ganzzahlendivision.

Die Geschwindigkeit des Balles wird auf Werte zwischen 0 und 9 abgebildet entsprechend der Intervallsgrenzen welche im Kapitel 4.2.1 bereits

beschrieben wurden.
Sobald alle diskretisierten Werte vorliegen, werden diese mit den Werten der Observationen, welche in der Agentenpolicy liegen, verglichen und es wird die Eintragsnummer aus dem Policyvektor des Agenten zurückgeliefert.

### 4.3.4 Speicher- und Einlesefunktionen

Um ein unterbrochenes Lernen zu ermöglichen, ist es notwendig, Methoden zur Speicherung und zum Einlesen von Policies zu entwerfen. Das Speichern der Policy ist in der Methode „void Agent::savePolicy()“ implementiert. Durch Aufruf dieser Methode werden 2 Dateien erzeugt, zum einen eine menschenlesliche textuelle Datei (policy.txt) und zum anderen eine binäre Datei (policy.log), welche zum Wiedereinlesen der Policy verwendet wird. Es werden alle Policy Elemente mit ihren Observations und den dazugehörigen Aktionen (sofern welche vorhanden sind) der Reihe nach in diese Dateien geschrieben. Ein Eintrag besteht demzufolge aus dem diskretisierten Wert von den drei Winkeln und der Geschwindigkeit. Zusätzlich wird noch die dazugehörige Aktion gespeichert, welche aus einem Kommandotypen (KICK / TURN) und jeweils einem Parameter beim Turnbefehl besteht oder aus zwei Parametern bei einem Kickbefehl. Zusätzlich zu der Policy wird noch eine Headerinformation gespeichert - aus dieser ist zu entnehmen welche Diskretisierung bei der gespeicherten Policy verwendet und wieviele Lernzyklen bisher durchlaufen wurden. In der textuellen Datei wird noch vermerkt, wieviel gute und wieviele schlechte Lernausgänge es bisher gegeben hat - diese Werte können zur Berechnung der Lernperformance herangezogen werden.

### 4.3.5 Der Trainer dieses Trainingsszenarios

Um dem Agenten das Lernen zu ermöglichen, wird ein Trainer benötigt, der den Agenten und evtl auch andere Objekte zu Beginn jeder Trainingsepisode, gemäß den Auflagen den Trainingsszenarios, positioniert. Um dies zu bewerkstelligen, wertet der Trainer den Semaphor des Agenten aus, um sich mit diesem zu synchronisieren. Dannach positioniert er zu Beginn jeder Trainingsepisode, den Agenten 40 Meter in Richtung des rechten Tores (bezogen auf den Spielfeldmittelpunkt). Die Körperrichtung des Agenten wird vom Trainier zufällig gesetzt. Zusätzlich wird der Ball zufällig in

einen Ring um den Agenten gesetzt, dessen Außen- und Innenrand durch die „kickable area“ und durch die „collision area“ des Agenten beschrieben wird. Die Ballgeschwindigkeit kann einen Wert $\geq 0$ aufweisen. Diese Werte sind durch „Defines“ im Coach-Quellcode festgelegt und können für die einzelnen Trainingsszenarien angepaßt werden. Die Zufallszahlen, welche der Trainer (und auch der Agent) benutzt, werden durch den Marsenne Twister berechnet [Marsenne].

### 4.3.6 Realisierungsprobleme

Hier werden die größten Probleme beschrieben, die vermuteten Ursachen und wie sie behoben wurden. Zu allererst sei festgestellt, daß diese Probleme ausschließlich die Synchronisation und den Datenaustausch zwischen Server, Agent und Trainer betrafen. Als primäre Fehlerindikation war zu bemerken, daß die Daten über den Zustand des Systems (also z.B. die Position bestimmter Objekte wie beispielsweise des Balles), fehlerhaft beim Agenten ankamen. Sie hinkten entweder dem tatsächlichen Zustand des Systems um einige Zyklen hinterher oder aber die Werte selber hatten eine Abweichung von über $+/-50\%$. Solche stark verfälschten Werte sind für das Lernen eine nicht hinzunehmende Fehlerquelle, da durch abweichende Daten eine Abbildungsfunktion, welche Zustände auf Observationen abbildet, völlig unterschiedliche Observationen für eigentlich gleiche Zustände zurückliefern kann. Durch die dadurch zustande kommende „Abbildungswillkür“ kam es im frühen Entwicklungsstadium dazu, daß die Policy des Agenten willkürlich irgendwelche Aktionen lernte, welche nur aufgrund einer Fehlabbildung in einen Endzustand gekommen waren. Weiterhin wurden auch durch die „Abbildungswillkür“ bereits richtig gelernte Observation- Aktionspaare wieder verlernt. Schlußendlich hatte dies zur Folge, daß sowohl die Lerneffektivität als auch die Lernperformance (was damit gemeint ist, wird im nächsten Kapitel erklärt) sehr schlecht waren.

Nachfolgend werden die vermeindlichen Fehlerursachen in chronologisch aufgetretener Reihenfolge beschrieben.

**4.3.6.1 UDP-Problem** Durch die Fehlerindikation, daß häufig Daten vom Server nicht in das Weltmodell des Spielers eingingen, war eine der ersten vermuteten Fehlerursachen, daß das UDP Subsystem die Daten nicht

verlässlich auslieferte. Bei der Kommunikation mit Hilfe von UDP zwischen Server und Klient erhält keine der beiden Seiten eine Information darüber, ob die gesendeten Datenpakete angekommen sind oder nicht (UDP ist ein Datagrammprotokoll). Zur Fehlerüberprüfung wurde der Datenverkehr zwischen Server und Klient aufgezeichnet und ausgewertet. Nach einer genauen Prüfung des Aufbaus der Datenpakete und deren Anzahl war klar, daß die Informationen nicht im UDP Subsystem verloren gehen. Weiterhin lagen auch keine sogenannten „Source quench messages“ vor, welche auf ein Verlorengehen der Datenpakete hingedeutet hätten.

**4.3.6.2 Desynchronisation zwischen Agenten und Trainer** Da die Fehlerindikation immer noch persistent war, wurde eine zeitliche Desynchronisation zwischen dem Trainer und dem Agenten vermutet. Solch eine Desynchronisation hätte zur Folge, daß sich der Lernablauf verschiebt und es z.B. passieren könnte, daß der Agent mitten in einer Trainingsepisode vom Trainer auf einen neuen Startzustand gesetzt wird. Es müßte also sichergestellt werden, daß eine Trainingsepisode vom Trainer begonnen wird und danach immer abwechselnd der Trainer und der Agent zum Zuge kommen. Dies wurde durch die Einführung eines Semaphores bewerkstelligt - er stellt sicher, daß die beiden Prozesse synchron laufen. Genau diese Ergänzung brachte einen guten Fortschritt in Bezug auf die Qualität der Daten des Agenten.

**4.3.6.3 Anpassung Serverparameter** Um eine weitere Verbesserung der Agentendaten zu erzielen, wurden auf Serverseite die Parameter so angepaßt, daß kein Rauschen in den Daten mit übermittelt wird. Weiterhin wurde eingestellt, daß sensorische Informationen 10 mal pro Zyklus an den Agenten übermittelt werden. Entgegen der angestellten Vermutungen, brachten diese Änderungen keinerlei Verbesserungen bei den Daten des Agenten.

**4.3.6.4 Erhöhung der Serverzyklen** Es lag die Vermutung nahe, daß durch eine kurze Zyklendauer, der Agent zu viel Rechenzeit in Anspruch nimmt und somit nicht in der Lage ist, die Informationen, welche er vom Server bekommt, zu verarbeiten. Um diese Möglichkeit auszuschließen, wurden die Serverzyklen verlängert. Doch auch diese Anpassung brachte keine Verbesserung der Informationen des Agenten.

**4.3.6.5 Datenträgheit** Nach den oben stehenden Änderungen war nun zu erkennen, daß die Informationen am Agenten zu einem bestimmten Grad zeitverzögert ankamen. Diese Trägheit könnte von einem Partikelfilter im Agenten ausgehen (siehe auch Abschnitt „Die Problematik der Partikelfilter“). Um dies zu prüfen, wurde ein sogenannter „FADE_IN“ und ein „FADE_OUT“ eingeführt. Diese sorgen am Anfang und am Ende einer Trainingsepisode dafür, daß der Partikelfilter eine bestimmte Anzahl an Zyklen Zeit bekommt, sich auf die neuen Informationen anzupassen. Diese Änderungen brachten tatsächlich eine beträchtliche Verbesserung der Daten - jedoch nur am Anfang einer Trainingsepisode. Sobald sich die Daten innerhalb einer Trainingsepisode veränderten, waren die Daten wieder sehr ungenau. Weiterhin brachten diese Änderungen einen extremen Nachteil mit sich, da am Anfang und am Ende einer jeder Trainingsepisode gewartet wurde, kam es zu einer Verdreifachung der Lernepisodendauer.

**4.3.6.6 Partikelfilter** Nach den vorangegangenen Verbesserungen und Ergebnissen, lag die Lösung für die Probleme auf der Hand: Der Partikelfilter auf der Agentenseite mußte abgeschaltet werden. Jedoch gibt es auf Agentenseite keine Möglichkeit den Partikefilter auszuschalten und mangels Dokumentation wäre es in annehmbarer Zeit auch nicht möglich gewesen dies durch eine Umprogrammierung des Agenten zu erreichen. Weiter Experimente an der Serverkonfiguration deckten eine Möglichkeit auf: Durch Setzen der beiden Serverkonfigurationsflags „fullstate_r“ und „fullstate_l “ auf 1 schaltet der Agent automatisch seinen Partikelfilter aus und hat so genau die Informationen, die der Server übermittelt. Nachdem der Partikelfilter ausgeschaltet war, wurden „FADE_IN“ und „FADE_OUT“ wieder aus dem Code genommen, um eine Steigerung der Performance zu erreichen. Durch das Ausschalten des Partikelfilter ergaben sich im Nachhinein jedoch weitere Probleme: So wird durch das Ausschalten bewirkt, daß viele Methoden zur Positionsbestimmung nicht mehr zur Verfügung stehen und so mußten diese durch neue ersetzt werden.

# 5 Lernverhalten und Leistungsfähigkeit

In diesem Kapitel werden das Lernverhalten und die Leistungsfähigkeit des vorgestellten Verfahrens an verschiedenen Trainingsszenarien vorgestellt. Es werden mehrere Parametrisierungen durchgegangen (wovon jede einem Trainingsszenario entspricht) und untersucht, ob sich ein Trend hin zu einer Optimalität ergibt. Optimal heißt in diesem Fall, daß der Performancewert eines Trainingsszenarios möglichst maximal und die Lerneffizienz möglichst hoch sein soll. In diesen Trainingsszenarien werden verschiedene Lernagenten dazu gebracht, eine möglichst optimale Policy zu erlernen. Die Lerneffizienz und die Performance der Policies in den einzelnen Szenarien wird aufgezeichnet und untersucht. Weiterhin soll später versucht werden, bereits gelernte Policies in leicht veränderten Szenarien einzusetzen. Ein Trainingsszenario wird im wesentlichen durch zwei Faktoren bestimmt. Dies ist zum einen ein Set an Parameteren des Agenten und des Trainers ( und in zweiter Linie auch die Parameter des Servers, diese sind jedoch über alle Trainingsszenarien gleich und wurden bereits beschrieben ) und zum anderen die bei diesem Parameterset erlernte Policy. Zum Schluss soll das Lernverhalten der hier verwendeten Lernmethode, mit einer Q-Learning Methode verglichen werden [CDoeven].

## 5.1 Szenario- und Auswertungsbeschreibung

Es wird nun beschrieben, welche Parameter auf Trainer- und Agentenseite zu einem Szenario gehören, und wie die Auswertungen der gelernten Policies erfolgen.

### 5.1.1 Parametersets

**5.1.1.1 Trainer Parameterset** Der Parameterset des Trainers besteht aus folgenden Einstellungsmöglichkeiten:

- *STEPS_PER_TRIAL*: Bezeichnet die Länge einer Lernepisode, entspricht dem $L$-Wert des eigentlichen Observation based RFL.
- *FADE_IN*: Mittlerweile ein nicht mehr benötigter Wert, er gab an, wieviele Zyklen vor dem Start einer Trainingsepisode gewartet werden soll( damit sich die Informationen des Partikelfilters glätten)

- *FADE_OUT*: Äquivalent zu FADE_IN jedoch am Ende einer Lernepisode.
- *RADIUS*: Bezeichnet den Radius der „kickable area“des Agenten.
- *MIN_RADIUS*: Bezeichnet die Größe der „collision area“des Agenten. Bildet zusammen mit $RADIUS$ die Innen- und Außenbegrenzung eines Ringes um den Agenten, in welchem der Ball positioniert werden darf.
- *MAX_VELOCITY*: Gibt die maximale Geschwindigkeit an, mit der sich der Ball in einem Startzustand bewegen darf. Auch wenn hier als Wert $0$ angegeben wurde, ist es möglich, daß sich der Ball innerhalb einer Lernepisode bewegt - er hat nur am Anfang keine Ausgangsgeschwindigkeit
- *DISTANCE*: Gibt an, auf wieviel Meter rechts vom Spielfeldmittelpunkt der Agent positioniert wird.
- *BODYANGLE*: Gibt an, um wieviel Grad der Körperwinkel nach links und rechts, gesehen von der Nulllinie aus, abweichen darf.

Wie zu sehen ist, gibt es bereits auf der Trainerseite eine Vielzahl an Parametern, welche den Lernvorgang mehr oder weniger beeinflussen können. Ein Vorteil ist vor allem, daß man gezielt durch das Justieren der Parameter $MAX_RADIUS,\ MAX_VELOCITY, BODY ANGLE$ den Zustandsraum des Agenten einschränken kann und so durch ein erneutes Nachjustieren den Zustandsraum schrittweise vergrößern kann. Das kann dazu führen, daß der Agent für einen Teil seines eigentlichen Zustandsraum bereits eine mehr oder weniger optimale Policy lernt - nach und nach wird dann der Zustandsraum vergrößert. Dies kann unter bestimmten Umständen dazu führen, daß der eigentliche Lernprozess beschleunigt wird. Jedoch sollte bedacht werden, daß auch genau das Gegenteil eintreten kann - dann es ist möglich, daß eine Policy für den vom Trainer eingeschränkten Zustandsraum zwar optimal sein kann, dies für den größeren Zustandsraum jedoch nicht mehr ist. Es würde also dazu führen, daß bei jeder Vergrößerung des Zustandraumes ein nicht zu vernachlässigender Teil der Policy neu gelernt werden muß. Genau dieses Verhalten war bei einigen Trainingsszenarien zu beobachten, nach jeder Vergrößerung war zu bemerken,

daß die Zahl der bis dato gelernten Zustand-Aktions Paare abnahm. Bei anderen Trainingsszenarien war dies nicht zu beobachten - es bleibt also noch offen zu prüfen was genau die Abnahme beeinflußt und was nicht.

**5.1.1.2 Agenten Parameterset** Im Gegensatz zu dem Parameterset des Trainers, hat das Set des Agenten mehr Einstellungsmöglichkeiten. Es besteht aus den folgenden Parametern:

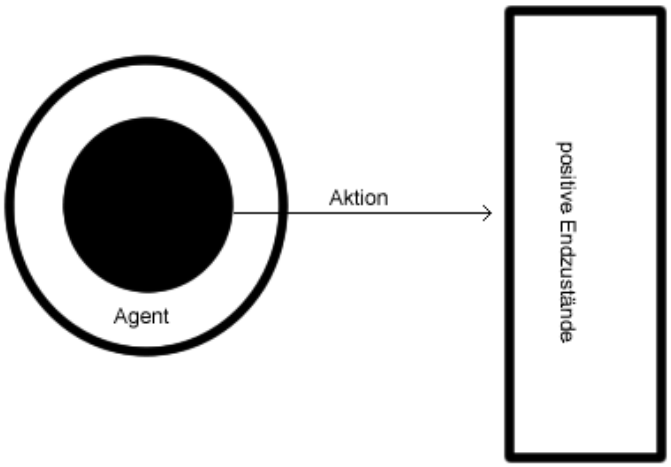
- *MAX_STEPS_PER_TRIAL*: Bezeichnet, wie lang die Trainingsepisode ist - diese muß auf den gleichen Wert gesetzt sein wie beim Trainer, da sich sonst diese beiden Prozesse nicht richtig synchronisieren und somit ein erfolgreiches Lernen nicht mehr möglich ist.
- *FADE_IN*: Siehe FADE_IN des Trainers.
- *FADE_OUT*: Siehe FADE_OUT des Trainers.
- *SEND_PER_STEP*: Wie früher bereits erwähnt, werden die sensorischen Informationen laut Serverkonfiguration 10 mal pro Zyklus an den Agenten gesendet. Dieser Wert gibt an, bei welchem Zeitschritt diese Informationen abgegriffen werden - er *muß* kleiner sein als der Wert in der Serverkonfiguration, da die Informationen nicht immer so oft beim Agenten eingehen, wie sie es laut Serverkonfiguration müßten.
- *MIN_SPEED*: Gibt an, welche Geschwindigkeit der Ball besitzen muß, wenn er die „kickable area“ verläßt. Es ist also ein Teil der Definition eines positiven Endzustandes.
- *Diskretisierung von $\alpha$:* $d(\alpha)$: Diskretisierung des Winkels $\alpha$. Was dieser Winkel ist, wurde bereits in einem früheren Kapitel beschrieben
- *Diskretisierung von $\beta$:* $d(\beta)$: Diskretiserung des Winkels $\beta$. Die Bedeutung wurde früher beschrieben.
- *Diskretisierung von $\gamma$:* $d(\gamma)$: Diskretisierung des Winkels $\gamma$. Die Bedeutung wurde früher beschrieben.
- *Diskretisierung von $v_{ball}$:* $d(v_{ball})$ : Diskretisierung von der Geschwindigkeit $v_{ball}$. Wurde früher bereits beschrieben.

- *Diskretisierung von $\delta$:* $d(\delta)$: Disrkretisierung des Winkels $\delta$. Wurde früher bereits beschrieben.
- *Diskretisierung von $\epsilon$:* $d(\epsilon)$: Diskretisierung des Winkels $\epsilon$. Wurde früher bereits beschrieben.
- *Diskretisierung von $p_{kick}$:* $d(p_{kick})$: Diskretisierung der Kickpower. Wurde früber bereits beschrieben.
- *Torbreite*: Gibt die Breite des Tores an, auf das geschossen werden soll. Das normale Tor ist knapp über 7 Meter in beide Richtungen vom Tormittelpunkt aus breit.

### 5.1.2 Lerneffektivität

Hier wird nun das Verfahren beschrieben, welches dazu benutzt wurde, um die Lerneffektivität des Trainingsszenarios zu messen. Als Lerneffektivität wird die Menge der gelernten Zustands(Observation)- Aktionspaare bezeichnet zu einem beliebigen Zeitpunkt $t_n$. Zu diesem Zweck wird alle 5000 Zyklen die Anzahl der gelernten Aktionen aufgezeichnet und abgespeichert. Zeichnet man über einen längeren Zeitraum immer wieder die gelernten Aktionen auf, so ergibt sich eine Kurve, welche die Effektivität des Trainingsszenarios widerspiegelt. Es sollte bedacht werden, daß die oberste Grenze der gelernten Aktionen *nicht* der Anzahl der Observationen entspricht, da sich in den hier vorgestellten Szenarien der Ball am Anfang nicht bewegt. Die Observationen, in denen sich der Ball bewegt, können in diesem Fall nur dann besucht werden, wenn der Agent eine Aktionskette lernt und somit in eine Observation mit einer Ballgeschwindigkeit übergeht. Sollte der Agent jedoch für jeden Startzustand genau eine Aktion lernen, welche ihn direkt in einen positiven Endzustand bringt, so kommt es dazu, daß nicht für alle Observationen Aktionen gelernt werden. In Abbildung 5 ist eine 2 dimensionale Abbildung des eigentlich höherdimensionalen Observationraumes des Agenten zu sehen. Dabei entspricht der komplett schwarze Kreis den Observationen, in denen der Ball ruht und der Ring um den schwarzen Kreis repräsentiert die Observationen, in denen der Ball eine Geschwindigkeit hat. Gebe es nun eine Aktion, welche aus allen Observationen, die in dem schwarzen Kreis liegen, den Agenten in einen positiven Endzustand führen, so würden keine Aktionen für die Observationen in dem äußeren Ring gelernt (weil sie keine Startzustände

Abbildung 5: Problematik Lerneffektivität




sind und weil keine Aktionskette über sie führen könnte). Dies würde dazu führen, daß nur für einen Teil aller möglicher Observationen Aktionen bekannt würden - dies reicht trotzdem aus, um in genau diesem Trainingsszenario den Agenten verläßlich in einen positiven Endzustand zu bringen, so fern keine weiteren Startzustände hinzukommen. Bei wieviel Prozent von den gesamten Observationen diese Grenze genau liegt, läßt sich nicht vorhersagen.

### 5.1.3 Performance

Als die Performance bezeichnet man das Verhältnis zwischen bekannten Aktionen, welche in einen positiven Endzustand übergehen und solche die in einen negativen Endzustand übergehen. Es wird immer über 5000 Zyklen aufgezeichnet, ob gespeicherte Aktionen in positive oder negative Endzustände übergehen und am Ende das Verhältnis der beiden Werte abgespeichert. Es ergibt sich eine Kurve, welche zu Zeitpunkten $t$ die zugehörigen Performancewerte $P(t)$ abbildet. Es hat sich herausgestellt, daß gute Policies Performancewerte von über 95% haben können.

### 5.1.4 Verteilungsmessung

Gibt es in einem Trainingsszenario bereits eine Policy, welche einen Großteil der besuchbaren Observationen abdeckt, lohnt es sich eine Verteilungsmessung durchzuführen. Es sei angemerkt, daß die Policies auf eine bestimmte Torbreite trainiert worden sind. So ist das Tor im Standardfall z.B. 14 Meter breit. Bei der Verteilungsmessung wird, sobald der Ball die „kickable area“ des Agenten verlassen hat der Schnittpunkt zwischen der Bewegungsrichtung des Balles und der Torlinie (des rechten Tores) berechnet. Von diesem Schnittpunkt wird jeweils der vorzeichenbehaftete Y-Koordinaten Wert gespeichert und zur Erstellung der Verteilungsfunktion benutzt. Es werden dann die Verteilungsklassen eingeteilt und die relativen Verteilungen berechnet. Diese werden dann in einem Verteilungsgraphen eingezeichnet. Die Verteilungsmessung gibt also die prozentualen Werte an, bei welchem Y-Wert der Ball das Tor treffen wird.

## 5.2 Die Trainingsszenarien

In diesem Abschnitt werden nun die einzelnen Trainingsszenarien vorgestellt und zu diesen Szenarien jeweils Tests durchgeführt. Es soll dadurch ein Trainingsszenario ermittelt werden, welches eine möglichst optimale Policy erreichen kann. Diese Policy wird dann nicht nur bei dem Ursprungsszenario getestet, sondern auch in anderen Szenarien - jedoch wird bei diesen Szenarien nicht die Diskretisierung geändert sondern vielmehr z.B. die Torbreite. Ob bereits gelernte Policies auch auf Szenarien anwendbar sind, welche eine andere Diskretisierung haben, bleibt immer noch zu prüfen. Es werden im Folgenden jeweils die Szenarienparameter vorgestellt und daran anschließend dann die entsprechenden Tests.

### 5.2.1 Die 12x36 Welt - Teil 1

Die Trainingsszenarien, die vorgestellt werden, werden immer in drei Teile gegliedert. Bei einigen kommt noch ein vierter Teil hinzu. Es wird grundsätzlich mit den Referenzparametern begonnen, da diese den Aufbau des Trainingsszenarios beschreiben. Dannach folgen dann die Lernperformance und die Lerneffektivität - beides sind Graphen welche die Leistungsfähigkeit des jeweiligen Trainingsszenarios am besten beschreiben. Die Überschrift soll mit ihrer Bezeichnung die wichtigsten Eckpunkte des Trainings-

szenarios wiedergeben, so bezeichnet die erste Ziffer die Diskretisierung der Winkel der Observationen und die zweite Ziffer bezeichnet die Diskretisierung der Aktionswinkel.

**5.2.1.1 Die Referenzparameter** Zunächst soll ein Blick auf die Parameter geworfen werden, welche das Szenario beschreiben. Wie zu erkennen

Abbildung 6: Referenzparamter der 12x36 Welt - Teil 1

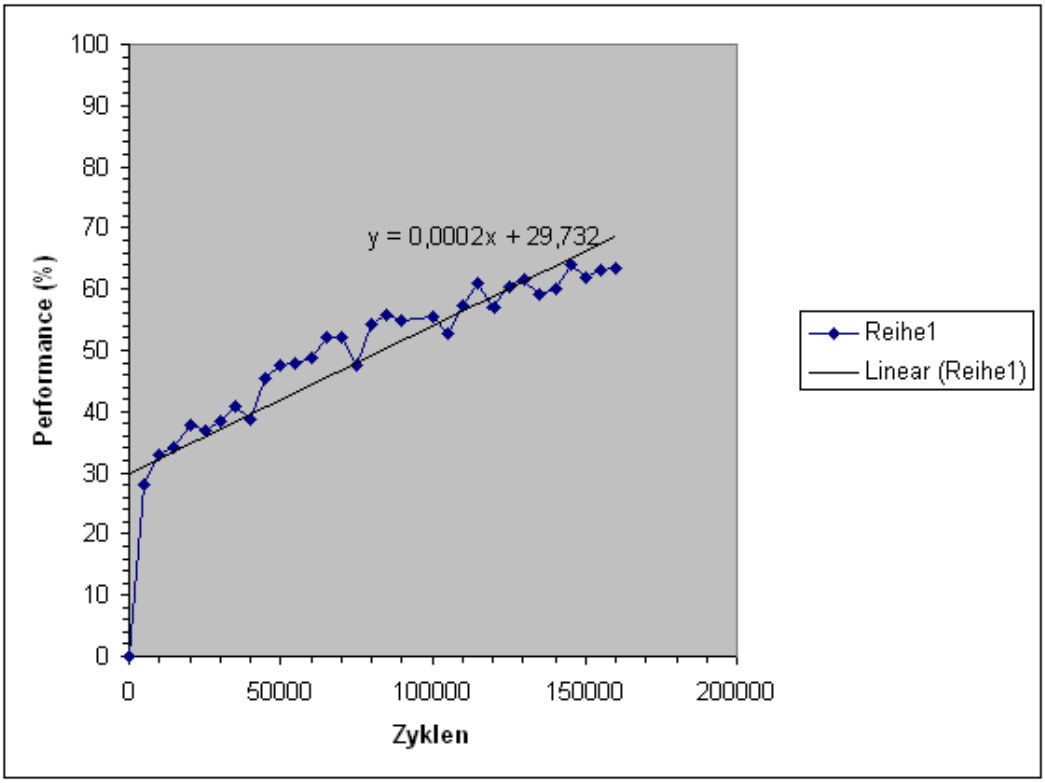
| Trainer / Agent | Parameter | Wert |
|---|---|---|
| T | STEPS_PER_TRIAL | 4 |
| T | FADE_IN | 0 |
| T | FADE_OUT | 0 |
| T | RADIUS | 1,085 |
| T | MIN_RADIUS | 0,386 |
| T | MAX_VELOCITY | 0 |
| T | DISTANCE | 40 |
| T | BODYANGLE | 180 |
| A | MAX_STEPS_PER_TRIAL | 4 |
| A | FADE_IN | 0 |
| A | FADE_OUT | 0 |
| A | SEND_PER_STEP | 10 |
| A | MIN_SPEED | 0,5 |
| A | $d(\alpha)$ | 12 |
| A | $d(\beta)$ | 12 |
| A | $d(\gamma)$ | 12 |
| A | $d(v_{ball})$ | 10 |
| A | $d(\delta)$ | 12 |
| A | $d(\epsilon)$ | 36 |
| A | $d(p_{kick})$ | 10 |
| A | Torbreite | 14 |

ist, ist die Diskretisierung der Observationswinkel noch sehr grob und somit ist auch der Zustandsraum selber noch sehr klein. Es gibt insgesamt $12^3 * 10 = 17280$ verschiedene Observationen und $12 + 36 * 10 = 372$ verschiedene Aktionen. Es stellt sich die Frage, ob es mit dieser Diskretisierung

eine gute Performance geben kann und ob diese Performance später auch gehalten werden kann.

**5.2.1.2 Lernperformance** Läßt man für dieses Traininsszenario eine Policy erlernen, so ergibt sich die Lernperformance wie im folgenden Graphen zu sehen ist: Es ist zu erkennen, daß bereits ab etwa 80.000 Zyklen

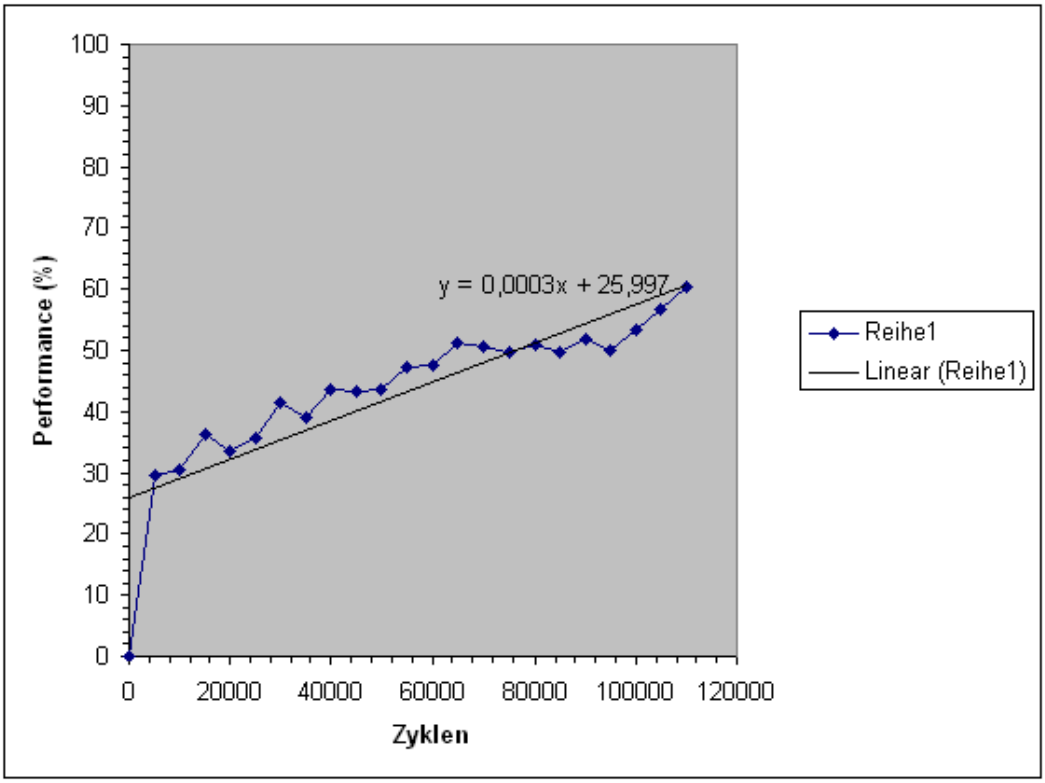
Abbildung 7: Lernperformance der 12x36 Welt - Policy 1




die Policy einen Performancewert von über 50% aufweisen kann. Berechnet man eine lineare Annäherungsfunktion für die Funktion der Performance, so ergibt sie sich zu $y = 0,0002x + 29,732$. Es ist klar, daß diese Funktion offenbar nur für die ersten 100.000 Schritte gilt, da ansonsten bei gleichbleibender Steigung eine Performance von über 100% erreicht werden könnte. Löst man die Gleichung nach x auf, ergibt sich:

$5000 * y - 148660 = x$. Dies würde bedeuten, daß eine Performance von 100% bei $5000 * 100 - 148660 = 351340$ Zyklen erreicht wäre. Aus späteren Trainingsszenarien ist jedoch bekannt, daß eine Performance von 100% nur in sehr gut ausgewählten Trainingsszenarien möglich ist. Um zu überprüfen, ob es bei anderen Policies auch lineare Annäherungsfunktionen gibt, welche im gleichen Trainingsszenario trainiert wurden, soll die Performance einer weiteren Policy (es handelt sich um eine weitere Policy - alle Parameter sind gleich geblieben) betrachtet werden: Wie auch hier zu sehen

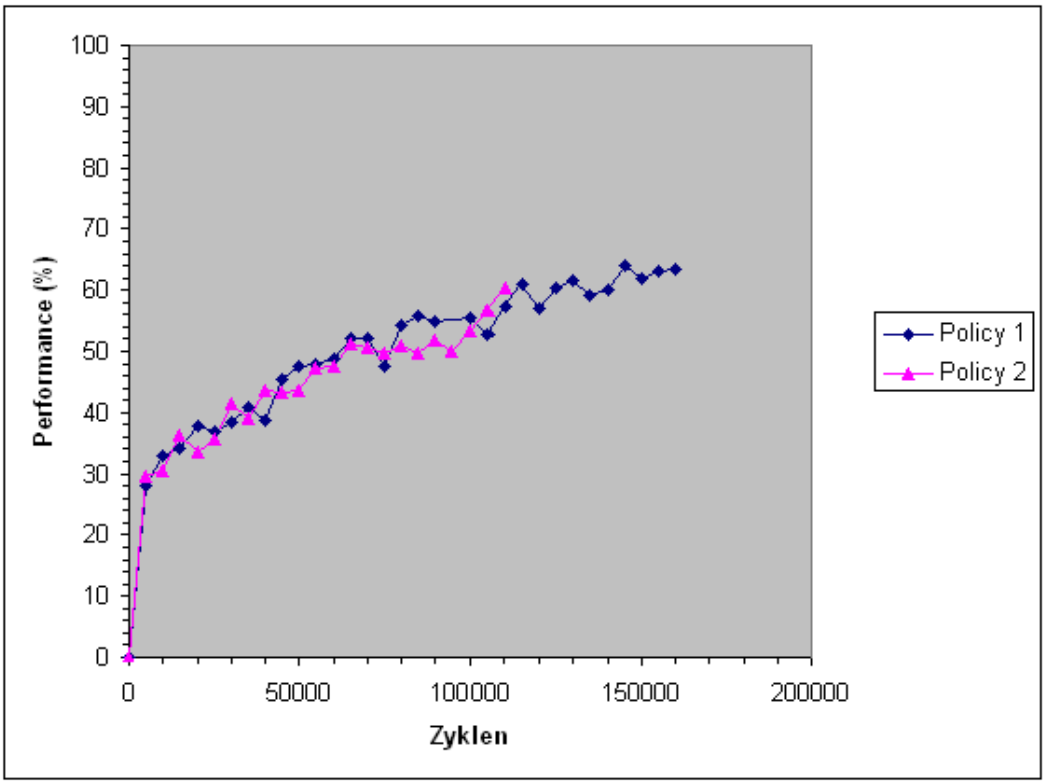
Abbildung 8: Lernperformance der 12x36 Welt - Policy 2




ist, hat die Performance auch in diesem Fall eine lineare Annäherungsfunktion, jedoch ist diese im Vergleich zur letzten Policy bereits etwas steiler. Berechnet man hier die lineare Annäherungsfunktion der Performance so ergibt sich $y = 0,0003x + 25,997$. Beachtlich ist bei beiden Graphen, daß

bei der Performance in beiden Fällen ein linearer Trend zu beobachten ist - es bleibt jedoch fraglich, ob dies bis zu einer 100%en Performance erhalten bleibt. Es ist vermutlich eher anzunehmen, daß die Funktion sich einem Maximalwert annähern wird, welcher zwischen 80 und 100% liegt. Dies ist bisher eine reine Spekulation, jedoch haben andere Testläufe gezeigt, daß der Grenzwert oft im Bereich oberhalb von 90% liegt. Betrachtet man beide

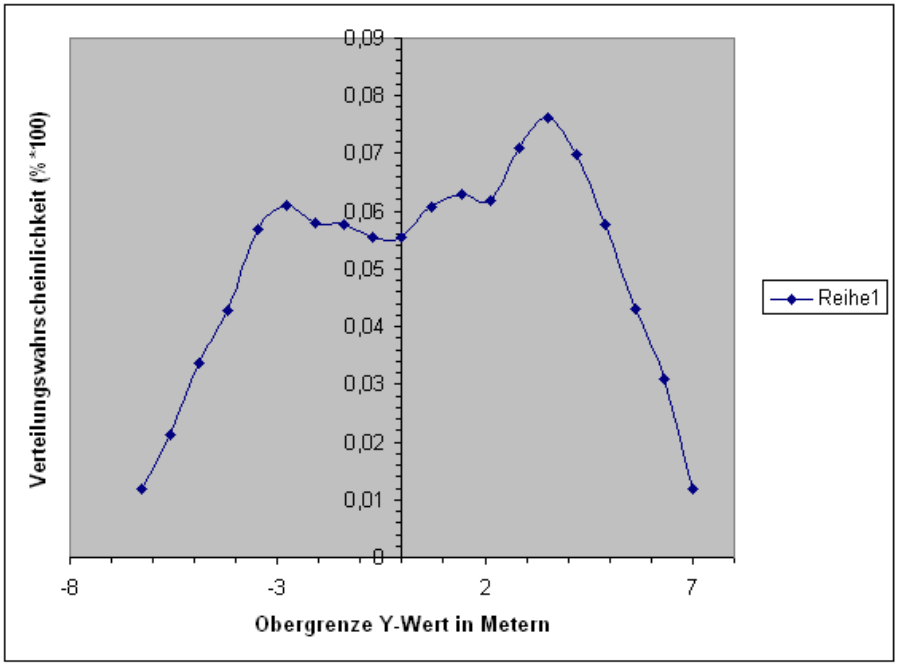
Abbildung 9: Lernperformance 12x36 Welt - Ein Vergleich




Performancegraphen in einem Schaubild, so ist deutlich zu sehen, daß beide Graphen sehr nah beieinander liegen. Es liegt also die Vermutung nahe, daß zumindestens im Bereich bis 200 000 Zyklen die Annäherungsfunktionen linear bleiben.

**5.2.1.3 Verteilungsmessung** Nachfolgend sollen die Verteilungsmessungen der beiden bisher bekannten Policies auf ein 14 Meter breites Tor betrachtet werden. Für die Verteilungsmessung wurden nur solche Schüsse aufgezeichnet, welche auch wirklich das Tor getroffen hätten, d.h. alle Aktionen, welche in einen negativen Endzustand führen, sind nicht in dieser Verteilungsmessung aufgenommen. Auf der X-Achse sind die Y-Werte aufgetragen, welche den Schnittpunkt des Balles mit der Torlinie repräsentiert. Der Wert 0 stellt hier also den Tormittelpunkt dar. Negative Werte stellen den Bereich unterhalb des Tormittelpunktes dar und positive Werte liegen oberhalb des Tormittelpunktes. Bei dieser Verteilung ist zu sehen, daß die Außenbereiche (gemeint sind die Bereiche, die näher am Torpfosten als am Tormittelpunkt liegen) des Tores selten getroffen werden und sich oberhalb vom Tormittelpunkt ein absolutes Treffermaximum herausbildet. Ansonsten ist die Verteilung im Innenbereich des Tores etwa gleich verteilt. Auch bei der Verteilung der zweiten Policy, ist die Wahrscheinlich-

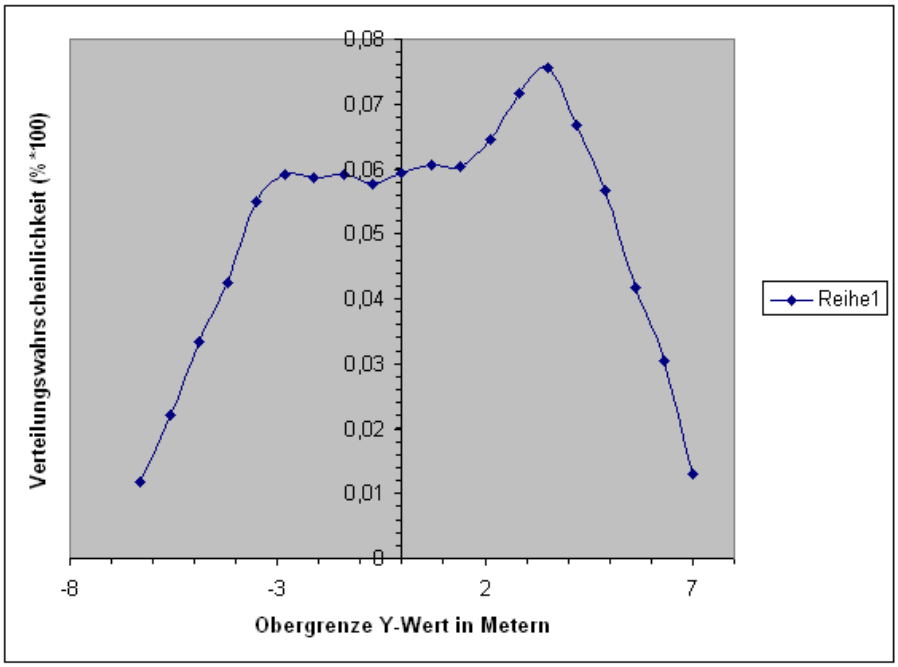
Abbildung 10: Verteilungsmessung der 12x36 Welt - Policy 1




keit für das Treffen der Außenbereiche des Tores eher unwahrscheinlich. Es

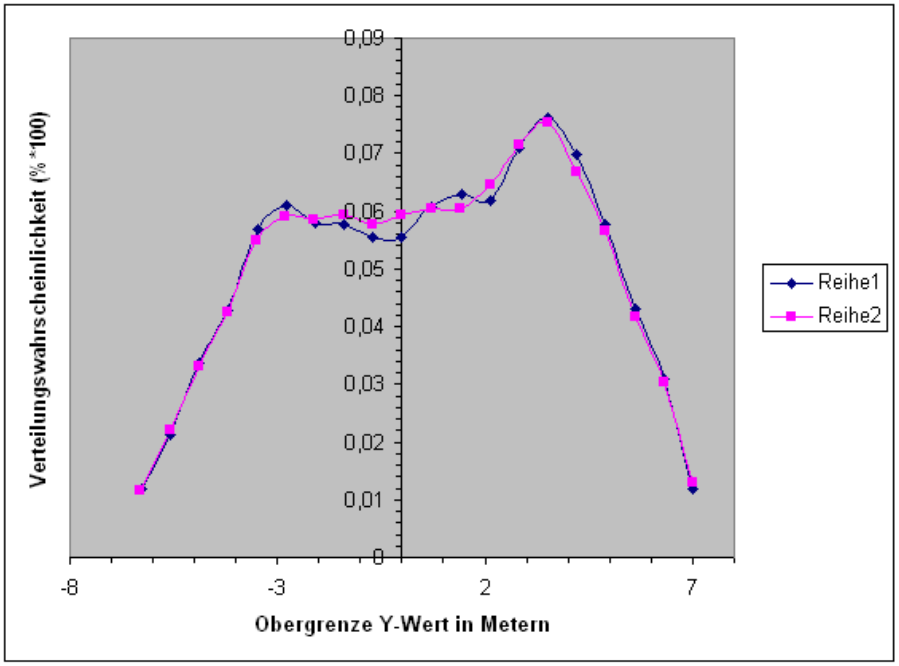
bildet sich auch hier wieder ein absolutes Maximum an einer Stelle, welche überhalb vom Tormittelpunkt liegt, heraus. Ebenfalls herrscht auch hier wieder fast eine Gleichverteilung in den Klassen, welche im Innenbereich des Tores liegen. Vergleicht man die beiden Verteilungsmessungen so fällt

Abbildung 11: Verteilungsmessung der 12x36 Welt - Policy 2



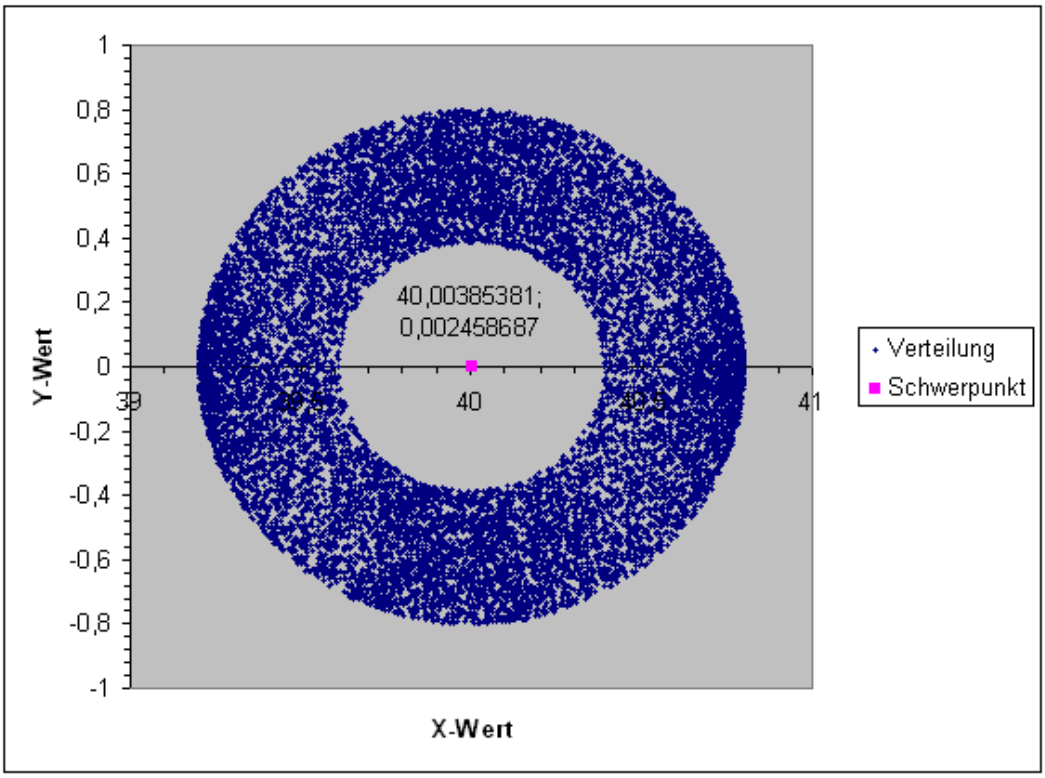

auf, daß sie fast identisch sind. Bei beiden Verteilungen bildet sich ein absolutes Maximum in den inneren Gebieten oberhalb des Tormittelpunktes heraus. Dies kann mehrere Ursachen haben. Eine mögliche Erklärung wäre es, daß die beiden Maxima völlig zufälig entstanden sind - jedoch scheint dies wenig plausibel, da beide Verteilungskurven fast identisch sind.

Abbildung 12: Verteilungsmessung der 12x36 Welt - Ein Vergleich



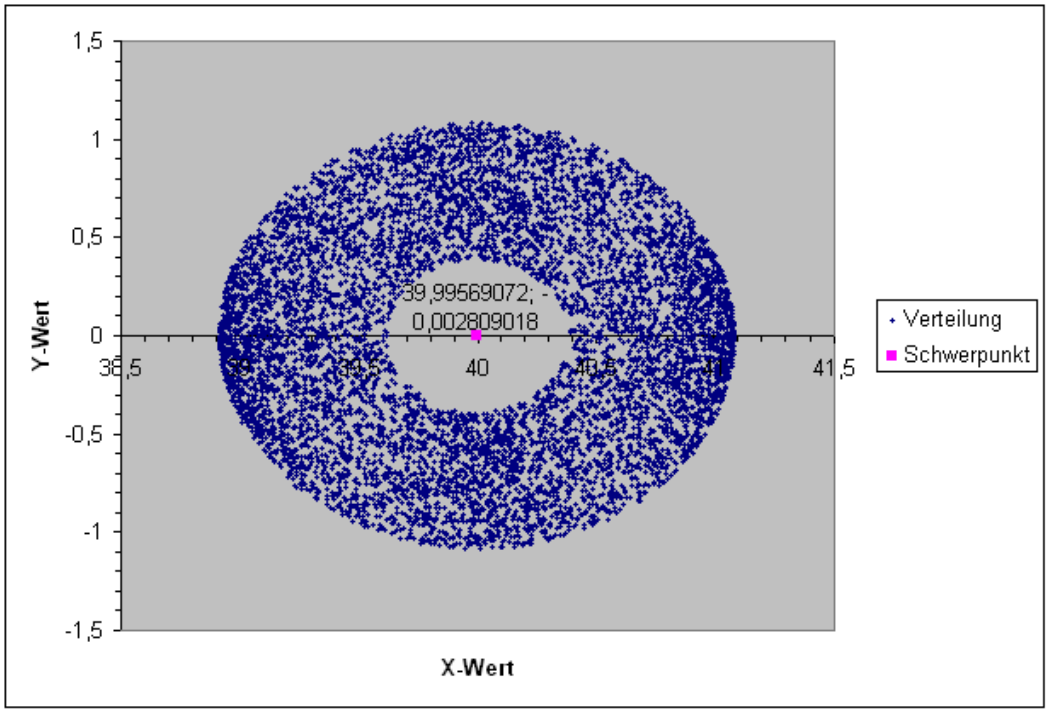

Eine weitere mögliche Erklärung wäre, daß die Verschiebung durch Aktionsketten zustande kommt - dies steht noch zur Prüfung aus.
Zur Aufklärung, ob diese Maxima zufällig entstanden sind, soll ein Blick auf die Startzustandsverteilung die der Trainer setzt geworfen werden. In diesen beiden Graphen wurde aufgezeichnet, wie der Trainer den Agenten auf dem Spielfeld positioniert. Jeder Datenpunkt im dargestellten Graphen entspricht den Koordinaten von jeweils einer Positionierung. Sollte bei der Trainerpositionierung eine Gewichtung in eine bestimmte Richtung feststellbar sein, so wäre dies eine mögliche Erklärung für die Verschiebung der Maxima bei der Verteilungsmessung. Schaut man sich nachfolgend jedoch die beiden wichtigsten Positionierungen an, einmal die Positionierung mit $0,8m$ Radius und einmal mit $1,085m$ Radius fällt auf, daß beide eine Häufung nahe der Y-Nullinie sowohl in Richtung des Tores als auch in Richtung der Mittellinie haben. Zusätzlich hat die Verteilung in Abbildung 13 noch Häufungen im oberen und unteren Bereich des Positionierungsringes. Bei beiden Positionierungsgraphen ist der Massenschwerpunkt der Messpunk-

Abbildung 13: Verteilung der Startzustände, max 0,8 Meter rund um den Agenten




te jedoch jeweils genau in der Mitte des Ringes (dort wo der Agent steht), d.h. es sollte keine Bevorzugung der Zufallszahlen in eine bestimmte Richtung geben. Damit ist praktisch ausgeschlossen, daß sich die Verschiebung beim Verteilungsgraphen zufällig ergeben hat. Weitergehend wurde untersuch, ob die gelernten Aktionen der beiden Policies einen positiven oder negativen Winkelwert bevorzugen. Bei den Policies stellte sich heraus, daß Aktionen mit positivem Winkel bevorzugt werden (Policy 1: positive Aktionen 1158, nagative 1036, Policy 2: positive Aktionen 1039, nagative 892). Dies kann eine mögliche Erklärung für das verschobene Maximum bei der Verteilungsmessung sein. Andererseits muß dies keine Erklärung sein, da

Abbildung 14: Verteilung der Startzustände, max 1,085 Meter rund um den Agenten




die Aktionen genau so angeordnet sein können, daß sie beim Treffen ins Tor eine Gleichverteilung erzeugen.

### 5.2.2 Die 12x36 Welt - Teil 2

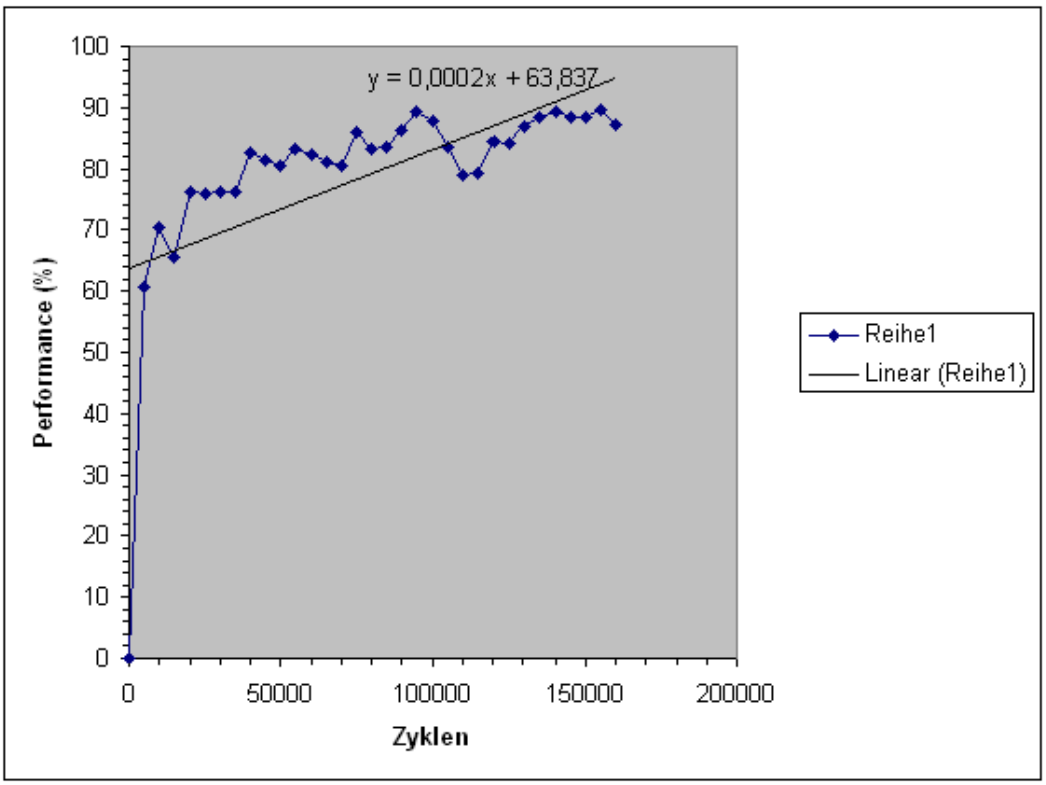
Abbildung 15: Referenzparameter der 12x36 Welt - Teil 2

| Trainer / Agent | Parameter | Wert |
|---|---|---|
| T | STEPS_PER_TRIAL | 4 |
| T | FADE_IN | 0 |
| T | FADE_OUT | 0 |
| T | RADIUS | 1,085 |
| T | MIN_RADIUS | 0,386 |
| T | MAX_VELOCITY | 0 |
| T | DISTANCE | 40 |
| T | BODYANGLE | **45** |
| A | MAX_STEPS_PER_TRIAL | 4 |
| A | FADE_IN | 0 |
| A | FADE_OUT | 0 |
| A | SEND_PER_STEP | 10 |
| A | MIN_SPEED | 0,5 |
| A | $d(\alpha)$ | 12 |
| A | $d(\beta)$ | 12 |
| A | $d(\gamma)$ | 12 |
| A | $d(v_{ball})$ | 10 |
| A | $d(\delta)$ | 12 |
| A | $d(\epsilon)$ | 36 |
| A | $d(p_{kick})$ | 10 |
| A | Torbreite | 14 |

**5.2.2.1 Die Referenzparameter** Wie zu sehen ist, hat sich bei diesem Szenario nur etwas an der Positionierung des Trainiers geändert. Es wird der Körperwinkel vom Agenten in den Startzuständen nur jeweils um 45 Grad in beiden Richtungen gedreht - somit fällt ein Großteil der Startzustände weg. Mit den Startzuständen fallen auch einige Observationen weg, welche nun nur noch sehr unwahrscheinlich besucht werden. Dies ist darauf zurückzuführen, daß einige Observationen nur durch Aktionsketten erreicht werden können (jene z.B. in denen der Ball eine Geschwindigkeit

hat). Bei diesem Parameterset sind jedoch nur Aktionsketten bis zu maximal 4 Aktionen möglich.

Abbildung 16: Lernperformance der 12x36 Welt - Policy 3




**5.2.2.2 Lernperformance** Es läßt sich zwar auch hier eine lineare Annäherungsfunktion bestimmen zu $y = 0,0002x + 63,837$. Unschwer zu erkennen ist jedoch, daß der Graph sich einem Grenzwert von etwa 90% annähert, anstatt weiter zu steigen. An früherer Stelle wurde ja schon einmal gesagt, daß gute Performancewerte zwischen 90 und 100% schwanken können, dies ist ein Beispiel für einen guten Performancewert. An diesem Graphen läßt sich gut zeigen, was eine Verkleinerung der möglichen Startzustände und der damit einhergehenden Verkleinerung der möglichen Observationen, für eine Verbesserung der Performance bringt. Dies

rührt daher, daß der Agent in diesem Fall wesentlich weniger Observation-Aktionspaare lernen muß, was durch die eingeschränkten Startzustände zustande kommt. In diesem Fall kann der Agent sehr früh die Policy optimieren - dies passiert immer dann wenn die gleichen Observationen, obwohl sie schon bekannte Aktionen haben, immer wieder durchlaufen werden. Die Optimierung kommt aus folgender Tatsache zustande: Dadurch, daß eine Observation mehrere Zustände auf sich abbildet, kann es dazu kommen, daß eine Aktion in einer Observation nicht immer in die gleiche Folgeobservation übergeht. Dieser Tatsache ist es zu verdanken, daß viele bereits schon gelernte Aktionen mehrer Male verlernt werden müssen, da es in der betreffenden Observation eine bessere Aktion gibt. Eine Optimierung kann nur dann stattfinden, wenn die Diskretisierung der Aktionen fein genug ist, so daß es für jeden Zustand innerhalb einer Observation mindestens eine Aktion gibt, die das System sofort, oder durch mögliche Nachfolgeaktionen in einen positiven Endzustand bringt. Nur wenn diese Tatsache in jeder möglichen Observation gegeben ist, kann eine Performance von 100% erreicht werden.

**5.2.2.3 Vergleich der Performancewerte** Um zu verdeutlichen, welche Auswirkungen eine Verkleinerung der Startzustände auf die Policyperformance haben kann, sollen alle 3 bisher vorgestellten Performancewerte in einem Graphen (Abbildung 17) verglichen werden. Wie unschwer zu erkennen ist, ist die Performance der dritten Policy (jene mit eingeschränktem Körperwinkel) von Anfang an um fast 20 Prozentpunkte besser als die beiden Performancewerte der Policies mit uneingeschränktem Startzustandsraum. Die Performance der dritten Policy scheint sich bei circa 89-90% einzupendeln.

**5.2.2.4 Lerneffektivität** Um zu prüfen, ob sich der höhere Performancewert der dritten Policy auf die gelernten Aktionen auswirkt, sollen nun von allen drei Policies die Lerneffektivitäten betrachtet werden. Es ist hier ganz klar zu erkennen, daß die hohe Performance eine immense Auswirkung auf die Lerneffektivität hat. So sind in den beiden Policies, welche einen vollen Startzustandsraum haben, nach 110.000 Zyklen bereits 1871 bzw 1901 Observation- Aktionspaare bekannt, der dritten Policy hingegen nur 706. Startet man ein Traininsszenario mit eingeschränkten Anfangszuständen so kann man gut im voraus berechnen, welche maximale Per-

Abbildung 17: Lernperformance der 12x36 Welt - Ein Vergleich

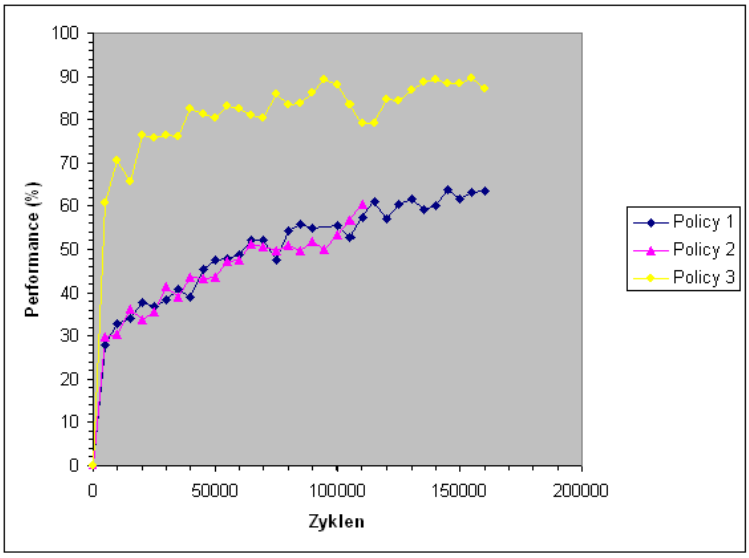
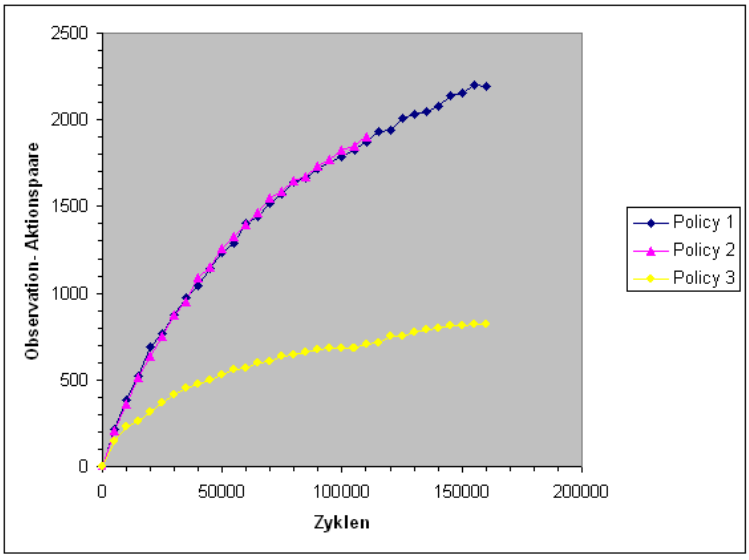



Abbildung 18: Lerneffektivität der 12x36 Welt - Ein Vergleich



formance das komplette Szenario maximal haben wird. Trotzdem sollte bedacht werden, daß es sich dabei um eine Vorhersage handelt, es gibt keine Garantie darüber, daß die endgültige Policy dem vorausberechneten Performancewert wirklich entgegenstrebt.

### 5.2.3 Die 24x72 Welt - Teil 1

In den letzten beiden Trainingsszenarien wurde untersucht, wie sich die Lernperformance und die Effektivität ändert, wenn die Diskretisierung gleich bleibt und nur die Startzustände begrenzt werden. Es sollte untersucht werden, in wie weit sich die Performance und die Effektivität ändern, wenn die Diskretisierung sowohl der Observationswinkel als auch der Aktionswinkel verfeinert wird. Zusätzlich zu der Änderung der Diskretisierung wurde das Tor auf 30 Meter verbreitert.

#### 5.2.3.1 Die Referenzparameter

**5.2.3.2 Lernperformance** Ähnlich den vorangegangenen Trainingsszenarien, läßt sich auch hier eine lineare Annäherungsfunktion berechnen, deren genauen Wert man Abbildung 20 entnehmen kann. Die Linearität der Annäherungsfunktion gilt jedoch nur in dem dargestellten Wertebereich des Graphen. Der eigentliche Graph steigt am Anfang sehr schnell und pendelt sich dann aber recht bald bei 90% Performance ein. Die Schwankungen haben dabei meistens die folgende Ursache: Wie bereits früher schon einmal erwähnt, bildet eine Observation mehrere Zustände auf sich ab. Es können nun Zustände getestet werden, die zwar in der gleichen Observation liegen wie der Zustand, für den die Aktion gespeichert wurde - wendet man auf diesen Zustand jedoch die gespeicherte Aktion an, so kann es dazu kommen, daß man in eine andere Folgeobservation kommt und somit in einen negativen Endzustand. Tritt dies häufig auf, so geht die Performance in diesem Messpunkt nach unten. Ein Indiez für eine nahezu optimale Policy ist demzufolge ein Performancegraph, der keine Schwankungen mehr hat - er kennt für jede Observation eine Aktion, welche so gut ist, daß sie nahezu immer in die gleiche Folgeaktion führt.

**5.2.3.3 Lerneffektivität** Es folgt nun ein Graph, welcher den Lernerfolg visualisiert.

Abbildung 19: Referenzparameter der 24x72 Welt

| Trainer / Agent | Parameter | Wert |
|---|---|---|
| T | STEPS_PER_TRIAL | 4 |
| T | FADE_IN | 0 |
| T | FADE_OUT | 0 |
| T | RADIUS | 1,085 |
| T | MIN_RADIUS | 0,386 |
| T | MAX_VELOCITY | 0 |
| T | DISTANCE | 40 |
| T | BODYANGLE | 180 |
| A | MAX_STEPS_PER_TRIAL | 4 |
| A | FADE_IN | 0 |
| A | FADE_OUT | 0 |
| A | SEND_PER_STEP | 10 |
| A | MIN_SPEED | 0,5 |
| A | $d(\alpha)$ | 24 |
| A | $d(\beta)$ | 24 |
| A | $d(\gamma)$ | 24 |
| A | $d(v_{ball})$ | 10 |
| A | $d(\delta)$ | 72 |
| A | $d(\epsilon)$ | 72 |
| A | $d(p_{kick})$ | 10 |
| A | Torbreite | 30 |

Wieder einmal mehr läßt sich beobachten, daß auch hier die Kurve am besten durch eine lineare Annäherungsfunktion beschrieben wird. Ihr exakter Wert kann dem Diagramm entnommen werden. Es sei zunächst festgehalten, daß es insgesamt $d(\alpha) * d(\beta) * d(\gamma) * d(v_{ball})$ Observationen gibt, also $24 * 24 * 24 * 10 = 138240$ Observationen. Geht man von der Annahme aus, daß von allen Observationen aus einer der positiven Endzustände mit genau einer Aktion erreichbar ist und der Ball im Anfangszustand keine Ausgangsgeschwindigkeit hat, so reduziert sich der Observationraum auf $d(\alpha) * d(\beta) * d(\gamma)$ was $24 * 24 * 24 = 13824$ entspricht. Dies ist der Fall, weil die Observationen, in denen der Ball eine Geschwindigkeit und einen Bewegungsvektor haben, nicht besucht werden und somit diese beiden Pa-

Abbildung 20: Lernperformance der 24x72 Welt

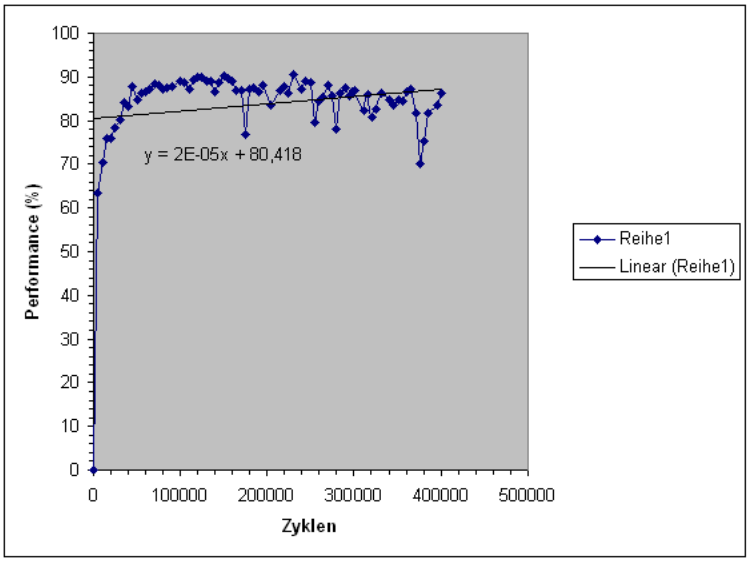
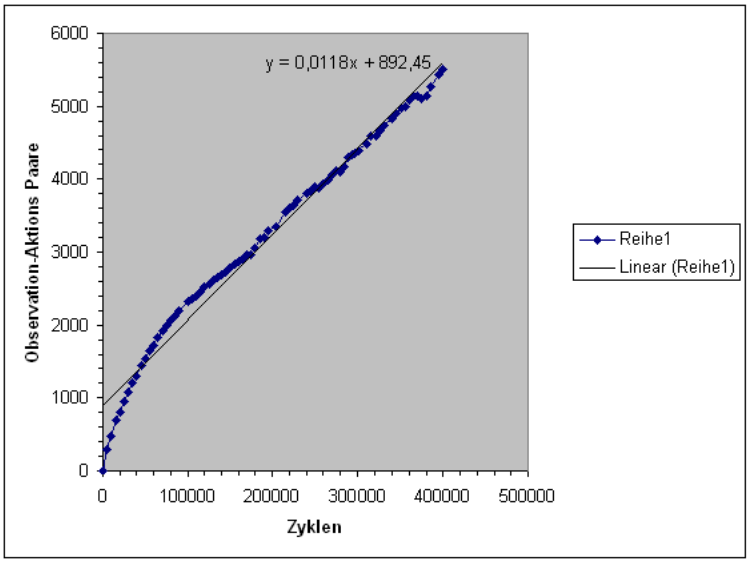



Abbildung 21: Lernereffektivität der 24x72 Welt



rameter mit jeweils dem Wert $1$ in die Berechnung eingehen. Schaut man sich den Lernerfolg unter der Prämisse des eingeschränkten Observationenraumes an, so fällt auf, daß bereits nach 400 000 Zyklen fast 50% der Observationen welche überhaupt besucht werden können, eine zugeordnete Aktion haben. Berechnet man über die Annäherungsfunktion die Zyklenzahl, bei der alle Observationen eine passende Aktion haben, so kommt man auf $(13824 - 892,45)/0,0188 = 687848$ Zyklen.

### 5.2.4 Die 24x72 Welt - Teil 2

Abbildung 22: Die Referenzparameter der 24x72 Welt - Teil 2

| Trainer / Agent | Parameter | Wert |
|---|---|---|
| T | STEPS_PER_TRIAL | 4 |
| T | FADE_IN | 0 |
| T | FADE_OUT | 0 |
| T | RADIUS | 1,085 |
| T | MIN_RADIUS | 0,386 |
| T | MAX_VELOCITY | 0 |
| T | DISTANCE | 40 |
| T | BODYANGLE | 180 |
| A | MAX_STEPS_PER_TRIAL | 4 |
| A | FADE_IN | 0 |
| A | FADE_OUT | 0 |
| A | SEND_PER_STEP | 10 |
| A | MIN_SPEED | 0,5 |
| A | $d(\alpha)$ | 24 |
| A | $d(\beta)$ | 24 |
| A | $d(\gamma)$ | 24 |
| A | $d(v_{ball})$ | 10 |
| A | $d(\delta)$ | 72 |
| A | $d(\epsilon)$ | 72 |
| A | $d(p_{kick})$ | 10 |
| A | Torbreite | 14 |

**5.2.4.1 Die Referenzparameter** Wie man sehen kann, wurde im Gegensatz zum vorherigen Szenario nur das Tor verkleinert auf die Größe, die es eigentlich im Spiel hat.

**5.2.4.2 Lernperformance** Gegenüber dem Szenario mit dem breiteren Tor scheint sich in diesem Fall die Performance auf circa 85% einzupendeln. Es ist durchaus möglich, daß durch weitere Policyoptimierung die Performance noch weiter nach oben geht, nur in diesem Graphen ist dieser Trend nicht zu beobachten. Vergleicht man die beiden Performancewerte,

Abbildung 23: Lernperformance der 24x72 Welt - Teil 2

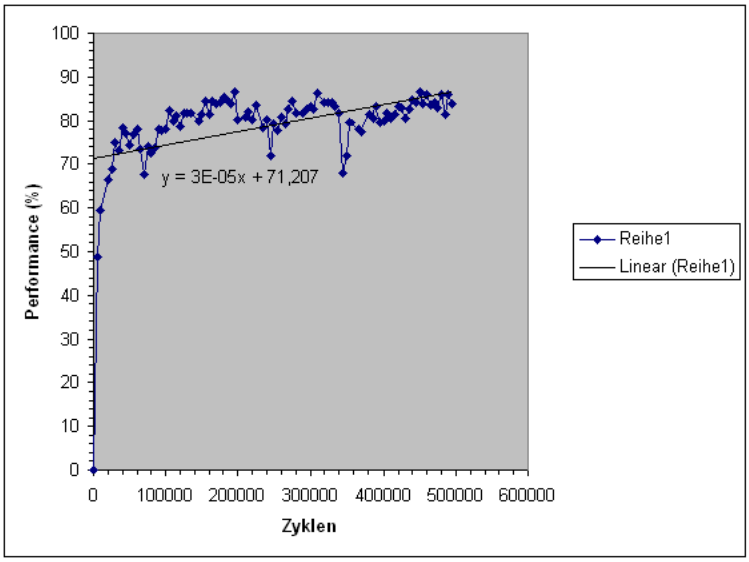
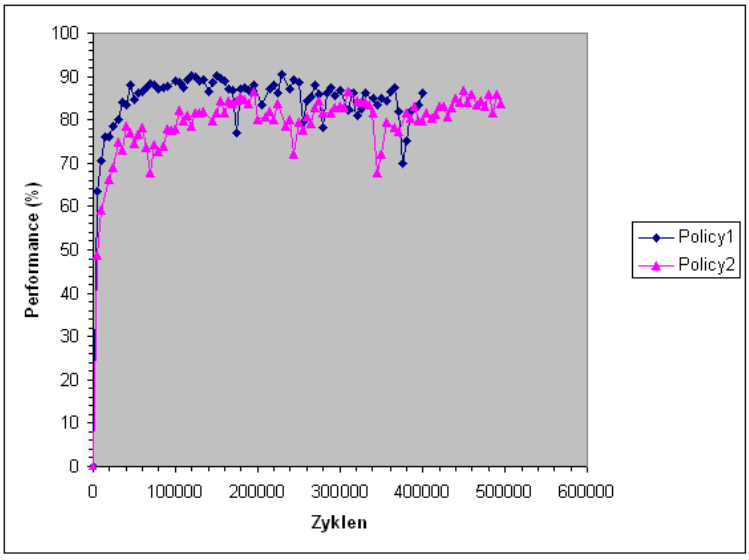
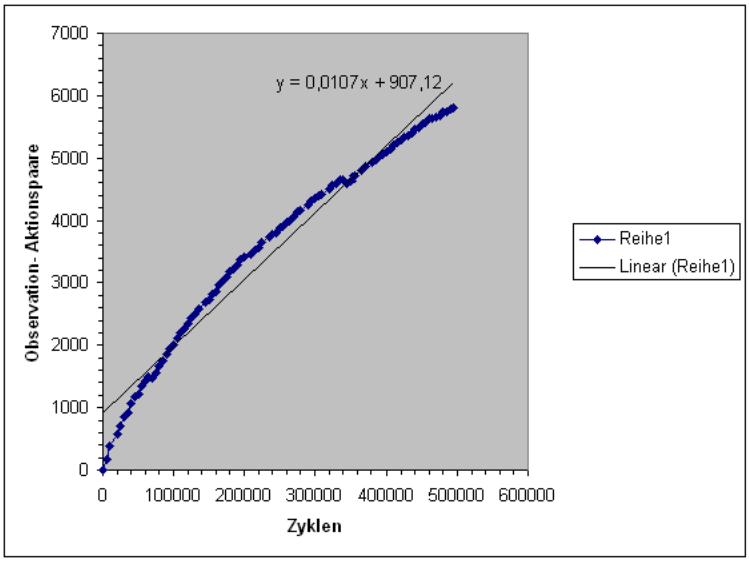



Abbildung 24: Lernperformance der 24x72 Welt - Ein Vergleich



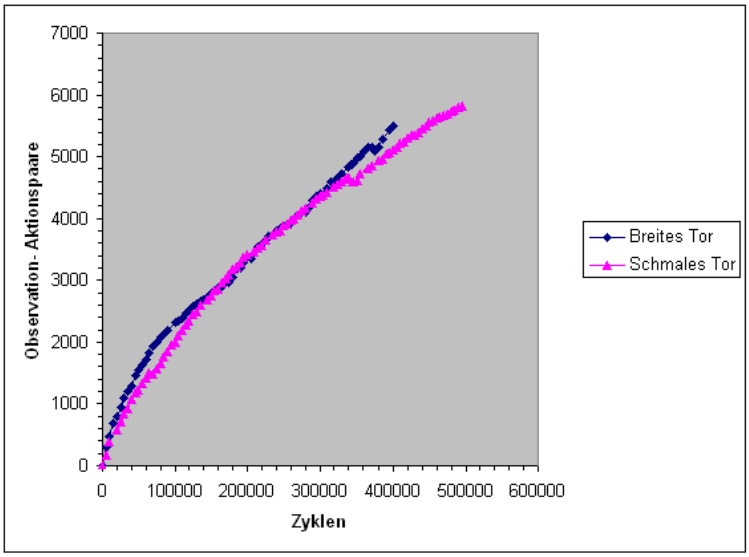
so sieht man, daß die Policy (Policy1), welche auf ein größeres Tor trainiert und getestet wurde, augenscheinlich eine bessere Performance hat. Dies verwundert nicht, da durch das größere Tor die Policy fehlertoleranter wird. Die höhere Fehlertoleranz ergibt sich aus der Tatsache, daß der Agent nicht so gute Aktionen lernen muß. Dies liegt begründet in der Tatsache, daß bei einem vergrößerten positiven Endzustandsraum für eine Observation mehrere Aktionen in einen Endzustand überführen können und die Wahrscheinlichkeit eine dieser Aktionen zu finden höher ist.

Abbildung 25: Lerneffektivität der 24x72 Welt - Teil 2




**5.2.4.3 Lerneffektivität** Auch in diesem Fall kann eine lineare Annäherungsfunktion berechnet werden. Wie jedoch auf dem Schaubild zu sehen ist, wird sich die eigentliche Lernkurve einem Maximalwert annähern, denn es läßt sich bereits gut erkennen, daß die Steigung der Funktion mit fortschreitender Zyklenzahl abnimmt. Dies ist nicht verwunderlich, da die maximale Observationzahl, wie weiter oben bereits erklärt bei circa 13.000 Observationen liegt und somit bereits circa 45% aller möglichen Observationen eine bekannte Aktion haben und das bereits nach 50.000 Zyklen. Es bleibt festzustellen, daß die Lerneffektivität bei der Policy mit dem schmaleren Tor nicht so hoch ist, als bei der Policy mit dem breiteren Tor.

Abbildung 26: Lerneffektivität der 24x72 Welt - Ein Vergleich




In Abbildung 26 ist zu sehen, daß die Lerneffektivität bei breitem Tor höher ist als bei schmalem Tor, bis auf den Bereich zwischen 150.000 - 300.000 Zyklen, wo beide gleich auf liegen.

### 5.2.5 Die 4x4 Welt - Ein Vergleich mit dem Q-Learning

In diesem Abschnitt soll die sogenannte 4x4 Welt vorgestellt werden mit ihren Besonderheiten und weiterhin soll ein Vergleich mit einem Q-Learning Szenario angestellt werden, welches das gleiche Problem lösen sollte.

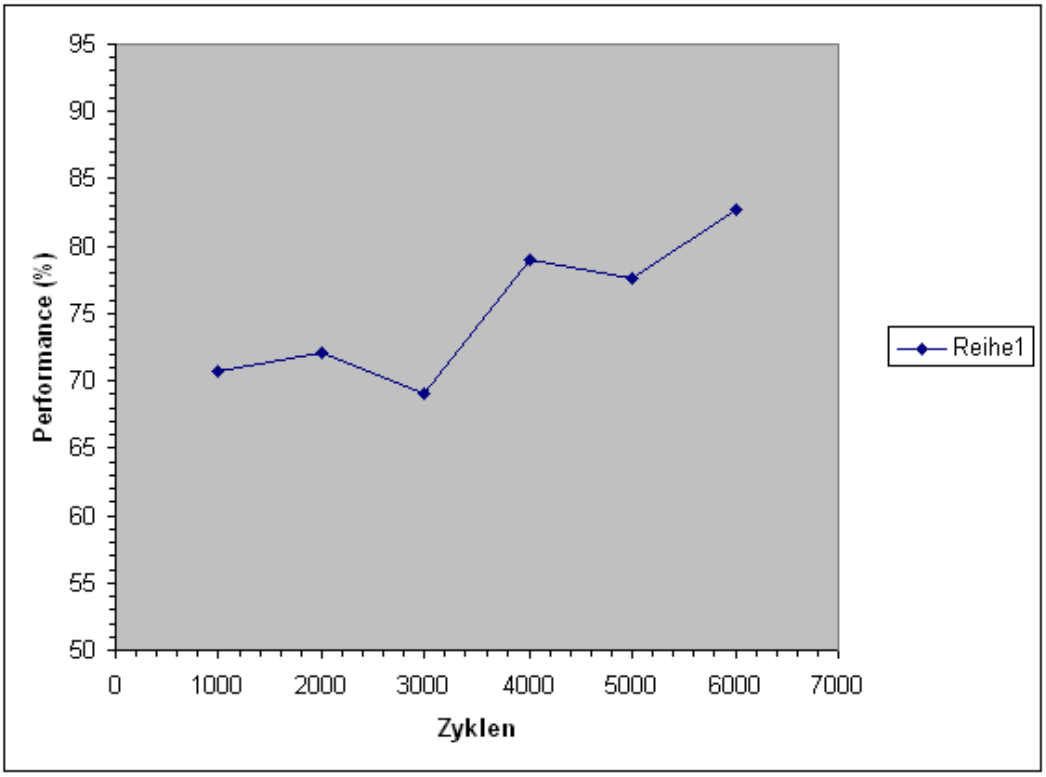
**5.2.5.1 Die Referenzparameter** Wie zu sehen ist (Abbildung 27), sind alle möglichen Winkel in diesem Szenario auf genau 4 Zustände beschränkt, welche jeweils den vier Sektoren in einem Koordinatensystem entsprechen. Eine Eigenart bei diesem Szenario ist, daß der Trainer den Agenten und den Ball immer auf feste Startzustände setzt. Daraus folgt, daß es genau 16 feste Startzustände in diesem Szenario für den Agenten gibt (4 feste Positionen für den Körperwinkel und den Ballwinkel, Geschwindigkeit und Geschwindigkeitsvektor sind nicht vorhanden $4 * 4 * 1 * 1 = 16$). Hieraus wiederum folgt, daß dieses Szenario deterministisch

Abbildung 27: Die Referenzparameter der 4x4 Welt

| Trainer / Agent | Parameter | Wert |
|---|---|---|
| T | STEPS_PER_TRIAL | 6 |
| T | FADE_IN | 0 |
| T | FADE_OUT | 0 |
| T | RADIUS | 1,085 |
| T | MIN_RADIUS | 0,386 |
| T | MAX_VELOCITY | 0 |
| T | DISTANCE | 40 |
| T | BODYANGLE | 180 |
| A | MAX_STEPS_PER_TRIAL | 4 |
| A | FADE_IN | 0 |
| A | FADE_OUT | 0 |
| A | SEND_PER_STEP | 10 |
| A | MIN_SPEED | 0,5 |
| A | $d(\alpha)$ | 4 |
| A | $d(\beta)$ | 4 |
| A | $d(\gamma)$ | 4 |
| A | $d(v_{ball})$ | 10 |
| A | $d(\delta)$ | 4 |
| A | $d(\epsilon)$ | 4 |
| A | $d(p_{kick})$ | 10 |
| A | Torbreite | 50 |

sein sollte, da eine Observation in diesem Fall genau einem Zustand entspricht. Das ganze Szenario hat insgesamt einen Observationsraum von $4^3 * 10 = 640$ Observationen, wovon wieder nicht alle besucht werden (die Gründe wurden früher schon mehrfacht genannt). Um die grobe Diskretisierung aufzufangen, wurde das Szenario auf ein 50 Meter breites Tor trainiert. Da die Episodendauer in diesem Szenario auf 4 beschränkt ist und es genau 16 Startzustände gibt, sollte es maximal möglich sein $16 * 4 = 64$ Aktionen zu lernen, *wenn* das Szenario *deterministisch* sein sollte.

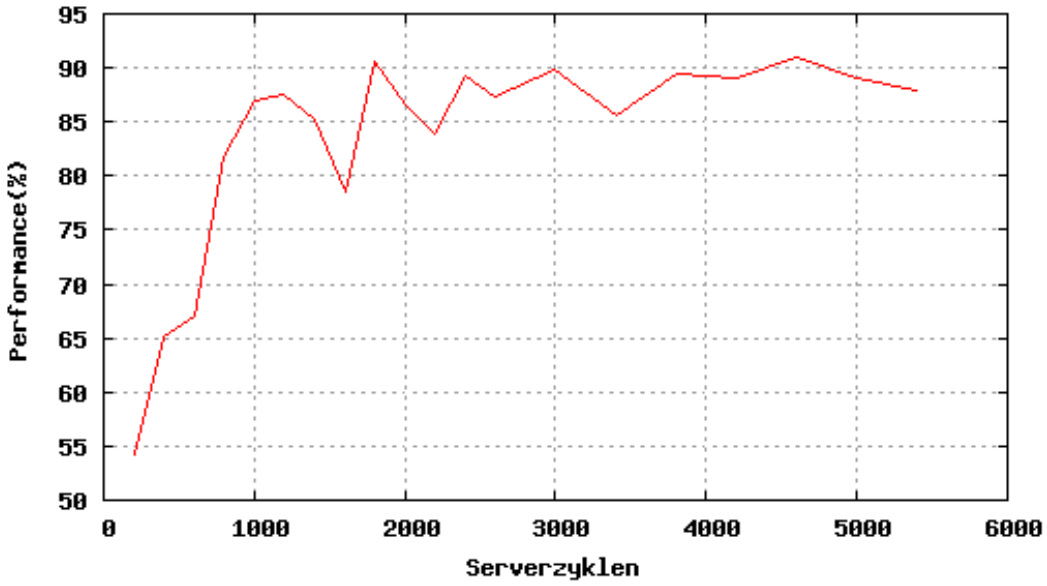
Abbildung 28: Lernperformance der 4x4 Welt - Mit Observation based RFL




**5.2.5.2 Lernperformance** Es ist hier zu erkennen, daß die Performance nach 6000 Zyklen bei 82% angekommen ist, der Performancegraph jedoch weiterhein einen steigenden Trend hat. So ist in Anlehnung an die nächste

Abbildung anzunehmen, daß die Performance auch weiterhin noch steigen wird. In Abbildung 29 ist der Performancegraph des gleichen Szenari-

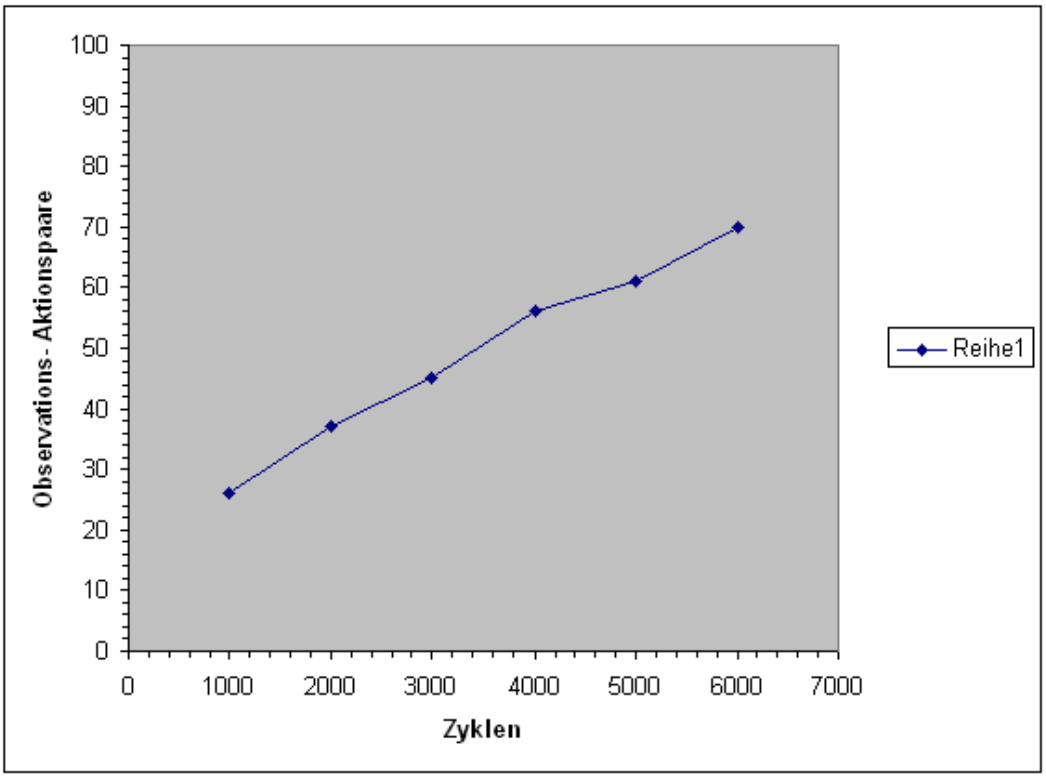
Abbildung 29: Lernperformance der 4x4 Welt - Mit Q-Learning




os zu sehen. Jedoch wurde hier ein Q-Learning Verfahren zum Trainieren des Agenten eingesetzt [CDoeven]. Es ist gut zu erkennen, daß bei dem zweiten Verfahren die Lernperformance schon bei knapp 1000 Zyklen die 82% überschreitet. Die Performance scheint sich hierbei knapp 90% einzuschwingen. Ob die Performance beim Observation based RFL sich gleich verhält, läßt sich nach dem aufgezeichneten Zeitraum noch nicht abschätzen, jedoch deutet die positive Steigung der Kurve darauf hin. Zur weiteren Untersuchung, warum die Performance beim Q-Learning so wesentlich besser ist, soll nun auf die Lerneffektivität der beiden Lernverfahren geblickt werden.

**5.2.5.3 Lerneffektivität** Hier ist zu erkennen, daß mit dem Observation based RFL nur 70 Aktionen gelernt sind nach 6000 Zyklen. Obgleich es sehr wenige Aktionen sind, können diese alle Startpositionen in einen

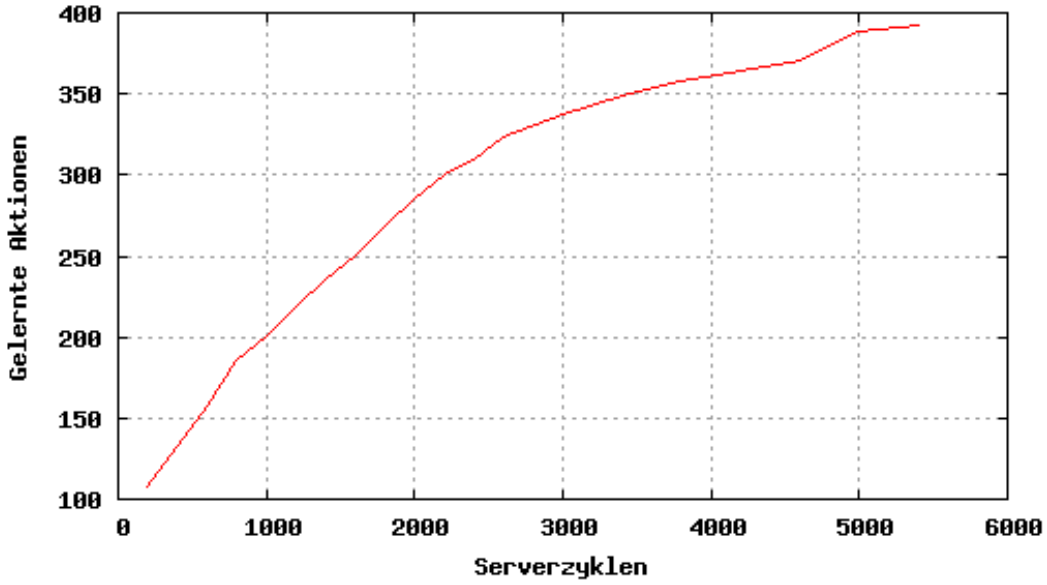
Abbildung 30: Lerneffektivität der 4x4 Welt - Mit Observation based RFL




Endzustand überführen, da es wie in der Einleitung beschrieben, in diesem Szenario nur 16 Startzustände gibt. Bei 70 insgesamt gelernten Aktionen muß es also zur Bildung von Aktionsketten gekommen sein - da, wenn dies nicht der Fall wäre, es genau 16 gelernte Aktionen geben dürfte - für jeden Anfangszustand genau eine Aktion, die in den Endzustand überführt. Durch die Begrenzung von $MAX_STEPS_PER_TRIAL$ auf $4$ ist es pro Anfangszustand nur möglich eine Aktionskette der maximalen Länge $4$ zu lernen, in diesem Fall wären das $16 * 4 = 64$ Aktionen. Gelernt wurden jedoch 70 Aktionen - wie kann dies sein? Es wurde bereits vermutet, daß dieses Szenario deterministisch sein soll. Das würde bedeuten, daß bei Ausführung einer Aktion in einer bestimmen Observation das System immer in die gleiche Folgeobservation übergeht. Ist es jedoch nichtdeterministisch,

so kann die Ausführung der gleichen Aktion in einer bestimmten Observation in mehrere verschiedene Folgeobservationen übergehen. Man kann sich diese Tatsache wie eine Weiche vorstellen: Unter bestimmten Vorraussetzungen führt es in eine Observation unter anderen Vorraussetzungen führt es in eine andere Observation. Erst durch diese Tatsache wird es möglich, hier mehr als 64 Aktionen zu lernen. Im Vergleich dazu kennt die Policy des

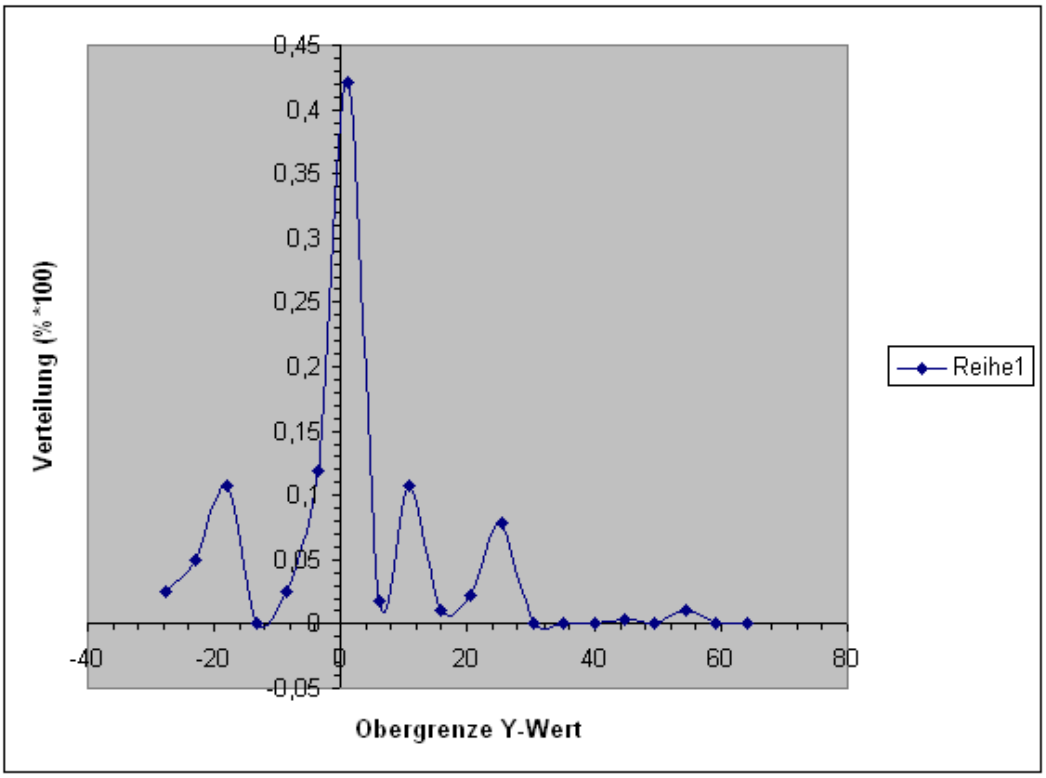
Abbildung 31: Lerneffektivität der 4x4 Welt - Mit Q-Learning




Q-Learning nach dem gleichen Lernzeitraum bereits knapp 400 Aktionen. Durch die Tatsache, daß auch hier viele Aktionen gelernt wurden, wird die oben geäußerte These, nämlich daß dieses Szenario nichtdeterminisitisch ist, untermauert.

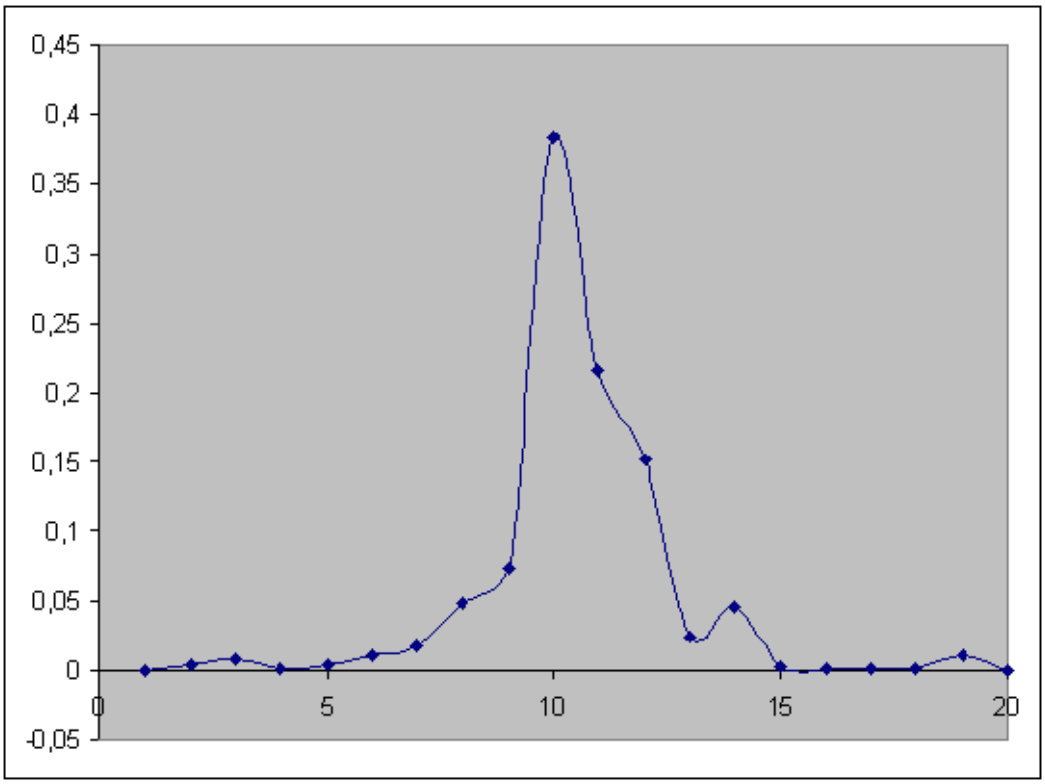
**5.2.5.4 Verteilungsmessung** Die Verteilungsmessung (Abbildung 32 und 33) wurde auf ein 100 Meter breites Tor aufgezeichnet. In der Verteilungsmessung ist zu erkennen, daß sich 3 lokale Maxima und 1 abolutes Maximum zwischen $+30$ und $-30$ Meter bilden, was genau der Größe des ursprünglichen Tores entspricht. In der Mitte des Tores ist eine Trefferchance von über 40% verzeichnet. Weiterhin ist eine leichte Verteilungs-

Abbildung 32: Verteilungsmessung der 4x4 Welt - Mit Observation based RFL




erhebung bei knapp $54m$ oberhalb des Tormittelpunktes zu sehen. Ähnlich wie bei der Verteilungsmessung des Observation based RFLs hat die Messung des Q-Learnings ihr absolutes Maximum auch im Mittelpunkt des Tores. Dort trifft der Ball mit circa 40%er Wahrscheinlichkeit auf. Ebenfalls ist auch hier wieder eine Verteilungshäufung bei knapp $50m$ oberhalb des Tormittelpunktes auszumachen. Diese Häufung ist bei dieser Verteilungsmessung auch auf der anderen Seite, also bei $50m$ (-50m ist in diesem Fall 0 und +50m ist 20) unterhalb des Tormittelpunktes zu sehen.

Abbildung 33: Verteilungsmessung der 4x4 Welt - Mit Q-Learning




### 5.2.6 Die 24x72 Welt - Ein Vergleich mit Q-Learning

In diesem Abschnitt soll das früher bereits vorgestellte 24x72 Szenario verglichen werden, mit einem gleichwertigen Szenario, in welchem mit Q-Learning gelernt wurde.

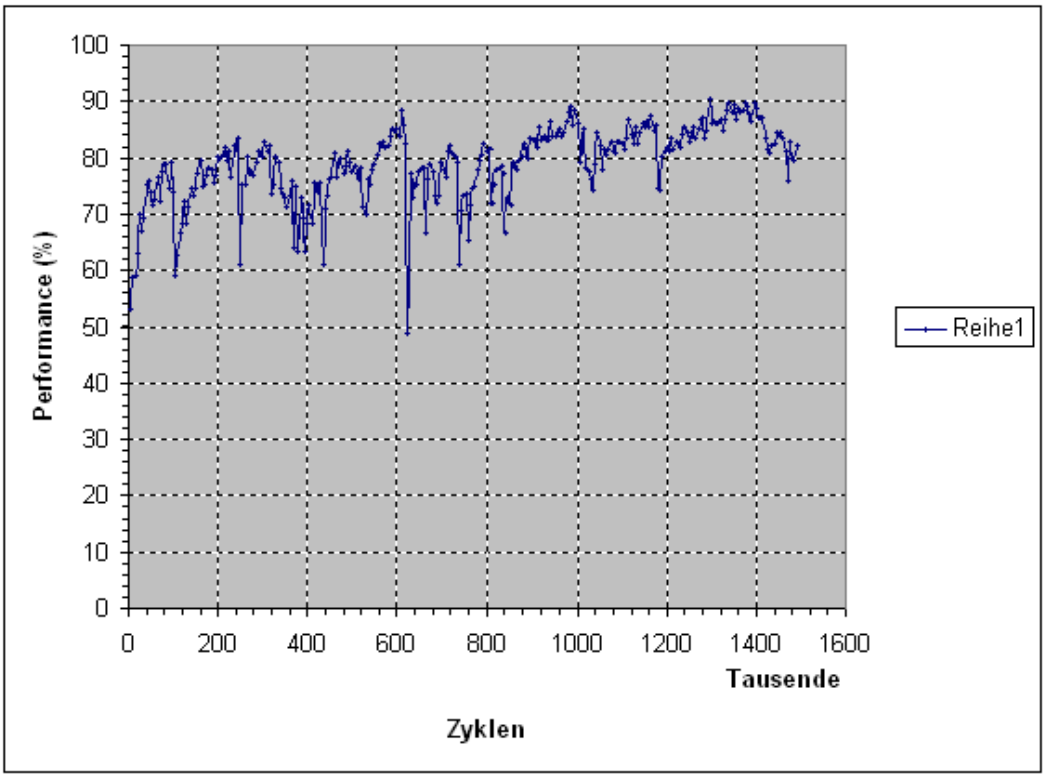
#### 5.2.6.1 Die Referenzparameter

**5.2.6.2 Lernperformance** Es ist gut zu sehen, daß bereits nach 200.000 Zyklen die Performance bei etwa 80% liegt. Sie steigt weiter, bis sie sich nach etwa 1,5 Millionen Zyklen zwischen 85 und 90% einschwingt. Blickt

Abbildung 34: Die Referenzparameter der 24x72 Welt

| Trainer / Agent | Parameter | Wert |
|---|---|---|
| T | STEPS_PER_TRIAL | 4 |
| T | FADE_IN | 0 |
| T | FADE_OUT | 0 |
| T | RADIUS | 1,085 |
| T | MIN_RADIUS | 0,386 |
| T | MAX_VELOCITY | 0 |
| T | DISTANCE | 40 |
| T | BODYANGLE | 180 |
| A | MAX_STEPS_PER_TRIAL | 4 |
| A | FADE_IN | 0 |
| A | FADE_OUT | 0 |
| A | SEND_PER_STEP | 10 |
| A | MIN_SPEED | 0,5 |
| A | $d(\alpha)$ | 24 |
| A | $d(\beta)$ | 24 |
| A | $d(\gamma)$ | 24 |
| A | $d(v_{ball})$ | 10 |
| A | $d(\delta)$ | 72 |
| A | $d(\epsilon)$ | 72 |
| A | $d(p_{kick})$ | 10 |
| A | Torbreite | 6 |

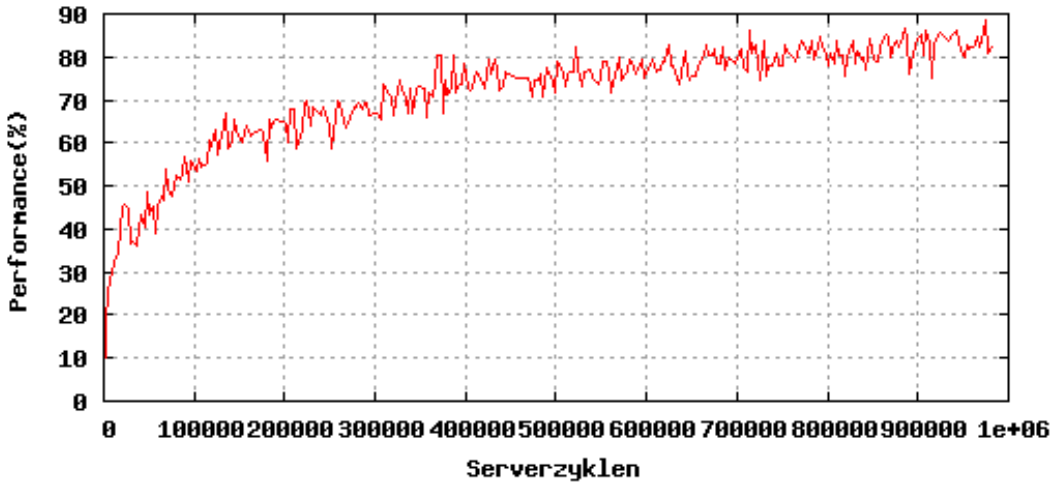
Abbildung 35: Lernperformance der 24x72 Welt - Mit Observation based RFL




man zum Vergleich auf die Performancekurve des Q-Learning, so kann festgestellt werden, daß sich auch hier die Performance zwischen $80$ und $90\%$ einzuschwingen scheint.

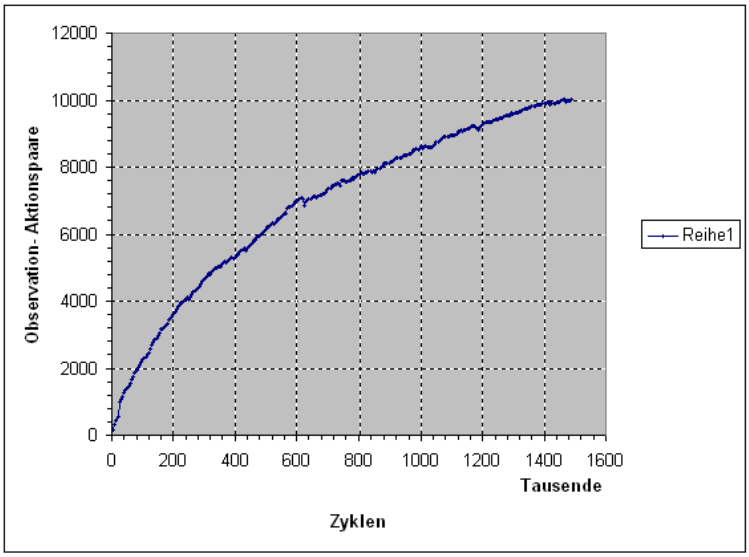
**5.2.6.3 Lerneffektivität** Vergleicht man die beiden Lerneffektivitäts Diagramme miteinander, so ist zu erkennen, daß nach knapp 1,6 Millionen Zyklen beim Observation based RFL etwa 10000 Observation- Aktionspaare bekannt sind, beim Q-Learning hingegen circa 17000 Paare. Mit der Annahme, daß dieses Szenario deterministisch ist, würden minimal $24 * 24 = 576$ (ruhender Ball, nur 2 varriierende Winkel. Daraus folgt: minimale Anzahl Aktionen = Anzahl der Startzustände) gelernte Aktionen und

Abbildung 36: Lernperformance der 24x72 Welt - Mit Q-Learning




maximal $24 * 24 * 4 = 2304$ (Anzahl Startzustände * Anzahl der maximal möglichen Aktionen pro Trial) gelernte Aktionen benötigt, um dieses Szenario völlig zu erschöpfen. Auch hier ist wieder einmal erkennbar, daß die Zahl der gelernten Observation- Aktionspaare mit beiden Lernverfahren bei weitem die maximale Anzahl der lernbaren Aktionen in einer deterministischen Welt übersteigt.

Abbildung 37: Lerneffektivität der 24x74 Welt - Mit Observation based RFL




Es soll nun noch ein Blick auf die Verteilungsmessungen geworfen werden, da bei der Performance die beiden Lernverfahren nahezu gleichwertig sind. Die Lerneffektivität ist hier nicht unbedingt ein Leistungsindikator, da die Menge der gelernten Aktionen nichts über ihre Güte aussagt.

**5.2.6.4 Verteilungsmessung** Die Policies wurden auf ein 6 Meter breites Tor angelernt und wurden dann für die Verteilungsmessung auf ein 30 Meter breites Tor getestet. Bei der Verteilungsmessung des Observation based Verfahren kann man erkennen, daß das absolute Verteilungsmaximum bei $+3m$ auf der Y-Achse liegt. Dies entspricht genau dem Bereich des oberen Pfosten des Tores (Die beiden Pfosten des Tores sind fast genau auf den beiden Datenpunkten welche einmal links und einmal rechts von der Y-Achse liegen). Außerhalb des Bereiches des ursprünglichen Tores nimmt die Trefferwahrscheinlichkeit jedoch nicht sonderlich schnell ab - ein großer Teil der Aktionen scheint den Bereich ausserhalb des Tores zu treffen. Addiert man die Einzelhäufigkeiten der Punkte, welche im eigentlichen Tor liegen auf, so erhält man eine Trefferwahrscheinlichkeit von 35% ins Tor. Leider liegt genau diese Verteilungsmessung aus dem Q-Learning

Abbildung 38: Lerneffektivität der 24x74 Welt - Mit Q-Learning

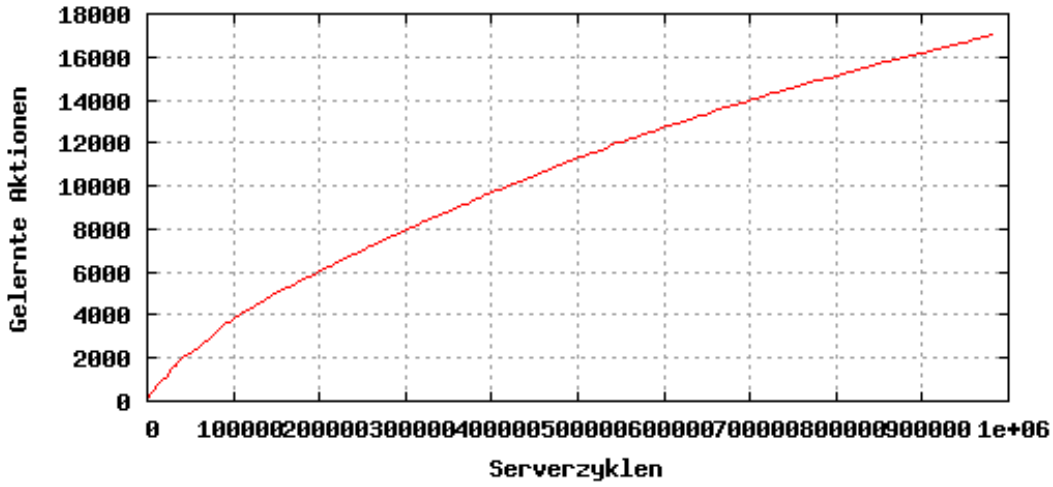
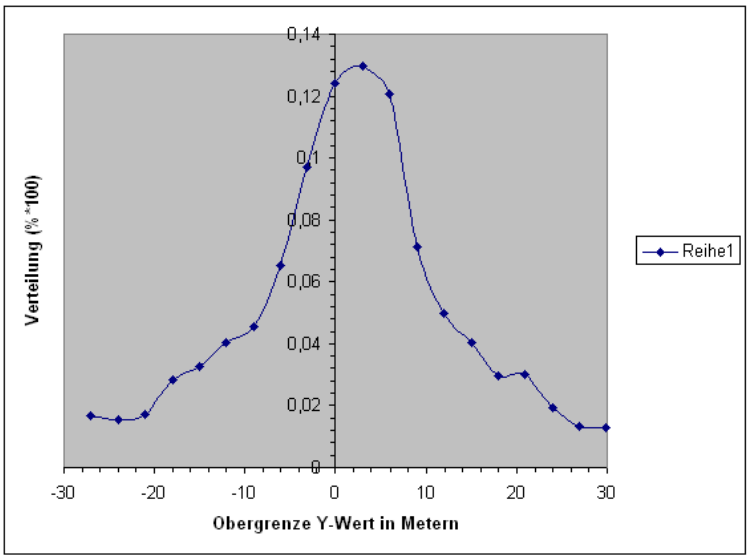



Abbildung 39: Verteilungsmessung der 24x74 Welt - Mit observation based RFL




nicht vor. Es bleibt zu prüfen, warum ein großer Teil der Schüsse das Tor nicht trifft.

# 6 Auswertung und Ausblick

In diesem Kapitel soll nun eine Auswertung der erstellten Ergebnisse erfolgen und weiterhin sollen Vorschläge gemacht werden, wie das Lernverfahren oder dessen Implementierung, weiterhin verbessert werden können. Auch soll an dieser Stelle noch eine kurze Wiederholung stattfinden in Hinblick darauf, was alles benötigt wird um das eingesetzte Lernverfahren anwenden zu können.

## 6.1 Auswertung der Testergebnisse

Betrachtet man als erstes die 4x4 Welt, so ist hier gut zu sehen, daß auch schon mit einer sehr groben Diskretisierung der Aktionen und der Zustandswinkel eine gute Performance zustande kommt. So hatte dieses Szenario mit dem Observation based RFL Verfahren nach 6000 Zyklen eine Performance von über 80% und mit Q-Learning gar eine Performance von über 90%.
Um zu testen, wie sich die Performance bei einer anderen Einteilung ändert, wurde das Trainingsszenario der 12x36 Welt herangezogen. Aus durchgeführten Tests, welche bedauerlicherweise nicht Bestandteil dieser Arbeit sind, war bekannt, daß für eine gute Performance die Diskretisierung der Aktionswinkel feiner sein mußte, als die Winkel der Observationen. Beim ursprünglichen Test dieses Szenarios lag jedoch die Performance nur bei knapp 55% (nach 10.000 absolvierten Zyklen). Diese konnte durch Verkleinerung des Startzustandraumes auf 90% verbessert werden. Ob bei uneingeschränktem Startzustandsraum in diesem Szenario auch eine Performance von knapp 90% erreicht werden kann, steht noch zu prüfen aus, da sich dieser Wert, wenn überhaupt, erst nach vielen hunderttausenden Zyklen einstellen wird.
Zur weiteren Überprüfung wurden die Diskretisierungen sowohl der Observationswinkel als auch der Aktionswinkel noch einmal verfeinert auf das 24x72 Szenario. In diesem Szenario stellt sich eine Performance von etwa 85% ein (nach 20.000 absolvierten Zyklen). Verbreitert man das Tor auf 30 Meter, so ist hier zu beobachten, daß die Performance weiter auf 90% steigt. Auch wurde dieses Szenario dazu herangezogen einen Vergleich mit dem Q-Learning zu tätigen. Nachdem mit beiden Verfahren etwa 1,5 Millionen Zyklen absolviert waren, pendelten sich sowohl das Q-Learning

Verfahren, als auch das Observation based RFL Verfahren zwischen 80 und 90% Performance ein. Die Performance scheint also in erster Linie von der Diskretisierung abzuhängen und in zweiter Linie erst vom eingesetzten Lernverfahren (Die Diskretisierung bestimmt die maximale Performance, das Lernverfahren bestimmt wie schnell man sich dieser Performance annähert). Der Performancewert ist bei diesem Szenario relativ hoch, wenn man bedenkt, daß hier auf ein nur 6 Meter breites Tor trainiert wurde.

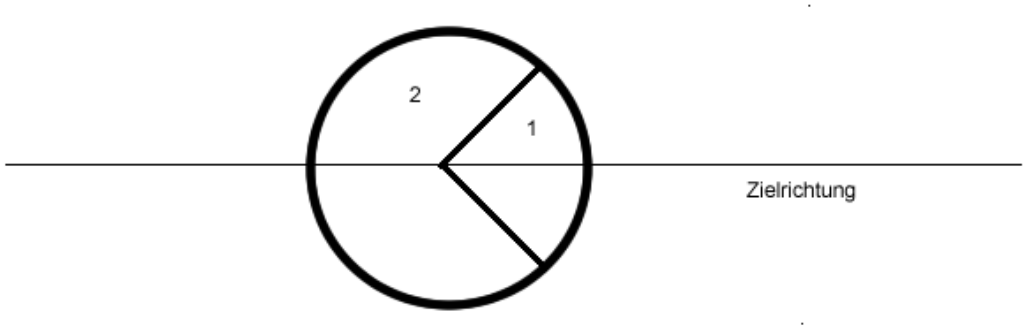
Betrachtet man die Verteilungsgraphen der verschiedenen Szenarien so fällt auf, daß nur das 4x4 Szenario ein Verteilungsmaximum im Tormittelpunkt hat. Alle anderen Szenarien haben bei der Verteilungsmessung eine Verschiebung des Maximums nach oben oder unten. Es wurde versucht, diese starken Asymmetrien zu untersuchen, aber zum jetztigen Zeitpunkt gibt es noch keine Erklärung für das Auftreten der Asymmetrien.
Um die Performance von Szenarien gezielt zu steigern, gibt es zwei Möglichkeiten: Zum einen kann die Torbreite erhöht werden und zum anderen kann der Startzustandsraum verkleinert werden. Um zu testen, wie gut die Performance eines Szenarios (wie gut die Diskretisierung) ist, muß ein Testlauf ausgeführt werden, solange bis ein Grenzwert in der Performance sichtbar wird - dies sollte am besten mit einem eingeschränkten Startzustandsraum passieren, um die Rechenzeit zu minimieren.

## 6.2 Verbesserungen am Lernverfahren

Es sollen nun Verbesserungsvorschläge zum eingesetzten Lernverfahren vorgebracht werden. Die größte Schwäche des eingesetzten Verfahrens ist, daß eine hohe Rechenzeit zum Erlernen der Policies nötig ist. Eine leicht durchzuführende Verbesserung sieht dabei wie folgt aus: Durch die feste Größe der Trainingsepisode (4 Zyklen in den hier vorgestellten Szenarien), welche sowohl auf Agenten, als auch auf Trainerseite zum Einsatz kommen, kann es vorkommen, daß der Agent mehrere Zyklen hintereinander nichts zu tun hat. Das beruht auf der Tatsache, daß der Agent bereits nach einem Zyklus einen Endzustand erreicht haben kann, für ihn ist damit die Lernepisode beendet, jedoch wird er erst nach dem Ablauf der 4 Zyklen auf einen neuen Startzustand gesetzt. Es könnten viele Zyklen gespart werden, würde der Agent auf einen neuen Startzustand gesetzt, direkt nachdem er sich in einem Endzustand befindet.

Ein anderer Ansatz der verfolgt werden könnte ist, daß durch die Reduzierung des Observationraumes die Lernkomplexität soweit verkleinert wird, daß ein essentieller Zeitvorteil entsteht. Die Performancetests zeigten, daß durch feinere Diskretisierung die Performance des Szenarios zunimmt - Wie wird es möglich sein den Observationraum zu verkleinern und trotzdem noch eine gute Performance zu erhalten? Der Ansatz besteht darin, daß eine neue Art der Generalisierung eingeführt wird. Diese erfordert es, daß die Lernaufgabe in zwei Teilbereiche aufgebrochen wird und diese unterschiedlich diskretisiert werden. Zu diesem Zweck wird ein Kreisausschnitt innerhalb der „kickable area“ des Agenten definiert, mit Richtung zum Anspielpunkt. Dieser Bereich ist in Abbildung (40) mit einer „1“ markiert. In

Abbildung 40: Neue Generalisierungsidee




diesem Bereich kann die Diskretisierung sehr fein sein, ohne daß der Observationsraum zu groß wird. Bereich „2“ hingegen in Abbildung (40) hat eine sehr grobe Diskretisierung und in diesem Bereich ist die Aufgabe des Agenten nur, den Ball irgendwie in dem Bereich „1“ zu bringen. Denkbar wäre hier z.B. sogar die Ballgeschwindigkeit und den Ballvektor aus dem Zustandsraum herauszunehmen und zu fordern, daß der Ball innerhalb von einem Zyklus im Bereich „1“ landen muß. Ist der Ball dann in Bereich

„1“ kann aufgrund der feinen Diskretisierung ein sehr präzieser Zielschuss abgegeben werden.

## 6.3 Ausblick

In diesem Abschnitt soll ein Ausblick darüber gewährt werden, wie die hier beschriebenen Ergebnisse in der Praxis eingesetzt werden können und wie dies zu erfolgen hat. Diese Diplomarbeit hat gezeigt, daß das hier vorgestellte Lernverfahren gut in der Lage ist, in einem angemessenen Zeitrahmen eine Policy zu erlenen, welche in der Lage ist, die gestellte Problemsituation mit einer guten Performance zu bewältigen. Um die mit diesem Lernverfahren erzeugten Policies zum Einsatz zu bringen, sind diese in eine Form zu bringen, welche von einem Agenten schnell zugänglich sind. Der Agent wird dafür eine Methode benötigen, welche sein momentanes Weltbild in die Observationen der gespeicherten Policy übersetzt und die dazu nötige Aktion ermittelt. Bevor jedoch an einen Einsatz in einem RoboCup Team gedacht werden kann, müssen unbedingt die weiter oben beschrieben Möglichkeiten zur Verringerung der Rechenzeit eingebaut werden, da es selbst für einfache Aufgabenstellungen zuweilen mehrere Tage dauert, eine passende Policy zu berechnen (z.B. wenn eine Policy 1,4 Millionen Zyklen á 10 ms erfordert, sind dies immerhin schon fast 39 Stunden). Sind diese Verbesserungen implementiert, so steht dem Einsatz in einem RoboCup Team nichts mehr im Wege, da die erzielten Policies eine gute Performance aufweisen. Der Agent kann schnell an andere Aufgabenstellungen angepaßt werden, da nur seine Abbildungsfunktion (welche das Weltmodell auf Observationen abbildet), seine Diskretisierung und die ihm zur Verfügung stehenden Aktionen geändert werden müssen. Es ist also möglich mit dem hier zum Einsatz gekommenden Agenten ohne viel Aufwand, viele „Skills“ für den RoboCup zu lernen.
Sollte jedoch gewünscht werden, daß hier vorgestellte Lernverfahren bei richtigen Robotern einzusetzen, so gibt es zwei Möglichkeiten: Zum einen wäre eine Simulation denkbar, in der genug Rechenzeit zur Verfügung steht um mehrere Millionen Zyklen lang zu lernen oder es muß versucht werden, die Komplexität des Zustandsraumes so weit zu verkleinern, daß eine erfolgreiche Policy bereits nach mehreren hundert oder tausend Zyklen zustande kommt.

# A Sourcecode - rli.h

```
 1 #pragma interface
 2
 3 #include <stdio.h>
 4 #include <unistd.h>
 5 #include <fcntl.h>
 6 #include <unistd.h>
 7 #include <errno.h>
 8 #include <sys/types.h>
 9 #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <ctime>
12 #include <vector>
13
14 #include "randomc.h"
15 #include "Geometry.h"
16 #include "SoccerTypes.h"
17 #include  "WorldModel.h"
18
19 #ifndef _INCL_RLI
20 #define _INCL_RLI
21
22
23 class Agent;
24 class Observation;
25 class PolicyElement;
26
27
28 /**
29  * Diese Klasse enthält alle relevaten Methoden und
      Informationen die der Lernagent für den Lernvorgang
      braucht. Wird diese Klasse instanziiert, so werden
      dabei die Diskretsierungen der Winkel festgelegt.
      Diese Klasse enthält die Policy wie auch das Set der
      für den Agenten möglichen Aktionen.
30  */
31 class Agent {
32
33    friend class Player;
34
35 public:
36    /**
37     * Konstuktor des Agenten. Bei seinem Aufruf werden
          die Winkeldiskretisierungen festgelegt und
```

```
            gleichzeitig, wird angegeben ob eine neue Policy
            erstellt werden soll, oder ob eine bereits
            vorhandene, alte Policy weiterverwendet werden
            soll. Weiterhin wird im Agenten ein
            Zufallszahlengenerator initialisiert. Dieser wird
            z.B. für das aussuchen zufälliger Aktionen
            gebraucht.
38          */
39          Agent(int angle_ball, int angle_body, int
               angle_velocity, bool j) {
40             disAngleBall = angle_ball;
41             disAngleBody = angle_body;
42             disAngleVelocity = angle_velocity;
43             readin = j;
44             goodAction = 0;
45             badAction = 0;
46                uint32 seed = time(0);
47             randomGen = new TRandomMersenne(seed);
48          }
49
50          /**
51           * Destruktor des Agenten. Hier wird der Semaphor,
                welcher zur synchronisation mit dem Trainer diente
                 freigegeben, der Zufallszahlengenerator wird
                entfernt und die Policy und das Aktionsset werden
                ebenfalls gelöscht.
52           */
53          ~Agent( void ) {
54             if (semctl(semKey, NULL, IPC_RMID) == -1)
55                perror("[AGENT] - Semaphor: ");
56             free((TRandomMersenne*)randomGen);
57             Policy.clear();
58             Actions.clear();
59          }
60
61
62
63          void  init (AngDeg threshold);
64          void  savePolicy();
65          void  readPolicyVector();
66          void  cleanup();
67          AngDeg   getAbortThreshold();
68          void  initActionVector();
69          void  initPolicyVector();
70          void  incStep();
```

```
71      void   writeSummarize();
72      void   incGood();
73      unsigned long int getGood();
74      unsigned long int getBad();
75      void   resetGood();
76      void   resetBad();
77      void   incBad();
78      int    getActionCount();
79      unsigned long int getStepsTaken();
80      ofstream mdata;
81      int    getPolicyVectorCount();
82      int    getCurrentObservation(WorldModel *WoMo);
83      SoccerCommand* getRandomAction();
84      SoccerCommand* currentObservationHasAction(WorldModel
            *WMo);
85
86  private:
87      int    disAngleBall;
88      int    disAngleBody;
89      int    disAngleVelocity;
90      int    semKey;
91      unsigned long int goodAction;
92      unsigned long int badAction;
93      bool   readin;
94      unsigned long int steps_taken;
95      AngDeg   abortThreshold;
96      TRandomMersenne *randomGen;
97      vector<PolicyElement*>  Policy;
98      vector<PolicyElement*>::iterator PolicyIterator;
99      vector<SoccerCommand*> Actions;
100     vector<SoccerCommand*>::iterator ActionsIterator;
101
102
103 };
104
105 /**
106  * Die Klasse Observation enthält, die der Name sagt eine
        Observation des Systemzustandes. In ihr werden die
       bereits diskretisierten Werte des Environments
       abgespeichert.
107  */
108 class Observation {
109
110 public:
111    /**
```

```
112      * Standardkonstruktor. Durch die übergebenen
             diskretisierten Werte, wird eine passende
             Observation erstellt.
113      */
114     Observation(unsigned int aBall, unsigned int aBody,
            unsigned int aVelocity, unsigned int vel) {
115       angleBall = aBall;
116       angleBody = aBody;
117       angleVelocity = aVelocity;
118       velocity = vel;
119     }
120
121     ~Observation      ( void         ) {}
122     bool  operator == ( const Observation &p  );
123
124     unsigned int   angleBall;
125     unsigned int   angleBody;
126     unsigned int   angleVelocity;
127     unsigned int   velocity;
128
129 };
130
131 /**
132  * Die Klasse PolicyElement enthält die Einzelteile,
       welche später eine komplette Policy ausmachen. Dies
       sind: Eine Observation, eine Aktion und ein Wert,
       welcher anzeigt ob diese Observation schon einmal
       besucht wurde, oder nicht (z). Der Wert d wird nur
       für analytische Zwecke gebraucht und ist nicht
       notwendig für den eigentlichen Lernenvorgang.
133  */
134 class PolicyElement {
135
136 public:
137    /**
138     * Standardkonstruktor. Es werden die bereits
           diskretisierten Werte der betreffenden Observation
            übergeben, aus denen dann ein Objekt der Klasse
           Observation erzeugt wird. Weiterhin wird z auf 0
           gesetzt und die Aktion "taken" wird mit NULL
           initialisiert.
139     */
140    PolicyElement (unsigned int aBall, unsigned int aBody,
           unsigned int aVelocity, unsigned int vel) {
141      z = 0;
```

```
142         d = 0;
143         taken = NULL;
144         omega = new Observation(aBall, aBody, aVelocity,
                vel);
145     }
146
147     PolicyElement() {}
148     ~PolicyElement ( void ) {}
149
150     void  operator =  (const PolicyElement &p );
151
152     Observation *omega;
153     bool z;
154     SoccerCommand *taken;
155     bool d;
156 };
157
158 #endif
```

# B Sourcecode - rli.cpp

```
1  #pragma  implementation
2
3
4  // -- Include files
5
6  #include "rli.h"
7  #include <cstdlib>
8  #include <ctime>
9  #include <math.h>
10 #include <string.h>
11 #include <stdio.h>
12 #include <fstream>
13
14
15 /*/
16  * Diese Agentenmethode initialisiert die Policy des
      Agenten. Sie wird entweder komplett neu angelegt,
      oder sie wird aus einer Datei eingelesen.
17  */
18 void Agent::init(AngDeg threshold)
19 {
20    abortThreshold = threshold;
21    initActionVector();
22    ofstream mdata("messdaten.txt",ios::trunc);
23    if (!mdata.is_open()) {
24       fprintf(stderr,"[AGENT]: coudln't open messdaten.
            txt\n");
25    }
26    if (readin == 1)
27       readPolicyVector();
28    else
29       initPolicyVector();
30 }
31
32 AngDeg Agent::getAbortThreshold() {
33    return abortThreshold;
34 }
35
36 /**
37  *Vergleicht zwei Observations auf Gleichheit.
38  */
39 bool Observation::operator == ( const Observation &p )
40 {
```

```
41      return ( ( angleBall == p.angleBall ) && ( angleBody
           == p.angleBody ) && ( angleVelocity == p.
           angleVelocity ) && ( velocity == p.velocity ) );
42  }
43
44  /**
45   * Hier werden die für den Agenten möglichen Aktionen
        erzeugt. Diese teilen sich in zwei Teile ein. Zum
        einen in Turnbefehle und zum anderen in Kickbefehle.
46   */
47  void Agent::initActionVector() {
48      int i=0, j=0;
49      for (i=0; i<7; i++) {
50          Actions.push_back(new SoccerCommand(CMD_TURN, (i
               *30)));
51      }
52      for (i=1; i<6; i++) {
53          Actions.push_back(new SoccerCommand(CMD_TURN, (i
               *(-30))));
54      }
55      for (i=0; i<37; i++) {
56          for (j=1; j<11; j++) {
57              Actions.push_back(new SoccerCommand(CMD_KICK, (j
                   *10), (i*5)));
58          }
59      }
60      for (i=1; i<36; i++) {
61          for (j=1; j<11; j++) {
62              Actions.push_back(new SoccerCommand(CMD_KICK, (j
                   *10), (i*(-5))));
63          }
64      }
65  }
66
67
68  /**
69   * Wenn eine neue Policy initialisiert werden soll, so
        passiert dies hier. Es werden PolicyElemente anhand
        der gegebenen Diskretisierungewerte erzeugt und in
        die Policy geschrieben.
70   */
71  void Agent::initPolicyVector() {
72      int j=0, k=0, l=0, m=0;
73      steps_taken = 0;
74      for (j=0; j<disAngleBall; j++) {
```

```
75          for (k=0; k<disAngleBody; k++) {
76             for (l=0; l<disAngleVelocity; l++) {
77                for (m=0; m<10; m++) {
78                   Policy.push_back(new PolicyElement(j, k, l
                        , m));
79                }
80             }
81          }
82       }
83       printf("Größe des Zustandsraumes: %i\n",Policy.size())
            ;
84       return;
85    }
86
87    /**
88     * Gibt die Größe der Policy wieder (Anzah der Elemente
          in der Policy).
89     */
90    int Agent::getPolicyVectorCount() {
91       return Policy.size();
92    }
93
94    /**
95     * Liefert die Anzahl der Aktionsabläufe wieder, die in
          der Policy gespeichert waren und welche in einen
          positiven Endzustand übergegangen sind.
96     */
97    unsigned long int Agent::getGood() {
98       return goodAction;
99    }
100
101   /**
102    * Liefert die Anzah der Aktionsabläufe wieder, die in
          der Policy gespeichert waren und welche in einen
          negativen Endzustand übergegangen sind.
103    */
104   unsigned long int Agent::getBad() {
105      return badAction;
106   }
107
108   /**
109    * Schließt das Logfile "mdata"
110    */
111   void Agent::cleanup() {
112      mdata.close();
```

```
113 }
114
115 /**
116  * Liefert die Anzahl der bisher absolvierten Schritte
        wieder.
117  */
118 unsigned long int Agent::getStepsTaken() {
119    return steps_taken;
120 }
121
122 /**
123  *  Schreibt die Rohdaten für die Performancemessung in
        die Daten "mdata"
124  */
125 void Agent::writeSummarize() {
126    printf("[DATA]: %i %i\n",goodAction, badAction);
127 }
128
129 /**
130  * Setzt den Wert der Aktionsabläufe, welche in einen
        positiven Endzustand geführt haben zurück auf 0
131  */
132 void Agent::resetGood() {
133    goodAction = 0;
134 }
135
136 /**
137  * Setzt den Wert der Aktionsabläufe, welche in einen
        negativen Endzustand geführt haben zurück auf 0
138  */
139 void Agent::resetBad() {
140    badAction = 0;
141 }
142
143 /**
144  * Inkrementiert den Wert goodAction
145  */
146 void Agent::incGood() {
147    goodAction++;
148 }
149
150 /**
151  * inkrementiert den Wert badAction
152  */
153 void Agent::incBad() {
```

```
154     badAction++;
155  }
156
157  /**
158   * Liefert die Anzahl der möglichen Aktionen des Agenten
          zurück
159   */
160  int Agent::getActionCount() {
161     int count=0;
162     int j=Policy.size();
163     for (int i=0; i<j; i++) {
164        if (Policy[i]->taken != NULL)
165           count++;
166     }
167     return count;
168  }
169
170  /**
171   * Erzeugt aus den Daten des Weltmodelles eine
          Observation und gibt deren Indizierungsnummer zurück.
172   */
173  int Agent::getCurrentObservation( WorldModel *WoMo ){
174     AngDeg zb, zk, bv;
175     unsigned int izb, izk, ibv;
176     double vel;
177     unsigned int ivel = 0, zyklus;
178     Observation *current;
179     zyklus = WoMo->getCurrentCycle();
180     zb = WoMo->getAgentGlobalBodyAngle() + WoMo->
           getRelativeAngle(OBJECT_BALL,1);
181     if (zb < 0)
182        zb = 360 + zb;
183     if (zb >= 360)
184        zb = zb - 360;
185     izb = (unsigned int)(zb / (360 / disAngleBall));
186     printf("[WoMo]: Zyklus: %i Winkeldifferenz Ball: %f, %
           f %f %i\n",zyklus, zb, WoMo->
           getAgentGlobalBodyAngle(), WoMo->getRelativeAngle(
           OBJECT_BALL, 1), izb);
187
188     zk = WoMo->getAgentGlobalBodyAngle();
189     if (zk < 0)
190        zk = 360 + zk;
191     izk = (unsigned int)(zk / (360 / disAngleBody));
```

```
192     printf("[WoMo]: Zyklus: %i Winkel Korper<>Tor: %f\n",
            zyklus, WoMo->getRelativeAngle(OBJECT_GOAL_R, 1));
193
194     bv = WoMo->getGlobalBodyAngle(OBJECT_GOAL_R) - WoMo->
            getBallDirection();
195     if (bv < 0)
196        bv = 360 + bv;
197     ibv = (unsigned int)(bv / (360 / disAngleVelocity));
198     printf("[WoMo]: Zyklus: %i Geschwindigkeitswinkel: %f
           %f\n", zyklus, (WoMo->getGlobalBodyAngle(
            OBJECT_GOAL_R)-WoMo->getBallDirection()), WoMo->
            getBallDirection());
199     vel = WoMo->getBallSpeed();
200     if (vel <0.5)
201        ivel = 0;
202     if ((vel>= 0.5) && (vel <1.0))
203        ivel = 1;
204     if ((vel>=1.0) && (vel<1.5))
205        ivel = 2;
206     if ((vel>=1.5) && (vel <1.75))
207        ivel = 3;
208     if ((vel>=1.75) && (vel<2.0))
209        ivel = 4;
210     if ((vel>=2.0) && (vel<2.1))
211        ivel = 5;
212     if ((vel>=2.1) && (vel<2.2))
213        ivel = 6;
214     if ((vel>=2.2) && (vel<2.3))
215        ivel = 7;
216     if ((vel>=2.3) && (vel<2.4))
217        ivel = 8;
218     if (vel>=2.4)
219        ivel = 9;
220     printf("[WoMo]: Zyklus: %i Geschwindigkeit: %f\n",
            zyklus, WoMo->getBallSpeed());
221     printf("[WoMo]: Zyklus: %i Current Observation is %i %
            i %i %i \n",zyklus,izb, izk, ibv, ivel);
222     current = new Observation(izb, izk, ibv, ivel);
223     int cp = Policy.size();
224     for (int i=0; i<cp; i++) {
225        if (*current == *(Policy[i]->omega)) {
226           free(current);
227           return i;
228        }
229     }
```

```
230     free(current);
231     return -1;
232 }
233
234 /**
235  * Wählt zufälig eine Aktion aus dem Set der möglichen
        Aktionen aus.
236  */
237 SoccerCommand* Agent::getRandomAction() {
238     int ca = 0;
239     ca = randomGen->IRandom(0,(Actions.size()-1));
240     switch (Actions[ca]->commandType) {
241         case CMD_TURN:
242             return ( new SoccerCommand(CMD_TURN, Actions[ca
                ]->dAngle) );
243             break;
244         case CMD_KICK:
245             return ( new SoccerCommand(CMD_KICK, Actions[ca
                ]->dPower, Actions[ca]->dAngle) );
246             break;
247         default:
248             break;
249     }
250     return NULL;
251 }
252
253 /**
254  * Prüft, ob für die momentan herrschende Observation
        bereits eine Aktion in der Policy abgelegt wurde.
255  */
256 SoccerCommand* Agent::currentObservationHasAction(
    WorldModel *WMo ) {
257     int current;
258     current = getCurrentObservation( WMo );
259     if ( (current < 0) || (current >= Policy.size())) {
260         printf("Warning: \"getCurrentObservation\" returned
              an erronous state! %i\n",current);
261         return (new SoccerCommand(CMD_ILLEGAL));
262     }
263     if (Policy[current]->taken != NULL) {
264         printf("There was an action, which i returned\n");
265         switch (Policy[current]->taken->commandType) {
266             case CMD_TURN:
267                 return (new SoccerCommand(CMD_TURN, Policy[
                    current]->taken->dAngle));
```

```
268             break;
269          case CMD_KICK:
270             return (new SoccerCommand(CMD_KICK, Policy[
                    current]->taken->dPower, Policy[current]->
                    taken->dAngle));
271             break;
272          default:
273             break;
274       }
275    }
276    return NULL;
277 }
278
279 /**
280  * Speichert die momentan Policy in eine Binärdatei und
        zusätzlich noch eine lesbare Textdatei.
281  */
282 void Agent::savePolicy() {
283    printf("Saving Policy now...\n");
284    ofstream archive("policy.log", ios::binary|ios::trunc)
          ;
285    if (!archive.is_open()) {
286       printf("Warning File was not opened!\n");
287    }
288    ofstream textual("policy.txt", ios::trunc);
289
290    if (!textual.is_open()) {
291       fprintf(stderr, "Textual isn't open although it
             should be\n");
292    }
293
294    int j = Policy.size();
295    CommandT dummy = CMD_ILLEGAL;
296    double d1 = 0, d2 = 0;
297    int count = 0, count_a = 0;
298
299    unsigned int b, c, d, e;
300    int f;
301    double g, h;
302
303    archive.write(reinterpret_cast<char *>(&steps_taken),
          sizeof(steps_taken));
304    archive.write(reinterpret_cast<char *>(&disAngleBall),
          sizeof(disAngleBall));
```

```
305     archive.write(reinterpret_cast<char *>(&disAngleBody),
            sizeof(disAngleBody));
306     archive.write(reinterpret_cast<char *>(&
            disAngleVelocity), sizeof(disAngleVelocity));
307     archive.write(reinterpret_cast<char *>(&abortThreshold
            ), sizeof(abortThreshold));
308     archive.write(reinterpret_cast<char *>(&goodAction),
            sizeof(goodAction));
309     archive.write(reinterpret_cast<char *>(&badAction),
            sizeof(badAction));
310
311     textual << "Steps taken: " << steps_taken <<"\n";
312     textual << "Ball Angle diskretisierung: " <<
            disAngleBall << "\n";
313     textual << "Body Angle diskretisierung: " <<
            disAngleBody << "\n";
314     textual << "Velocity Angle diskretisierung: " <<
            disAngleVelocity << "\n";
315     textual << "Abort Threshold: " << abortThreshold << "\
            n";
316     textual << "Good: " << goodAction << " Bad: " <<
            badAction << "\n";
317
318     textual << "\n--------Data---------\n";
319     for (int i=0; i<j; i++) {
320        count++;
321        b = Policy[i]->omega->angleBall;
322        c = Policy[i]->omega->angleBody;
323        d = Policy[i]->omega->angleVelocity;
324        e = Policy[i]->omega->velocity;
325        archive.write(reinterpret_cast<char *>(&b), sizeof(
               b));
326        archive.write(reinterpret_cast<char *>(&c), sizeof(
               c));
327        archive.write(reinterpret_cast<char *>(&d), sizeof(
               d));
328        archive.write(reinterpret_cast<char *>(&e), sizeof(
               e));
329
330        if (Policy[i]->taken == 0) {
331           archive.write(reinterpret_cast<char *>(&dummy),
                  sizeof(dummy));
332           archive.write(reinterpret_cast<char *>(&d1),
                  sizeof(d1));
```

```
333             archive.write(reinterpret_cast<char *>(&d2),
                    sizeof(d2));
334         }
335         else {
336             if (Policy[i]->z == 0) {
337                 f = Policy[i]->taken->commandType;
338                 g = Policy[i]->taken->dAngle;
339                 h = Policy[i]->taken->dPower;
340                 if ( (f==CMD_TURN) || (f==CMD_KICK))
341                     printf("KICK oder TURN: %f %f\n",g, h);
342                 count_a++;
343                 archive.write(reinterpret_cast<char *>(&f),
                        sizeof(f));
344                 archive.write(reinterpret_cast<char *>(&g),
                        sizeof(g));
345                 archive.write(reinterpret_cast<char *>(&h),
                        sizeof(h));
346                 textual << b << "\t" << c << "\t" << d << "\t
                        " << e << "\t";
347                 textual << f << "\t" << g << "\t" << h << "\n
                        ";
348             }
349             else {
350                 archive.write(reinterpret_cast<char *>(&dummy
                        ), sizeof(dummy));
351                 archive.write(reinterpret_cast<char *>(&d1),
                        sizeof(d1));
352                 archive.write(reinterpret_cast<char *>(&d2),
                        sizeof(d2));
353             }
354         }
355     }
356     archive.close();
357     textual.close();
358     printf("Saved %i policyelements containing %i actions
            in %i bytes\n",count, count_a, count*(sizeof(b)+
            sizeof(c)+sizeof(d)+sizeof(e)+sizeof(f)+sizeof(g)+
            sizeof(h)));
359     printf("Used %i steps so far\n",steps_taken);
360     printf("Size 1: %i 2: %i 3: %i 4: %i 5: %i 6: %i 7: %i
             \n", sizeof(b), sizeof(c), sizeof(d), sizeof(e),
            sizeof(f), sizeof(g), sizeof(h));
361     return;
362 }
363
```

```
364 /**
365  * Restauriert eine gespeicherte Policy aus einer
        Binärdatei.
366  */
367 void Agent::readPolicyVector() {
368
369    unsigned int  aball, abody, avelocity, velocity;
370    int  count = 0, count_a = 0;
371         CommandT commandType;
372    double dAngle, dPower;
373    printf("Trying to restore policy.\n");
374
375    ifstream archive("policy.log",ios::binary);
376    archive.read(reinterpret_cast<char *>(&steps_taken),
          sizeof(steps_taken));
377    archive.read(reinterpret_cast<char *>(&disAngleBall),
          sizeof(disAngleBall));
378    archive.read(reinterpret_cast<char *>(&disAngleBody),
          sizeof(disAngleBody));
379    archive.read(reinterpret_cast<char *>(&
          disAngleVelocity), sizeof(disAngleVelocity));
380    archive.read(reinterpret_cast<char *>(&abortThreshold)
          , sizeof(abortThreshold));
381    archive.read(reinterpret_cast<char *>(&goodAction),
          sizeof(goodAction));
382    archive.read(reinterpret_cast<char *>(&badAction),
          sizeof(badAction));
383    printf("Options extracted: %i %i %i %i %f\n",
          steps_taken, disAngleBall, disAngleBody,
          disAngleVelocity, abortThreshold);
384    do {
385       archive.read(reinterpret_cast<char *>(&aball),
             sizeof(aball));
386       archive.read(reinterpret_cast<char *>(&abody),
             sizeof(abody));
387       archive.read(reinterpret_cast<char *>(&avelocity),
             sizeof(avelocity));
388       archive.read(reinterpret_cast<char *>(&velocity),
             sizeof(velocity));
389       archive.read(reinterpret_cast<char *>(&commandType)
             , sizeof(commandType));
390       archive.read(reinterpret_cast<char *>(&dAngle),
             sizeof(dAngle));
391       archive.read(reinterpret_cast<char *>(&dPower),
             sizeof(dPower));
```

```
392         printf("Restored: %u %u %u %u %u %f %f\n", aball,
                abody, avelocity, velocity, commandType, dAngle,
                 dPower);
393         Policy.push_back(new PolicyElement(aball, abody,
                avelocity, velocity));
394         switch ( commandType ) {
395            case CMD_ILLEGAL:
396               printf("CMD_ILLEGAL, no new action will be
                      created\n");
397               break;
398            case CMD_TURN:
399               printf("CMD_TURN, new turn action will be
                      created\n");
400               Policy[Policy.size()-1]->taken = new
                      SoccerCommand(CMD_TURN, dAngle);
401               count_a++;
402               break;
403            case CMD_KICK:
404               printf("CMD_KICK, new kick command will be
                      created\n");
405               Policy[Policy.size()-1]->taken = new
                      SoccerCommand(CMD_KICK, dPower, dAngle);
406               count_a++;
407               break;
408            default:
409               break;
410         }
411         count++;
412      } while ( !archive.eof() );
413      printf("Restored %i policyelements containing %i
             actions\n",count, count_a);
414      archive.close();
415      return;
416   }
417
418   /**
419    * Inkrementiert die Anzahl der bereits genommenen
          Schritte um eins.
420    */
421   void Agent::incStep() {
422      steps_taken++;
423      return;
424   }
425
```

# C Sourcecode - PlayerTeams.cpp

```
 1 #include "Player.h"
 2 #include "rli.h"
 3 #include <sys/sem.h>
 4 #include <sys/ipc.h>
 5 #include <sys/types.h>
 6 #include <signal.h>
 7 #include <errno.h>
 8
 9 #define MAX_STEPS_PER_TRIAL 4
10 #define FADE_IN 0
11 #define FADE_OUT 0
12 #define SEND_PER_STEP 6
13 #define MIN_SPEED 0.5
14
15 /**
16  * Dies ist die Methode in dem das komplette Lernen
      stattfindet. Sie wird pro Serverzyklus mehrere male
      aufgerufen. Wie das Lernen genau von statten geht,
      ist an anderer Stelle nachzuschlagen.
17  */
18 SoccerCommand Player::deMeer5( )
19 {
20
21    SoccerCommand soc(CMD_ILLEGAL);
22    VecPosition   posAgent = WM->getAgentGlobalPosition();
23    VecPosition   posBall  = WM->getBallPos();
24    int           iTmp;
25    static bool createObjects = 0;
26    static bool trialEnd = 0;
27    static bool trialOverride = 0;
28    static long int   laststep = -1;
29    static long int sends_per_step = 0;
30    static int last_observation = - 2;
31    struct sembuf operations;
32    struct sembuf bla;
33    static bool stepOverride = 0;
34
35    if (!createObjects) {
36       SimAgent = new Agent(24, 24, 24, 1);
37       createObjects = 1;
38       key_t ka = ftok ("/home/usadel/Robocup/team_train/
            src/PlayerTeams.cpp",5);
39       if (ka == -1) {
```

```
40          perror("[AGENT] - ftok");
41        }
42        if ( (SimAgent->semKey = semget(2005, 1, IPC_CREAT|
              IPC_EXCL|0777)) == -1)
43          perror("[AGENT] - Semaphor(semget)");
44        operations.sem_num = 0;
45        operations.sem_op = 1;
46        operations.sem_flg = 0;
47        if ( (semop(SimAgent->semKey,&operations,1)) == -1)
48          perror("[AGENT] - Semaphor(semop (0))");
49        printf("New Agent was created\n");
50        SimAgent->init(30);
51      }
52
53      operations.sem_num = 0;
54      operations.sem_op = -1;
55      operations.sem_flg = 0;
56      if ( (semop(SimAgent->semKey,&operations,1)) == -1)
57        perror("[AGENT] - Semaphor(semop (1))");
58
59      Time kaffe = WM->getTimeLastRecvSenseMessage();
60      printf("[AGENT] Last time i received new information
            was %i\n",kaffe.getTime());
61      if (sends_per_step < (SEND_PER_STEP -1 )) {
62        fprintf(stderr,"-");
63        if ( (laststep+1) == WM->getCurrentCycle() ) {
64          printf("[AGENT]: sends_per_step reset due to
                laststep\n");
65          sends_per_step = 0;
66          stepOverride = 0;
67        }
68        if (stepOverride == 1)
69          sends_per_step = 0;
70        printf("[AGENT]: --Mark1--\n");
71        printf("[AGENT]: Warning: desynchronisazion
              detected\n");
72        printf("[AGENT]: Confidence: %f\n",WM->
              getConfidence(OBJECT_BALL));
73        printf("[AGENT]: Send per Steps: %i\n",
              sends_per_step);
74        if ( (laststep+1) == WM->getCurrentCycle()) {
75          printf("[AGENT]: obpressed due to laststep\n");
76        } else {
77          printf("[AGENT]: opressed due to SEND_PER_STEP\n
                ");
```

```
78          }
79          sends_per_step++;
80          if (WM->getCurrentCycle() == 0) {
81              printf("[AGENT]: sends_per_step reset due to
                    cycle 0\n");
82              sends_per_step = 0;
83          }
84      }
85      else {
86          fprintf(stderr,"+");
87          stepOverride = 1;
88          if (0) {
89          }
90          else {
91              if (WM->getCurrentCycle() % 5000 == 0) {
92                  printf("[AGENT]: Wrote to logfile\n");
93                  fprintf(stderr, "[DATA]\t%i\t%i\n",WM->
                        getCurrentCycle(), SimAgent->
                        getActionCount() );
94                  fprintf(stderr, "[DATA]\t%i\t%i\n",SimAgent->
                        getGood(), SimAgent->getBad());
95                  SimAgent->writeSummarize();
96                  SimAgent->resetGood();
97                  SimAgent->resetBad();
98              }
99              last_observation = SimAgent->
                    getCurrentObservation(WM);
100             fprintf(stderr,"*****Summarize for step %i
                    ******\n",WM->getCurrentCycle());
101             printf("[AGENT]: --Mark2--\n");
102             printf("[AGENT]: cycle %i start\n",WM->
                    getCurrentCycle());
103             printf("[AGENT]: There were %i omitted receive
                    steps\n",sends_per_step);
104             sends_per_step = 0;
105             printf("[AGENT]: Current data: %f %f %x\n",WM->
                    getRelativeAngle(OBJECT_BALL, 1), WM->
                    getAgentGlobalBodyAngle(), WM->isBallKickable
                    ());
106             printf("[AGENT]: Global Angle: %f\n",WM->
                    getAgentGlobalBodyAngle());
107             printf("[AGENT]: BallDir: %f\n",(WM->
                    getGlobalBodyAngle(OBJECT_GOAL_R) - WM->
                    getBallDirection()));
```

```
108         VecPosition leckmichdochamarsch = WM->getBallPos
                ();
109         VecPosition goalPos = WM->getGlobalPosition(
                OBJECT_GOAL_R);
110         printf("[AGENT]: Goalposition %f %f\n",goalPos.
                getX(), goalPos.getY());
111         printf("[AGENT]: Ballposition %f %f\n",
                leckmichdochamarsch.getX(),
                leckmichdochamarsch.getY());
112         printf("[AGENT]: Zuverlaessigkeit: %f\n",WM->
                getConfidence(OBJECT_GOAL_R));
113         printf("[AGENT]: Zuverlaessigkeit: %f\n",WM->
                getConfidence(OBJECT_BALL));
114         if ( WM->getCurrentCycle() % (
                MAX_STEPS_PER_TRIAL)  == (MAX_STEPS_PER_TRIAL
                 - 1)) {
115            printf("[AGENT]: --Mark3--\n");
116            printf("[AGENT]: End of trainingstrial\n");
117            trialEnd = 1;
118            int fu3 = SimAgent->Policy.size();
119            int gna3 = 0;
120            for (int sdf3 =0; sdf3<fu3; sdf3++) {
121               if (SimAgent->Policy[sdf3]->taken != NULL)
122               gna3++;
123            }
124            fprintf(stderr,"--end %i\n",gna3);
125         }
126         if ( WM->getCurrentCycle() % (
                MAX_STEPS_PER_TRIAL) == 0) {
127            printf("[AGENT]: --Mark4--\n");
128            trialEnd = 0;
129            trialOverride = 0;
130            printf("[AGENT]: Starting new
                   trainingsepisode\n");
131            int fu4 = SimAgent->Policy.size();
132            int gna4 = 0;
133            for (int sdf4 =0; sdf4<fu4; sdf4++) {
134               if (SimAgent->Policy[sdf4]->taken != NULL)
135               gna4++;
136            }
137            fprintf(stderr,"--begin %i\n",gna4);
138         }
139
140         if (trialEnd == 1) {
141            fprintf(stderr,"Trial has ended ");
```

```
142             printf("[AGENT]: Trial End.\n");
143             printf("[AGENT]: --Mark8--\n");
144             SimAgent->incStep();
145             int ii = SimAgent->getPolicyVectorCount();
146             AngDeg thresh = WM->getRelativeAngle(
                    OBJECT_BALL, 1) + WM->
                    getAgentGlobalBodyAngle();
147             printf("[AGENT]: THRESHOLD %f %f %f\n",
                    thresh, WM->getRelativeAngle(OBJECT_BALL,
                    1), WM->getAgentGlobalBodyAngle());
148             printf("[AGENT]: THRESHOLD %f\n",WM->
                    getBallDirection());
149             Line BallLine = Line::
                    makeLineFromPositionAndAngle(WM->
                    getBallPos(),WM->getBallDirection());
150             Line GoalLine = Line::makeLineFromTwoPoints(
                    WM->getGlobalPosition(OBJECT_FLAG_G_R_T),
                    WM->getGlobalPosition(OBJECT_FLAG_G_R_B));
151             VecPosition sp = BallLine.getIntersection(
                    GoalLine);
152             double by = sp.getY();
153             printf("[AGENT]: Y-Offset was %f\n",by);
154             printf("[AGENT]: Eckpunkte %f %f\n",WM->
                    getGlobalPosition(OBJECT_FLAG_G_R_T).getY
                    (), WM->getGlobalPosition(
                    OBJECT_FLAG_G_R_B).getY());
155             if ( (!WM->isBallKickable()) && (fabs(sp.getY
                    ()) < 7) && (WM->getBallSpeed() >=
                    MIN_SPEED) && (fabs(WM->getBallDirection()
                    ) <= 90)) {
156                 printf("[AGENT]: --Mark9--\n");
157                 printf("[AGENT]: FABS 1 was %f and the
                        Threshold was %f\n",fabs(WM->
                        getRelativeAngle(OBJECT_BALL, 1) - WM->
                        getAgentGlobalBodyAngle()), SimAgent->
                        getAbortThreshold());
158                 printf("[AGENT]: BallDirection: %f\n",WM->
                        getBallDirection());
159                 fprintf(stderr, "[GAUSS]: %i %f\n",WM->
                        getCurrentCycle(), sp.getY());
160                 int kap = 0;
161                 for (int ij = 0; ij < ii; ij++) {
162                     if ( SimAgent->Policy[ij]->z == 1) {
163                         SimAgent->Policy[ij]->z = 0;
164                         if (SimAgent->Policy[ij]->d == 1)
```

```
165                         SimAgent->incGood();
166                     SimAgent->Policy[ij]->d = 0;
167                     kap++;
168                 }
169             }
170             fprintf(stderr, "and %i Actions were saved
                    \n",kap);
171
172         }
173         else {
174             printf("[AGENT]: --Mark10--\n");
175             printf("[AGENT]: FABS 1 was %f and the
                    Threshold was %f\n",fabs(WM->
                    getRelativeAngle(OBJECT_BALL, 1) - WM->
                    getAgentGlobalBodyAngle()), SimAgent->
                    getAbortThreshold());
176             int kapp = 0;
177             for (int ik = 0; ik < ii; ik++) {
178                 if (SimAgent->Policy[ik]->z == 1) {
179                     SimAgent->Policy[ik]->z = 0;
180                     if (SimAgent->Policy[ik]->d == 1)
181                         SimAgent->incBad();
182                     SimAgent->Policy[ik]->d = 0;
183                     free((SoccerCommand*)SimAgent->
                            Policy[ik]->taken);
184                     SimAgent->Policy[ik]->taken = NULL;
185                     kapp++;
186                 }
187             }
188             fprintf(stderr, "and %i Actions were
                    deleted\n",kapp);
189         }
190     }
191
192     if ( trialOverride == 1 ) {
193         printf("[AGENT]: --Mark7--\n");
194         bla.sem_num = 0;
195         bla.sem_op = 1;
196         bla.sem_flg = 0;
197         if ( (semop(SimAgent->semKey,&bla,1)) == -1 )
198             perror("[AGENT] - Semaphor(semop (2))");
199         laststep = WM->getCurrentCycle();
200         int cd = SimAgent->getPolicyVectorCount();
201         int kappp = 0;
202         fprintf(stderr,"Trial is beeing overriden");
```

```
203             for (int ce = 0; ce < cd; ce++) {
204                 if ( SimAgent->Policy[ce]->z == 1) {
205                     SimAgent->Policy[ce]->z = 0;
206                     if (SimAgent->Policy[ce]->d == 1)
207                         printf("[DATA] d failure\n");
208                     SimAgent->incBad();
209                     SimAgent->Policy[ce]->d = 0;
210                     free((SoccerCommand*)SimAgent->Policy[
                            ce]->taken);
211                     SimAgent->Policy[ce]->taken = NULL;
212                     kappp++;
213                 }
214             }
215             fprintf(stderr,"and %i Actions where deleted\
                    n",kappp);
216             return soc;
217         }
218
219         if (trialEnd != 1) {
220             printf("[AGENT]: --Mark11--\n");
221             SimAgent->incStep();
222             AngDeg basket = WM->getRelativeAngle(
                    OBJECT_BALL, 1) + WM->
                    getAgentGlobalBodyAngle();
223             printf("[AGENT]: THRESHOLD: %f %f %f\n",
                    basket, WM->getRelativeAngle(OBJECT_BALL,
                    1), WM->getAgentGlobalBodyAngle());
224             printf("[AGENT]: THRESHOLD %f\n",WM->
                    getBallDirection());
225             int jj = SimAgent->getPolicyVectorCount();
226             Line BallLine = Line::
                    makeLineFromPositionAndAngle(WM->
                    getBallPos(),WM->getBallDirection());
227             Line GoalLine = Line::makeLineFromTwoPoints(
                    WM->getGlobalPosition(OBJECT_FLAG_G_R_T),
                    WM->getGlobalPosition(OBJECT_FLAG_G_R_B));
228             VecPosition sp = BallLine.getIntersection(
                    GoalLine);
229             double by = sp.getY();
230             printf("[AGENT]: Y-Offset was %f\n",by);
231             printf("[AGENT]: Eckpunkte %f %f\n",WM->
                    getGlobalPosition(OBJECT_FLAG_G_R_T).getY
                    (), WM->getGlobalPosition(
                    OBJECT_FLAG_G_R_B).getY());
```

```
232             if ( (!WM->isBallKickable()) && (fabs(sp.getY
                    ()) < 7) && (WM->getBallSpeed() >=
                    MIN_SPEED) && (fabs(WM->getBallDirection()
                    ) <= 90)) {
233                 printf("[AGENT]: --Mark9--\n");
234                 printf("FABS 2 was %f and the THreshold
                        was: %f \n",fabs(WM->getRelativeAngle(
                        OBJECT_BALL, 1) - WM->
                        getAgentGlobalBodyAngle()), SimAgent->
                        getAbortThreshold());
235                 printf("[AGENT]: BallDirection: %f\n",WM->
                        getBallDirection());
236                 trialOverride = 1;
237                 int gzk = 0;
238                 fprintf(stderr,"positiv terminalstate
                        reached ");
239                 printf("[GAUSS]: %i %f\n",WM->
                        getCurrentCycle(), sp.getY());
240                 for (int ji = 0; ji < jj; ji++) {
241                     if (SimAgent->Policy[ji]->z == 1) {
242                         SimAgent->Policy[ji]->z = 0;
243                         if (SimAgent->Policy[ji]->d == 1)
244                             printf("[DATA] d failure\n");
245                         SimAgent->incGood();
246                         SimAgent->Policy[ji]->d = 0;
247                         gzk++;
248                         printf("[AGENT]: Positiver
                                Endzustand, date Policy up...\n")
                                ;
249                     }
250                 }
251                 fprintf(stderr,"saved %i Actions\n",gzk);
252             }
253             else {
254                 printf("[AGENT]: --Mark12--\n");
255                 if ( WM->isBallKickable() ) {
256                     printf("Constructive Branch, terminal
                            state not yet reached\n");
257                     SoccerCommand* ac = SimAgent->
                            currentObservationHasAction( WM );
258                     int fu = SimAgent->Policy.size();
259                     int gna = 0;
260                     for (int sdf =0; sdf<fu; sdf++) {
261                         if (SimAgent->Policy[sdf]->taken !=
                                NULL)
```

```
262                         gna++;
263                     }
264                     printf("There were %i mapped Actions\n"
                            ,gna);
265                     fprintf(stderr,"There are %i mapped
                            Actions\n",gna);
266                     if ( ac != NULL ) {
267                         printf("deterministic action, c-
                                Branch\n");
268                         fprintf(stderr,"Taking deterministic
                                 action\n");
269                         SimAgent->Policy[SimAgent->
                                getCurrentObservation(WM)]->z =
                                1;
270                         SimAgent->Policy[SimAgent->
                                getCurrentObservation(WM)]->d =
                                1;
271                         soc = *ac;
272                         ACT->putCommandInQueue( soc );
273                     }
274                     else {
275                         ac = SimAgent->getRandomAction();
276                         printf("Random Action, c-Branch\n");
277                         fprintf(stderr,"taking random action
                                \n");
278                         printf("%i %i\n",ac->dAngle, ac->
                                dPower);
279                         int whatever = SimAgent->
                                getCurrentObservation(WM);
280                         switch (ac->commandType) {
281                             case CMD_TURN:
282                                 SimAgent->Policy[whatever]->
                                        taken = new SoccerCommand(
                                        CMD_TURN, ac->dAngle);
283                                 break;
284                             case CMD_KICK:
285                                 SimAgent->Policy[whatever]->
                                        taken = new SoccerCommand(
                                        CMD_KICK, ac->dPower, ac->
                                        dAngle);
286                                 break;
287                             default:
288                                 break;
289                         }
290                         SimAgent->Policy[whatever]->z = 1;
```

```
291                         SimAgent->Policy[whatever]->d = 0;
292                         soc = *ac;
293                         ACT->putCommandInQueue( soc );
294                      }
295                      free(ac);
296                   }
297                   else {
298                      printf("[AGENT]: --Mark13--\n");
299                      trialOverride = 1;
300                   }
301                }
302             }
303             else {
304             }
305             printf("[AGENT]: cycle %i stop\n",WM->
                   getCurrentCycle());
306          }
307          int fu2 = SimAgent->Policy.size();
308          int gna2 = 0;
309          for (int sdf =0; sdf<fu2; sdf++) {
310             if (SimAgent->Policy[sdf]->taken != NULL)
311                gna2++;
312          }
313          fprintf(stderr,"so far %i actions\n",gna2);
314       }
315       laststep = WM->getCurrentCycle();
316       bla.sem_num = 0;
317       bla.sem_op = 1;
318       bla.sem_flg = 0;
319       if ( (semop(SimAgent->semKey,&bla,1)) == -1 )
320          perror("[AGENT] - Semaphor(semop (2))");
321       return soc;
322    }
323    <...>
```

# D Sourcecode - BasicCoach.cpp

```
1  #include"Parse.h"
2  #include <math.h>
3  #include <sys/sem.h>
4  #include <sys/ipc.h>
5  #include <sys/types.h>
6  #include <errno.h>
7
8  #ifdef WIN32
9  #include <windows.h>
10 #else
11 #include <sys/poll.h>
12 #endif
13
14 #define STEPS_PER_TRIAL 4
15 #define FADE_IN 0
16 #define FADE_OUT 0
17 #define RADIUS 1.085
18 #define MIN_RADIUS 0.386
19 #define MAX_VELOCITY 0
20 #define DISTANCE 40
21 #define BODYANGLE 180
22 #define NORMAL 1
23 //#define SIMPLE 0 /*!< Wenn gesetzt, wird der Trainer
       den Ball nur an 4 verschiedene Startpositionen
       platzieren.
24
25 extern Logger Log; /*!< This is a reference to the Logger
        to write loginfo to*/
26
27 /**
28  * Initialisierungsmethode des Trainers, unter anderem
       wird hier der Semaphor, welcher zum abgleichen mit
       dem Agenten benutzt wird, initialisiert.
29  */
30 BasicCoach::BasicCoach( ActHandler* act, WorldModel *wm,
       ServerSettings *ss,
31        char* strTeamName, double dVersion, bool isTrainer
              )
32
33 {
34    char str[MAX_MSG];
35
36    ACT       = act;
```

```
37      WM          = wm;
38      SS          = ss;
39      bContLoop = true;
40      WM->setTeamName( strTeamName );
41
42      if( !isTrainer ) {
43          sprintf( str, "(init %s (version %f))", strTeamName
                , dVersion );
44          printf("I'am not the trainer\n");
45      }
46      else {
47          sprintf( str, "(init (version %f))", dVersion );
48          printf("I'am the trainer\n");
49      }
50      srand(static_cast<unsigned>(time(0)));
51      uint32 seed = time(0);
52      RendGen = new TRandomMersenne(seed);
53      key_t ka = ftok("/home/usadel/Robocup/team_train/src/
            PlayerTeams.cpp",5);
54      if (ka == -1) {
55          perror("[COACH] - ftok");
56      }
57      if ( (this->SemKey = semget(2005, 1, 0)) == -1)
58          perror("[COACH] - Semaphor(semget)");
59      ACT->sendMessage( str );
60  }
61
62  BasicCoach::~BasicCoach( )
63  {
64  }
65
66  /**
67   * In dieser Hauptmethode des Trainers, werden Ball und
        Spieler selber auf Zufällig ausgewählte
        Startpositionen gesetzt. In wie fern sich die
        Positionierung der einzelnen Parameter beeinflußen
        läßt, ist an früherer Stelle der Diplomarbeit zu
        entnehmen.
68   */
69  void BasicCoach::mainLoopNormal( )
70  {
71  #ifdef WIN32
72      Sleep( 1000 );
73  #else
74      poll( 0, 0, 1000 );
```

```
75 #endif
76
77     bool bSubstituted   = false;
78     double x=0;
79     double y=0;
80     double threshold_y = 0;
81     double threshold_vy = 0;
82     char command[MAX_MSG];
83     int bodyAngle = 0;
84     double velocityY = 0;
85     double minY = 0;
86     double velocityX = 0;
87     static unsigned long int laststep = 0;
88
89    ACT->sendMessageDirect( "(eye on)" );
90
91 #ifdef WIN32
92     Sleep( 1000 );
93 #else
94     poll( 0, 0, 1000 );
95 #endif
96
97     while( WM->getPlayMode() != PM_TIME_OVER  && bContLoop
           )
98     {
99         if (WM->getCurrentCycle() == 0) {
100           printf("Starting..\n");
101           ACT->sendMessageDirect("(change_mode play_on)");
102        }
103        Log.log( 1, "in loop %d %d %f",
104              WM->getTimeLastSeeGlobalMessage().getTime(),
105              bSubstituted,
106              WM->isConfidenceGood( OBJECT_TEAMMATE_11 )) ;
107        if( WM->waitForNewInformation() == false )
108        {
109           printf( "Shutting down coach\n" );
110           bContLoop = false;
111        }
112        printf("[COACH]: cycle %i start\n",WM->
               getCurrentCycle());
113        if ( (laststep == WM->getCurrentCycle()) && (WM->
               getCurrentCycle() != 0)) {
114           fprintf(stderr,"desynchonisation omitting step\n
                  ");
115        }
```

```
116         else {
117             if ( (WM->getCurrentCycle() % (STEPS_PER_TRIAL +
                     FADE_IN + FADE_OUT) ) == 0 ) {
118                 struct sembuf operations;
119                 operations.sem_num = 0;
120                 operations.sem_op = -1;
121                 operations.sem_flg = 0;
122                 if ( (semop(this->SemKey, &operations, 1)) ==
                         -1 ) {
123                     perror("[COACH] - Semaphor(semop(0))");
124                 }
125                 else {
126                     fprintf(stderr,"[COACH] - Semaphor geholt\
                             n");
127                 }
128                 //Create random body direction for the player
129                 #ifndef SIMPLE
130                 if (RendGen->IRandom(0,1) == 0) {
131                     bodyAngle = RendGen->IRandom(0,BODYANGLE);
132                     sprintf(command, "(move (player train 1) %
                             i 0 %i)",DISTANCE, bodyAngle);
133                     //printf("Bodyangle: %i\n",bodyAngle);
134                 }
135                 else {
136                     bodyAngle = (-1)*RendGen->IRandom(0,
                             BODYANGLE);
137                     sprintf(command, "(move (player train 1) %
                             i 0 %i)",DISTANCE, bodyAngle);
138                 }
139                 ACT->sendMessageDirect( command );
140                 //Create Random start position for the Ball
141                 #else
142                 int sector = RendGen->IRandom(0,3);
143                 if (sector == 0)
144                     sprintf(command, "(move (player train 1) %
                             i 0 0)",DISTANCE);
145                 if (sector == 1)
146                     sprintf(command, "(move (player train 1) %
                             i 0 90)",DISTANCE);
147                 if (sector == 2)
148                     sprintf(command, "(move (player train 1) %
                             i 0 180)",DISTANCE);
149                 if (sector == 3)
150                     sprintf(command, "(move (player train 1) %
                             i 0 -90)",DISTANCE);
```

```
151
152             ACT->sendMessageDirect( command );
153             #endif /*SIMPLE*/
154             #ifndef SIMPLE
155             if (RendGen->IRandom(0, 1) == 0) {
156                 x = RendGen->Random()*RADIUS;
157                 threshold_y = sqrt( (RADIUS*RADIUS) - (x*x
                        ) );
158                 printf("[COACH]: Data: %f %f\n",
                        MIN_RADIUS, x);
159                 if ((x*x) >= (MIN_RADIUS*MIN_RADIUS)) {
160                     minY = 0;
161                     printf("[COACH]: Data: %f\n",minY);
162                 }
163                 else {
164                     minY = sqrt( (MIN_RADIUS*MIN_RADIUS) -
                            (x*x) );
165                     printf("[COACH]: Data: %f %f\n",(
                            MIN_RADIUS*MIN_RADIUS),(x*x));
166                 }
167
168                 printf("[COACH]: Data: %f %f\n",
                        threshold_y, minY);
169                 y = RendGen->Random()*(threshold_y-minY);
170                 printf("[COACH]: Data: %f %f\n",y, minY);
171                 y += minY;
172
173                 if (RendGen->IRandom(0, 1) == 0)
174                     y=y;
175                 else
176                     y = (-1)*y;
177                 printf("[COACH]: Data: %f %f\n",x,y);
178             }
179             else {
180                 x = (-1)*RendGen->Random()*RADIUS;
181                 threshold_y = sqrt( (RADIUS*RADIUS) - (x*x
                        ) );
182                 printf("[COACH]: Data: %f %f\n",
                        MIN_RADIUS, x);
183                 if ( (x*x) >= (MIN_RADIUS*MIN_RADIUS)) {
184                     minY = 0;
185                     printf("[COACH]: Data: %f\n",minY);
186                 }
187                 else {
```

```
188                     minY = sqrt( (MIN_RADIUS*MIN_RADIUS) -
                            (x*x) );
189                     printf("[COACH]: Data: %f %f\n",(
                            MIN_RADIUS*MIN_RADIUS),(x*x));
190                 }
191
192                 printf("[COACH]: Data: %f %f\n",
                        threshold_y, minY);
193                 y = RendGen->Random()*(threshold_y-minY);
194                 printf("[COACH]: Data: %f %f\n",y, minY);
195                 y += minY;
196
197                 if (RendGen->IRandom(0, 1) == 0)
198                     y = y;
199                 else
200                     y = (-1)*y;
201                 printf("[COACH]: Data: %f %f\n",x,y);
202             }
203             //Create random velocity for the ball
204             if (MAX_VELOCITY == 0) {
205                 if (NORMAL) {
206                     sprintf(command, "(move (ball) %f %f 0
                            0 0)",(DISTANCE+x),(0+y));
207                     printf("[VERT]: %f %f\n",(DISTANCE+x)
                            ,(0+y));
208                 }
209                 else
210                     sprintf(command, "(move (ball) %f %f 0
                            0 0)",(DISTANCE+0.4),(0));
211                 ACT->sendMessageDirect( command );
212             }
213             else {
214                 if (RendGen->IRandom(0, 1) == 0) {
215                     velocityX = RendGen->Random()*
                            MAX_VELOCITY;
216                     threshold_vy = sqrt( (MAX_VELOCITY*
                            MAX_VELOCITY) - (velocityX*velocityX
                            ) );
217                     if (RendGen->IRandom(0, 1) == 0)
218                         velocityY = RendGen->Random()*
                                threshold_vy;
219                     else
220                         velocityY = (-1)*RendGen->Random()*
                                threshold_vy;
221                 }
```

```
222                 else {
223                     velocityX = (-1)*RendGen->Random()*
                            MAX_VELOCITY;
224                     threshold_vy = sqrt( (MAX_VELOCITY*
                            MAX_VELOCITY) - (velocityX*velocityX
                            ) );
225                     if (RendGen->IRandom(0, 1) == 0)
226                         velocityY = RendGen->Random()*
                                threshold_vy;
227                     else
228                         velocityY = (-1)*RendGen->Random()*
                                threshold_vy;
229                 }
230                 if (NORMAL) {
231                     sprintf(command, "(move (ball) %f %f 0
                            %f %f)",(DISTANCE+x),(0+y),
                            velocityX, velocityY);
232                     printf("[VERT]: %f %f\n",(DISTANCE+x)
                            ,(0+y));
233                 }
234                 else
235                     sprintf(command, "(move (ball) %f %f 0
                            %f %f)",(DISTANCE+0.4),(0),
                            velocityX, velocityY);
236                 ACT->sendMessageDirect( command );
237             }
238             #else
239             int sec = RendGen->IRandom(0,3);
240             if (sec == 0)
241                 sprintf(command, "(move (ball) %f %f 0 0
                        0)",(DISTANCE+0.5),(0+0.5));
242             if (sec == 1)
243                 sprintf(command, "(move (ball) %f %f 0 0
                        0)",(DISTANCE+0.5),(0-0.5));
244             if (sec == 2)
245                 sprintf(command, "(move (ball) %f %f 0 0
                        0)",(DISTANCE-0.5),(0+0.5));
246             if (sec == 3)
247                 sprintf(command, "(move (ball) %f %f 0 0
                        0)",(DISTANCE-0.5),(0-0.5));
248
249             ACT->sendMessageDirect( command );
250             #endif /*SIMPLE*/
251             operations.sem_num = 0;
252             operations.sem_op = 1;
```

```
253                 operations.sem_flg = 0;
254                 if ( (semop(this->SemKey, &operations, 1)) ==
                        -1) {
255                    perror("[COACH] - Semaphor(semop (1))");
256                 }
257                 else {
258                    fprintf(stderr,"[COACH] - Semaphor zuruck\
                          n");
259                 }
260                 printf("[COACH]: cycle %i stop\n",WM->
                       getCurrentCycle());
261                 if( Log.isInLogLevel(  456 ) )
262                    WM->logObjectInformation( 456,
                          OBJECT_ILLEGAL);
263                 if( SS->getSynchMode() == true )
264                    ACT->sendMessageDirect( "(done)" );
265              }
266           }
267           laststep = WM->getCurrentCycle();
268        }
269        return;
270     }
271     <...>
```

# Literatur

[Robo01] *The Goals of RoboCup*
RoboCup Federation http://www.robocup.org/overview/22.html. Verifiziert am 12. Februar 2001

[RoCu01] *RoboCup-97: Robo Soccer World Cup I, Seiten 20-41. Springer-Verlag 1997*
tsuki Noda, Shoji Suzuki, Hitoshi Matsubara, Minoru Asada und Hiroaki Kitano.

[IAS2004] *Intelligent Autonomous Systems Group*
http://www.science.uva.nl/research/ias/

[UvA2003] *UvA Basiscode 2003*
http://staff.science.uva.nl/ jellekok/robocup/2003/trilearn_base_sources-3.3.tar.gz

[SoServer] *Soccer Server Manual 2003*
http://sserver.sourceforge.net/

[Marsenne] *Mersenne Twister*
http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html

[Cost] *Cost-to-go functions*
http://www.cis.temple.edu/ pwang/525-NC/Lecture/Ch12.htm

[CDoeven] *Reinforcement Learning in der Robocup Soccerserver Umgebung - Q-Lernen in einer Nichtdeterministischen Welt*
Cand. Ing. Celal Döven

[SBarto] *Reinforcement Learning: An Introduction*
Richard S. Sutton and Andrew G. Barto
MIT Press, Cambridge, MA, 1998

[Harmon] *Reinforcement Learning: A Tutorial*
Mance E. Harmon and Stephanie S. Harmon
Wright State University

[Rmiller] *Generalization in Reinforcement Learning and the Use of Observation-Based Learning*

Martin Lauer and Martin Riedmiller
Lehrstuhl Informatik I, Universität Dortmund.

[Schleginski] *Robocup World Model Dokumentation*
Mark Schleginski
HS Weigarten

[Rein] *Glossary of Terminology in Reinforcement Learning*
http://www-anw.cs.umass.edu/rlr/terms.html