Technische Hochschule
Brandenburg
University of
Applied Sciences
Fachbereich
Informatik und Medien

Masterarbeit

Real-time reinforcement learning von Handlungsstrategien für humanoide Roboter

Fachbereich:	Informatik
Studiengang:	Master Informatik
Vorgelegt von:	Colin Christ
1. Gutachter:	Diplom-Informatiker Ingo Boersch
2. Gutachter:	Prof. Dr. Jochen Heinsohn
Vorgelegt am:	22.12.2017

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Real-time reinforcement learning von Handlungsstrategien für humanoide Roboter

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Brandenburg an der Havel, den 22.12.2017

Unterschrift

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Motivation . . . 3
- 1.2 Aufgabenstellung . . . 3
- 1.3 Analyse der Aufgabenstellung . . . 4
  - 1.3.1 Funktionale Ziele . . . 4
  - 1.3.2 Nichtfunktionale Ziele . . . 5
- 1.4 Abgrenzung . . . 5
- 1.5 Aufbau der Arbeit . . . 5

2 Nao **7**
- 2.1 Hardware . . . 7
- 2.2 API . . . 7
- 2.3 Zusammenfassung . . . 10

3 Szenario **11**
- 3.1 MDP . . . 12
  - 3.1.1 Zustandsraum . . . 12
  - 3.1.2 Aktionen . . . 14
  - 3.1.3 Reward . . . 14
  - 3.1.4 Übergangswahrscheinlichkeiten . . . 16
- 3.2 Zusammenfassung . . . 16

4 Verwandte Arbeiten **17**
- 4.1 Zusammenfassung . . . 20

5 Theoretische Vorbetrachtung **23**
- 5.1 Reinforcement Learning . . . 23
- 5.2 Temporal Difference Learning . . . 24
  - 5.2.1 Q-Learning . . . 24
  - 5.2.2 SARSA . . . 25
  - 5.2.3 Q(λ) . . . 26
- 5.3 Exploration . . . 28
  - 5.3.1 ε-greedy . . . 28
  - 5.3.2 Softmax . . . 28
- 5.4 Estimator . . . 29
  - 5.4.1 Q-Tabelle . . . 29
  - 5.4.2 Function Approximation . . . 29
- 5.5 Planer . . . 31
  - 5.5.1 Prioritized Sweeping . . . 31
- 5.6 Zusammenfassung . . . 32

6 Konzeption des Agenten **33**
- 6.1 Zusammenfassung . . . 34

7 Implementierung **37**
7.1 Systemumgebung . . . 37
7.2 Zustand . . . 37
7.3 Nao . . . 38
7.4 Aktionen . . . 40
7.5 Agent . . . 41
7.5.1 Erzeugung eines Agenten-Objekts . . . 41
7.6 Rewarder . . . 41
7.7 Logger . . . 42
7.8 LearnerValidatorPair . . . 44
7.9 GUI-Klassen . . . 44
7.9.1 PanelController . . . 45
7.10 Koordinatenlisten . . . 45
7.11 Nebenläufigkeit . . . 46
7.12 Umsetzung der RL-Algorithmen . . . 46
7.12.1 Explorator Funktionen . . . 47
7.12.2 Estimator Funktionen . . . 47
7.12.3 Temporal Difference Learning Funktionen . . . 48
7.12.4 Planende Funktionen . . . 48
7.13 Zusammenfassung . . . 49

8 Grafische Benutzeroberfläche **51**
8.1 Erläuterung der GUI . . . 51
8.2 Visualisierung des Lernvorgangs . . . 52
8.3 Konfiguration des Roboters . . . 55
8.4 Abstrakte Darstellung des Nao . . . 56
8.5 Zusammenfassung . . . 56

9 Versuche **59**
9.1 Erste Versuchsreiche . . . 59
9.1.1 Durchführung . . . 59
9.1.2 Ergebnis . . . 59
9.2 Zweite und dritte Versuchsreihe . . . 62
9.2.1 Durchführung . . . 62
9.2.2 Ergebnis . . . 62
9.3 Vierte Versuchsreihe . . . 65
9.3.1 Durchführung . . . 65
9.3.2 Ergebnis . . . 65
9.4 Zusammenfassung . . . 71

10 Auswertung **73**
10.1 Fazit . . . 73
10.2 Ausblick . . . 73
10.2.1 Grafische Benutzeroberfläche . . . 74
10.2.2 Reinforcement Learning . . . 74
10.2.3 Überhitzungsproblem . . . 75
10.2.4 Änderung des Zustandaktionsraumes . . . 76

Literaturverzeichnis **II**

A Ausführen des Programms **a**

B Argumentenübergabe von der GUI bis zum Agenten **a**

C Erste Versuchsreihe **d**

D Zweite Versuchsreihe **m**

E Dritte Versuchsreihe **r**

F Vierte Versuchsreihe **A**

Abbildungsverzeichnis

1 Gelenke und Servomotoren [Ald17a] 8
2 Koordinatensysteme des Nao [Ald17a] 9
3 Der Nao führt einen *Ballwechsel* durch. 9
4 Der Nao hält einen Ball 10
5 Der Versuchsaufbau aus der Vogelperspektive. Der lernende Roboter ist rot, der validierende Roboter ist blau. 11
6 Positionen des Armes 13
7 Versuchsskizze aus [LLN17] 17
8 Versuchaufbau aus [SCSD17] 19
9 Zwei Roboter lernen das Türen-öffnen unter Verwendung von asynchronen NAF. Die finale Policy konnte, wenn zwei Roboter genutzt wurden, mit einer Wahrscheinlichkeit von 100% über alle 20 Versuche erreicht werden. Bild und Bildunterschrift sind aus [GHLL17] 19
10 Agent-Umgebung Interaktion in RL (angelehnt an [SB12]) 23
11 2D- und 3D-Ansicht von drei überlappenden Tilings, die einen 2D-Zustandsraum abdecken. Die Gesamtanzahl der Features in diesem Beispiel ist 27, wobei nur 3 Features zurzeit aktiv sind, was durch den grauen Punkt, bzw. Linie, in den Tiles symbolisiert wird. Die Grafik ist aus [TP17]. 30
12 Flowchart generalisierter Algorithmus des Agenten 35
13 UML-Ansicht der Klasse `State` 37
14 UML-Ansicht der Klassen `Nao`, `NaoMock`, `B2M Thread` und `B2M Watchdog` . 38
15 UML-Ansicht der Aktionen 40
16 UML-Ansicht der Klasse `Agent` 41
17 UML-Ansicht der Klasse `Rewarder` 42
18 UML-Ansicht der Klasse `Logger` 43
19 UML-Ansicht der Klasse `LearnerValidatorPair` 43
20 UML-Ansicht der GUI-Klassen 44
21 Grafische Benutzeroberfläche 51
22 Abbildung eines Lernverlaufs in Form eines Diagramms, bei dem sich auf der X-Achse die Episoden – und auf der Y-Achse die Anzahl der Aktionen pro Episode befinden 53
23 Animation des Nao 53
24 Beispielhafte Übersicht des Lernschritts, es wird die Anzahl der vergangenen Episoden, Epsilon und die aktualisierten Tiles (siehe Tilecoding) angezeigt 54
25 Graph als Darstellungsvariante des Lernens 54
26 Ansicht zur Konfiguration des Roboters 55
27 Grafische Visualisierung der Zustände 57
28 Bestandteile der Zustandsgrafik 58
29 Lernverläufe der Algorithmen in der ersten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet 60
30 Versuchreihe 1, Q-Table, Aktionen über den gesamten Lernverlauf 61
31 Versuchreihe 1, Q-Feature mit PS, Aktionen über den gesamten Lernverlauf 63
32 Versuchreihe 1, Q-Feature, Aktionen über den gesamten Lernverlauf 64
33 Vergleich der Lernverläufe von Q-Feature mit PS über die Versuchsreihen eins bis drei 65

34 Lernverläufe der Algorithmen in der zweiten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet 66
35 Lernverläufe der Algorithmen in der dritten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet 66
36 Lernverläufe der vierten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet 67
37 Vergleich Q-Feature aus der ersten Versuchsreihe mit Q-Learning PS aus der vierten Versuchsreihe 68
38 Versuchreihe 4, Q-Learning mit PS, Aktionen über den gesamten Lernverlauf 69
39 Q-Learning mit PS aus der vierten Versuchsreihe. Der Verlauf von `epsilon` ist blau dargestellt. 70
40 Lernverläufe der Algorithmen in der ersten Versuchsreihe . . . . . d
41 Q-Feature Lernverlauf in der ersten Versuchsreihe. Zyklisches Ansteigen und Abfallen von `epsilon` . . . . . e
42 Lernverlauf von Q-Learning in der ersten Versuchsreihe . . . . . f
43 Lernverlauf von Q(λ) in der ersten Versuchsreihe . . . . . g
44 Lernverlauf von Q-Learning mit Prioritized Sweeping in der ersten Versuchsreihe . . . . . h
45 Lernverlauf von Q-Feature in der ersten Versuchsreihe i
46 Lernverlauf von Q-Freature mit Prioritized Sweeping in der ersten Versuchsreihe . . . . . j
47 Q-Feature in der realen Umgebung mit dem Nao. Es wurden die ausgefüllten Feature-Werte von dem Q-Feature der ersten Versuchsreihe genutzt. `epsilon` wurde auf null gesetzt. . . . . k
48 Q-Feature in der realen Umgebung mit dem Nao. Es wurden die ausgefüllten Feature-Werte von dem Q-Feature der ersten Versuchsreihe genutzt. `epsilon` wurde auf 0.55 gesetzt. l
49 Lernverlauf von Q-Feature in der zweiten Versuchsreihe . . . . . m
50 Lernverlauf von Q-Freature mit Prioritized Sweeping in der zweiten Versuchsreihe . . . . . n
51 Lernverlauf von Q(λ) in der zweiten Versuchsreihe . . . . . o
52 Lernverlauf von Q-Learning in der zweiten Versuchsreihe . . . . . p
53 Lernverlauf von Q-Learning mit Prioritized Sweeping in der zweiten Versuchsreihe . . . . . q
54 Lernverlauf von Q-Feature in der dritten Versuchsreihe . . . . . r
55 Lernverlauf von Q-Freature mit Prioritized Sweeping in der dritten Versuchsreihe . . . . . s
56 Lernverlauf von Q(λ) in der dritten Versuchsreihe . . . . . t
57 Lernverlauf von Q-Learning in der dritten Versuchsreihe . . . . . u
58 Lernverlauf von Q-Learning mit Prioritized Sweeping in der dritten Versuchsreihe v
59 Lernverläufe der vierten Versuchsreihe . . . . . A
60 Lernverlauf von Q-Feature in der vierten Versuchsreihe . . . . . B
61 Lernverlauf von Q-Learning in der vierten Versuchsreihe C
62 Lernverlauf von Q-Learning mit Prioritized Sweeping in der vierten Versuchsreihe . . . . . D
63 Vollständiger Graph von Q-Learning mit PS . . . . . E
64 Oberer Graph von Q-Learning mit PS, roter Ball Zyklus . . . . . F

65 Mittlerer Graph von Q-Learning mit PS, Übergang zwischen rotem und blauem Ball . . . . . G
66 Unterer Graph von Q-Learning mit PS, blauer Ball Zyklus . . . . . G

Tabellenverzeichnis

1 Dekodierungstabelle der Zustände 13
2 Übersicht mit Beschreibung der Aktionen 14
3 Übersicht, in welchen Zuständen eine Aktion ausgeführt werden soll. Eine leere Zelle bedeutet, dass der Zustand des Features für die Aktion irrelevant ist. 15
4 Konfiguration der Tilings. Die angegebenen Zahlen beziehen sich auf die Dimensionen des Zustandsraumes, siehe Abschnitt 3.1.1 48
5 Verwendete Methoden in der ersten Versuchsreihe 59

Zusammenfassung

Das Lernen von Handlungsstrategien ohne vorgefertigte Daten, wie beispielsweise beim Supervised Learning (SL), stellt in der Robotik eine große Herausforderung dar. Diese Aufgabe wird mit **Reinforcement Learning** (RL) gelöst, bei dem der Lerner mit *Trial and Error* seine Möglichkeiten erkundet, um einen guten Lösungsweg zu finden.

RL birgt ein hohes Potenzial in der Robotik. Da der Lerner keine vorgefertigten Daten nutzt, kann er seine Umwelt *frei* erforschen, wodurch er mit RL unkonventionelle und auch bessere Lösungswege, als durch andere Methoden des maschinellen Lernens, finden kann. RL in der Robotik hat jedoch den Nachteil, dass eine gute Handlungsstrategie langsam gelernt wird.

Ziel der Arbeit ist es, RL-Algorithmen zu finden und zu kombinieren, um das Erlernen einer Handlungsstrategie schneller zu gestalten.

In dieser Arbeit wird die Implementierung eines Programms beschrieben, welches das parallele Durchführen von RL-Versuchen mit verschiedenen Algorithmen ermöglicht. Hierbei können auch kompatible Algorithmen miteinander kombiniert werden, um ein besseres Ergebnis zu erzielen. Der Lernvorgang kann mit den humanoiden Nao-Robotern oder in einer Simulation durchgeführt werden. Dies erhöht die Flexibilität und Wiederverwendbarkeit des Programms. Zusätzlich, um den Lernvorgang zu überwachen und ein Verständnis für RL durch dieses Programm vermitteln zu können, wird das Lernen auf einer grafischen Benutzeroberfläche visuell dargestellt.

Um das Ziel zu erreichen, wird das Programm für die Durchführung von Versuchsreihen genutzt, bei denen verschiedene RL-Algorithmen-Kombinationen getestet werden. Der Lerner soll die gestellte Aufgabe in unter einer Stunde lösen. Die Versuchsreihen ergaben, dass die Aufgabe am besten durch **Q-Learning** in Verbindung mit **Prioritized Sweeping** gelöst werden konnte. Der Lerner entdeckte hierbei eine gute Handlungsstrategie in ca. 40min.

1 Einleitung

1.1 Motivation

Computer lösen schon seit Jahrzehnten Probleme für den Menschen. Das Fortschreiten der technologischen Entwicklung ermöglichte uns Menschen, immer komplexere Probleme von Computern lösen zu lassen. Einige Probleme werden so komplex, dass sie sich nicht mehr vom Menschen implementieren lassen.

Eine Methode, solche Probleme zu lösen ist, der Maschine *beizubringen*, welche Lösungen richtig sind, indem Beispiele für richtige Lösungen gezeigt werden. Dieses Prinzip wird beim **Supervised Learning** (SL) umgesetzt. SL wird heutzutage beispielsweise in der Bilderkennung verwendet. Wenn der Maschine beigebracht werden soll, eine Schachpartie zu gewinnen, könnte die Maschine vom Menschen Spielzüge *beigebracht* bekommen, wodurch die Maschine lernen könnte, wie ein Mensch Schach oder Go, siehe [SHM+16], zu spielen. Die Maschine kann durch SL Spielzüge lernen, die den Menschen zum Ziel geführt haben. Es gibt aber, beispielsweise beim Schach, Spielzüge und Strategien, die so komplex sind, dass sie noch nie ein Mensch angewendet hat. Solche Spielzüge können bei SL nicht von der Maschine entdeckt werden, da die Maschine vom Menschen gelernt hat und dadurch nie *viel besser* als der Mensch sein wird.

Um den Menschen zu übertreffen, muss die Maschine ihre eigenen Lösungen finden. Das Ergebnis ist hierbei durch eine *Belohnung* definiert. Die Maschine kann selbst, über *Trial and Error* Wege finden, das Problem zu lösen. Diese Handlungsstrategie werden zu Beginn des Lernens noch sehr umständlich sein, jedoch kann die Maschine hierdurch lernen, Probleme bestmöglich zu lösen. Die erlernte Handlungsstrategie kann besser sein, als alle vom Menschen entwickelten Lösungsstrategien. Sie kann *übermenschlich* sein. Die beschriebene Lernmethode heißt **Reinforcement Learning** (RL), was sich in das Deutsche mit *bestärkendes Lernen* übersetzen lässt. Der Lernende, welcher im weiteren Agent genannt wird, führt eine Aktion aus und bekommt eine positive oder negative *Belohnung*. Ziel des Agenten ist es, die Belohnungen langfristig zu maximieren, siehe Abschnitt 5.1.

RL in der Robotik stellt auch heute noch eine besondere Herausforderung dar. Bewegungen des Agenten sind nicht immer präzise und auf andere Gegebenheiten, wie beispielsweise Batteriestand oder Motorüberhitzung, muss geachtet werden. Hinzu kommt, dass das finale RL nicht in einer Simulation stattfinden kann, sondern auf die reale Motorik und Sensorik des Roboters zurückgreifen muss und daher in realtime stattfindet. Aktionen benötigen dementsprechend für die Ausführung mehr Zeit, als bei RL ohne Roboter. Dies hat zur Folge, dass eine Episode lange dauern kann und ein Reinforcement Algorithmus die wenigen Daten mit hoher Effizienz verwerten muss, sodass ein Lernen in akzeptabler Zeit stattfinden kann.

In dieser Arbeit wird eine Aufgabe mittels humanoider Roboter und RL gelöst. Anhand dieser Aufgabe werden Probleme analysiert und mit neuesten Erkenntnissen aus dem Bereich RL gelöst.

1.2 Aufgabenstellung

Ziel der Arbeit ist die Entwicklung einer Applikation zur Demonstration von Reinforcement Learning auf autonomen, humanoiden Robotern. Demonstriert werden soll das Erlernen einer erfolgreichen Handlungsstrategie in einem einfachen, realen Szenario. Das Szenario kann selbst gewählt werden, bspw. das Sortieren von Bällen. Das Szenario soll im Wesentlichen deterministisch sein, kann aber in seltenen Fällen stochastisch reagieren.

Der Lernvorgang soll unbeaufsichtigt selbständig laufen können und in kurzer Zeit (bspw. einer Stunde) zu einer erfolgreichen Policy führen. Ein zweiter Applikationsmodus soll das unbegrenzte Ausführen der erlernten Policy ermöglichen.

Eine besondere Schwierigkeit stellt die angestrebte Autonomie des Lernvorgangs dar, da das Robotersystem weder sich, noch die Welt beschädigen darf. Hier ist bspw. auf Anomalien im Betrieb zu achten.

Für den Lernvorgang darf der Agent keine fremderstellte Simulation verwenden, für Evaluierung und Test der Applikation ist eine Simulation natürlich erlaubt. Damit besteht die zweite Schwierigkeit in der geringen Anzahl von Interaktionen mit dem realen Szenario, sodass Maßnahmen zur Effizienzsteigerung klassischer RL-Ansätze verwendet müssen. Hilfreich wäre eine geeignete Visualisierung des Lernvorganges bzw. der Policy oder transparenter Wertefunktionen, um Besuchern und Studenten den Ablauf zu verdeutlichen und die Programm-Entwicklung zu unterstützen. Der Demonstrator soll modular, einfach bedienbar, auch nach länger Nichtnutzung leicht in Betrieb zu nehmen und auf den Rechnern des KI-Labors lauffähig sein. Parameter sollten deklarativ und einfach änderbar repräsentiert werden. Das erstellte System soll geeignet dokumentiert und evaluiert werden.

1.3 Analyse der Aufgabenstellung

1.3.1 Funktionale Ziele

Folgende funktionale Ziele lassen sich aus der Aufgabenstellung ableiten:

1. Primäres Ziel der Arbeit ist es, eine gute Policy in möglichst kurzer Zeit zu erlernen. Eine Policy wird als gut bezeichnet, wenn eine Episode möglichst kurz ist. Es ist nicht notwendig, dass der Agent aus jedem möglichen Zustand den kürzesten Weg findet, die Policy soll sich jedoch optimal verhalten, wenn sich der Agent in einem oft besuchten Zustand befindet.

2. Es soll ein Progamm entwickelt werden, mit dem die Lernvorgänge durchgeführt und analysiert werden sollen.

3. Das Programm soll über zwei Modi verfügen. Der erste Modus soll das RL von Beginn an zeigen. Der zweite Modus soll veranschaulichen, wie sich der Agent nach dem Lernen verhält.

4. Um die Algorithmen miteinander vergleichen zu können, oder das Lernen zu beschleunigen, soll es möglich sein, dass mehrere Agenten gleichzeitig lernen.

5. Einem Agenten soll zur Laufzeit Funktionalität, in Form von RL-Methoden, hinzugefügt werden können.

6. RL-Methoden, die der Agent nutzt, sollen zur Laufzeit durch entsprechende Parameter konfiguriert werden können. Mit dieser Konfiguration wird anschließend das Lernen initialisiert.

7. Das Lernen soll sowohl in der realen Welt als auch in simulierter Form möglich sein.

8. Handlungsvorgänge sollen abgespeichert und geladen werden können.

9. Lernverläufe sollen abgespeichert werden können.

10. Die Anwendung soll eine grafische Benutzeroberfläche besitzen, an die wiederum folgende Anforderungen gestellt werden:

    (a) **Intuitive Bedienung**: Die Oberfläche soll ohne Anleitung oder Handbuch bedienbar sein. Die einzige Voraussetzung ist, dass sich der Anwender im RL auskennt.

    (b) **Ausprobieren ohne Konsequenzen**: Die Oberfläche soll den Anwender benachrichtigen, wenn er eine Aktion ausführen möchte, die den Versuchsaufbau gefährden könnte.

    (c) **Professionelles Aussehen**: Da das Programm in Vorführungen genutzt werden soll, muss die Benutzeroberfläche ansprechend und professionell gestaltet sein.

    (d) **Visualisierung der Daten**: Damit das RL besser vom Anwender verstanden werden kann, sollen die Daten visuell aufbereitet werden.

1.3.2 Nichtfunktionale Ziele

Folgende nichtfunktionale Ziele lassen sich aus der Aufgabenstellung ableiten:

1. Die Anwendung soll auf den Rechnern des KI-Labors lauffähig sein.

2. Um die Austauschbarkeit von Programmteilen zu ermöglichen, soll das Programm modular aufgebaut sein. Dabei soll die Kohäsion möglichst hoch sein.

3. Das Programm muss so dokumentiert werden, dass eine Benutzung und Erweiterung des Programms auch nach langen Phasen der Nichtbenutzung möglich ist.

1.4 Abgrenzung

Diese Arbeit bildet nur einen Teil des gesamten Projekt ab. Der Beginn des Projekts und somit die Grundlage dieses Projekts, ist bereits in [Chr17] ausführlich behandelt worden und wird deswegen in dieser Arbeit nur an relevanten Stellen aufgegriffen. Sowohl Bilderkennung als auch Überhitzungsprobleme werden in dieser Arbeit nicht behandelt. Der Kern dieser Arbeit beschäftigt sich einerseits mit dem Erstellen eine grafischen Benutzeroberfläche, mit welcher Lernvorgänge konfiguriert, gestartet, gespeichert und überwacht werden können und andererseits mit dem Implementieren von verschiedenen RL-Algorithmen, die in vorangegangenen Forschungen gute Resultate erzielten und in der gegebenen Bearbeitungszeit, in Berücksichtigung ihrer Komplexität, umsetzbar erscheinen.

1.5 Aufbau der Arbeit

Im Folgendem wird der humanoide Roboter *Nao* vorgestellt, der für das RL in der realen Welt verwendet wird. Anschließend wird in Abschnitt 3 auf Seite 11 das Szenario vorgestellt. Die Aufgabe, die der Agent lösen soll, sowie der *Markov Decision Process* (MDP) im Allgemeinen und der MDP des Szenarios, werden erläutert. In Abschnitt 4 auf Seite 17 werden verwandte Arbeiten vorgestellt und Bezüge zu dieser Arbeit hergestellt. RL-Konzepte, die für das Verstehen der Arbeit wichtig sind, sind im Abschnitt 5 auf Seite 23 erklärt.

Abschnitt 6 auf Seite 33 befasst sich mit der Konzeption des Programms und im Besonderen mit der Konzeption des Agenten. Dadurch wird ein besseres Verständnis für den Aufbau des Programms vermittelt. Anschließend werden besondere Details bei der Implementierung des Programms besprochen. Da in dieser Arbeit eine grafische Benutzeroberfläche entwickelt wurde, wird diese in Abschnitt 8 auf Seite 51 erläutert.

Das Ergebnis der Arbeit ist ein Programm, welches mit RL eine gute Policy π in kurzer Zeit findet. Die Policy wird hierbei als *gut* erachtet, wenn der Agent wiederholt die minimale Anzahl von Aktionen in einer Episode benötigt und die definierte Aufgabe dabei erfolgreich löst.

Um herauszufinden, welche Parameter und Methoden sich am besten eignen, um möglichst schnell zu lernen, werden Versuchsreihen durchgeführt, die in Abschnitt 9 auf Seite 59 beschrieben und ausgewertet werden. Vielversprechende Methodiken, um das Lernen zu beschleunigen, werden miteinander verglichen, sodass am Schluss ermittelt wird, welche Methoden am besten sind, um diese Aufgabenstellung zu lösen. Weitere Ergebnisse der Versuchsreihen sind im Anhang zu finden. Abschließend wird ein Fazit in Abschnitt 10.1 auf Seite 73 gezogen und ein Ausblick für Folgearbeiten gegeben.

2 Nao

In diesem Abschnitt wird der Nao vorgestellt, der in dieser Arbeit beim RL in der realen Welt genutzt wird. Der Nao ist ein humanoider Roboter des Unternehmens Aldebaran [Ald17a]. Er wird zu pädagogischen Zwecken genutzt, beispielsweise kann er über eine visuelle Programmiersoftware programmiert werden. Schüler oder Studenten können hierdurch am Nao programmieren lernen. In der Hilton Hotelkette wird der Nao eingesetzt, um Gäste über Sehenswürdigkeiten o. Ä. zu informieren [Ald17b].

2.1 Hardware

Der Nao ist 573mm groß [Ald17a]. Der Nao besitzt verschiedene Gelenke und Servomotoren, die in Abb. 1 abgebildet sind. Die Gelenke und Motoren sind in Gliedern zusammengefasst [Ald17a]:

- **Head**: HeadPitch, HeadYaw.
- **RArm**: RShoulderRoll, RShoulderPitch, RElbowYaw, RElbowRoll, RWristYaw, RHand.
- **LArm**: LShoulderRoll, LShoulderPitch, LElbowYaw, LElbowRoll, LWristYaw, LHand.
- **RLeg**: RHipPitch, RHipRoll, RKneePitch, RAnklePitch, RAnkleRoll.
- **LLeg**: LHipYawPitch, LHipPitch, LHipRoll, LKneePitch, LAnklePitch, LAnkleRoll.

Wenn die Motoren aktiv sind, fließt durch sie Strom. Je stärker der Strom ist, der durch die Motoren fließt, desto leistungsfähiger werden diese. Durch den Strom werden die Motoren heiß und müssen deshalb regelmäßig entlastet werden, um nicht zu überhitzen. Wenn der Roboter eine Aufgabe erfüllt, bei der er nicht laufen muss, empfiehlt es sich, dass er sitzt um die Motoren in den Beinen zu schonen.

2.2 API

Um den Nao zu programmieren, stellt Aldeberan eine API namens `naoqi` bereit, welche für die Programmiersprachen C++, Java und Python zur Verfügung steht [Ald17a]. Um den Nao zu steuern oder Informationen abzufragen können sogenannte *Proxys* genutzt werden. Ein Proxy ist hierbei für einen Bereich zuständig, beispielsweise um Motoren zu bewegen oder das Kamerabild empfangen. Um einen Arm zu bewegen wird eine Instanz des `ALMotion-ALProxy`s benötigt. Um einen Proxy zu initialisieren wird die IP-Adresse des Nao benötigt sowie der Port, welcher standardmäßig `9559` ist und der Name des benötigten Proxys. Bei der Initialisierung wird versucht, über das Netzwerk[1] eine Verbindung zum Nao aufzubauen. Funktioniert dies nicht, wird eine Exception zurückgegeben. Der `ALMotion`-Proxy verfügt über eine Methode namens `positionInterpolations`, die es ermöglicht, ein – oder mehrere Körperteile gleichzeitig zu bewegen. Folgende Parameter werden für diese Methode benötigt:

1. `effectorNames`, String-Liste von Gliedern, die bewegt werden sollen.

[1] Sofern das Programm nicht auf dem Nao läuft

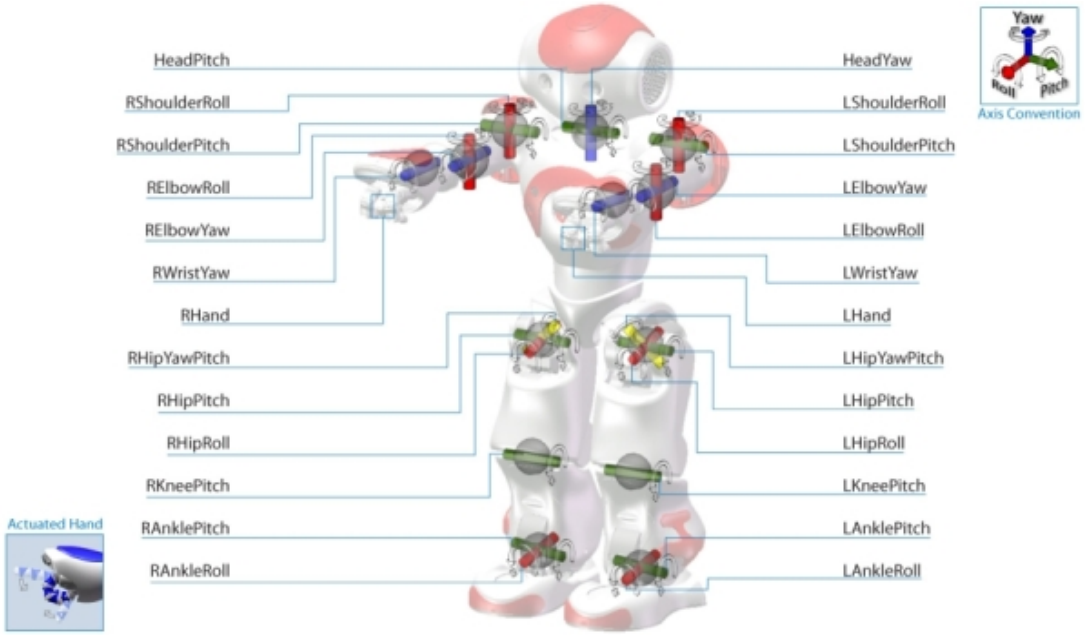


Abbildung 1: Gelenke und Servomotoren [Ald17a]

2. `taskSpaceForAllPaths`, Koordinatensystem, `{FRAME_TORSO = 0, FRAME_WORLD = 1, FRAME_ROBOT = 2}`. Je nach Aufgabe können verschiedene Koordinatensysteme genutzt werden, siehe Abb. 2:

   - `FRAME_TORSO`, der Nullpunkt liegt im Torso. Das Koordinatensystem ist nützlich für Bewegungen des Kopfes oder der Arme, wenn die Beinposition ignoriert werden soll.
   - `FRAME_ROBOT`, der Nullpunkt liegt zwischen den Füßen des Nao. Das Koordinatensystem kann verwendet werden, wenn der Roboter Bewegungen mit Beinen und Armen ausführen soll.
   - `FRAME_WORLD`, der Nullpunkt liegt zwischen den Füßen des Nao zu Beginn des Programms. Der Nullpunkt bewegt sich nicht, wenn der Nao läuft, dementsprechend ist dieses Koordinatensystem zu gebrauchen, wenn sich der Nao fortbewegt.

3. `paths`, Liste mit 6 Floats [x,y,z,wx,wy,wz]. Die ersten drei Floats bilden die Koordinate im Koordinatensystem. Die letzten drei Floats bilden einen Vektor, der die Richtung des Glieds im Koordinatensystem angibt. Es kann alternativ eine Liste von 6D-Vektoren angegeben werden um einen komplexeren Pfad zu bestimmen.

4. `axisMasks`, Integer im Wertebereich von 0 bis 63. Wenn `axisMasks` in binär umgerechnet wird, ergibt sich eine sechsstellige binäre Maske. Die Maske gibt an, welche Dimensionen von `paths` genutzt werden sollen. Beispielsweise wird bei $63 \rightarrow 111111$ der gesamte `paths` genutzt. Wenn `axisMasks` $7 \rightarrow 111000^2$ ist, werden nur die Positionskoordinaten genutzt.

5. `relativeTimes`, Float. Gibt die Dauer einer Bewegung an.

6. `isAbsolute`, Boolean. Gibt an, ob die Bewegung absolut oder relativ ist.

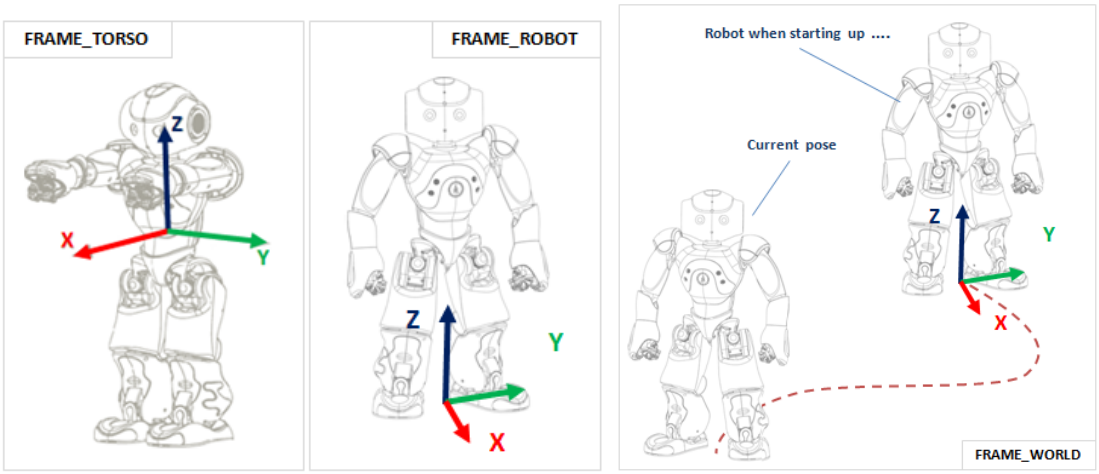


Abbildung 2: Koordinatensysteme des Nao [Ald17a]

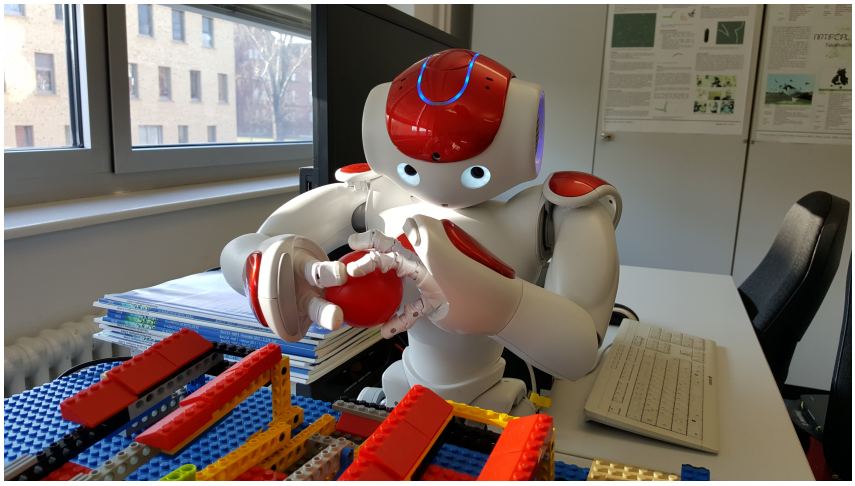


Abbildung 3: Der Nao führt einen *Ballwechsel* durch. Hierbei müssen die Hände dicht zusammen kommen. Dies ist nur umsetzbar, wenn die Kollisionskontrolle abgeschaltet wird, da sich andernfalls die Hände nicht so nahe kommen können.

Der Nao kann sich standardmäßig nicht selbst berühren. Aktionen, die beispielsweise das Berühren der Hände erfordern, wie in Abb. 3 zu sehen ist, sind nicht ohne weiteres durchführbar. Das liegt daran, dass der Nao über eine Kollisionskontrolle mit sich selbst verfügt, die ihn vor Schäden schützen soll. Um eben genannte Aktion ausführbar zu machen, muss diese Kollisionskontrolle abgeschaltet werden. Zu Beginn des Lernvorgangs wird der Nao in eine initiale Position gebracht. Währenddessen wird auch die Kollisionskontrolle mit den Methoden
`self.motionProxy.setCollisionProtectionEnabled('LArm', False)` und
`self.motionProxy.setCollisionProtectionEnabled('RArm', False)` abgeschaltet.
`motionProxy` ist hierbei eine `ALMotion`-Instanz. Die Kollisionskontrolle wird nur für die Arme abgeschaltet, da nur diese sich gegenseitig berühren können sollen. Nach dem Lernvorgang wird die Kollisionskontrolle wieder angeschaltet.

Um das Szenario, siehe Abschnitt 3 auf Seite 11, erfolgreich zu lösen, muss der Nao einen Plastikball greifen und halten können wie es in Abb. 4 zu sehen ist. Die `naoqi` bietet die Methoden `openHand` und `closeHand` an, die ein Öffnen und Schließen der Hände

[2]Das *least significant Bit* ist hierbei das Erste in Leserichtung

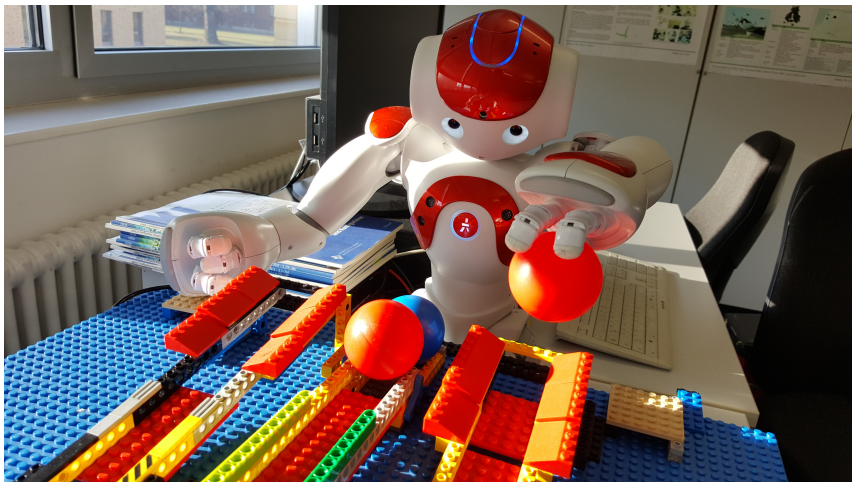


Abbildung 4: Der Nao hält einen Ball

ermöglichen [Ald17a]. Die Methoden sind aber für das Lösen der Aufgabe unzureichend, da sie die Motoren nach dem Schließen der Hand abschalten. Folglich kann der Nao den gegriffenen Ball nicht hochheben.

Das Problem wurde mit der unten gezeigten Methode `closeHand` gelöst, die als Eingabeparameter eine `ALMotion`-Instanz und einen Handnamen, also `"RHand"` oder `"LHand"` benötigt:

```
def closeHand(motionProxy,hand):
  ###
  # Summary: it will close Naos right or left hand
  # Parameters: motionProxy
  # Return: --
  ###
  # close: 0.01
  # open: 0.98
  motionProxy.stiffnessInterpolation([hand], 1, 1)
  motionProxy.angleInterpolation([hand],[0.01],[0.5],True)
```

Durch `stiffnessInterpolation` wird der Motor der Hand auch nach dem Greifen mit Strom versorgt. Die Methode `angleInterpolation([hand],[0.01],[0.5],True)` bewirkt ein Schließen der Hand.

2.3 Zusammenfassung

In diesem Abschnitt wurde der Nao beschrieben. Es wurde gezeigt, dass es wichtig ist, Motoren, die während des Lernens nicht aktiv sind, mit weniger Strom zu versorgen, oder abzuschalten. Des Weiteren wurde erklärt, dass der Nao über spezielle *Proxies* gesteuert wird. Im Detail wurde die Methode `positionInterpolations` der Klasse `ALMotion` behandelt, mit der die Gliedmaßen des Naos zu festgelegten Koordinaten bewegt werden können. Zwei Problematiken wurden beschrieben. Das Berühren der Hände wird bewerkstelligt, indem die Kollisionskontrolle beim Lernvorgang abgeschaltet wird. Für das Halten des Balls wurde eine Methode genutzt, die dafür sorgt, dass die Motoren nach dem Greifvorgang aktiv bleiben.

3 Szenario

Im Folgenden wird das *Szenario* aufgezeigt. Es wird beschrieben, welche Aufgabe der Lerner lösen soll. Anschließend wird die Aufgabe unter Gesichtspunkten des *MDP*s besprochen.

Der Nao, siehe Abschnitt 2 auf Seite 7, soll mittels RL bzw. Q-Learning, siehe Abschnitt 5.2.1 auf Seite 24, rote und blaue Bälle sortieren. Die Ballfarbe soll mittels Naos Kamera bestimmt werden. Ein Versuchsaufbau soll entwickelt werden, welcher beliebig lange ausgeführt werden kann. Das Programm wird in Python entwickelt und soll modular aufgebaut sein, sodass ein Austausch oder Hinzufügen von Zustände, Aktionen oder andere RL Algorithmen möglich ist. Es soll eine optimale Policy π^*, mit einem Minimum an Aktionsausführungen, gefunden werden[Chr17].

Der Versuchsaufbau wurde in [Chr17] ausführlich beschrieben. In Abb. 5 ist der Versuchsaufbau zu sehen. Die mittlere Schiene wird Source –, die beiden umliegenden Schienen werden Destination genannt.

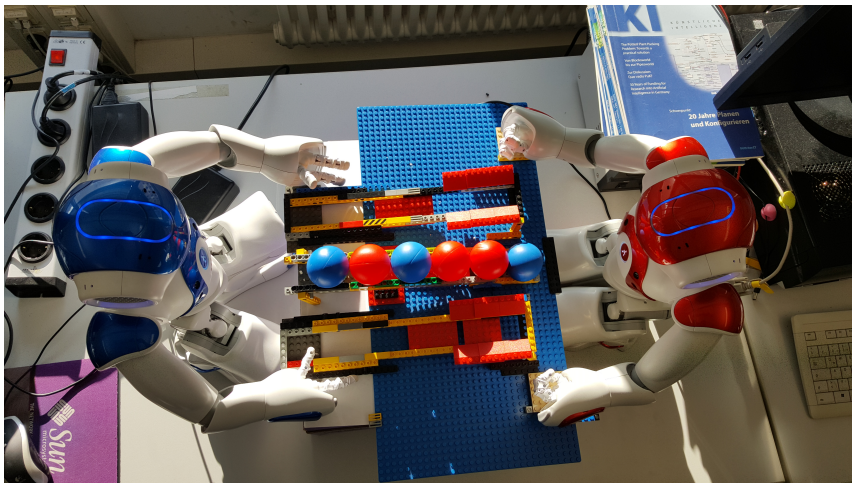


Abbildung 5: Der Versuchsaufbau aus der Vogelperspektive. Der lernende Roboter ist rot, der validierende Roboter ist blau.

Aufgabe des Lerners Der lernende Nao soll die blauen Bälle auf die von ihm aus linke Seite – und die roten Bälle auf die rechte Seite sortieren. Hierfür kann er in den einzelnen Zuständen bestimmte Aktionen ausführen. In Tabelle 2 sind die Aktionen aufgelistet und beschrieben.

Aufgabe des Validators Der validierende Nao (in Abb. 5 ist es der blaue Roboter) hat die Aufgabe die Bälle zurück zu sortieren und zu erkennen, ob der ankommende Ball in die richtige Schiene eingeordnet wurde. Zu welchem Zeitpunkt er die Bälle zurück sortiert, wird in Abschnitt 7.6 auf Seite 41 diskutiert.

Ist die Aufgabe episodisch? In [SB12, S. 49-51] ist eine Aufgabe episodisch, wenn sich die Interaktion intuitiv in einzelne Abschnitte unterteilen lässt. Diese Aufgabe lässt sich in die einzelnen Balleinsortierungsvorgänge unterteilen und kann deshalb als episodisch gesehen werden. Dennoch ist es möglich zu argumentieren, dass der Zustand nach einer Einsortierung weder zurückgesetzt, noch der Agent in einen anderen zufälligen Zustand

gebracht wird. Das Lernen geht nach dem Einsortieren kontinuierlich weiter. Daher kann die Aufgabe auch als kontinuierlich bezeichnet werden. In dieser Arbeit werden die einzelnen Einsortierungsvorgänge, aus oben genannten Gründen, als Episoden bezeichnet. Dennoch ist zu beachten, dass zwischen den Episoden kein Zurücksetzen des Rewards, siehe Abschnitt 3.1.3, stattfindet.

3.1 MDP

Der *Markov Decision Process* (MDP) ist eine RL-Aufgabe, die die *Markov-Eigenschaft* erfüllt. Als *Zustand* werden in dieser Arbeit alle Informationen bezeichnet, über die der Agent zu einem Zeitpunkt verfügt. In generellen, kausalen Zusammenhängen kann der aktuelle Zustand von allen vorangegangenen Zuständen und Aktionen abhängen. In diesem Fall ist die Wahrscheinlichkeit, in diesen Zustand zu gelangen, von der gesamten Historie des Agenten abhängig [SB12]:

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{1}$$

Wenn der Zustand jedoch die Markov-Eigenschaft erfüllt, hängt die Antwort der Umgebung bei $t + 1$ nur von der Zustands- und Aktionsdarstellung im Zeitpunkt t ab [SB12]:

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{2}$$

Ein MDP besteht aus

- einer Menge von Zuständen, $\mathcal{S}$,
- einer Menge von Aktionen, $\mathcal{A}$,
- einer Reward-Funktion $R : \mathcal{S} \times \mathcal{A} \to \Re$ und
- einer Zustands-Transitions-Funktion $T : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$, $\Pi(\mathcal{S})$ ist ein Teil einer Wahrscheinlichkeitsdistribution über die Menge $\mathcal{S}$. $T(s, a, s')$ ist die Wahrscheinlichkeit einer Transition von s nach s', unter der Nutzung der Aktionen a. [KLM96]

Im Folgenden wird der MDP der Aufgabe beschrieben.

3.1.1 Zustandsraum

Der Zustandsraum S ist fünfdimensional und diskret. Ein Zustand setzt sich aus Zustand der linken– und rechten Hand, Position des linken– und rechten Arms und Zustand an der Source zusammen. Teilzustände bzw. Features sind mit Ziffern kodiert, wobei die Kodierung für jeweils Hände und Arme gleich ist. Die Kodierung ist in Tabelle 1 zu finden. Die Reihenfolge, in der die Features in einem Zustand angegeben werden, ist fest und lautet wie folgt:

$(Hand_{rechts}, Hand_{links}, Arm_{rechts}, Arm_{links}, BallAtSource)$

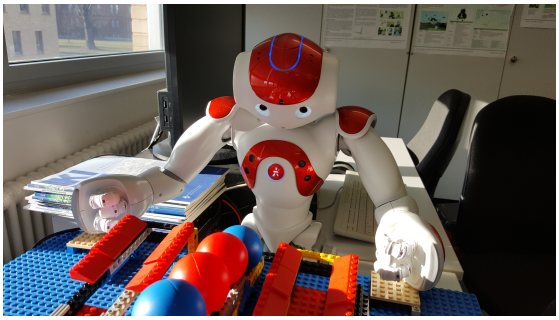
Die Unterteilung der Zustände in diese fünf Dimensionen kann vor allem im Bereich *Function Approximation* genutzt werden (siehe Abschnitt 5.4.2). Der Zustandsraum ist, ohne Abzug von unzulässigen Zuständen, 2500 Zustände groß.

Tabelle 1: Dekodierungstabelle der Zustände

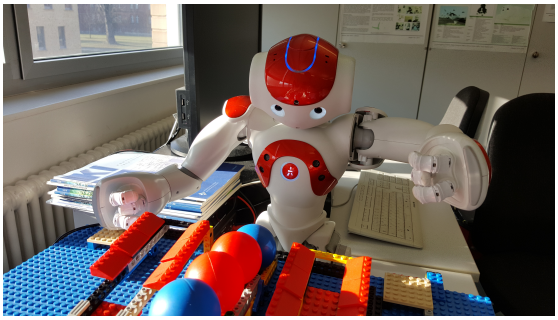
	Hand	Arm	Source
0	HandOpen	Arm an der *Start* Position	NoBall
1	HandClosedNoBall	Arm an der *Up* Position	UnknownBall
2	UnknownBall	Arm an der *Over Ball* Position	RedBall
3	RedBall	Arm an der *Source*	BlueBall
4	BlueBall	Arm über dem Ziel/*Destination*	—

Abkürzungen:

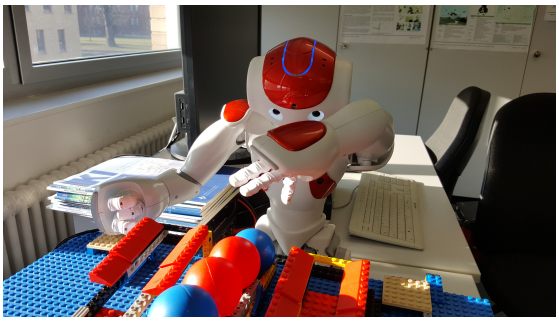
Destination = Äußere Schiene
Source = Mittlere Schiene



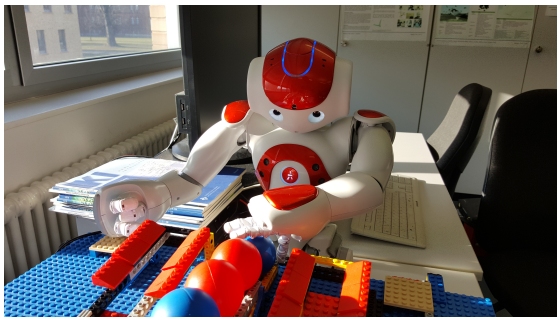
(a) Linker Arm an der *Start*-Position



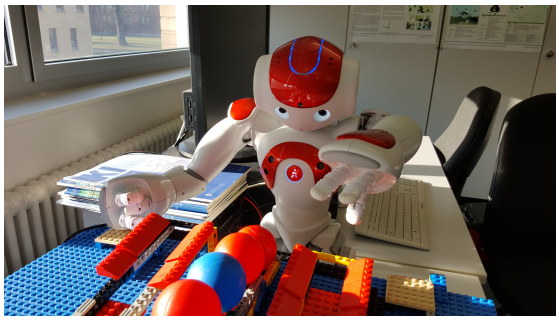
(b) Linker Arm an der *Up*-Position



(c) Linker Arm an der *Over Ball*-Position



(d) Linker Arm an der *Source*-Position



(e) Linker Arm an der *Over Destination*-Position

Abbildung 6: Positionen des Armes

3.1.2 Aktionen

Der Aktionsmenge $\mathcal{A}$ umfasst 19 Aktionen, welche in Tabelle 2 beschrieben sind. Bis auf die Aktion `BallDetectAtSource`, sind alle Aktionen doppelt vorhanden, also einmal für den rechten und einmal für den linken Arm. Eine Aktion kann nur ausgeführt werden, wenn sich der Agent in einem Zustand befindet, in dem er diese Aktion ausführen darf. Dies soll einer Beschädigung des Roboters, unvorhergesehenem Ballverlust und einen Übergang in unzulässige Zustände vermeiden. In Tabelle 3 ist vermerkt, in welchen Zuständen eine Aktion ausgeführt werden kann.

In der Tabelle sind `LHandOpen` und `RHandOpen` als Sonderfälle gekennzeichnet. Der Grund hierfür ist, dass beim Zustand eine Fallunterscheidung gemacht werden muss: Wenn die Hand ohne Ball geschlossen ist, darf der Agent die Hand aus jeder Armposition öffnen. Wenn ein Ball in der Hand ist, darf der Agent die Hand nur öffnen, wenn sich der Arm an der Position `Arm@Ball`, also an der Source, oder an der Position `ArmOverDestination` befindet. Dies soll einen Ballverlust im Laufe des Experiments vermeiden.

Tabelle 2: Übersicht mit Beschreibung der Aktionen

Name	Beschreibung
`LHandOpen`	Öffnen der linken Hand
`RHandOpen`	Öffnen der rechten Hand
`LHandClose`	Schließen der linken Hand
`RHandeClose`	Schließen der rechten Hand
`LArm2Start`	Linker Arm bewegt sich zur Startposition
`RArm2Start`	Rechter Arm bewegt sich zur Startposition
`LArm2Up`	Linker Arm bewegt sich zur Position „Arm@Up“
`RArm2Up`	Rechter Arm bewegt sich zur Position „Arm@Up“
`LArmOverBall`	Linker Arm bewegt sich zur Position „ArmOverBall“
`RArmOverBall`	Rechter Arm bewegt sich zur Position „ArmOverBall“
`LArm2Ball`	Linker Arm bewegt sich zur Position „Arm@Ball“
`RArm2Ball`	Rechter Arm bewegt sich zur Position „Arm@Ball“
`LArmOverDestination`	Linker Arm bewegt sich zur Position „ArmOverDestination“
`RArmOverDestination`	Rechter Arm bewegt sich zur Position „ArmOverDestination“
`BallDetectAtSouce`	Erkennt die Farbe des Balls auf der mittleren Schiene (Source)
`LBallDetect`	Erkennt die Farbe des Balls in der linken Hand
`RBallDetect`	Erkennt die Farbe des Balls in der rechten Hand
`SwapBallL2R`	Wechselt den Ball von der linken in die rechte Hand
`SwapBallR2L`	Wechselt den Ball von der rechten in die linke Hand

3.1.3 Reward

$\mathcal{R}(s, a, s')$ ist der erwartete Reward, wenn der Agent im Zustand s die Aktion a ausführt und in den Zustand s' gelangt. Der Reward ist eine *Rückmeldung*, die der Agent von der Umgebung bekommt, siehe Abb. 10, die dem Agenten mitteilt, ob die ausgeführte Aktion a im Zustand s, welche in Zustand s' führte, *gut* oder *schlecht* war. Der Agent erhält einen entsprechend positive oder negative meist reelle Zahl von der Umgebung. In diesem Szenario ist s' für den Reward unwichtig. Deshalb gilt hier $\mathcal{R}(s, a)$. Standardmäßig ist der Schrittreward auf -15 festgelegt. Einen negativen Schrittreward zu nutzen führt dazu, dass der Agent den kürzesten Weg finden muss, um seinen Return zu maximieren [SB12].

Tabelle 3: Übersicht, in welchen Zuständen eine Aktion ausgeführt werden soll. Eine leere Zelle bedeutet, dass der Zustand des Features für die Aktion irrelevant ist.

	links		rechts	
Name	Hand	Arm	Hand	Arm
LHandOpen	Sonderfall			
RHandOpen	Sonderfall			
LHandClose	0			
RHandeClose			0	
LArm2Start		1, 4		
RArm2Start				1, 4
LArm2Up		0, 2, 4		
RArm2Up				0, 2, 4
LArmOverBall		1, 3, 4		0, 1, 4
RArmOverBall		0, 1, 4		1, 3, 4
LArm2Ball	0	2		0, 1, 4
RArm2Ball	0	0, 1, 4		2
LArmOverDestination		1, 2		
RArmOverDestination				1, 2
BallDetectAtSouce		0, 1, 4		0, 1, 4
LBallDetect	2, 3, 4	1, 4		
RBallDetect			2, 3, 4	1, 4
SwapBallL2R	2, 3, 4	1, 4	0, 1	1, 4
SwapBallR2L	0, 1	1, 4	2, 3, 4	1, 4

Aus Abschnitt 1.3.1 geht hervor, dass das Lernen mit einem Roboter stattfinden kann, oder simuliert wird. Dies sind zwei verschiedene, jedoch gleichzeitig ähnliche MDPs. Die Vergabe des Rewards unterscheidet sich bei den MDPs.

Folgende Zustände geben bei dem simulierten MDP einen besonderen Reward:

1. `((4, X, 4, X, X), 'LHandOpen')`
2. `((X, 3, X, 4, X), 'RHandOpen')`
3. `((3, X, 4, X, X), 'LHandOpen')`
4. `((X, 4, X, 4, X), 'RHandOpen')`

Das X kennzeichnet hierbei die irrelevanten Dimensionen. Die Zustände 1 und 2 geben einen sehr großen Reward, Standardmäßig 500. Die Zustände 3 und 4 geben einen sehr kleinen Reward, standardmäßig -500. Der Wertebereich des Rewards, der in einem Schritt erlangt werden kann, geht also von -500 bis +500.

Wenn der Agent in der Simulation im Glauben ist, einen roten Ball auf die rechte Seite zu sortieren, dann gilt dies in der Simulation als Wahrheit und der Agent bekommt einen hohen Reward für die Ausführung der Aktion in diesem Zustand. Wenn der Agent in der realen Welt das Gleiche macht, erhält er nur den Reward, wenn der Validator erkennt, dass der Agent den Ball einsortiert hat.

Wenn der Agent in der Simulation einen unbekannten Ball einsortiert, bekommt er mit einer Wahrscheinlichkeit von 0,5 einen Reward von 500 und mit der gleichen Wahrscheinlichkeit einen Reward von -500. Bei dem Agenten in der realen Welt hängt in dieser Situation nach wie vor die Höhe des Rewards von der Beobachtung des Validators ab.

Abgesehen davon, dass ein *guter* Reward positiv und ein *schlechter* Reward negativ ist, ist die Höhe der Reward bisher relativ willkürlich bestimmt worden. Eine optimale Höhe des Rewards wurde bisher nicht ermittelt. Generell ändert die Anpassung des Rewards die Aufgabe, die der Agent lösen soll. Je nachdem, wie hoch der Reward ist, verhält der Agent sich anders. Dies zeigt sich besonders, wenn *Funktion Approximation* genutzt wird (siehe Abschnitt 5.4.2). Vorangegangene Erkenntnisse aus [Chr17] zeigten, dass sich der Reward mit -500 und +500 sehr gut im Estimator, siehe Abschnitt 5.4, über andere Zustandsaktionen ausbreitet.

3.1.4 Übergangswahrscheinlichkeiten

Es wird davon ausgegangen, dass der Versuch weitgehend deterministisch ist. Selbstverständlich ist der Versuch in der realen Welt nicht deterministisch, da Motoren nicht hundertprozentig akkurat funktionieren und es durch unvorhergesehene Probleme zu Fehlern etc. kommen könnte. In der simulierten Welt führen 16 von 19 Aktionen immer zu einem deterministischen Zustandsübergang. Drei Aktionen, `BallDetectAtSouce`, `LBallDetect`, `RBallDetect`, erkennen die Farbe der Bälle und führen dazu, dass ein Ball mit unbekannter Farbe, als blauer oder als roter Ball erkannt wird. Dementsprechend ist hier der Übergang nichtdeterministisch. In der simulierten Welt wird ein Ball mit unbekannter Farbe mit der Wahrscheinlichkeit von 0,5 als roter Ball, oder mit der gleichen Wahrscheinlichkeit als blauer Ball erkannt. In der realen Welt hängt die Ballerkennung von der Farbe des Balles ab, den der Nao in der Hand hält, da er die Ballfarbe durch seine Kamera ermittelt. Die Wahrscheinlichkeiten der Ballfarben, die erkannt werden, hängen in der realen Welt mit dem Verhältnis zwischen roten und blauen Bällen, die im Versuch vorhanden sind, zusammen.

3.2 Zusammenfassung

In diesem Abschnitt wurde das Szenario beschrieben, in welchem der Lerner verschiedenfarbige Bälle sortieren soll. Der validierende Nao erkennt, ob die Bälle richtig eingeordnet wurden. Die Aufgabe wurde mit einem *Markov Decision Process*(MDP) beschrieben, bei dem die Zustandsräume und die Aktionsräume diskret sind. Es gibt einen Schritt Reward von -15, positiven Reward von 500, wenn ein Ball richtig einsortiert wurde und negativen Reward von -500. Die Festlegung des Rewards ist eine Designentscheidung basierend auf Test und Erkenntnissen aus [Chr17], das wissenschaftliche Ermitteln eines angemessenen Rewards sollte jedoch Bestandteil einer folgenden Arbeit werden. Der MDP ist in der Simulation weitgehen deterministisch, Ausnahmen bilden die Aktionen, in denen die Farbe des Balls erkannt wird.

4 Verwandte Arbeiten

In diesem Abschnitt werden verwandte Arbeiten zusammengefasst. Außerdem werden Ähnlichkeiten und Unterschiede zwischen den verwandten Arbeiten und dieser Arbeit aufgezeigt. Verwandte Arbeiten wurden unter Berücksichtigung folgender Kriterien ermittelt:

- Die Arbeit löst eine Aufgabenstellung unter Verwendung von RL.
- Für die Aufgabe werden Roboter verwendet.
- Die Zustandsräume sind diskret.

Folgende Fragen sollten in den Arbeiten beantwortet werden:

- Welche RL-Algorithmen wurden genutzt?

  Welche Ergebnisse haben sie erzielt?
- Wie groß ist der Zustandsraum?
- Wie wurde der aktuelle Zustand des Agenten erfasst?

In [LLN17] wurde Q-Learning genutzt, um einem Nao beizubringen, beim Laufen Objekte zu umgehen. Für die Steuerung des Naos wurde das Robot Operating System (ROS) verwendet. Der Zustandsraum umfasst 5 Zustände. In Abb. 7 ist gezeigt, dass die Zustände davon abhängen, wie der Nao zu dem Objekt steht, welches er umgehen soll. Die Position des Naos zum Objekt wird über das Sonar des Naos ermittelt. Es gibt zwei Aktionen: Links und Rechts. Der Nao bewegt sich dementsprechend nach links oder nach rechts und wechselt seinen Zustand. Für jede Aktion gibt es einen Schrittreward von -1. Werden die terminalen Zustände $S0$ oder $S4$ erreicht, bekommt der Agent einen Reward von 10. Der Versuch wurde zuerst simuliert und anschießend mit dem Nao durchgeführt. Das Ergebnis war eine Q^*-Matrix. Durch die geringe Zustandsaktionsmenge wurde Q^* bereits nach 21 Episoden erreicht. [LLN17] weißt drei Parallelen mit dieser Arbeit auf:

1. Der Nao wurde genutzt um ein Problem zu lösen,
2. als RL-Methode wurde Q-Learning genutzt und
3. es gab einen negativen Schrittreward.

Da in dieser Arbeit der Zustandsraum um ein Vielfaches größer als in [LLN17] ist, wird das bloße Anwenden der Methodik von [LLN17] in dieser Arbeit vermutlich nicht ausreichen.

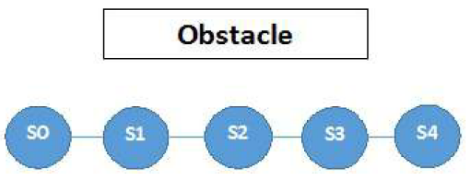


Abbildung 7: Versuchsskizze aus [LLN17]

In [SCSD17] wurde eine Aufgabe mit zwei unterschiedlichen Industrierobotern gelöst. Beide Industrieroboter verfügen über die gleichen Aktionen und den gleichen Zustandsraum. Sie befinden sich im gleichen Versuchsaufbau, wie auf Abb. 8 zu sehen ist, und müssen zwei Aufgaben lösen. Im Versuch befinden sich zwei *Werksstücke*, die durch einen RFID-Chip von den Robotern unterschieden werden können. Die Aufgabe der Roboter ist es, das erste Werkstück nacheinander auf die Positionen $S_1 - S_3 - S_3$ und das zweite Werksstück auf die Positionen $S_2 - S_3 - S1$, zu platzieren. Es ist möglich, dass sich die Industrieroboter behindern oder die Aufgaben kooperativ lösen. Beide Roboter werden über einen Siemens S7 gesteuert. In dieser Arbeit wird zwar über Multi-Agent Systeme gesprochen, es wird aber nicht weiter darauf eingegangen, ob beide Roboter von eigenständigen Agenten gesteuert werden, oder ob ein Roboter beide Agenten steuert. In der Arbeit wurde Q-Learning mit einer ε-greedy-Explorationsstrategie verwendet. Interessant war die Wahl des Rewards, welcher sich folgendermaßen zusammensetzt:

$$r_t = \frac{\alpha_1}{\Delta T_{t,1} + \Delta T_{t,2} + p_a + N_c w_c} \tag{3}$$

α_1 ist eine Konstante. $\Delta T_{t,1}$ und $\Delta T_{t,2}$ sind die Bearbeitungszeiten der beiden Roboter in dem Iterationsschritt. p_a ist eine *Bestrafung*, die auftritt, wenn eine Aktion nicht ausgeführt werden kann, beispielsweise wenn ein Werkstück an eine Position gelegt werden soll, die von einem anderen Werksstück blockiert wird. Die Roboter können bestimmen, in welcher Position sie sich befinden, verfügen aber sonst über keine zusätzliche Sensorik. Um eine Kollision der Roboter zu vermeiden, sind die Roboter von *Sicherheitszonen* umgeben. Wenn sich die Sicherheitszonen beider Roboter kreuzen, weicht der Roboter mit der geringeren Priorität zurück. Die Prioritäten sind vordefiniert. In Gleichung (3) ist N_c die Anzahl der Überschneidungen der Sicherheitszonen in einem Ausführungsschritt. w_c ist ein Wichtungsfaktor. Dementsprechend wirken sich Überschneidungen der Sicherheitszonen schlecht auf den Reward aus. Im Gegensatz zu anderen RL Lösungen in der Robotik, wie beispielsweise [DCP+13], ist sowohl der Zustands - als auch der Aktionsraum diskret. Was, wie in Abschnitt 3.1.1 gezeigt, auch auf den Zustands- und Aktionsraum dieser Arbeit zutrifft. [SCSD17] ergab, dass die ein Lernfortschritt erkennbar ist, es wurde jedoch im Lernverlauf ein Werkstück öfter genutzt, was dazu führte, dass das andere Werkstück, im späteren Verlauf, nicht mehr genutzt wurde. Des Weiteren lernten die Roboter, sich gegenseitig nicht in die Quere zu kommen.

In [GHLL17] wurden mittels Industrierobotern verschiedene Aufgaben in kontinuierlichen Zustandsaktionenräumen gelöst. In der Arbeit wurde eine asynchrone Variante von *Normalized Advantage Functions* (NAF) genutzt, was eine Alternative zu *Policy Gradient* und *Actor-Critic*-Methoden darstellt. Mit NAF kann Q-Learning, durch *Experience Replay*, für kontinuierliche Aufgaben verwendet werden [GLSL16]. Es sollten verschiedene Aufgaben gelöst werden:

- Zu einem zufälligen Ziel gelangen,
- einen Stab greifen un zu einer festgelegten Position bringen,
- eine Tür öffnen und
- eine Tür schließen.

Die Aufgaben wurden ausschließlich mit RL gelöst. Die Versuche wurden zuerst simuliert und danach in der Realität ausgeführt. Wie in [LPKQ16] wurden mehrere Roboter verwendet, um eine Aufgabe zu lernen. Für eine Aufgabe gibt es einen *Learner Thread* und

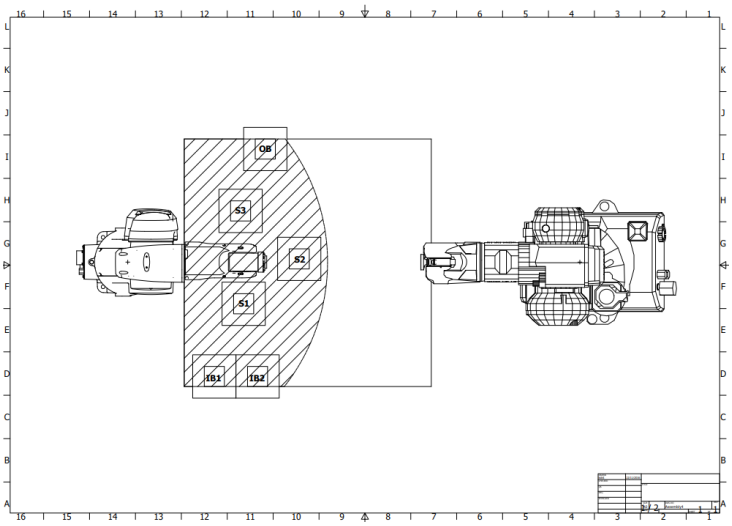


Abbildung 8: Versuchaufbau aus [SCSD17]

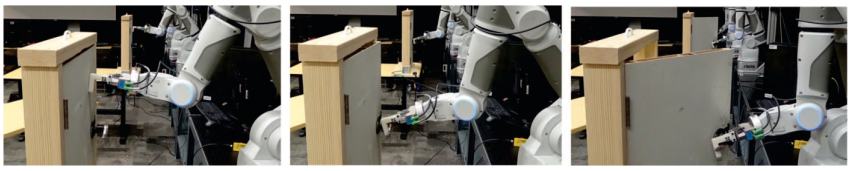


Abbildung 9: Zwei Roboter lernen das Türen-öffnen unter Verwendung von asynchronen NAF. Die finale Policy konnte, wenn zwei Roboter genutzt wurden, mit einer Wahrscheinlichkeit von 100% über alle 20 Versuche erreicht werden. Bild und Bildunterschrift sind aus [GHLL17]

N *Experience Collecting Worker Threads*. Der Learner Thread läuft auf einem zentralen Server und sendet Updates der Policy-Parameter an die Worker Threads. Die Worker Threads laufen auf den ausführenden Robotern und senden in jedem Iterationsschritt Observation, Aktion und Reward an den Learner Thread, der die Daten in den *Replay Buffer* schreibt. Der Learner Thread nutzt den Replay Buffer, um asynchron Updates auf dem *Deep Neural Network Q-Function Approximator* auszuführen. Durch die Trennung von Aktionsausführung und Update der neuronalen Netze kann die Aktionsausführung in Echtzeit laufen und wird nicht durch hohen Rechenaufwand verlangsamt. Der Learner Thread aktualisiert die Policy-Parameter der Worker Threads zu Beginn jeder Episode. Für die Aufgabe *Tür öffnen* beinhalten die Zustandsfeatures die sieben Gelenkwinkel des Industrieroboters, deren Zeitableitung, End-Effector-Position, die Position der Türklinke, die Position des Türrahmens, der Winkel der Tür, der Winkel der Türklinke und weitere. Ein Zustand enthält 25 Dimensionen. Die Rewardfunktion für einen Zustandsaktionspaar (x,u)[3] lautet:

$$r(x,u) = -c_1 d(h, e(x)) + c_2(-d(q_0, q(x)) + d_i) - c_3 u^T u \tag{4}$$

$d(h,e)$ ist der Abstand zwischen der Position des End-Effectors e und der Position der Türklinke h. $d(q_0, q(x))$ ist der Unterschied zwischen den aktuellen Winkel der Türklinke und dem Winkel der Türklinke, wenn sich die Tür öffnet. d_i ist Der Abstand zwischen der geschlossenen und der aktuellen Türposition. Dementsprechend wird der Reward größer, wenn der Roboter die Tür öffnet. In der Arbeit wurden für das Öffnen der Tür zwei Worker, die gleichzeitig lernten, genutzt. Das Ziel wurde nach ca 2,5 Stunden erreicht. Auf Abb. 9 sind Industrieroboter zu sehen, die unter Verwendung von asynchronen NAF, gelernt haben, eine Tür zu öffnen.
Die Rewardfunktion Gleichung (4) gibt genaue Angaben über die Umgebung und lässt eine vordefinierte Lösungsstrategie erahnen:

1. Bewege dich zur Türklinke.
2. Ändere den Winkel der Türklinke.
3. Bewege die Tür.

Für ein Problem, dessen Lösung für den Menschen offensichtlich ist, mag eine eng definierte Rewardfunktion eine gute Lösung sein. Es kann jedoch für schwierigere Probleme von Nachteil sein, die Rewardfunktion so eng zu definieren, da diese die Exploration von anderen Lösungstrategien, die das Problem eventuell besser lösen, behindern können. Wenn es beispielsweise einen Weg gäbe, die Tür zu öffnen ohne die Türklinke zu benutzen, könnte diese Lösung aufgrund der Rewardfunktion nicht gefunden werden. Aus diesem Grund wird bei den Versuchen in dieser Arbeit die Rewardfunktion *offener* formuliert, siehe Abschnitt 3.1.3.

4.1 Zusammenfassung

In diesem Abschnitt wurden drei Arbeiten vorgestellt. Alle Arbeiten befassten sich mit RL in der Robotik und benutzten Q-Werte, um den erwarteten Return von einem Zustandsaktionspaar zu ermitteln. Bei [SCSD17] und [GHLL17] wurde die Rewardfunktion mit

[3] x bedeuted, dass sich der Zustand in einem kontinuierlichen Zustandsraum befindet. u bedeutet, dass sich die Aktion in einem kontinuierlichen Aktionsraum befindet. Wenn die Räume diskret sind, werden Zustände mit s – und Aktionen mit a dargestellt

vielen Parametern gewählt, die bestimmte Lösungswege zu stark begünstigen könnten, während andere *gute* Lösungswege nicht entdeckt werden könnten. In dieser Arbeit wird desswegen die Rewardfunktion allgemeiner gehalten.

5 Theoretische Vorbetrachtung

In diesem Abschnitt wird zunächst RL erläutert. Anschließend werden einzelne Methoden des RL erklärt, die in dieser Arbeit verwendet oder beschrieben wurden, mit der Intention, dass sie in Folgearbeiten verwendet werden, siehe Abschnitt 10.2 auf Seite 73. Um einen besseren Überblick über die Methoden zu erreichen, wurden die Methoden in dieser Arbeit in folgende Kategorien unterteilt:

- Temporal Difference Learning in Abschnitt 5.2,
- Exploration in Abschnitt 5.3,
- Estimator in Abschnitt 5.4 und
- Planer in Abschnitt 5.5.

5.1 Reinforcement Learning

Abbildung 10: Agent-Umgebung Interaktion in RL (angelehnt an [SB12])

Beim Reinforcement Learning ist der Agent mit seiner Umgebung durch eine Interaktion von Wahrnehmung und Aktionen verbunden. Die Interaktion erfolgt in diskreten Zeitschritten, $t = 0, 1, 2, 3, ...$ In jeden Schritt t bekommt der Agent von der Umgebung einen Zustand, $s_t \in \mathcal{S}$, übermittelt. In diesem Zustand führt er eine Aktion, $a_t \in \mathcal{A}$, aus. Im darauffolgenden Zeitschritt erhält der Agent eine Reaktion der Umgebung, in Form eines numerischen Rewards, $r_{t+1} \in \mathbb{R}$, und bekommt einen neuen Zustand, s_{t+1}, von der Umgebung mitgeteilt. Anschließend kann der Agent eine neue Aktion, a_{t+1}, wählen, die er ausführt. Diese Interaktion zwischen Agent und Umgebung ist auf Abb. 10 dargestellt. Die Aktionen werden entsprechend der aktuellen Policy π, also seiner momentanen Handlungsstrategie, gewählt.[KLM96][Wat89][SB12]

Das Reinforcement Learning Model besteht aus:

- einer diskreten Menge von Zuständen, $\mathcal{S}$;
- einer diskreten Menge von Aktionen, $\mathcal{A}$; und
- einer Menge von Reinforcement Signalen; meistens {0,1}, oder reelle Zahlen.

[KLM96]

Aufgabe des Agenten ist es, den Reward, den er bekommt, langfristig zu maximieren. Oft ist jede durchgeführte Aktion mit einem negativen Reward, also mit Kosten verbunden. In einem Zielzustand befindet sich in der Regel ein positiver Reward. Eine Policy π^* gilt als optimal, wenn die Summe des Rewards r, Return R genannt, in einer Episode maximal ist [SB12]. Ziel des Agenten ist es, den summierten Reward, den er von der Umwelt bekommt, langfristig zu maximieren.[KLM96]

Zu Beginn des Lernens muss der Agent mit *Trial and Error* herausfinden, welche Aktionen ausgeführt werden müssen, um einen guten Reward zu erhalten. Wenn der Agent eine gute Strategie gefunden hat, um den Reward zu bekommen, steht er vor einem Problem: Entweder nutzt er seine Handlungsstrategie, also seine Policy aus, d.h. er macht das, was er kennt, um einen guten Reward zu erhalten, oder er wählt andere Aktionen aus und sucht neue Wege den Reward zu maximieren. Dieses Problem ist als *Exploration vs. Exploitation*-Problem bekannt und wird Abschnitt 5.3 auf Seite 28 näher behandelt.

Beim *Supervised Learning* (SL) lernt der Agent durch das Wissen eines externen Supervisors. Es ist zwar eine sehr wichtige Art des Lernens, jedoch für unerforschte Aufgaben, bei denen es keine gelabelten Daten gibt, von denen gelernt werden kann ist SL nicht möglich. Für solche Probleme ist RL besser geeignet, da der Agent von seinen eigenen Erfahrungen lernen kann.[SB12]

5.2 Temporal Difference Learning

Unter *Temporal Difference Learning*(TD-Learning) werden RL-Methoden zusammengefasst die in jedem Iterationsschritt ihre Policy anpassen. Im Gegensatz zu *Monte Carlo*-Methoden, die ihre Policy erst aktualisieren, wenn sie ein Return erhalten haben, also erst am Ende der Episode, nutzen TD-Methoden einen *Temporal Error*. Der Temporal Error ist der Unterschied zwischen der alten und der neuen Schätzung einer Value-Function. Der aktuelle Reward wird hierbei berücksichtigt. Des Weiteren werden nur die Werte eines Nachfolgers berücksichtigt, anstelle die Werte aller Nachfolger miteinzubeziehen.[SB12][KP12]

5.2.1 Q-Learning

Q-Learning, vorgestellt in [Wat89], ist ein modellfreier, off-Policy temporal difference Algorithmus. Das heißt, er benötigt kein Modell, die Policy hat keinen Einfluss auf den Algorithmus solange genügend exploriert wird. Außerdem ist der Algorithmus in der Lage eine optimale Policy π^* zu entwickeln. Dies gilt auch, wenn der Reward für eine Handlung, nicht sofort, sondern erst im späteren Verlauf auftritt.[SB12]

Im Gegensatz zur Value Funktion V benötigt Q den Zustand s und die dazugehörige Aktion **a** um den erwarteten Return zu bestimmen. $Q(s, a)$ gibt den erwarteten Return der Episode an, wenn der Agent in Zustand s die Aktion a ausführt.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma * \max_{a'}(Q(s',a')) - Q(s,a)) \tag{5}$$

s Aktueller Zustand

a Ausgeführte Aktion

$Q(s, a)$ Erwarteter Return im Zustand-Aktions-Paar (s,a)

α Lernrate

γ Diskontierungsfaktor

r Reward nach Ausführung von a

s' Folgezustand

a' Aktion im Folgezustand

Wenn jede Aktion, in jedem Zustand, in unendlich vielen Durchgängen, unendlich oft ausgeführt wird und die Lernrate α passend abgesenkt wird, konvergiert Q mit einer Wahrscheinlichkeit von 1 gegen Q^*, der optimalen Q-Funktion. [KLM96][Wat89]

Die optimale Policy $\pi^*(a)$ lässt sich, wie in Gleichung (6) gezeigt, ermitteln. Der Vorteil, gegenüber anderen Methoden, beispielsweise das Nutzen der Value Funktion V, siehe [SB12] [KLM96], ist, das kein Wissen über Transitionsfunktionen zu anderen Zuständen, nötig ist [KP12].

$$\pi^*(s) = \underset{a}{argmax}(Q^*(s,a)) \tag{6}$$

In Algorithmus 1 ist das Q-Learning zu sehen. S ist hierbei der aktuelle Zustand, A ist die ausgewählte Aktion im Zustand S, R ist der Reward, den der Agent erhielt, nach der Ausführung der Aktion A im Zustand S, S' ist der Folgezustand.

Algorithmus 1 : Q-Learning aus [SB12]

```
1 Initialize Q(s,a), ∀s ∈ S, a ∈ A(s), arbitrarily, and Q(terminal-state,·)=0
2 while True do // Wiederhole für jede Episode
3     Initialize S
4     repeat for each step of episode
          // Befehl(e) vor Aktionsausführung
          Choose A from S using policy derived from Q (e.g., ε-greedy)
          // Führe Aktion aus
5         Take action A, observe R, S'
          // Befehl(e) nach Aktionsausführung
6         Q(S,A) ← Q(S,A) + α[R + γ max_a Q(S',a) − Q(S,A)]
7         S ← S'
8     until S is terminal
```

5.2.2 SARSA

SARSA nutzt wie Q-Learning eine Funktion Q, die den erwarteten Return eines (s,a)Paares schätzt. Im Gegensatz zu Q-Learning ist SARSA eine on-Policy Lernmethode. Der Name SARSA ist zusammengesetzt aus den Bestandteilen eines Lernschritts, welche für das Update von $Q(s,a)$ verwendet werden: Im Zustand s wird die Aktion a ausgeführt, wodurch der Return r erhalten wird. Im folgenden Zustand *s'* wird Aktion *a'* gewählt. [SB12]

Anstatt den maximalen Q-Wert im Zustand *s'* zu ermitteln, wird in *s'* eine Aktion *a'*, durch die Policy oder durch die Explorationsmethode, gewählt. Der Q-Wert von *(s',a')* wird genutzt, um den Q-Wert von *(s,a)* zu aktualisieren. Gleichung (7) zeigt die SARSA Update-Gleichung.[SB12]

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma * Q(s',a') - Q(s,a))) \tag{7}$$

5.2.3 Q(λ)

Um die Wissensausbreitung von Q-Learning zu beschleunigen, ist es möglich, alle Zustandsaktionspaare einer Episode in Abhängigkeit von dem Diskontierungsfaktor γ und dem *trace-decay parameter* λ mit dem Reward, der im Iterationsschritt erlebt wurde, zu aktualisieren. Hierfür wird eine Funktion $Z(s, a)$ benötigt, die angibt, wie weit jedes Zustands-Aktions-Paar im gesamten Raum zurückliegt, bzw. wie relevant der aktuelle Reward für dieses Zustands-Aktions-Paar ist. $Z(s)$ ist in [SB12] folgendermaßen definiert:

$$Z_t(s) = \begin{cases} \gamma\lambda Z_{t-1}(s) & \text{if } s \neq S_t \\ \gamma\lambda Z_{t-1}(s) + 1 & \text{if } s = S_t \end{cases} \tag{8}$$

Da in diesem Projekt ausschließlich der Zustandsaktionsraum betrachtet wird, muss Gleichung (8) auf diesen Raum angepasst werden:

$$Z_t(s, a) = \begin{cases} \gamma\lambda Z_{t-1}(s, a) & \text{if } s \neq S_t \text{ and } a \neq A_t \\ \gamma\lambda Z_{t-1}(s, a) + 1 & \text{if } s = S_t \text{ and } a = A_t \end{cases} \tag{9}$$

Bei Gleichung (8) und Gleichung (9) ist zu sehen, dass die *Relevanz* des aktuellen Rewards für ein (s,a) von λ und γ abhängt. Beide Parameter liegen in einem Wertebereich zwischen 0 und 1. Je höher die Werte liegen, desto höher ist die Relevanz des aktuellen Rewards für (s,a), die weit in der Vergangenheit liegen. Damit Z *zerfällt*, muss die Gleichung (9) Bestandteil in jedem Lernschritt sein, wobei theoretisch über den gesamten Zustands-Aktions-Raum iteriert wird. Da alle $Z(S, A)$ nach einer Episode auf 0 gesetzt werden, werden nur die $Z(S', A')$ die in der Episode vorkamen aktualisiert.

Es gibt drei unterschiedliche Ansätze um $Q(\lambda)$ umzusetzen.

Watkins $Q(\lambda)$ Algorithmus 2 zeigt Watkins $Q(\lambda)$. Im Gegensatz zu anderen $TD(\lambda)$ Methoden, werden die (s,a)-Paare nur bis zur nächsten explorativen Aktion ausgeführt. Sobald eine nicht-gierige Aktion ausgeführt wird, werden alle $Z(s, a)$ auf null gesetzt und das *normale* one-step Q-Learning wird ausgeführt. [SB12]

Der Grund hierfür ist, dass durch Exploration viel schlechter Reward gesammelt wird, welcher sich negativ auf die vorangegangenen Zustandaktionen, welche der Policy folgten, auswirken könnten und dadurch die gelernte Policy zerstören [Wat89]. Bei häufiger Exploration werden die Traces dabei jedoch kürzer, was geringere Backupsequenzen zur

Folge hat.

Algorithmus 2 : $Q(\lambda)$ nach Watkins aus [SB12]

Initialize Q(s,a) arbitrarily, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
while *True* **do**
  Z(s,a) = 0, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  repeat for each step of episode
    `// Führe Aktion aus`
    Take action A, observe R, S'
    `// Befehl(e) nach Aktionsausführung`
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $A^* \leftarrow argmax_a Q(S', a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
    $\delta \leftarrow R + \gamma Q(S', A^*) - Q(S, A)$
    $Z(S, A) \leftarrow Z(S, A) + 1$
    forall $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ **do**
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta Z(s, a)$
      if $A' = A^*$ **then** $Z(s, a) \leftarrow \gamma\lambda Z(s, a)$ **else** $Z(s, a) \leftarrow 0$
    $S \leftarrow S'$; $A \leftarrow A'$
  until *S is terminal*

Pengs $Q(\lambda)$ Im Gegensatz zu Watkins $Q(\lambda)$ werden bei Pengs $Q(\lambda)$ nicht die Traces abgeschnitten sobald exploriert wird. Pengs $Q(\lambda)$ wird in Algorithmus 3 gezeigt. Anzumerken ist hierbei, dass $\hat{V}(s) = max_a(\hat{Q}(s, a))$ ist. Das Explorationsproblem, welches in Abschnitt 5.2.3 beschrieben wurde, wird bei Pengs $Q(\lambda)$ dadurch gelöst, dass zwei verschiedene Error, e_t und e'_t, erzeugt werden. e'_t wird mit $\hat{Q}(s_t, a_t)$ gebildet und wird nur für das one-step Q-update genutzt. e_t wird aus $\hat{V}(s)$, bzw. $max_a(\hat{Q}(s, a))$, gebildet und wird deswegen nicht schlecht beeinflusst wenn eine *schlechte* Aktion gewählt wird.

Algorithmus 3 : Pengs $Q(\lambda)$ aus [PW96]. Ein Zustand s ist hier mit x und Z mit Tr gekennzeichnet.

$\hat{Q}(x, a) = 0$ and $Tr(x, a)0$ for all x and a
while *True* **do**
  `// Befehl(e) vor Aktionsausführung`
  $x_t \leftarrow$ the current state
  Choose an action a_t according to current exploration policy
  `// Führe Aktion aus`
  Carry out action a_t in the world. Let the short-term reward be r_t, and the new state be x_{t+1}
  `// Befehl(e) nach Aktionsausführung`
  $e'_t = r_t + \gamma\hat{V}_t(x_{t+1}) - \hat{Q}_t(x_t, a_t)$
  $e_t = r_t + \gamma\hat{V}_t(x_{t+1}) - \hat{V}_t(x_t)$
  foreach *state action pair* **do**
    $Tr(x, a) = \gamma\lambda Tr(x, a)$
    $\hat{Q}_{t+1}(x, a) = \hat{Q}_t(x, a) + \alpha Tr(x, a)e_t$
  $\hat{Q}_{t+1}(x_t, a_t) = \hat{Q}_{t+1}(x_t, a_t) + \alpha e'_t$
  $Tr(x_t, a_t) = Tr(x_t, a_t) + 1$

Naives $Q(\lambda)$ Als Naives $Q(\lambda)$ oder auch McGovern $Q(\lambda)$ wird eine Mischung aus Pengs und Watkins $Q(\lambda)$ bezeichnet. Der Algorithmus ist wie Watkins $Q(\lambda)$, nur dass hier Traces nicht nach der Exploration auf null gesetzt werden.

5.3 Exploration

Im Gegensatz zum supervised learning (SL) muss der Agent beim RL seine Umgebung selbst erkunden[KLM96]. Der Agent steht in jedem Iterationsschritt vor der Wahl ob er seiner Policy folgt, also den Weg nimmt, der ihm nach seinem aktuellen Wissensstand den besten Return bringt, oder ob er exploriert, d.h., dass er policyunabhängig Aktionen ausführt um dadurch eventuell Wege zu finden um einen besseren Return zu erzielen. Es ist nicht möglich in einem Zeitschritt zu Explorieren und gleichzeitig der Policy zu folgen. Dieses Problem nennt sich „Exploration vs. Exploitation"[SB12, S. 27].

Auf kurze Sicht gesehen ist es günstiger gierig vorzugehen, also die Policy auszunutzen, da es den momentan besten Return verspricht. Jedoch auf lange Sicht ist es besser genügend zu Explorieren um bessere Wege zu finden und den Return zu maximieren[SB12, S. 27].

5.3.1 ε-greedy

Die ε-greedy Methode nutzt einen Parameter ε der Werte zwischen 1 und 0 annehmen kann. Dabei ist der Parameter die Wahrscheinlichkeit, mit der eine zufällige Aktion ausgewählt und dadurch exploriert wird. Mit der Gegenwahrscheinlichkeit, also $\varepsilon - 1$, wird dementsprechend der Policy gefolgt also die „gierige" (engl. greedy) Aktion gewählt.

Zu Beginn des Lernens kann viel Exploration betrieben werden, um gute Wege durch den Zustandsraum zu finden. Mit der Zeit, wenn genügend Exploration betrieben wurde, kann ε abgesenkt werden um den Return zu steigern.

5.3.2 Softmax

In Zustandsräumen, in denen eine ungünstige Aktion schwerwiegende Folgen, beispielsweise großen Regret[4], haben können, ist ε-greedy eine ungünstige Explorationsvariante, da, wenn exploriert wird, die Wahrscheinlichkeit eine günstige oder eine ungünstige Aktion auszuwählen gleichgroß sind[SB12, S. 30]. Für genanntes Szenario ist Softmax eine bessere Lösung. Hier findet die Exploration unter Berücksichtigung der Policy statt. Eine Aktion, die einen höheren Return verspricht, hat eine höhere Wahrscheinlichkeit, gewählt zu werden. Die verbreiteteste Softmax Formel lautet wie folgt:[SB12, S. 31]

$$\frac{e^{Q_t(a)}/\tau}{\sum_{i=1}^{n} e^{Q_t(a)}/\tau} \tag{10}$$

τ ist ein positiver Parameter, der „Temperatur" genannt wird. Ist die Temperatur hoch, sind die Aktionswahrscheinlichkeiten fast gleich. Wenn die Temperatur niedrig ist sind die Unterschiede der Wahrscheinlichkeiten größer. Dementsprechend wird der Agent immer gieriger, desto geringer die Temperatur wird.

[4]Regret ist negativer Reward

5.4 Estimator

Schätzer (engl. estimator) schätzen den erwarteten Return bei Erreichen eines Zustands oder wenn ein Zustands-Aktionspaar ausgeführt wird.

5.4.1 Q-Tabelle

Die Tabelle ist der primitivste Schätzer. Neue Q-Werte werden in eine Tabelle eingespeist, von der sie wieder ausgelesen werden können. Da die Daten nicht generalisiert werden, treten auch damit eingehende, sowohl positive, als auch negative, Effekte nicht auf. Nachteile sind, dass die Daten nicht komprimiert werden, sodass die Q-Tabelle sehr groß werden kann und der Agent bei Zuständen in denen er noch nicht war, seine bisher gesammelte Erfahrung auch nicht nutzen kann.

5.4.2 Function Approximation

Wie in Abschnitt 3.1.1 erwähnt ist der Zustandsraum 2.500 Zustände groß, abzüglich unzulässiger Zustände. Angenommen in jedem Zustand dürften alle 19 Aktionen ausgeführt werden, wäre der Zustands-Aktionsraum 47.500 Zustandsaktionspaare (sa-Paare) umfassen. Wenn der Agent pro Sekunde eine Aktion ausführt und er nie zweimal zu einem sa-Paar käme, könnte der Agent nur rund 13% des gesamten Zustandsaktionsraums in einer Stunde auskundschaften. Zwar muss der Agent, um zu einem terminalen Zustandsaktionspaar zu gelangen, nicht den ganzen Zustandsaktionsraum durchforsten, dennoch gleicht dieses Unterfangen *der Suche nach der Nadel im Heuhaufen.*

Um Speicherplatz zu sparen und bei ähnlichen Zuständen, bzw. sa-Paaren, ähnliche Q-Werte zu erhalten, müssen die Zustände *generalisiert* werden. Eine Generalisierung einer Funktion, wie beispielsweise der Q-Funktion, wird *Function Approximation* genannt. Function Approximation ist ein Teilgebiet des *supervised learning* und unter anderem in den Bereichen machine learning, artificial neural networks und pattern recognition behandelt [SB12, S. 213].

Um Zustände zu generalisieren, werden sie in *Features* unterteilt. Ein Feature ist in diesem Zusammenhang eine Eigenschaft eines Zustandes. Mehrere Zustände können dasselbe Feature besitzen. Ein Zustand kann in der Regel mit mehreren Features beschrieben werden.

Jedes Feature hat eine Wichtung. Die Wichtungen der Features eines Zustands sind in dem Vektor $\mathbf{w} = (w_1, w_2, ..., w_n)^\top$ enthalten. Die approximierte Value-Funktion wäre dementsprechend $\hat{v}(s, \mathbf{w})$.

Eine Klasse von Lernmethoden in diesem Bereich sind die **Gradienten-Abstieg-Methoden**. Es ist unwahrscheinlich, dass durch die Approximation alle $\hat{v}(s, \mathbf{w})$-Werte identisch mit den wahren $v_\pi(s)$-Werten sind. Es gibt eigentlich immer eine Abweichung zwischen den realen Werten und der Annäherung. Diese Abweichung wird als *root-mean-square-error*(RMSE) dargestellt:

$$RMSE(w) = \sqrt{\sum_{s \in S} d(s)[v_\pi(s) - \hat{v}(s, w)]^2} \tag{11}$$

wobei d : $S \to [0, 1]$, so das $\sum_s d(s) = 1$. $d(s)$ spezifiziert die relative Wichtigkeit des Fehlers im Zustand s. [SB12, S 215]

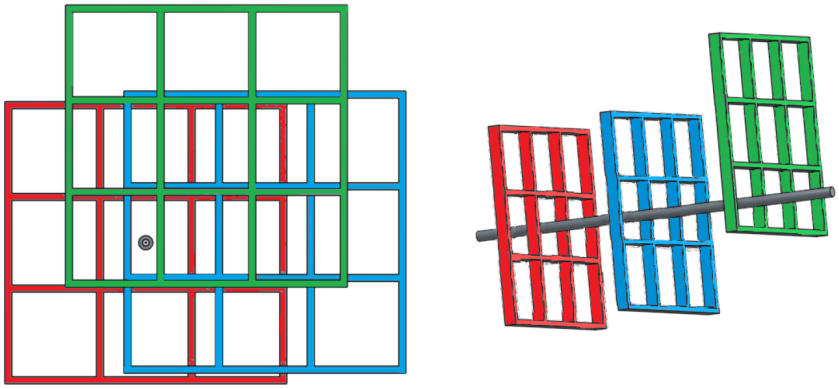


Abbildung 11: 2D- und 3D-Ansicht von drei überlappenden Tilings, die einen 2D-Zustandsraum abdecken. Die Gesamtanzahl der Features in diesem Beispiel ist 27, wobei nur 3 Features zurzeit aktiv sind, was durch den grauen Punkt, bzw. Linie, in den Tiles symbolisiert wird. Die Grafik ist aus [TP17].

Beim Gradienten-Abstieg wird der RMSE minimiert, indem der Wichtungsvektor w angepasst wird.

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla_{wt}[[v_\pi(s) - \hat{v}(s,w)]^2 = w_t + \alpha[v_\pi(s) - \hat{v}(s,w)]\nabla_{wt}\hat{v}(S_t, w_t) \tag{12}$$

α ist der Schrittweiten-Parameter und $\nabla_{wt}f(w_t)$ ist die partielle Ableitung eines Vektors mit einer Funktion f:

$$(\frac{\partial f(w_t)}{\partial w_{t,1}}, \frac{\partial f(w_t)}{\partial w_{t,2}}, \ldots, \frac{\partial f(w_t)}{\partial w_{t,n}})^\top \tag{13}$$

[SB12]

Tile Coding *Tile Coding*, auch *Cerebellar Model Articulation Controller* (CMAC) genannt, erstmals vorgestellt in [Alb75] nach [KLM96] und [SB12], ist eine Unterart des *Coarse Coding*. Coarse Coding ermöglicht die Repräsentation eines kontinuierlichen Zustandes mit überlappenden Features. *Tile Coding* ist für Probleme mit wenigen, also weniger als ca 100, binären Features ausgelegt ist. Ein d-dimensionaler Zustandsraum wird in m überlappende Partitionen, den sogenannten *Tiling*s unterteilt. Jedes Tiling besteht aus n *Tile*s. [TP17][SB12]

Ein Zustand enthält genau ein Tile aus jedem Tiling, dementsprechend ist die Anzahl der Features bei jedem Zustand gleich. Dies erlaubt es die Lernrate auf $\alpha = \frac{1}{m}$ zu setzen, wobei m die Anzahl der Tilings ist. Für ein langsames Lernen kann die Lernrate auch weiter abgesenkt werden, beispielsweise auf $\alpha = \frac{1}{10m}$. Um die Funktion zu schätzen, werden die einzelnen Tiles summiert. [SB12]

Um Interaktionen zwischen Dimensionen zu erkennen, müssen die Tilings den gesammten Zustandsraum so das jedes Tiling einen kleinen d-dimensionalen hyper-cube repräsentiert. Dies führt zu exponentiell vielen Features: $dim(\phi) = m * n^d$.[TP17]

Tilings können willkürlich, in einem Raster oder strategisch gesetzt werden. Es ist auch möglich sie spezifisch zu formen, um eine spezielle Form der Generalisierung zu erreichen. [SB12]

Um Speicherplatz zu sparen, ist es außerdem möglich, große Tilings in wenigen kleinen Tiles zu hashen. Dies kann jedoch eine Verschlechterung der Performance zu Folge

haben[SB12]. Da exponentielle Wachstum der Tiles kann durch das hashing begrenzt werden[TP17].

Eine Alternative zu Tile Coding wäre Kanerva Coding, welche komplexer als Tile Coding ist, sich jedoch besser für Zustandsräume mit vielen Dimensionen (mehr als hundert) eignet. [SB12][TP17]

5.5 Planer

Bei Q-Learning wird im jedem Schritt nur ein Q-Wert aktualisiert. Dieses neu gewonnene Wissen kann erst weiter verbreitet werden, wenn der Agent erneut in den Zustand gelangt. Um das Wissen schneller verbreiten zu können, werden planende Strategien eingesetzt, welche auf ein Modell der Umwelt zurückgreifen [SB12], die aus Erfahrungen des Agenten heraus entstehen können. Modelle können durch gerichtete Graphen beschrieben werden.

5.5.1 Prioritized Sweeping

Prioritized Sweeping(PS) wurde in [MA93] erstmals vorgestellt. Ziel von PS ist es den Rechenaufwand auf die *interessanten* Teile eines Problems zu richten. Ein Zustand gilt als *interessant*, wenn sich bei einem Update viel ändert, bzw. der Fehler bei einem Update groß ist. Dies könnte zu einem Update der Werte, der Vorgänger-Zustände, führen und somit lohnt es sich, die Vorgänger auch zu aktualisieren.[MA93]

Die Idee hinter PS ist die Q- oder V-Werte entsprechend ihrer *Interessantheit*, bzw. *Dringlichkeit*, zu aktualisieren. Hierfür wird eine Queue verwendet, in der jedes (s,a)-Paar[5] enthalten ist, von dem vermutet wird, dass es sich aufgrund des vorangegangenen Errors stark ändert. Die Queue ist priorisiert. Je *interessanter* ein (s,a)-Paar ist, desto höher ist die Priorität und desto höher wird das (s,a)-Paar in der Queue eingeordnet. Die Queue wird von hoher Priorität zu niedriger Priorität abgearbeitet[SB12].

Algorithmus 4 zeigt PS für Q-Learning. Der Error ist hierbei die Priorität. Wenn die Priorität einen Schwellwert θ überschreitet werden alle bekannten (s,a)-Paare, die direkt zum aktualisierten (s,a)-Paar gehören, mit der zuvor ermittelten Priorität zur Queue hinzugefügt. Aus Algorithmus 4 ist nicht erkennbar, wie der Reward in dem Model gespeichert wird. Entweder wird der Reward jedesmal, wenn ein (s,a)-Paar erlebt wurde, im Model überschrieben, oder das Model bildet den Mittelwert über alle erlebten Rewards. In [SB12] wird hierauf nicht weiter eingegangen. [MA93] bildet hingegen im Model einen Mittelwert über alle eingegangenen Rewards für ein (s,a)-Paar, wenn PS in einem MDP

[5](s,a)-Paar im Kontext von Q-Learning

angewendet wird.

Algorithmus 4 : Prioritized Sweeping für Q-Learning aus [SB12]

Initialize Q(s,a), Model(s,a), for all s,a, and $PQueue$ to empty
while *True* **do**
    `// Befehl(e) vor Aktionsauführung`
    $S \leftarrow$ current (nonterminal) state
    $A \leftarrow policy(S, Q)$
    `// Führe Aktion aus`
    Execute action A; observe resultant reward, R, and state, S'
    `// Befehl(e) nach Aktionsauführung`
    $Model(S, A) \leftarrow R, S'$
    $P \leftarrow |R + \gamma max_a Q(S', a) - Q(S, A)|$
    if $P > \theta$ **then** instert S, A into $PQueue$ with priority P
    for *n times,* ***while*** *PQueue is not empty* **do**
        $S, A \leftarrow first(PQueue)$
        $R, S' \leftarrow Model(S, A)$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
        forall $\bar{S}$, $\bar{A}$ *predicted to lead to* S **do**
            $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
            $P \leftarrow |\bar{R} + \gamma max_a Q(S, a) - Q(\bar{S}, \bar{A})|$
            if $P > \theta$ **then** insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

5.6 Zusammenfassung

In diesem Abschnitt wurden RL-Algorithmen vorgestellt, die relevant für diese Arbeit sind. Die Algorithmen wurden unterteilt in Temporal Difference Learning, Exploration, Estimator und Planer. In den folgenden Abschnitten wird die Umsetzung von einigen hier genannten RL-Algorithmen beschrieben. Beim RL entwickelt der Agent in einer Umgebung mit *Trial and Error* eine Handlungsstrategie, auch Policy genannt, die den erfahrenen Reward langfristig maximieren soll. Um die Policy weiterzuentwickeln, muss der Agent auch explorieren, d.h., dass er nicht immer seiner aktuellen Policy folgt, sondern auch davon abweicht, um bessere Wege zu finden, die Aufgabe zu lösen und mehr Reward zu erhalten. Die Exploration kann mit einer gewissen Wahrscheinlichkeit ε erfolgen, wie bei ε-greedy oder auch in Abhängigkeit von den Q-Werten, wie bei Softmax. Wenn der Zustandsraum sehr groß ist, können die Zustände generalisiert werden. Dadurch können Zustände, die die gleichen Eigenschaften haben, ähnliche Q-Werte zugewiesen werden. Wenn der Agent ein Modell der Umgebung besitzt, bzw. sich eines aufgrund von Erfahrungen erstellt hat, kann Prioritized Sweeping genutzt werden, um wichtige Erfahrungen, also (s,a)-Paare die sich stark geändert haben, an vorangegangene (s,a)-Paare im Estimator weiterzugeben.

6 Konzeption des Agenten

Im Folgenden wird die grobe Konzeption des Programms besprochen um den Agenten im Kontext des Programms einordnen zu können. Da in dieser Arbeit das Programm aus [Chr17], ist eine vollständige Beschreibung dort zu finden. Hier werden wichtige und geänderte Bestandteile erläutert.

Im Verlauf der Arbeit wird wie folgt vorgegangen: Zuerst wird das bestehende Programm auf die neuen Anforderungen angepasst. Insbesondere die Agenten-Klasse muss umstrukturiert werden, um mehr Modularität zu bieten. Zudem muss eine GUI konzipiert und umgesetzt werden. Sobald dies abgeschlossen ist, können vielversprechende RL-Methoden implementiert werden. Kriterien für die Auswahl der RL-Methoden sind dabei, wie kompliziert die Methoden zu implementieren sind und ob sie gute Resultate in anderen Arbeiten erzielten.

Das Programm ist in Schichten aufgebaut. Die oberste Schicht ist das GUI, welches über das Backend mit der Hauptanwendung verbunden ist. Das Backend der GUI greift ausschließlich auf die sogenannte `NaoSortingReinforcementAPI`, welche alle Funktionen des Hauptprogramms kapselt. Über die `NaoSortingReinforcementAPI` kann ein Lern-Thread gestartet werden. Der Lern-Thread besteht aus dem Agenten und dem Validator. Der Validator ist eine `Nao`- oder `NaoMock`-Instanz, siehe Abschnitt 2. Der Agent enthält eine `Nao`- oder `NaoMock`-Instanz.

Die unterste Schicht ist die `naoqi`, welche genutzt wird, um den Nao zu steuern. Die `Nao` Klasse greift auf die `naoqi` zu. Wenn die `naoqi` nicht auf dem ausführenden Computer vorhanden ist oder der Versuch nur simuliert werden soll[6], wird die `NaoMock` Klasse genutzt, welche ein Mockup der Nao-Klasse ist. Das heißt, dass die Methodenköpfe in beiden Klassen identisch sind, die `NaoMock` Klasse leitet jedoch die Anweisungen nicht an den Nao weiter.

Entscheidend für einen präzisen Vergleich der genannten Techniken ist ein flexibler Agent, der modular aufgebaut ist. Der Agent führt bei jeder Iteration Befehlssequenzen aus, die zur Laufzeit verändert werden können. Um dies zu erreichen, müssen zunächst verschiedene Reinforcement Algorithmen untersucht werden.

Algorithmus 1 auf Seite 25, Algorithmus 2 auf Seite 27, Algorithmus 3 auf Seite 27 und Algorithmus 4 auf Seite 32 sind RL-Algorithmen aus [SB12] und [PW96], die so, oder in veränderter Form, auch im Programm umgesetzt wurden. Auffällig sind die Gemeinsamkeiten im Ablauf der Algorithmen. So beinhaltet jeder Algorithmus die Ausführung einer Aktion, welche in den Algorithmen mit dem Kommentar `// Führe Aktion aus` markiert ist. Oftmals werden vor der Aktionsausführung Befehle ausgeführt, beispielsweise wird entschieden, ob exploriert wird. Diese Befehle sind mit dem Kommentar `// Befehl(e) vor Aktionsausführung` markiert. Nach der Aktion werden Schätzer und Modelle aktualisiert. Diese Befehle sind in den RL-Algorithmen mit dem Kommentar `// Befehl(e) nach Aktionsausführung` markiert. Gezeigte Algorithmen lassen sich also in gleiche Phasen unterteilen.

Es ist sinnvoll Parameter wie ε (siehe Abschnitt 5.3.1) zu verändern. Dies ist aber nicht unbedingt in den drei eben genannten Phasen notwendig. Parameter für die Exploration oder Lernrate können erst nach ein oder mehreren Episoden geändert werden. Daher kann noch eine Phase genutzt werden, die Methoden ausführt, wenn eine Episode endet.

Diese Sequenzen werden zu verschiedenen Zeitpunkten oder Ereignissen ausgeführt:

- Bevor die eigentliche Aktion ausgeführt wird

[6] also ohne Verwendung der Naos

  - Zu Beginn wird die laut Policy gewinnbringendste Aktion ermittelt und entschieden, ob exploriert wird

- Wenn das Ende einer Episode erreicht ist
  - Hier können Parameter angepasst und Observer benachrichtigt werden
  - Das Zustandsaktionspaar kann, für die Auswertung, als terminaler Interationsschritt gespeichert werden.

- Wenn das Ende noch nicht erreicht ist
  - Das Zustandsaktionspaar kann, für die Auswertung, als nicht terminaler Interationsschritt gespeichert werden.

- Nach der Ausführung der Aktion
  - Nach jeder Aktion beispielweise Policy und Model aktualisiert

Abb. 12 zeigt die Ausführungsschleife des Agenten für die Iterationsschritte. Sie zeigt, wie der Agent gestaltet werden kann, um die benötigte Flexibilität zu erreichten. Alle Funktionen, die der Agent ausführen soll, können zur Laufzeit, bei der Erzeugung des Agenten, in die entsprechenden Listen, als Funktionszeiger, hinzugefügt werden. Da die verschiedenen Methoden unterschiedliche Parameter benötigen, muss der Agent seine Parameter auch zur Laufzeit erzeugen können. Dies lässt sich mit einer assoziativen Liste umsetzten, bei der der *Schlüssel* der Name des Parameters ist. Die Umsetzung ist in Abschnitt 7.5 auf Seite 41 detailliert beschrieben.

6.1 Zusammenfassung

In diesem Abschnitt wurde gezeigt, dass sich RL-Algorithmen in gleiche Phasen unterteilen lassen. Aus diesem Ansatz wurde ein Lösungsweg entwickelt um den Agenten flexibler zu gestalten, sodass zur Laufzeit Methoden in die Ausführungsschleife des Agenten hinzugefügt werden können. Der Lösungsweg wird in Abschnitt 7.5 auf Seite 41 umgesetzt.

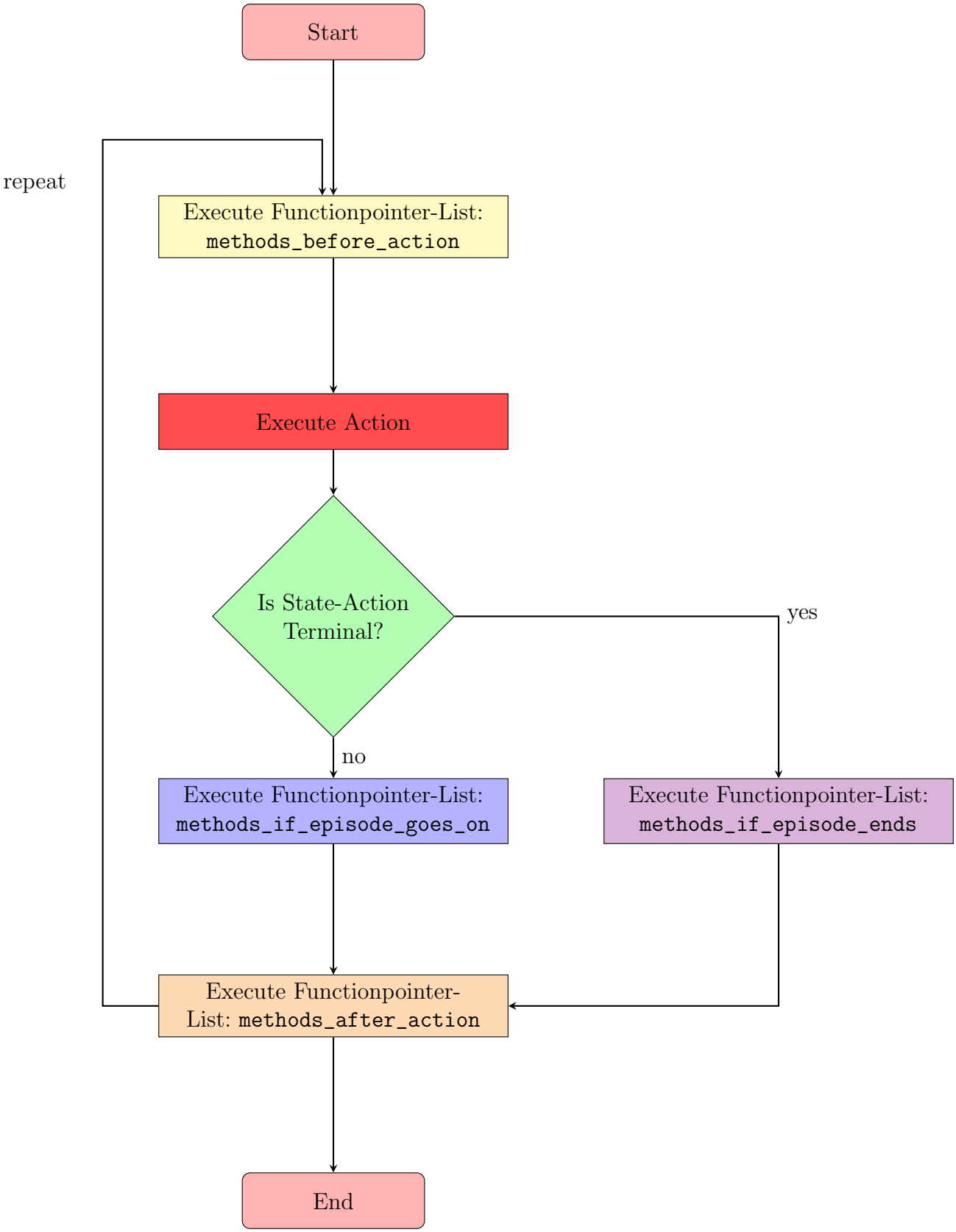


Abbildung 12: Flowchart generalisierter Algorithmus des Agenten

7 Implementierung

In diesem Abschnitt wird die Implementierung des Programmes betrachtet. Zuerst werden die wichtigsten Klassen des Programms erläutert. In Abschnitt 7.10 wird erklärt, wieso es notwendig ist die Koordinatenlisten, welche von Nao genutzt werden, gesondert in Dateien abzuspeichern und in welcher Form die Daten zur Laufzeit im Programm vorhanden sind. Anschließend wird in Abschnitt 7.11 darauf eingegangen, welche Teile des Programms nebenläufig sind und wie diese Daten austauschen. Abschließend wird die Umsetzung der RL Algorithmen, die in dieser Arbeit verwendet wurden, näher behandelt. Das Programm wird auf einem Rechner des KI-Labors implementiert und getestet.

7.1 Systemumgebung

Um mit dem Nao zu interagieren wird die Python Bibliothek `naoqi` genutzt. Dementsprechend wird der Großteil des Programms in Python 2.7.13 geschrieben. Das Graphical User Interface (GUI) wird in HTML und Javascript geschrieben. Der Vorteil ist, dass das GUI auf allen Systemen gleich aussieht und es einfache Frameworks gibt um ein GUI zu erstellen. Als Schnittstelle zwischen Python und HTML bzw. Javascript wird die Python Bibliothek `HtmlPy` genutzt. Das Programm kann sowohl auf einem Windows – als auch auf einem Linux Betriebssystem laufen.

7.2 Zustand

Die `State`-Klasse können Zustände repräsentiert werden. In der Klasse sind die Wertebereiche der einzelnen Dimensionen gespeichert. Wenn die Setter der Klasse genutzt

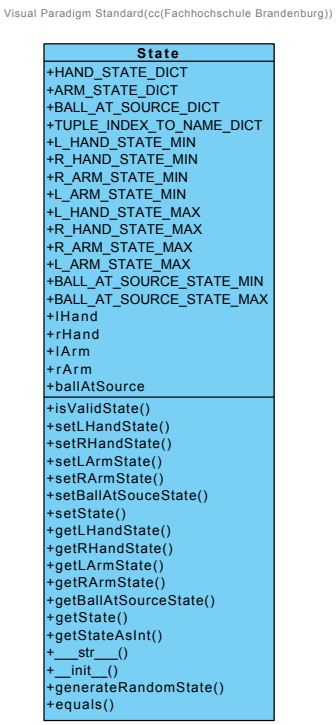


Abbildung 13: UML-Ansicht der Klasse `State`

werden, wird überprüft ob der Zustand valide ist. Sollte dies nicht der Fall sein, wird eine

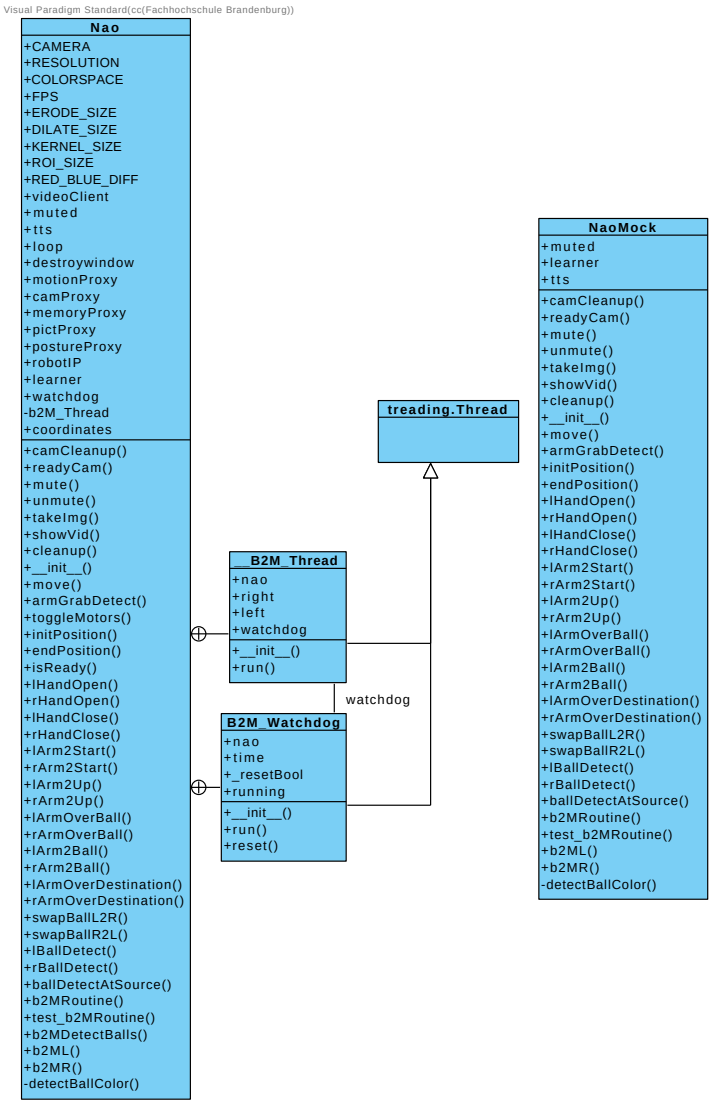


Abbildung 14: UML-Ansicht der Klassen `Nao`, `NaoMock`, `B2M Thread` und `B2M Watchdog`

entsprechende Exception geworfen. Im Programmcode ist die Unterscheidung von `State`-Instanzen und `State`-Tupeln wichtig. `State` Instanzen sind Objekte der Klasse `State`. Sie werden während Lernvorgängen genutzt und verändert. Wenn ein (s,a)-Paar an einen Estimator übergeben wird, muss die `State`-Instanzen in ein Tupel geändert werden, da s ein Zustandstupel und keine interne Objektadresse beinhalten soll. Das Tupel kann mit der `getState`-Methode von der Instanz erzeugt werden. Ein Tupel kann über den Konstruktor oder der `setState`-Methode in eine Instanz überführt werden.

7.3 Nao

Die Klasse `Nao` stellt das Bindeglied zwischen Roboter und Programm dar. Alle Aktionen, die der Nao ausführen soll und alle Informationen, die das Programm über den Nao benötigt, werden über diese Klasse gesteuert. Jede `Nao`-Klasse ist für genau einen Nao zuständig. Bei der Initialisierung der Klasse, wird werden `ALProxy`s eingerichtet, über die der Nao gesteuert wird. Da jeder Roboter einen eigenen Satz von Koordinaten benötigt,

werden diese bei der Initialisierung aus einer CSV-Datei geladen, siehe Abschnitt 7.10 auf Seite 45. Sollte bei der Initialisierung keine explizite Datei für die Koordinaten angegeben werden, werden die passenden Koordinaten anhand der IP des Naos ermittelt. Weiterhin wird bei der Initialisierung angegeben, ob der Roboter ein Lerner ist. Wenn der Roboter kein Lerner ist, ist er ein Validator. Ein Validator, darf die Bälle zurücksortiern. Ein Lerner verweigert die Ausführung des Befehls, da sonst der Versuchsaufbau beschädigt werden könnte.

In den ersten Versuchen ergaben sich zwei Probleme beim Zurücksortieren der Bälle:

1. Wenn oft Bälle einsortiert werden, muss der Validator die Arme oben halten, um schnell zu reagieren. Wenn die Bälle selten einsortiert werden, ist es jedoch schlecht, wenn der Validator die Arme oben hält, da die Arme in dieser Position schnell überhitzen.

2. Wenn mehrere Bälle auf einer Destination-Schiene liegen müssen alle einsortiert werden. Es ist jedoch ungünstig, wenn der Lerner wartet, bis der Validator alle Bälle einsortiert hat.

Punkt 1 wird durch einen `B2M_Watchdog`-Thread gelöst. Wenn der `B2M_Watchdog`-Thread nicht aktiv ist, wird er aktiviert, sobald die Methode `b2MRoutine` ausgeführt wird. Sobald der Thread aktiviert wurde, läuft im Thread ein Timer ab. Sollte die Methode `b2MRoutine` vor Ablauf des Timers ausgeführt werden, wird der Timer zurückgesetzt. Sobald der Timer abläuft, werden die Arme des Validators in eine Ruheposition gefahren.

Punkt 2 wird durch den `__B2M_Thread` gelöst. Die Methode `b2MRoutine` ermittelt auf welcher Seite Bälle liegen. Am Ende der `b2MRoutine` wird ein `__B2M_Thread` gestartet. Anschließend beendet sich die `b2MRoutine` Routine und gibt zurück, auf welcher Seite Bälle liegen. Der gestartete `__B2M_Thread` sortiert alle Bälle zurück. Da `__B2M_Thread` ein Thread ist, muss der Lerner nicht auf den Validator warten, bis die Bälle zurücksortiert wurden.

Da `__B2M_Thread` und `B2M_Watchdog` nur in der Klasse Nao genutzt werden, wurde beide Klassen in der Nao Klasse als innere Klassen implementiert, was in Abb. 14 zu sehen ist.

Sowohl der Validator, als auch der Lerner müssen die Ballfarben erkennen können. Um den Aufwand zu minimieren wurde die Erkennung durch *Regions of Interest*(ROI) gelöst. In den Koordinatenlisten der Naos ist die obere linke Ecke für die quadratische Region, welche eine Kantenlänge von 30 Pixeln besitzt, enthalten, siehe Abschnitt 8.3 auf Seite 55. Eine ROI ist ein Ausschnitt des Kamerabildes des Nao. Um die Farbe zu ermitteln, werden die RGB Werte der Region zusammengerechnet. Anhand von Schwellwerten, die in den Koordinatenlisten enthalten sind, wird bestimmt, ob die Region eher rot, eher, blau oder eher gar nichts von beidem ist. Dementsprechend wird geschlussfolgert, dass an dieser Stelle im Bild ein roter, ein blauer oder kein Ball ist. Der Nachteil dieser Methode ist, dass sie, durch die festen Schwellwerte, in dunklen Lichtverhältnissen unzuverlässig ist.

Die Klasse `NaoMock` ist ein *Mockup*, also eine *Attrappe*, der Klasse `Nao`. `NaoMock` besitzt alle relevanten Schnittstellen der Klasse `Nao`. Da `NaoMock` den Nao nicht ansteuert, wird weder Roboter, noch `naoqi` benötigt. In anderen Programmiersprachen müssten Interfaces verwendet werden, um eine Klasse zu simulieren. In Python genügt es jedoch, wenn die verwendeten Schnittstellen vorhanden sind. Eine anderer Lösungsansatz wäre gewesen, dass die Klasse `NaoMock` von `Nao` erbt. Dies ist jedoch problematisch, da die `naoqi` importiert werden müsste, wenn `NaoMock` verwendet wird. Dies ist ein Nachteil, da

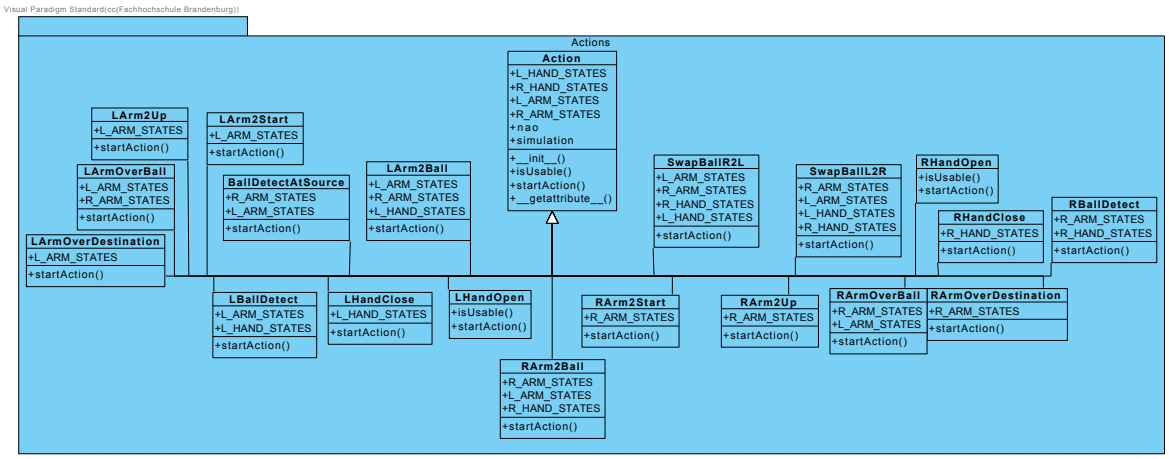


Abbildung 15: UML-Ansicht der Aktionen

die `naoqi` nicht auf jedem Computer vorhanden ist, auf dem das Programm ausgeführt werden soll. Wenn die `naoqi` auf einem Computer nicht installiert ist, kann das Programm ausschließlich Lernsimulationen ausführen.

7.4 Aktionen

Aktionen werden in diesem Projekt als Klassen implementiert, die von einer Klasse `Action` erben, wie in Abb. 15 gezeigt ist. Alternativ hätten Aktionen auch als Funktionen implementiert werden können, dann wäre jedoch Vererbung nicht möglich gewesen, was sich an manchen Stellen als sinnvoll erweist. Eine Aktionsklassen besitzen zwei Methoden: `isUsabe(currentState)` gibt eine Eins zurück, wenn die Aktion in diesem Zustand ausgeführt werden kann, ansonsten gibt die Methode eine Null zurück. Um die Ausführbarkeit zu ermitteln, besitzt jede Klasse vier Listen `L_HAND_STATES`, `R_HAND_STATES`, `L_ARM_STATES` und `R_ARM_STATES`. Jede Liste steht für jeweils eine Dimension des Zustandsraumes, ausgenommen ist die Dimension *BallAtSource*, da diese für die Ausführbarkeit der Aktionen keine Rolle spielt. In der Klasse `Action` sind in jeder Liste die jeweiligen Teilzustände enthalten. Beispielsweise enthält die Liste `L_HAND_STATES` in `Action` die Teilzustände 0, 1, 2, 3, 4. Die Listen geben an in welchen Teilzuständen die Aktion nutzbar ist. Laut Tabelle 3 sehen die Listen für die Aktion `LArmOverDestination` folgendermaßen aus:

- `L_ARM_STATES = (1, 3, 4)`
- `R_ARM_STATES = (0,1,4)`

Die anderen beiden Listen bleiben unverändert.

`startAction(currentState)` führt die jeweilige Aktion aus, wenn die Aktion im aktuellen Zustand genutzt werden kann. Alle Aktionsklassen halten entweder eine `Nao` – oder eine `NaoMock` Instanz. In der Aktions-Methode `startAction(currentState)` wird die entsprechende Methode dieser `Nao`/`NaoMock` Instanz ausgeführt. Anschließend wird der aktuelle Zustand (`currentState`) angepasst.

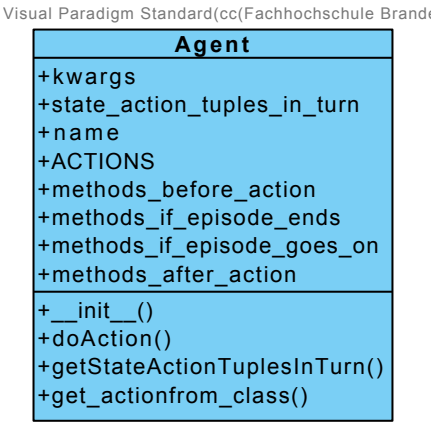


Abbildung 16: UML-Ansicht der Klasse `Agent`

7.5 Agent

Entsprechend Abschnitt 6 wurde Klasse `Agent` implementiert. Wie in Abb. 16 zu sehen ist, besitzt die `Klasse` vier Listen, in die Funktionen hinzugefügt werden können. Funktionslisten werden, wie in Abb. 12 gezeigt ist, zu verschiedenen Zeitpunkten im Iterationsschritt ausgeführt.

7.5.1 Erzeugung eines Agenten-Objekts

Da ein Agentenobjekt in der Erzeugung kompliziert ist, soll das Objekt ausschließlich durch die Methode `create_agent(...)` erzeugt werden. Die Methode entspricht dem *Erbauer* Entwurfsmuster [GHJV95]. D.h., dass die Methode alle gegebenen Parameter an die richtigen Stellen des Agenten einbaut. Der Agent besitzt kaum eigene Parameter. Grund hierfür sind die oben genannten Befehlssequenzen.

Es ist problematisch, die passenden Parameter für jede Methode bereitzustellen, da diese zur Laufzeit hinzugefügt werden können. Deswegen werden alle wichtigen Parameter in einem `dictionary` namens `kwargs` (keyword-argruments) bereitgestellt, in dem die Parametername und Parameterwert ein Key-Value-Paar darstellen. Die Übergabe und das Aussehen dieser List ist in Anhang B auf Seite a gezeigt.

7.6 Rewarder

Die `Rewarder`-Klasse vergibt den Reward für das ausgeführte (s,a)-Paar. Außerdem bestimmt der Rewarder ob ein terminales (s,a)-Paar erreicht wurde, oder nicht. Zwar gehört der Rewarder in der RL-Theorie, Abschnitt 5.1, zur Umgebung, jedoch müssen sich Agent und Rewarder untereinander austauschen. Deswegen enthält eine Instanz der Klasse `Agent` eine Referenz auf eine Instanz der `Rewarder`-Klasse. Wie in Abb. 17 zu sehen ist, enthält der Rewarder eine Variable namens `validator`. Dahinter verbirgt sich eine `Nao` – oder `NaoMoch`-Instanz, welche validiert, ob der Lerner erfolgreich war. Hier bieten sich nun zwei Optionen für die Umsetzung.

1. Entweder lässt der Rewarder den Validator nach jeder ausgeführten Aktion schauen, ob Bälle auf den Destination-Schienen liegen,

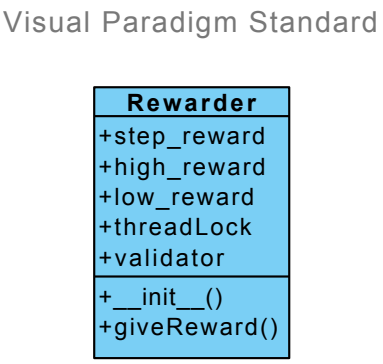


Abbildung 17: UML-Ansicht der Klasse `Rewarder`

2. oder der Rewarder lässt den Validator erst nachschauen, wenn der Agent ein (s,a)-Paar ausgeführt hat, indem es möglich ist, dass beim Validator ein Ball ankommt.

Punkt 1 ist die *realistischere* Lösung, es ist möglich, dass unerwartete Aktionen zu einem Ziel führen. Es gibt jedoch Probleme mit dieser Lösung: Wenn ein Ball vom Agenten fallen gelassen wird, benötigt er ca. zwei Sekunden, bis der Ball am Validator angekommen ist. Hierfür gibt es wiederum zwei Lösungsansätze: Entweder wartet der Validator in jedem Schritt zwei Sekunden, was offensichtlich zu einer erheblichen Verlangsamung des Lernablaufes führt, oder der Validator schaut sofort nach jedem Schritt, was zur Folge hat, dass der Reward kurzfristig mit einem *falschen* (s,a)-Paar in Verbindung gebracht wird. Dies ist jedoch aus theoretischer Sicht kein Hindernis, da TD-Learning, siehe Abschnitt 5.2, auch für solche Szenarien funktioniert. Das Problem ist, dass Policy in diesem Szenario langsamer konvergiert, da der Agent erst herausfinden muss wofür er den Reward bekam. Ähnliche Probleme ließen sich auch mit Tile Coding, siehe Abschnitt 5.4.2, Ansätzen entdecken.

Punkt 2 ist wesentlich leichter, da der Validator nur selten aktiviert wird, kann er zwei Sekunden auf den Ball warten. Dadurch ist gewährleistet, dass der Reward mit dem *richtigen* (s,a)-Paar in Verbindung gebracht wird.

In diesem Projekt wird der Ansatz von Punkt 2 aus Gründen der Einfachheit umgesetzt. Der Ansatz von Punkt 1 ist dennoch interessant und kann deshalb in zukünftigen Arbeiten in Angriff genommen werden.

7.7 Logger

Die Aufgabe der Klasse `Logger` ist die Erstellung einer Lernhistorie für die spätere Auswertung der Versuche und das Triggern von Events beim `EventHandler`, siehe Abschnitt 7.11 auf Seite 46. Die Methoden `appendActionTuple` und `finishTurn`, in Abb. 18 zu sehen, werden bei der Ausführung der Methode `create_agent` in die entsprechenden Funktionslisten des Agenten hinzugefügt.

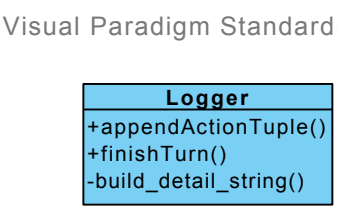


Abbildung 18: UML-Ansicht der Klasse `Logger`

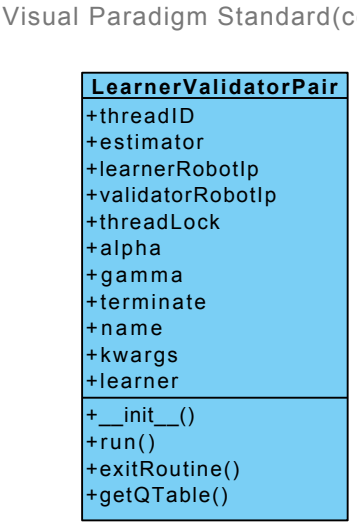


Abbildung 19: UML-Ansicht der Klasse `LearnerValidatorPair`

Visual Paradigm Standard(cc(Fachhochschule Brandenburg))

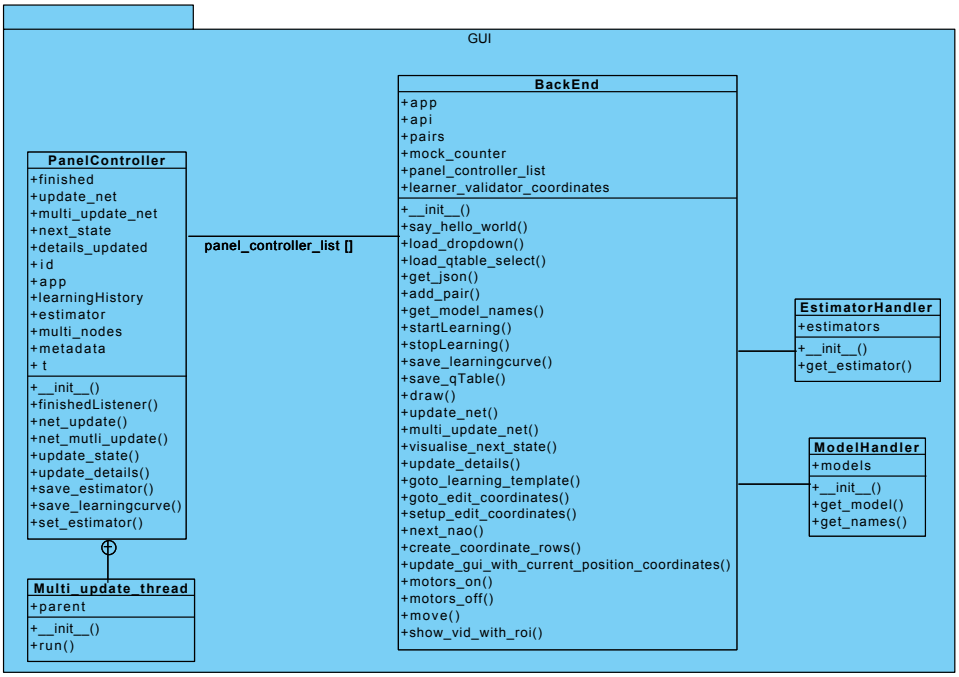


Abbildung 20: UML-Ansicht der GUI-Klassen

7.8 LearnerValidatorPair

Die Klasse `LearnerValidatorPair` erbt von `treading.Thread`. Im `LearnerValidatorPair`-Thread führt der Agent die Lernschritte innerhalb einer Schleife aus. Der Thread initialisiert den Agenten über die `create_agent`-Methode, siehe Abschnitt 7.5.1 und den Validator, welcher eine Instanz der Klasse `Nao` oder `NaoMock` ist, siehe Abschnitt 2. Bei der Initialisierung des Threads wird dem Thread eine assoziative Liste von Parametern übergeben. Diese Liste wird an die Methode `create_agent` weitergegeben und genutzt, um den Agenten entsprechend der Parameter zu initialisieren. Der Thread lässt sich über die boolean Variable `terminate` beenden. Die Naos werden beim Beenden des Threads in eine Position gebracht, in der sie die Motoren abschalten können, ohne Schaden zu nehmen.

7.9 GUI-Klassen

In Abb. 20 sind die Python-Klassen der GUI gezeigt. Die GUI besteht aus drei Teilen: Einem Python-, einem JavaScript- und einem HTML-Teil. Der HTML-Teil bildet die grobe Struktur der GUI ab. Der JavaScript-Teil reichert den HTML-Code mit Informationen und interaktiven Elementen an und steht in Verbindung mit der `BackEnd`-Instanz des Python-Teils, sodass er Methoden dieser Instanz ausführen kann. Es existiert genau eine `BackEnd`-Instanz zur Programmlaufzeit. Aus Sicht des JavaScript-Teils kapselt das `BackEnd` die Funktionalitäten des Hauptprogramms. Dementsprechend fungiert das `BackEnd` als Adapter zwischen Hauptprogramm und JavaScript-Teil. Das `BackEnd` enthält, wie in Abb. 20 zu sehen ist, eine `app`-Instanz, welche die Verbindung zum HTML- und Javascript Teil ermöglicht. Die Variable `panel_controller_list` enthält alle Instanzen

der Klasse `PanelController`, siehe Abschnitt 7.9.1. Wenn das Lernen über die Methode `startLearning()` vom JavaScript-Teil gestartet wird, bekommt diese einen String übergeben, der sich in eine assoziative Liste, also ein `dict`[7], parsen lässt. Die Liste enthält als Schlüssel einen Variablennamen und als Wert den entsprechenden Variablenwert. Es wurde eine Liste gewählt, anstatt normale Variablen zu verwenden, da die Liste flexibler ist. Verschiedene Algorithmen benötigen unterschiedliche Variablen, daher ist es günstig eine assoziative Liste zu verwenden. Da es möglich sein soll, dass die Agenten auf einer gemeinsamen Wissensbasis arbeiten, beispielsweise einem gemeinsamen Model oder einem gemeinsamen Estimator, gibt es zwei Klassen, die die Vergabe der Objekte verwalten. Der `EstimatoreHandler` ist für die Vergabe des Estimators zuständig. Der `ModelHandler` ist für die Vergabe des Models zuständig.

7.9.1 PanelController

Wenn in der GUI ein Panel erstellt wird, wird im `BackEnd` eine neue `PanelController`-Instanz initialisiert, welche das Panel verwaltet. Die Instanz meldet sich beim `Eventhandler`, siehe Abschnitt 7.11, als Observer an und leitet Informationen an das Panel weiter. Wenn ein Agent *Prioritized Sweeping* verwendet, siehe Abschnitt 7.12.4 und Abschnitt 5.5.1, und im Panel gleichzeitig ein Graph, siehe Abschnitt 8.2, angezeigt werden soll, können Performanceprobleme entstehen, die das Programm zum Absturz bringen können. Es werden zu viele Knoten aktualisiert. Der Versuch diese Menge der Knoten in der kurzen Zeit darzustellen überlastet das `HtmlPy`-Framework. Das Problem wurde durch drei Maßnahmen gelöst:

1. Da das Programm immer abstürzt, wenn die Anzahl der Knoten zu hoch ist, wurde eine Obergrenze für Knoten eingeführt. Wenn diese Obergrenze überschritten wird, werden alle Knoten im Graph gelöscht. Dies wurde mit JavaScript gelöst.

2. Für die *normale* Aktualisierung und die PS-Aktualisierung werden sowohl auf Python-Seite, wie auch auf JavaScript-Seite, unterschiedliche Methoden verwendet. In Python verwendet die `PanelController`-Instanz die Methode `net_multi_update()`, welche den Thread `Multi_update_thread` startet. Der Thread wartet eine festgelegte Zeitspanne, in der er Aktualisierungen *sammelt*, um diese anschließend gesammtelt als Json-String an den JavaScript-Teil schickt. Die JavaScript-Methode `net_multi_update()` aktualisiert den Graph mit dem Json-String.

3. Da nicht alle Knoten, die aktualisiert werden sollen, im Graph vorhanden sein müssen, wird in der JavaScript Methode `net_multi_update()` geprüft, ob die jeweiligen Knoten vorhanden sind. Aktualisierungen von Knoten, die nicht im Graph enthalten sind, werden ignoriert.

7.10 Koordinatenlisten

Der Versuch ist darauf ausgelegt, dass mehrere Agenten gleichzeitig lernen können. Entweder kann ihr Lernfortschritt miteinander verglichen werden, oder sie können gemeinsam auf einer Wissensbasis, bzw. Estimator, lernen um so schneller eine gute Policy zu entwickeln. So ist es möglich, dass vier Naos, also zwei Lerner-Validator-Paare gleichzeitig

[7] Kurzform für Dictionary

betrieben werden können. Jedes Paar benötigt seinen eigenen Versuchsaufbau und obwohl jeder Aufbau von seiner Struktur her gleich ist, gibt es aufgrund von abweichenden Positionierungen andere Koordinaten, welche die Paare benutzen müssen.

Daher besitzt jedes Paar eine eigene CSV-Datei, in der alle relevanten Koordinaten und sonstige Daten, wie beispielsweise Schwellwerte zur Farbenerkennung, enthalten sind. Die Dateien werden in der Klasse `Nao` verwendet und bei der Initialisierung der `Nao`-Instanz geladen. Hierbei sind die Daten zur Laufzeit in einem `dictionary` vorhanden. Die Koordinaten werden aus dem Datentyp mit einem `key`, in diesem Fall ein `String`, vom `dictionary` ausgelesen.

Wenn die Roboter neu verstellt werden, muss die CSV-Datei aktualisiert werden. Das Aktualisieren kann auch über die GUI erfolgen, siehe Abschnitt 8.3.

7.11 Nebenläufigkeit

Agenten sollen, wie in Abschnitt 1.3.2 auf Seite 5 gefordert, gleichzeitig lernen können. Um dies zu ermöglichen werden Threads verwendet. Zur Laufzeit des Programms sind mehrere Threads aktiv, die untereinander kommunizieren müssen. Es ist immer ein GUI-Thread aktiv der die grafische Benutzeroberfläche verwaltet. Zusätzlich ist ein Thread für jedes Lerner-Validator-Paar zuständig. Wenn ein Lerner Prioritized Sweeping betreibt, wird hierfür zusätzlich ein Thread genutzt. Die Threads kommunizieren über einen `EventHandler`, welcher ein *Singleton* ist. Da das Programm stark modular sein soll und die Sicherheit der Daten weitestgehend vernachlässigt werden kann, wurde das klassische Observer Muster umgestaltet. Anstatt ein klassisches *Observable* Interface zu nutzen, kann der `EventHandler` an jeder Stelle des Programms aufgerufen werden. Über den `Eventhandler` können Events ausgelöst werden. Die Events besitzen eine Kennung, beispielsweise einem zusammengesetzten `String` der aus dem Namen des Agenten und einem Eventnamen besteht. Observer können sich beim `EventHandler` auf diese Kennungen mit einer Funktion, die entsprechend ausgelöst wird, anmelden. Die einzelnen Event Namen sind schwer im Code zu finden, was einen Nachteil bei diesem Vorgehen darstellt. Deswegen müssen alle Events zentral dokumentiert werden. Eine alternative Lösungsmöglichkeit wäre es das vorgefertigte `Signal/Slot` Verfahren von QT zu verwenden, was in diesem Programm bisher nur auf der GUI Ebene Anwendung findet. Nachteil beim `Signal/Slot` Verfahren sind die festen Parameterlisten, die definiert werden müssen, wodurch das Programm weniger modular gestaltet werden kann.

7.12 Umsetzung der RL-Algorithmen

In diesem Abschnitt wird die programmiertechnische Umsetzung der eingebundenen Reinforcement Algorithmen vorgestellt. Zunächst wurden alle Algorithmen in die Kategorien unterteilt:

- Explorator Funktionen,
- Estimator Funktionen,
- Temporal Difference Learning Funktionen und
- Planning Funktionen.

Aus diesen Kategorien kann jeweils höchstens eine Funktionalität verwendet werden.

7.12.1 Explorator Funktionen

Diese Funktionen befassen sich mit dem *exploration vs exploitation*-Problem und bestimmen **wann** und **wie** exploriert wird. Es wurden bisher, in dieser Kategorie, zwei Algorithmen implementiert.

Der erste Algorithmus ist ε-greedy (siehe Abschnitt 5.3.1 auf Seite 28). Der Verlauf von `epsilon` ist in [Chr17] beschrieben. `epsilon` besitzt einen Ausgangswert, ein `epsilonImprovementValue`, welches positiv und kleiner als eins ist, und ein `epsilonLimit`, welches positiv und kleiner als `epsilon` ist. Alle drei Episoden wird `epsilon` mit `epsilonImprovementValue` multipliziert und wird somit geringer. Sobald `epsilon` kleiner als `epsilonLimit` ist, verhält sich der Agent so lange gierig, bis seine Episodenlänge im Durchschnitt größer ist, als die geringste durchschnittlich erfasste Episodenlänge im gesamten Lernvorgang. Der Durchschnitt muss immer über mindestens drei Epsisoden gebildet werden, da sich Episoden in diesem Szenario gegenseitig beeinflussen können. Beispielsweise könnte der Agent in einer Episode einen roten Ball mit der rechten Hand einsortieren, wärend er in der linken Hand einen blauen Ball über dem Ziel, bzw. Destination, siehe Abb. 6e auf Seite 13, hält. Sobalt er den roten Ball einsortiert, endet die erste Episode. Wenn er anschießend den blauen Ball einsortiert, beinhaltet die zweite Episode nur eine Aktion. Daher ist es notwendig einen Mittelwert über mehrere Episoden zu bilden. Der Verlauf von `epsilon` im Verhältnis zum Lernverlauf ist auf Abb. 41 auf Seite e zu sehen.

Der zweite Algorithmus ist *Softmax*, der in Abschnitt 5.3.2 auf Seite 28 beschrieben wurde. Hier wurde die Gleichung (10) auf Seite 28 verwendet.

7.12.2 Estimator Funktionen

Schätzer wurden genutzt um Q-Werte abzuspeichern und abzurufen, bzw. zu schätzen. Die Q-Tabelle (siehe Abschnitt 5.4.1 auf Seite 29) wurde als einfachste Variante implementiert. Aus dem Bereich Function Approximation, welcher auf Abschnitt 5.4.2 auf Seite 29 beschrieben ist, wurde ein Schätzer mit einem *Tile Coding*-Ansatz programmiert. In der Umsetzung von *Tile Coding* in dieser Arbeit nicht nur Features aus dem Zustandsraum, sonder auch Features aus dem Aktionsraum verwendet, also letztlich (s,a)-Features und der Estimator Q-Werte schätzt, wird der *Tile Coding*-Estimator im weiteren Verlauf der Arbeit `QFeature` genannt.

Da Tile Coding, wie in Abschnitt 5.4.2 auf Seite 30 beschrieben, eine Variante des Coarse Coding ist, welches wiederum auf kontinuierliche Zustandsräume ausgelegt ist, ist Tile Coding dementsprechend auch auf kontinuierliche Zustandsräume ausgelegt. Da der Zustandsraum, der in dem Versuch genutzt wird, nicht kontinuierlich ist, wird nun begründet, wieso Tile Coding trotzdem verwendet wird.

In kontinuierlichen Zustandsräumen haben Zustände, die nebeneinander liegen, ähnliche Eigenschaften und können dementsprechend teilweise über die gleichen Tiles generalisiert werden. In diesem Versuch ist der Zustandsraum zwar diskret, es gibt aber dennoch viele Zustände die, wenn man diese auf wenige Dimensionen, reduziert gleich sind, d.h., dass es in diskreten Zustandsräumen, genau wie in kontinuierlichen Zustandsräumen, bei zwei Zuständen mit ähnlichen Dimensionen eine gute *Idee* sein kann, dieselbe Aktion auszuführen. Im Grunde besteht der Zweck des Tile Codings darin, gemeinsame Eigenschaften zwischen den Zuständen zu finden, um diese dadurch zu generalisieren. Dies ist auch in diskreten Zustandsräumen möglich und oft auch wünschenswert.

Es wurden drei Tilings festgelegt. Da in diesem Projekt Q-Learning verwendet wird,

Tabelle 4: Konfiguration der Tilings. Die angegebenen Zahlen beziehen sich auf die Dimensionen des Zustandsraumes, siehe Abschnitt 3.1.1

Tilings	1	2	3
Rechter-Arm-Aktionen	(0,1,2,3,4)	(1,3)	(1,3,4)
Linker-Arm-Aktionen	(0,1,2,3,4)	(0,2)	(0,2,4)
Zwei-Arm-Aktionen	(0,1,2,3,4)	(4)	(0,1)

werden die Tilings nicht auf den Zustandsraum –, sondern auf den Zustandsaktionsraum angewendet. Alle Aktionen wurden in drei Kategorien unterteilt: Rechter-Arm-Aktionen, linker-Arm-Aktionen und zwei-Arm-Aktionen, in die alle Aktionen eingeteilt wurden, die nicht genau einen Arm benutzen. Das erste Tiling beinhaltet alle möglichen (s,a)-Paare und ist mit dem normalen Q gleichzusetzen. Zwar geht dadurch ein großer Vorteil von Function Approximation, weniger Speicherplatzverbrauch, verloren, jedoch die größte Gefahr von Function Approximation, zu starke Abstraktion, wird dadurch minimiert. Die anderen beiden Tilings werden jeweils durch die drei Aktionskategorien unterteilt. Je nachdem zu welcher Kategorie die ausgewählte Aktion gehört werden andere Dimensionen in den Zuständen betrachtet.

7.12.3 Temporal Difference Learning Funktionen

Diese Funktionen aktualisieren den Schätzer. Bislang sind drei Funktionen implementiert. Die erste Funktion ist das normale one-step Q-Learning wie es in Abschnitt 5.2.1 beschrieben ist. Dieses Update wird im weiteren Verlauf der Arbeit `QUpdate` genannt. Da sich die Update-Funktionen von tabellarischen und approximativen Schätzern unterscheiden, wurde die Tile Coding Update-Funktion implementiert. Diese Funktion wird im weiteren Verlauf der Arbeit `QFeatureUpdate` genannt. Sie ist jedoch nicht auf der GUI separat zu finden, da es nicht möglich ist einen solchen Approximator ausschließlich mit one-step Q-Learning zu aktualisieren, und es deswegen redundant wäre, dies als Funktion auswählbar zu machen. Die letzte Funktion ist $Q(\lambda)$ für tabellarische Schätzer. Hier gab es wie in Abschnitt 5.2.3 beschrieben drei unterschiedliche Methoden. Da während des Lernens viel exploriert wird, eignet sich Watkins $Q(\lambda)$ nicht, da, wie in Abschnitt 5.2.3, die Traces bei großer Exploration klein bleiben, was zu einer geringen Backup-Rate führt. Pengs $Q(\lambda)$ ist hier zwar besser geeignet, da die Traces auch bei hoher Exploration nicht auf null gesetzt werden, wie in Zeile 14 bereits gezeigt wurde. Der Naive $Q(\lambda)$ Algorithmus ist simpler als Pengs $Q(\lambda)$ und zeigt laut [SB12] in Tests gute Ergebnisse, deswegen wurde der Naive $Q(\lambda)$ Algorithmus implementiert. $Q(\lambda)$ hängt, wie alle Temporal Difference Learning Funktionen vom Schätzer ab. Deswegen funktioniert das implementierte $Q(\lambda)$ nur für tabellarische Schätzer. Im Programmcode wird der Algorithmus für das naive $Q(\lambda)$ durch die Methode `naiv_q_lambda_update` umgesetzt.

7.12.4 Planende Funktionen

Prioritized Sweeping (PS) wurde bisher als einzige planende Funktion hinzugefügt.PS wurde im Programmcode durch die Methode `prioritized_sweeping` realisiert. Durch die Priorisierung bietet PS einen Vorteil gegenüber andere planende Funktionen, wie beispielsweise Dyna [SB12]. Der in Algorithmus 4 gezeigte Algorithmus wurde mit einer Abweichung implementiert: Die Abarbeitung der Priority Queue (PQueue) wurde in einen

eigenständigen Thread ausgelagert. Diese Parallelisierung macht den *Lernthread* schlanker.

7.13 Zusammenfassung

In diesem Abschnitt wurde die Implementierung des Programms betrachtet. Es wurde beschrieben, dass das Programm aus drei Teilen besteht: Dem Haupteil, der in Python geschrieben ist, dem HTML-Teil, der die Grundstruktur der GUI abbildet und den JavaSkript-Teil, der die GUI mit Informationen anreichert. Zustände werden im Programm entweder mit einem Tupel oder mit der `State`-Klasse abgebildet. Die `Nao`-Klasse ist das Bindeglied zwischen Programm und Roboter. Die Klasse lädt die benötigten Koordinaten zur Initialisierung aus einer CSV-Datei. Für Simulationen kann die Klasse `NaoMock` genutzt werden, die das Nutzen des Programms auch ohne Verwendung des Naos möglich macht. Jede Aktion, die der Agent verwenden kann, wurde als Klasse implementiert. Alle Aktionen besitzen die Methode `isUsable`, mit der getestet werden kann, ob die Methode im aktuellen Zustand ausführbar ist. Der Agent wurde im Programm mit gleichnamiger Klasse `Agent` implementiert. Mit der Methode `create_agent` kann eine Instanz dieser Klasse erzeugt werden. Die Klasse `Rewarder` ermittelt den Reward im Iterationsschritt und leitet diesen an den Agenten weiter. Die Aufgabe der Instanz der Klasse `Logger` ist es, die Lernhistorie für eine Versuchsauswertung, zu sichern. Die Instanzen der Klasse `LearnerValidatorPair` sind Threads in denen sich Lerner und Validator befinden. In dem Thread wird die Lernschleife ausgeführt. Hauptbestandteile der GUI-Klassen im Python-Teil sind `BackEnd` und `PanelController`. Die Instanz der Klasse `BackEnd` ist für den Datenaustausch zwischen JavaScript-Teil in Python-Teil zuständig. Eine `PanelController`-Instanz ist für die Darstellung des Lernfortschritts eines Agenten zuständig. Die *Singleton* Klasse `EventHandler` ermöglicht den Datenaustausch zwischen den Threads. Des Weiteren wurde die Umsetzung der RL-Algorithmen beschrieben. Umgesetzt wurden: ε-geedy, Softmax, die Q-Tabelle als Klasse `QTable`, Tile Coding als Klasse `QFeature`, das naive $Q(\lambda)$ durch die Methode `naiv_q_lambda_update` und PS durch die Methode `prioritized_sweeping`.

8 Grafische Benutzeroberfläche

Dieser Abschnitt behandelt die grafische Benutzeroberfläche. Zu Beginn wird die Benutzeroberfläche in Abschnitt 8.1 erläutert. Da die Visualisierung des Lernens entscheidend für das Verstehen von Lernvorgängen und das Erkennen von Fehlern ist, wird diese in Abschnitt 8.2 näher behandelt. Da der Roboter nach langer Nicht-Benutzung des Programms wieder kalibriert werden soll, ist dies über eine besondere Ansicht möglich. Die Bestandteile dieser Ansicht werden in Abschnitt 8.3 erklärt. Im Programm wird der Nao abstrahiert als Grafik dargestellt. Was diese Grafiken im Detail bedeuten und wie sich diese zusammensetzen ist in Abschnitt 8.4 erklärt.

Die grafische Benutzeroberfläche (engl. graphical user interface, GUI) ist der programmiertechnisch aufwändigste Part des Projekts. Zwar hat die GUI technisch nichts mit dem eigentlichen RL zu tun, sie ist aber trotzdem ein wichtiger Bestandteil, wenn es um die Usability des Programms und das Aufbereiten der Daten geht. Die visuelle Aufbereitung soll dem Anwender oder dem Zuschauer bei einer Demonstration des Projekts, verdeutlichen, wie RL funktioniert, und welche Auswirkungen die einzelnen RL-Methoden und Strategien auf das Lernen haben. Eine gute Visualisierung der Daten wichtig, um Schwächen und Fehler im Lernvorgang zu erkennen.

8.1 Erläuterung der GUI

In Abb. 21 ist die grafische Benutzeroberfläche abgebildet. Die Lerner-Validator-Paare können über das Drop-Down-Menü ausgewählt werden, siehe Abb. 21 Nummer 1. Es können die Naos oder Nao-Mocks ausgewählt werden, bei letzterem findet das Lernen in der Simulation statt. Um eine Fehlbedienung des Programms zu vermeiden, können die Roboterpaare jeweils nur einmal ausgewählt werden.

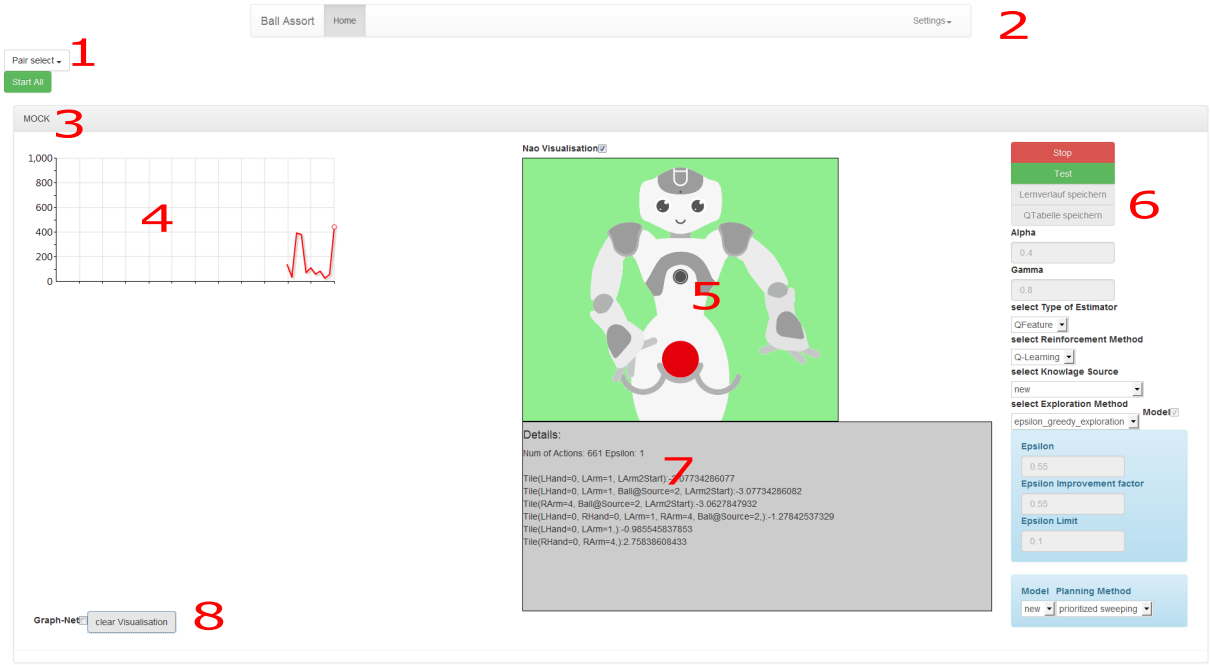


Abbildung 21: Grafische Benutzeroberfläche

Sobald ein Paar selektiert wurde, öffnet sich ein Panel, wie auf Abb. 21 Nummer 3, zu sehen ist. Im Kopf des Panels steht der Name des lernenden Roboters oder Mock, wenn

das Lernen simuliert wird. Der Name des lernenden Roboters ist zur Programmlaufzeit eindeutig und wird deshalb zur Identifikation von Objekten und Daten genutzt, die einem Lernvorgang zugewiesen sind. Das Panel ist in zwei Teile unterteilt:

Auf der rechten Seite wird der Lernfortschritt grafisch aufbereitet und auf der linken Seite können Einstellunmöglichkeiten getroffen werden.

Optionen Die Optionen befinden sich auf Abb. 21 bei Nummer 6 und sind so aufgebaut, dass nur relevante Optionen gezeigt werden. Wenn beispielsweise eine `epsilon greedy exploration` ausgewählt wurde, werden Einstellungsmöglichkeiten für Epsilon etc. angezeigt, welche wieder verschwinden, wenn eine andere Explorationsstrategie gewählt wird. Des Weiteren werden eingegebene Zahlenwerte überprüft, sollte ein Wert nicht der Norm entsprechen wird dieser nicht vom Programm akzeptiert. Um unbekannte Parameter zu erläutern, erscheinen Tooltips, wenn der Mauszeiger über ein Feld fährt. Das Lernen kann entweder über den START-Button oder über den START ALL-Button, der sich unter dem Drop-Down-Menü PAIR-SELECT befindet, ausgelöst werden. Wenn der Agent lernt sind die Einstellungsmöglichkeiten für die Lern-Parameter gesperrt, da es nicht möglich sein soll, Parameter wie die QTabelle während des Lernens manuell zu ändern. Während des Lernvorgangs ändert sich der Start-Button in einen Stopp-Button, der dazu dient, das Lernen zu terminieren. Nach dem Lernvorgang ist es möglich über die entsprechenden Button den Lernverlauf sowie die QTabelle zu speichern. Die QTabelle kann anschließend über das Drop-Down-Menü SELECT KNOWLAGE SOURCE geladen werden. Wenn verschiedene Agenten die gleiche QTabelle geladen haben arbeiten sie beide auf eben dieser Instanz und nicht auf Kopien. Dies ermöglicht das Vervollständigen einer Tabelle mit mehreren Robotern.

8.2 Visualisierung des Lernvorgangs

Das Lernen wird auf vier verschiedene Weisen visualisiert:

Kurve Da der Agent sein Ziel in möglichst wenigen Schritten erreichen soll, wurde der Graph des Lernverlaufs, wie er auf Abb. 22 auf der nächsten Seite zu sehen ist, hinzugefügt. An ihm kann überprüft werden, ob sich der Agent über die Zeit verbessert, also immer weniger Iterationen benötigt, um sein Ziel zu erreichen. Auf Abb. 21 bei Nummer 4 befindet sich die Abbildung des Lernverlaufs in Form eines Diagramms.

Animation des Nao Um das Verhalten des Agenten auch bei Simulationen leichter zu erkennen, wird der Zustand des Agenten in Form einer Grafik, wie auf Abb. 23 auf der nächsten Seite, dargestellt. Wenn sich der Zustand ändert, wird auch die Grafik aktualisiert, sodass die Bewegungsabläufe erkennbar sind. Auch für den realen Test am Roboter ist diese Animation hilfreich, da sie die wahrgenommene Umwelt des Nao zeigt, welche sich von der Realität unterscheiden kann. Auf Abb. 21 befindet sich die Animation des Nao bei Nummer 5.

Detailansicht des Lernschritts Die Detailansicht des Lernschritts ermöglicht es alle relevanten Informationen bezüglich des vorangegangenen Lernschritts zu zeigen. Auf Abb. 24 auf Seite 54 ist eine Detailansicht abgebildet, welche die Aktionsanzahl der Episode, Epsilon und die aktualisierten Tiles anzeigt. Die Ansicht kann sich entsprechend der

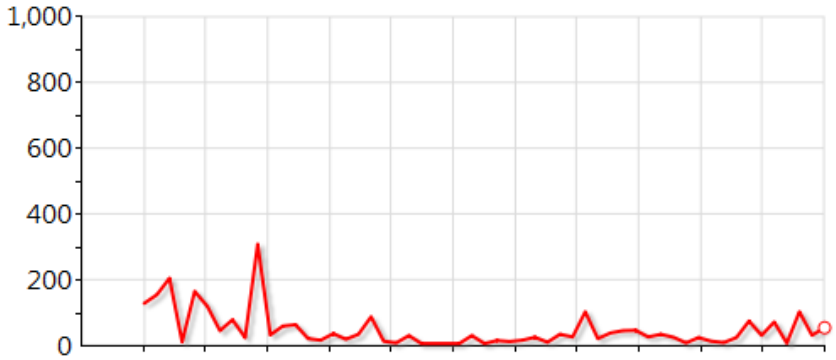


Abbildung 22: Abbildung eines Lernverlaufs in Form eines Diagramms, bei dem sich auf der X-Achse die Episoden – und auf der Y-Achse die Anzahl der Aktionen pro Episode befinden

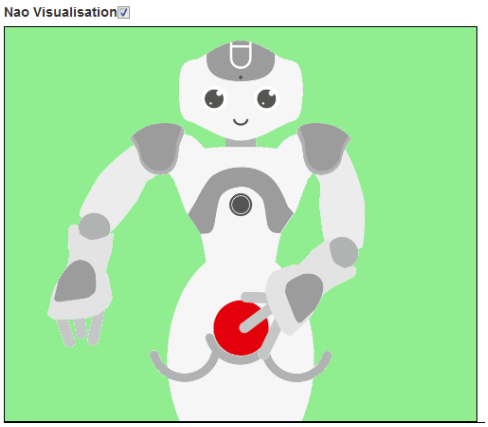


Abbildung 23: Animation des Nao

Details:

Num of Actions: 2Epsilon: 0.3025

Tile(RHand=0, RArm=4, RHandClose):-3.43647288776
Tile(RHand=0, RArm=4, Ball@Source=3, RHandClose):1.77359440653
Tile(LHand=3, RHand=0, LArm=0, RArm=4, Ball@Source=3, RHandClose):-2.39957951893

Abbildung 24: Beispielhafte Übersicht des Lernschritts, es wird die Anzahl der vergangenen Episoden, Epsilon und die aktualisierten Tiles (siehe Tilecoding) angezeigt

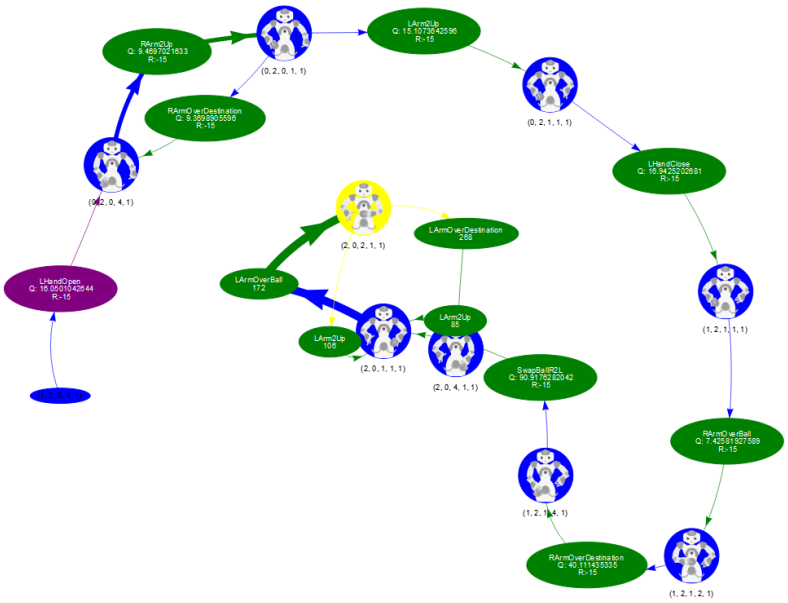


Abbildung 25: Graph als Darstellungsvariante des Lernens

vorgegebenen Parameter unterscheiden. Auf Abb. 21 befindet sich die Detailansicht des Lernschritts bei Nummer 7.

Graph Um das Verhalten des Agenten über mehrere Episoden zu betrachten, kann ein bipartiter gerichteter Graph eingeblendet werden. Ein beispielhafter Graph ist auf Abb. 25 zu sehen. Da das Szenario, siehe Abschnitt 3 auf Seite 11 keine deterministische Umgebung besitzt, werden sowohl die Zustände, als auch die Aktionen als Knoten dargestellt, somit können auch von einer Aktion mehrere Kanten zu Zuständen führen. In dem Graph sind die Zustände blau und die Aktionen grün markiert. Ausnahmen bilden hierbei der aktuelle Zustand, welcher gelb markiert ist – und Aktionen, die durch *Prioritized Sweeping* aktualisiert wurden, welche lila gekennzeichnet sind. Die Aktionsknoten haben immer genau eine Kante, die von einem Zustand zu ihnen führt. Dementsprechend sind die Aktionsknoten eher Zustandsaktionsknoten und können daher auch den Reward R und die Qualität Q für das Zustandsaktionspaar anzeigen.

Die Dicke der Pfeile zeigt an, wie oft der Agent einen Pfad beschritten hat, so können Dauerschleifen und optimale Handlungsstrategien erkannt werden.

In Abb. 21 auf Seite 51 bei Nummer 8 sind eine Checkbox und ein Button mit der Aufschrift CLEAR VISUALISATION zu sehen. Mit der Checkbox lässt sich das Netz ein-

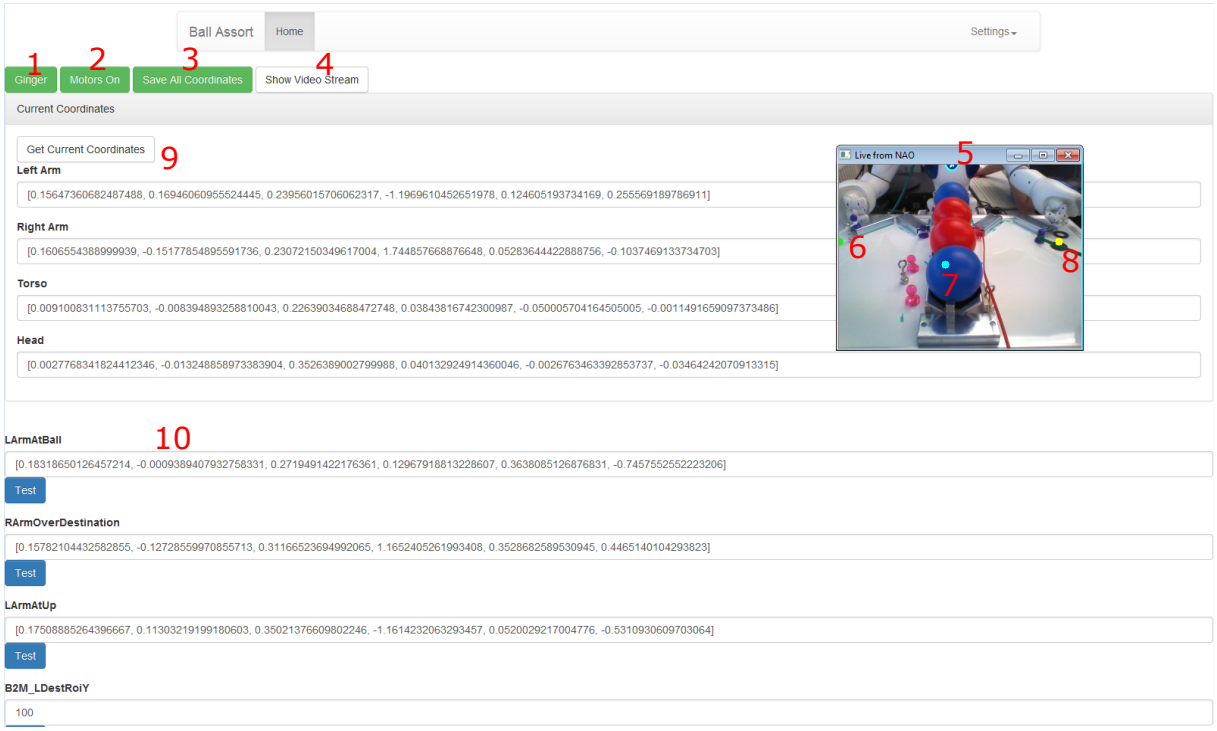


Abbildung 26: Ansicht zur Konfiguration des Roboters

und ausblenden. Das Drücken des Buttons bewirkt das Zurücksetzen des Graphen. Dies wird benötigt, wenn zu viele Objekte angezeigt werden, sodass sich die Performance des Computers verschlechtert. Des Weiteren werden aus diesem Grund auch alle Objekte gelöscht, sobald zu viele Objekte angezeigt werden.

8.3 Konfiguration des Roboters

In der Navigationsleiste befindet sich unter dem Menü-Punkt Settings der Unterpunkt Edit Coordinates, welche eine Ansicht öffnet, die es ermöglicht die Koordinaten für den Nao permanent zu ändern.

Im Folgenden werden die Elemente der Ansicht anhand von Abb. 26 besprochen:

1. Dieser Button zeigt den Namen des aktuellen Naos an. Wenn der Button gedrückt wird, wird der nächste aktive Nao ausgewählt. Jedes Lerner-Validator-Paar besitzt seinen eigenen Koordinatensatz. Dementsprechend werden auch andere Koordinaten angezeigt, wenn ein Nao eines anderes Paares ausgewählt wird.

2. Um die Koordinaten präzise zu erfassen oder Bewegungen durchführen zu lassen, müssen die Motoren angeschaltet werden, welches durch das Drücken auf diesen geschieht. Durch erneutes Drücken lassen sich die Motoren wieder abschalten.

3. Durch Drücken dieses Buttons, lassen sich alle Änderungen speichern.

4. Dieser Button öffnet ein Fenster, welches den Videostream der unteren Kamera des Nao anzeigt.

5. Dieses Fenster zeigt den Videostream der unteren Kamera des Nao an. Das Fenster lässt sich mit drücken der ESC-Taste schließen.

6. Dieser Punkt zeigt die *Region of Interest* (ROI) der rechten Destination an. Diese ROI wird vom Validator genutzt um zu ermitteln, welche Farbe der Ball hat, der in der rechten Destination[8]liegt. Der Punkt markiert hierbei die obere linke Ecke der quadratischen Region, die eine Seitenlänge von 30 Pixeln besitzt.

7. Dieser Punkt zeigt die ROI der Source. Der Lerner nutzt diese ROI um die Farbe des Balls, welcher sich in der Source befindet zu ermitteln.

8. Dieser Punkt zeigt die ROI der linken Destination an. Diese ROI wird vom Validator genutzt um zu ermitteln, welche Farbe der Ball hat, der in der linken Destination liegt.

9. Durch diesen Button werden die Koordinaten des linken Armes, rechten Armes, Torsos und des Kopfes in den darunterliegenden Textfeldern angezeigt.

10. Diese Textfelder zeigen die aktuellen Koordinaten an, der darunter liegende *Test*-Button lässt die Koordinate mit dem entsprechenden Körperteil anfahren. Die Koordinaten können überschrieben und mit drücken des Save All Coordinates-Button gespeichert werden.

8.4 Abstrakte Darstellung des Nao

Wie in Abb. 23 und Abb. 25 wurde der Nao für die Visualisierung abstrahiert dargestellt. Andere Optionen, wie nur den Zustand in kodierter From, beispielsweise `(1,2,1,2,1)` oder abstrakte Symbole zu verwenden, würden das Projekt für Zuschauer unzugänglicher gestalten. Die verwendeten Grafiken lassen sich leicht auf den realen Roboter übertragen, wie auf Abb. 27 zu sehen ist.

Da der Zustandsraum, ohne Abzug von unzulässigen Zuständen, 2500 Zustände umfasst, wurde nicht jedes Bild händisch gestaltet. Ein Bild lässt sich aus vier Komponenten zusammensetzen, wie auf Abb. 28 zu sehen ist. Jede Komponente ist ein PNG-Bild. Alle Bilder sind gleichgroß, sodass jede Komponente am richtigen Platz ist, wenn die Bilder übereinander gelegt werden. Wenn ein Ball von einer Hand gegriffen wurde, ist die Position des gegriffenen Balles abhängig von der Position des Armes. Da es aufwändiger ist, Einzelbilder vom gegriffenen Ball und dem Arm zusammenzusetzen, wurden alle Arm-Ball-Positionen händisch erstellt. Es gibt nur Bilder für den rechten Arm, um den linken Arm darzustellen, wird ein Bild des rechten Arms entsprechend gespiegelt. Alle möglichen und unmöglichen Zustände wurden einmal erstellt, damit diese nicht zur Programmlaufzeit generiert werden müssen.

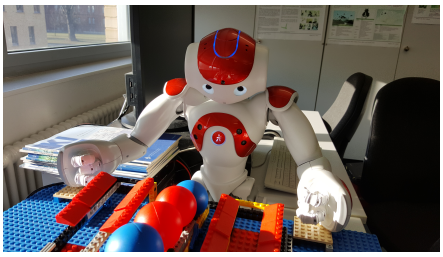
8.5 Zusammenfassung

In diesem Abschnitt wurde die GUI im Detail betrachtet. Die GUI ermöglicht es, den Lernfortschritt während des Lernvorgangs zu beobachten. Hierfür werden vier Arten der Visualisierung zur Verfügung gestellt:

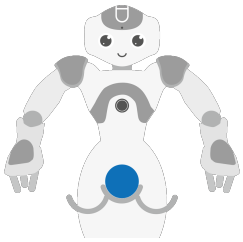
- Der Lernverlauf wird als Diagramm angezeigt, in welchem die Y-Achse die Anzahl der Aktionen pro Episode ist,

- das Verhalten des Nao wird grafisch dargestellt,

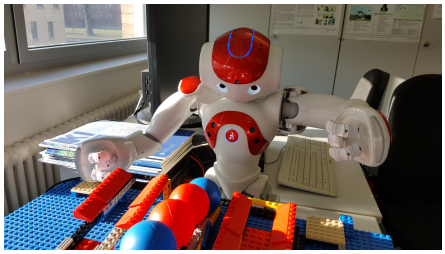
[8]Aus der Sicht des Validators



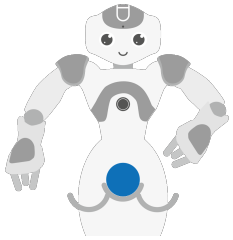
(a) Linker Arm an der Startposition, Realität



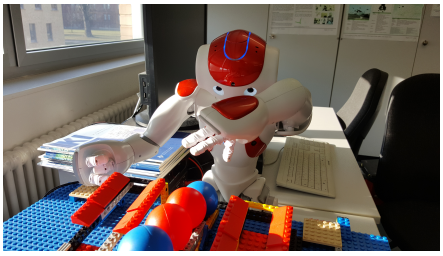
(b) Linker Arm an der Startposition (1,1,0,0,3), Grafik



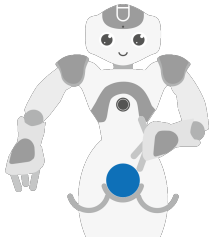
(c) Linker Arm an der „Up“-Position, Realität



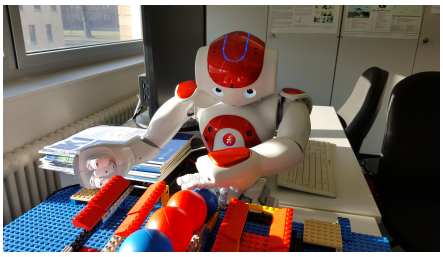
(d) Linker Arm an der „Up“-Position (1,1,1,0,3), Grafik



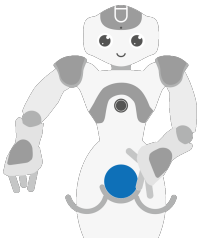
(e) Linker Arm an der „Over Ball“-Position, Realität



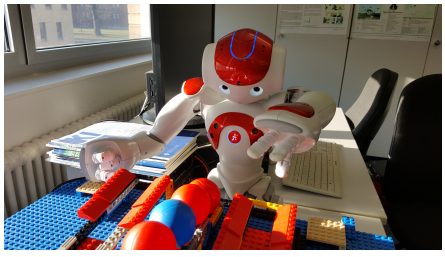
(f) Linker Arm an der „Over Ball“-Position (0,1,2,0,3), Grafik



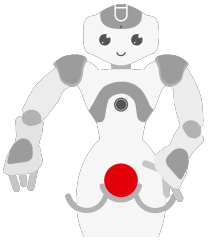
(g) Linker Arm am Ball, Realität



(h) Linker Arm am Ball (0,1,3,0,3), Grafik



(i) Linker Arm an der „Over Destination“-Position, Realität



(j) Linker Arm an der „Over Destination“-Position (0,1,4,0,2), Grafik

Abbildung 27: Grafische Visualisierung der Zustände

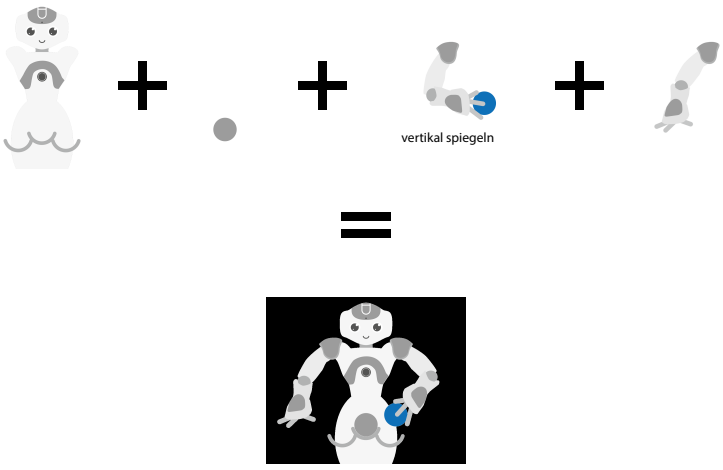


Abbildung 28: Bestandteile der Zustandsgrafik

- eine Detailansicht zeigt Einzelheiten zu dem aktuellen Lernschritt an,
- ein gerichteter bipartiter Graph zeigt die letzten Aktionen und die letzten Zustände, in denen der Agent gewesen ist, an.

Des Weiteren wurde beschrieben, wie die Koordinaten für die Naos, in der GUI angepasst werden können. Abschließend wurde gezeigt, dass die grafische Darstellung des Naos aus vier zusammengesetzten Teilen besteht.

9 Versuche

Das implementierte Programm wird im Folgenden für Versuche genutzt. Es soll ermittelt werden, welche RL-Algorithmen das Problem am schnellsten lösen und welche Parameter sich hierfür am besten eignen. Versuche aus [Chr17] zeigten, dass der Agent die Aufgabe mit sieben Aktionen pro Episode lösen kann. Bis ein Agent jedoch die Policy ermittelte, um das Problem so lösen zu können, benötigte er ca. 33.000 Aktionen. In dieser Arbeit soll der Agent schneller eine gute Policy finden.

Es werden mehrere Versuchsreihen durchgeführt. Eine Versuchsreihe enthält mehrere gleichzeitig durchgeführte Lernvorgänge, mit teilweise unterschiedlichen Parametern. In dieser Arbeit werden die Versuche simuliert, da die Roboter bereits nach wenigen Minuten überhitzen. Alle Versuchsreihen laufen eine Stunde lang. Jede Aktion eines Agenten dauert mindestens eine Sekunde, wobei sich die Dauer verlängern kann, wenn der Algorithmus in jedem Aktionsschritt eine hohe Rechenleistung benötigt. Da simuliert wird, hätte die Aktionsdauer auch auf null gesetzt und eine Anzahl von Aktionen festgelegt werden können. Dadurch hätten die Experimente schneller getestet werden können. Die Nachteile hierbei wären:

1. Während des Lernvorgangs hätten die Visualisierungen nicht genutzt werden können.

2. Algorithmen, die viel Rechenzeit benötigen, hätten genau so vielen Aktionen ausführen können, wie andere Algorithmen. Dies könnte zu einer Verfälschung des Ergebnisses führen können.

9.1 Erste Versuchsreiche

9.1.1 Durchführung

Bei der ersten Versuchsreihe wurden folgende Werte nicht verändert: $\alpha := 0.7$, $\gamma := 0.7$, $\varepsilon := 0.55$, `Epsilonfactor` $:= 0.55$, `EpsilonLimit` $:= 0.1$. Die Estimator Funktionen, Temporal Difference Learning Funktionen und planende Funktionen wurden wie in Tabelle 5 geändert.

Tabelle 5: Verwendete Methoden in der ersten Versuchsreihe

Versuche	1	2	3	4	5
Esimator	Q-Tabelle	Q-Tabelle	Q-Tabelle	Q-Feature	Q-Feature
TD-Learning Funktion	Q-Update	Q-Lambda	Q-Update	Q-Feature-Update	Q-Feature-Update
Planende Funktion	–	–	PS	–	PS

9.1.2 Ergebnis

Die Lernverläufe der ersten Versuchsreihe sind auf Abb. 29 abgebildet. Da die Lernverläufe sehr volatil sind, wurden sie mit einem gleitenden Mittelwert über 10 Episoden geglättet. Das Diagramm zeigt, dass die verschiedenen Agenten eine unterschiedliche Anzahl von Episoden während der Versuchsdauer absolvierten. Dies hängt mit der Anzahl der Aktionen per Episode zusammen. Alle Agenten konnten in der Zeitspanne *ungefähr*, mit einer Abweichung von ca. 200 Aktionen, die gleiche Anzahl von ca. 3.500 Aktionen ausführen.

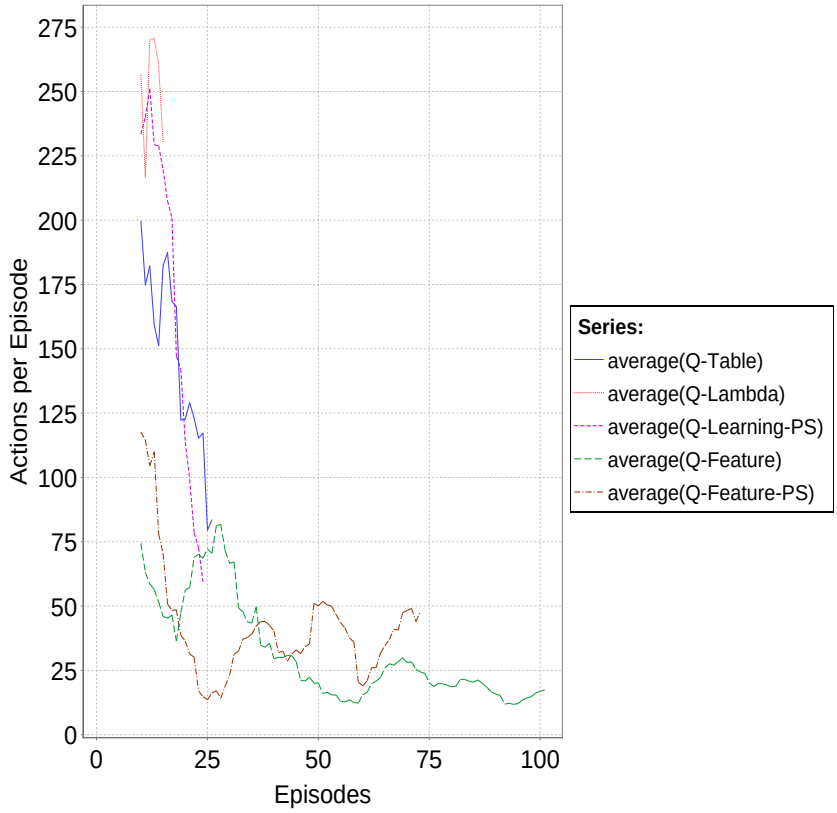


Abbildung 29: Lernverläufe der Algorithmen in der ersten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet

Bei einigen Agenten benötigten die Episoden eine höhere Anzahl von Aktionen, um beendet werden zu können.

Aufgrund der vergleichsweise vielen Aktionen pro Episode und keiner sichtbaren Leistungssteigerung bezüglich des Erfüllens der Aufgabe, wie auf Abb. 43 auf Seite g zu sehen ist, schnitt der Q-Lambda-Algorithmus am schlechtesten ab. Dies kann an der Implementierung des naiven-Q-Lambda liegen. Aus verschiedenen Internetquellen, beispielweise [Bar06], geht hervor, dass in [MS97], die nähere Implementation beschrieben ist. In [Bar06] ist beschrieben, dass immer der maximale Q-Wert in des aktuellen Zustands für die Backups genutzt wird. In [SB12] wird dies hingegen nicht erwähnt.

Von den Agenten, die eine Q-Tabelle nutzten, schnitt der Agent mit Priositised Sweeping in dieser Versuchsreihe am besten ab, was Episodenlänge betrifft. Die Episoden dauerten zu Beginn länger als bei dem *normalen* Q-Learning, dafür fiel die Lernkurve im weiteren Verlauf schneller. Alle Algorithmen mit Q-Tabelle konnten die Aufgabe nicht zuverlässig lösen. Die Anzahl von Erfolg und Misserfolg ist ähnlich hoch, siehe Anhang C auf Seite d.

Die Agenten die Q-Feature, also Function Approximation, statt der Q-Tabelle nutzen, benötigten wesentlich weniger Aktionen pro Episode. Sogar zu Beginn des Lernens war dies der Fall. Der Grund hierfür könnte darin liegen, dass beim Q-Feature-Learning ein *Kontext* hinzugegeben wurde. Vordefinierte Gemeinsamkeiten zwischen den einzelnen Zuständen werden berücksichtigt, sodass der Agent, um den Return zu maximieren, *gezwungen* wird Zustände mit anderen Eigenschaften zu finden, da er erlebt hat, dass ihm Zustände mit immer gleichen Eigenschaften keinen guten Reward bringen. Q-Feature-PS fällt zu Beginn stark, jedoch verbessert sich danach nicht mehr merklich. Grund hierfür könnte $\alpha = 0,7$.

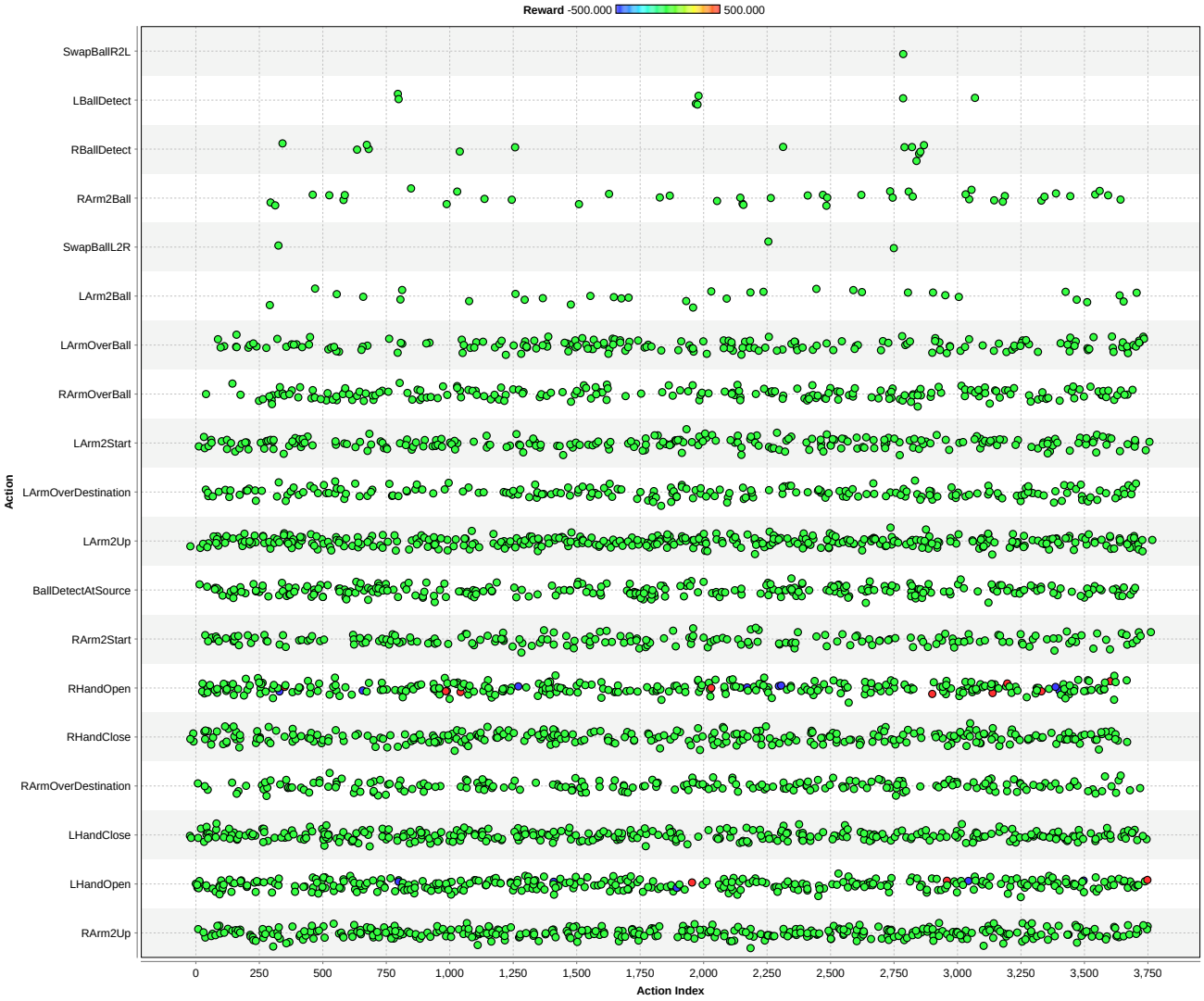


Abbildung 30: Versuchreihe 1, Q-Table, Aktionen über den gesamten Lernverlauf

Der Parameter ist wahrscheinlich zu hoch, was zur Folge hat, dass neues Wissen zu stark gewichtet wird und dementsprechend altes Wissen zu schwach gewichtet ist. Dadurch wird anfangs schneller gelernt, das Gelernte wird jedoch ebenso schnell wieder vergessen.

Im Folgenden werden Abb. 30, Abb. 32 und Abb. 31 näher betrachtet. Diese Streudiagramme zeigen das Auftreten der einzelnen Aktionen über die Zeit hinweg. Grüne Symbole, geben an, dass die Aktion nur einen *normalen* Schrittreward erhielt. Blaue Punkte zeigen einen besonders schlechten und rote Punkte einen besonders guten Reward an. Rote und blaue Punkte können nur bei den Aktionen `LHandOpen` und `RHandOpen` vorkommen.

Bei Abb. 30 ist zu sehen, dass alle Aktionen, die einen Ball involvieren sehr selten sind. Die Aktionen `LArm2Ball` und `RArm2Ball` **müssen** durchgeführt werden, um die Aufgabe zu lösen. Jedoch ist zu sehen, dass die beiden Aktionen zu Beginn sehr selten sind und die Häufigkeit von `RArm2Ball` mit der Zeit geringfügig zunimmt. Der Grund, dass diese Aktionen selten ausgeführt werden, ist auf Tabelle 3 zu finden. Um die Aktion `LArm2Ball` auszuführen, muss die linke Hand offen sein, der linke Arm muss sich auf der *Over Ball*-Position befinden. Gleichzeitig darf der rechte Arm nicht im Weg sein.

D.h., er darf sich nicht auf den Positionen 2 und 3, sondern muss sich stattdessen auf den Positionen 0,1 oder 4 befinden. Da dies vergleichsweise selten vorkommt, besteht nur selten die Möglichkeit, die Aktion auszuführen. Und selbst wenn die Aktion ausgeführt wird, muss der Ball anschließend gegriffen und hochgenommen werden. Bei der Aktion `RArm2Ball` verhält sich dies ähnlich.

Bei Abb. 31 ist zu erkennen, dass der Agent eine Präferenz für die linke Hand und den linken Arm entwickelte. Dies lässt sich daran erkennen, dass der Agent die Aktion `LArm2Ball` wesentlich öfter ausführt, als die Aktion `RArm2Ball`. Des Weiteren wurden die Aktionen `LBallDetect` und `SwapballL2R` im Vergleich zu Abb. 30 und Abb. 32 sehr häufig genutzt. Dies weist darauf hin, dass der Agent oft mit der linken Hand einen unbekannten Ball aufgriff, danach mit `LBallDetect` ermittelte welche Farbe der Ball hat und diesen, wenn es ein roter Ball war, mit `SwapballL2R` in die rechte Hand überführte, um ihn auf die rechte Seite einzusortieren. Wieso der Agent die linke Hand präferiert kann nicht eindeutig gesagt werden. Wahrscheinlich entwickelte sich nach ca. 900 Aktionen der Trend diesen Weg öfter zu gehen und breitete sich dementsprechend aus. Die hohe Lernrate α, der hohe Diskontierungsfaktor γ und Prioritised Sweeping verstärken dies.

In Abb. 32 sind die Aktionen, im Vergleich zu Abb. 30, mehr auf die Aktionen mit Bällen verteilt. Die Häufigkeit dieser Aktionen nimmt im Lernverlauf zu. Im Gegensatz zu Abb. 31 sind keine stark ausgeprägten Präferenzen zu erkennen. Jedoch wird `LArm2Ball` zum Ende des Versuches öfter genutzt. Ein anschließender Versuch, bei welchem das ausgefüllte `QFeature` in Verbingung mit dem Nao und den gleichen Parametern aus der ersten Versuchsreihe genutzt wurde, zeigt, dass der Agent, wenn er nicht exploriert, häufig ein ähnliches vorgehen, wie der Agent mit `QFeature` und PS, aufweist. Das Streudiagramm zu diesem Versuch ist auf Abb. 47 auf Seite k zu sehen. Darauf hin wurde noch ein Versuch mit der gleichen Konfiguration gestartet, mit der Ausnahme, dass diesmal wie in der ersten Versuchsreihe exploriert wurde. Das dazugehörige Streudiagramm ist auf Abb. 48 auf Seite l zu sehen. Durch die Exploration griff der Agent die roten Bälle manchmal mit der rechten Hand und sortierte sie richtig ein. Auffällig ist, dass die Episode länger dauert, wenn ein roter Ball einsortiert werden soll. Dies kann unter anderem auf die vermehrte Nutzung der linken Hand zurückgeführt werden. Da es der Agent geschafft hat, mit Exploration, den roten Ball mit der rechten Hand aufzuheben, ist es wahrscheinlich, dass er nach weiterem lernen den kürzesten Pfad, um den roten Ball einzusortieren, finden wird.

9.2 Zweite und dritte Versuchsreihe

9.2.1 Durchführung

In den beiden folgenden Versuchreihen wurde die Lernrate α verändert. Alle anderen Algorithmen und Parameter blieben gleich. In der zweiten Versuchsreihe wurde α auf 0.1 und in der Dritten auf 0.01 gesetzt. Es soll hierdurch bessere Resultate bei PS-Algorithmen erzielt werden.

9.2.2 Ergebnis

Abb. 34 zeigt den Lernverlauf der zweiten Versuchsreihe. Abb. 35 zeigt den Lernverlauf der dritten Versuchsreihe. Alle Lernverläufe der beiden Versuchsreihen hatten einen längere durchschnittliche Episodendauer im Gegensatz zur ersten Versuchsreihe. Im Diagramm auf Abb. 33 werden die Q-Feature-Lernverläufe der Versuchreihen miteinander verglichen.

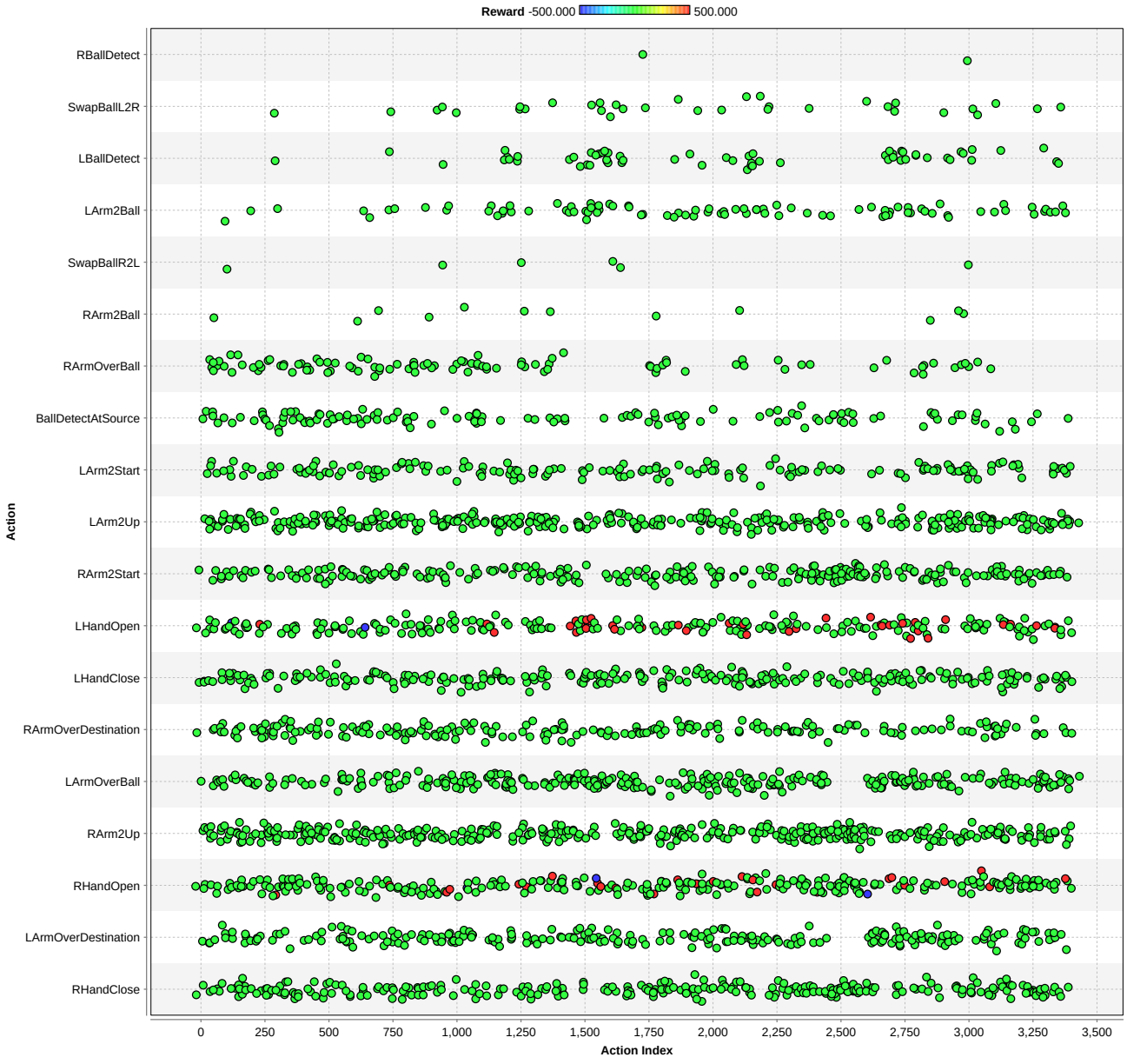


Abbildung 31: Versuchreihe 1, Q-Feature mit PS, Aktionen über den gesamten Lernverlauf

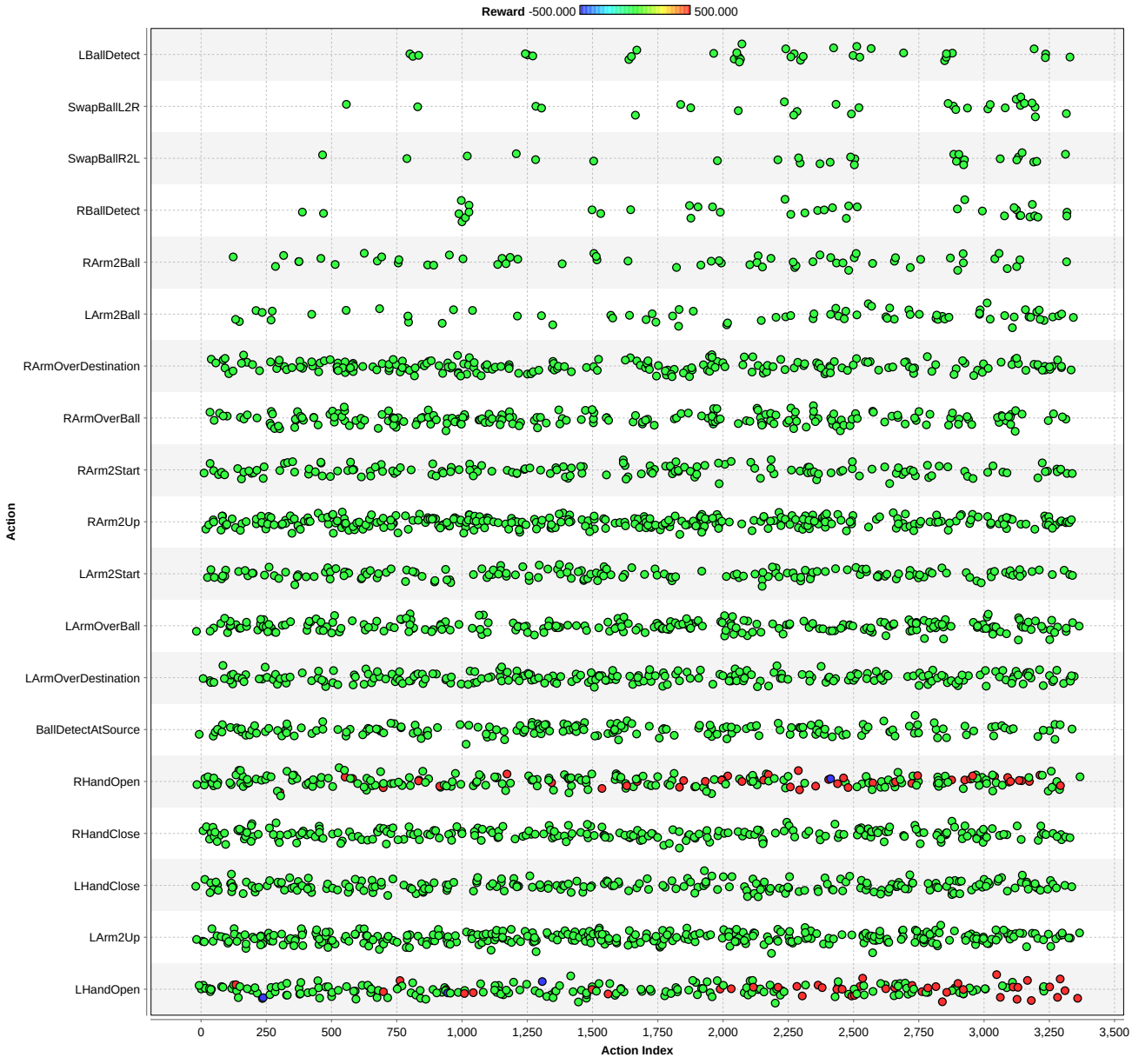


Abbildung 32: Versuchreihe 1, Q-Feature, Aktionen über den gesamten Lernverlauf

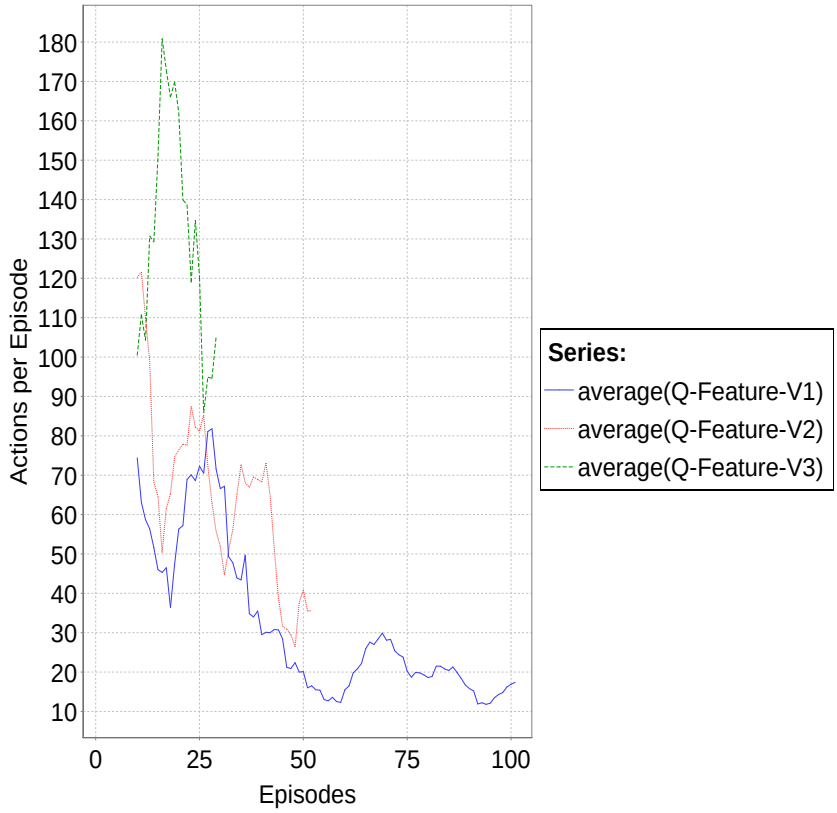


Abbildung 33: Vergleich der Lernverläufe von Q-Feature mit PS über die Versuchsreihen eins bis drei

Es ist zu sehen, das sich das Lernen mit abfallen von α verschlechtert. Q-Feature wurde hier als Beispiel genommen, die anderen Lernverläufe verhalten sich ähnlich.

Bei Q-Feature und Q-Feature-PS lässt sich dennoch ein zuverlässiges Erfüllen der Aufgabe, mit wenigen Fehlern feststellen. Dies ist in Anhang D auf Seite m und in Anhang E auf Seite r zu erkennen.

9.3 Vierte Versuchsreihe

9.3.1 Durchführung

Da die zweite und dritte Versuchsreihe eine Verschlechterung der Performance aufzeigten, werden α und γ auf 0,9 gesetzt. Die hohen Werte sollen das Lernen beschleunigen. Da die simulierte Umgebung, wie in Abschnitt 3.1.4 auf Seite 16 bereits angesprochen wurde, *weitgehend* deterministisch ist, kann diese Wahl der Parameter zu einem guten Ergebnis führen. In einer sehr stochastischen Welt, in der die Ausführung eines (s,a)-Paars stark variierende Rewards ergibt, wäre die Wahl einer solchen Lernrate und Diskontierungsfaktors kontraproduktiv, da das erlangte Wissen zu stark vernachlässigt werden würde, sodass die Q-Werte stark volatil wären.

9.3.2 Ergebnis

Abb. 59 auf Seite A und Abb. 36 zeigen die Lernverläufe der vierten Versuchsreihe an, wobei das letztere Diagramm geglätteter Kurven enthält. In dieser Versuchsreihe wurden die Algorithmen Q-Lambda und Q-Feature mit PS nicht genutzt. Q-Lambda wies in den

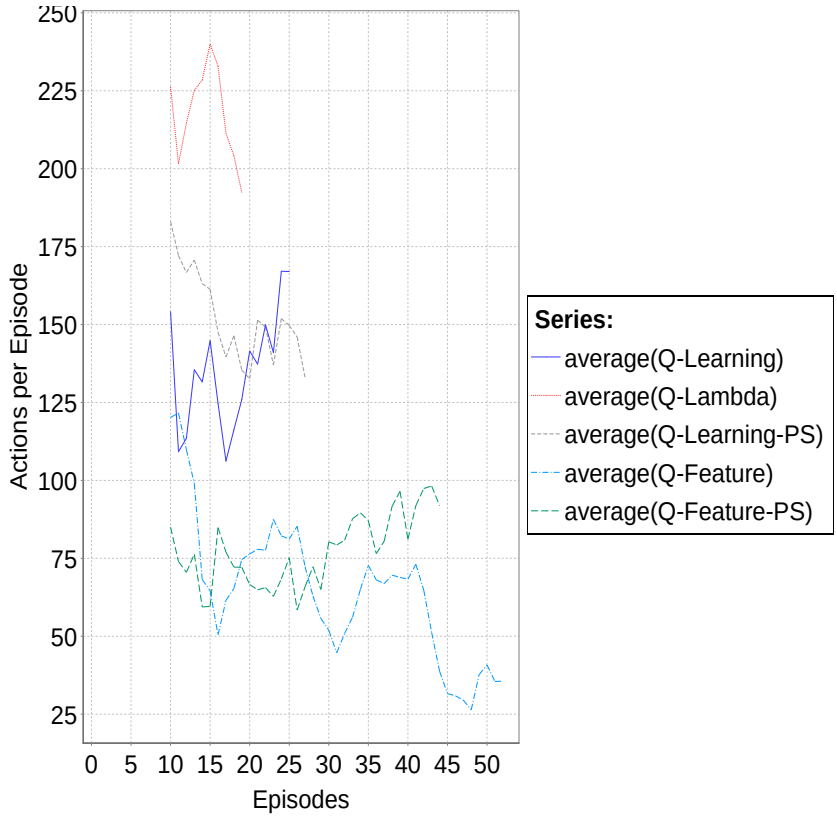


Abbildung 34: Lernverläufe der Algorithmen in der zweiten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet

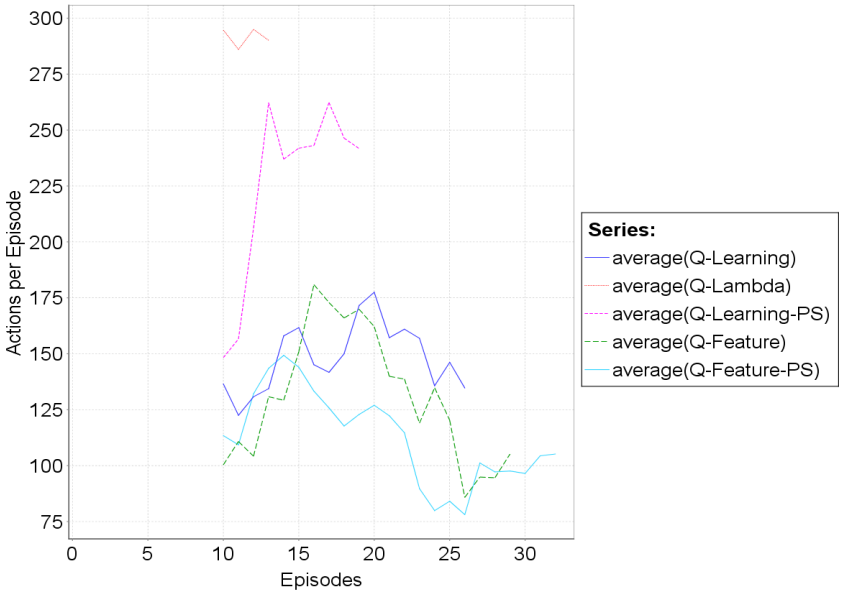


Abbildung 35: Lernverläufe der Algorithmen in der dritten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet

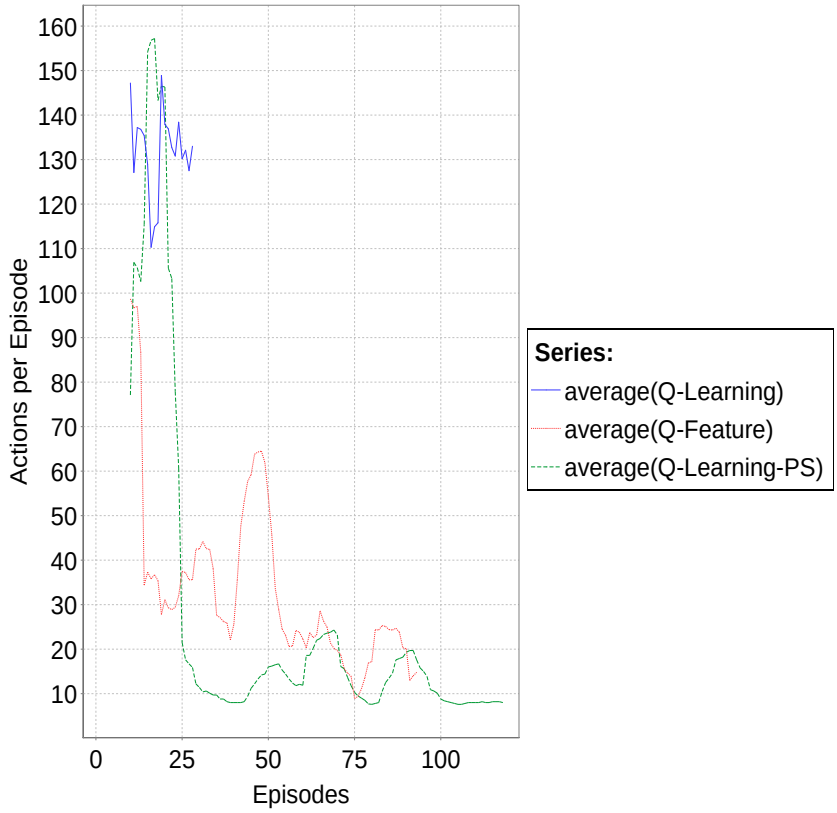


Abbildung 36: Lernverläufe der vierten Versuchsreihe. Die Lernverläufe sind mit einem gleitenden Mittelwert über 10 Episoden geglättet

vorangegangenen Versuchen eine zu schlechte Performance auf und sollte in zukünftigen Arbeiten angepasst werden. Das Programm ist bei Q-Feature mit PS immer nach ca. 40 Minuten abgestürzt[9]. Wahrscheinlich lag dies an der Menge der Updates, die durchgeführt werden mussten. Dieses Problem kann erst zu einem späteren Zeitpunkt behoben werden.

In Abb. 59 ist zu sehen, dass Q-Learning mit PS gute Ergebnisse erzielt. Der Algorithmus findet schneller eine gute Policy als der Q-Feature-Versuch der ersten Versuchreihe, der zuvor die besten Ergebnisse erzielte. Dies wird in Abb. 37 verdeutlicht. Im Streudiagramm auf Abb. 38 ist zu sehen, dass der Algorithmus um letzten Drittel des Versuchs, dies geschah nach ca 40 Min., eine gute Policy entdeckte. Eine starke Zunahme von hohen Rewards ist sichtbar. Die Aktionen `LArm2Ball` und `RArm2Ball` sind gut ausbalanciert. Die Aktionen `SwapBallR2L` und `SwapBallL2R` wurden kaum vom Agenten genutzt. Eventuell hätte sich vorerst eine schlechtere Policy wie bei Abb. 32 ergeben, wenn er öfter von diesen Aktionen Gebrauch gemacht hätte.

Es stellt sich die Frage wieso das Lernen so *abrupt* konvergiert. Die Antwort liegt in der gewählten Explorationsstrategie. Der Parameter für die Exploration `epsilon` (ε) fällt nicht konstant. Er fällt nach vier Episoden um den Faktor `epsilonImprovementValue`, bis der Wert kleiner als der Parameter `epsilonLimit` ist. Wenn dies eintritt, wird nicht mehr exploriert, `epsilon` wird als null betrachtet. Nach weiteren vier Episoden, wird der Durchschnitt der Länge der letzten Episoden gebildet und mit dem, in diesem Versuch gemessenen, Minimum der durchschnittlichen Episodenlängen verglichen. Solange die Epi-

[9]Zur Laufzeit ließ sich dennoch keine gute Performance erkennen. Hierzu gibt es jedoch keine Daten oder Aufzeichnungen und wird deswegen nicht weiter berücksichtigt.

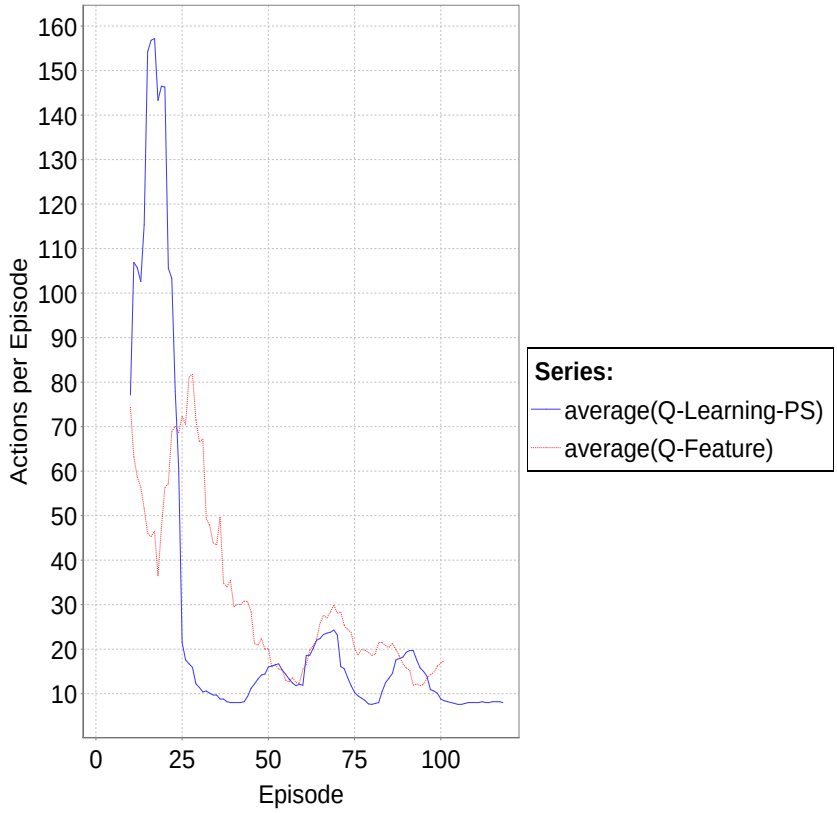


Abbildung 37: Vergleich Q-Feature aus der ersten Versuchsreihe mit Q-Learning PS aus der vierten Versuchsreihe

sodenlänge kleiner oder gleich diesem Minimum ist, wird nicht exploriert. Sobald der Wert überschritten wird, wird Epsilon wieder auf den ursprünglichen Wert zurückgesetzt. Gerade zu Beginn, kann es vorkommen, dass die gewählte Policy schlechter ist, als zufällig Aktionen zu wählen. Deswegen wird Epsilon auf eins gesetzt, wenn eine Episode länger als 300 Aktionen ist. Dies soll nur in Ausnahmesituationen geschehen und soll einen Versuch, der in einer Dauerschleife o.Ä. gefangen ist, *retten*. Wie sich epsilon im Detail verhält, ist in [Chr17] beschrieben.

Auf Abb. 39 ist der Verlauf von Epsilon, in Abhängigkeit von Q-Learning mit PS, dargestellt. Im ersten Drittel wurde stark exploriert. Als eine Episode, die Grenze von Aktionen überschritt, wurde `epsilon` auf eins gesetzt. Dementsprechend wurde ausschließlich exploriert, was lange Episoden zur Folge hatte. Nach dieser Explorationsphase ist eine starke Verbesserung des Lernens sichtbar. In den Phasen, in denen nicht exploriert wird, ist die Episodenlänge minimal oder ach annähernd minimal. Die Episoden werden nur signifikant länger, wenn exploriert wird.

In einem folgenden Versuch, wurde die Q-Tabelle erneut in einen Agenten geladen, sodass beim Lernen der Graph betrachtet werden konnte. Der Graph ist in Abb. 63 auf Seite E, Abb. 64, Abb. 65 und Abb. 66 zu sehen. Der Graph zeigt, dass der Agent einen Ball mit nur sieben Aktionen einsortieren kann. Er nutzt für den roten Ball nur die rechte und für den blauen Ball nur die linke Hand. Die Hand, die er nicht nutzt, verweilt an der Startposition. Dementsprechend kann er zwar gleichfarbige Bälle in einem minimalen Zyklus einordnen, wie auch in Abb. 64 und Abb. 66 zu sehen ist, benötigt jedoch für verschiedenfarbige Bälle länger, da die *inaktive* Hand nicht an der optimalen Position ist. Um von einem in den anderen Zyklus zu wechseln, werden wird der aktive Arm in die

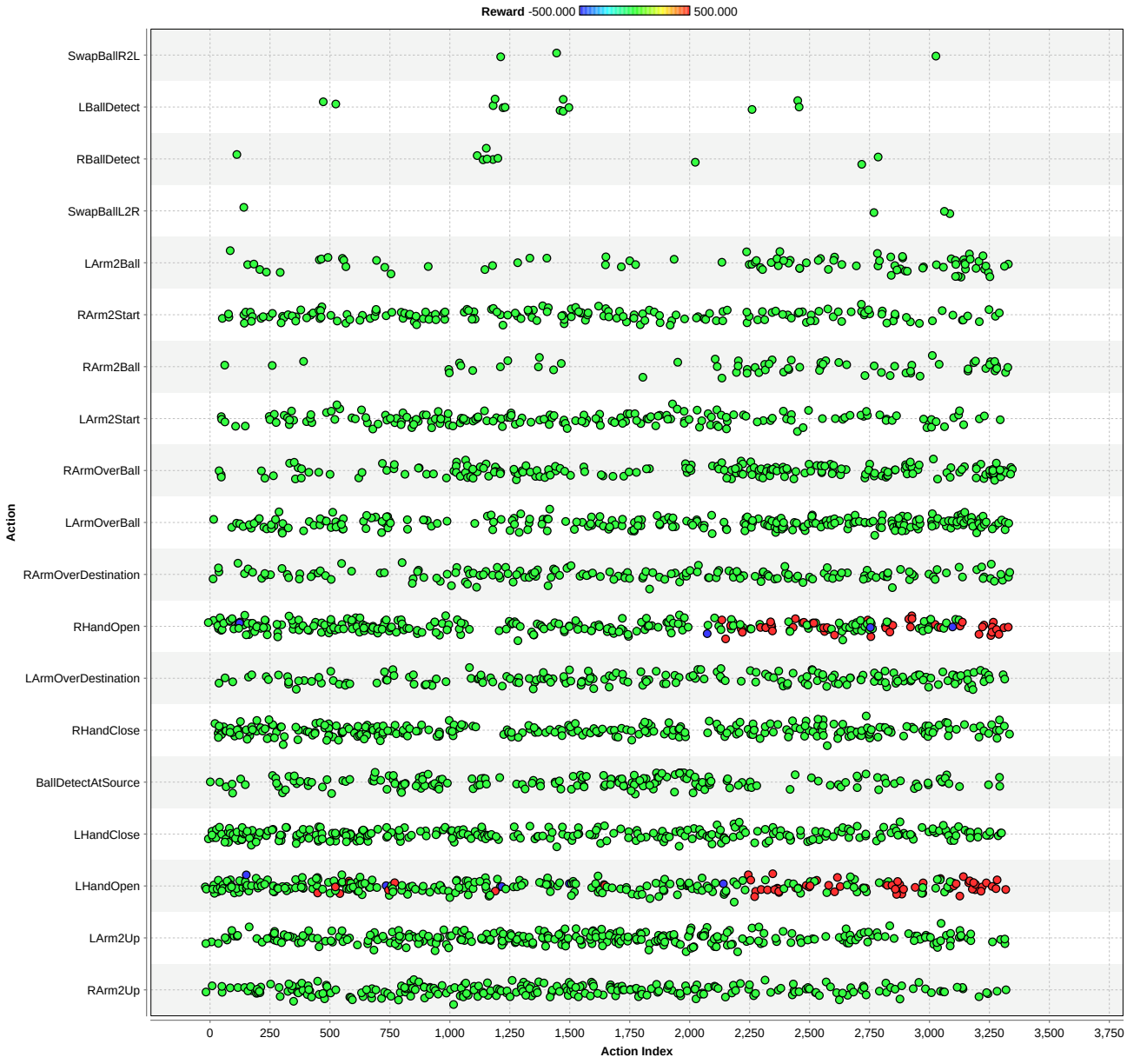


Abbildung 38: Versuchreihe 4, Q-Learning mit PS, Aktionen über den gesamten Lernverlauf

Abbildung 39: Q-Learning mit PS aus der vierten Versuchsreihe. Der Verlauf von `epsilon` ist blau dargestellt.

Start-Position gebracht. Danach wird der andere Arm aktiv und hebt den Ball auf. Das Ballwechsel-Verhalten ist auf Abb. 65 zu sehen. Das Verhalten kann auf zwei mögliche Ursachen zurückgeführt werden:

- Der Agent hat noch nicht genügend exploriert und dadurch noch keinen effizienteren Weg gefunden.
- Die Zustände werden nicht generalisiert, deswegen breitet sich das Wissen, bzw. die Q-Werte, nicht auf ähnliche Zustände aus.

9.4 Zusammenfassung

In diesem Abschnitt wurde die Durchführung der Versuche beschrieben und anschließend ausgewertet. Es wurden vier Versuchsreihen mit jeweils unterschiedlichen Parametern durchgeführt. In der ersten Versuchsreihe waren α und γ auf 0.7 gesetzt. Q-Feature schnitt am besten ab. Es wurde vermutet, dass Algorithmen mit PS besser lernen, wenn α kleiner ist. Dementsprechen wurde α in Versuchsreihe 2 auf 0.1 und in Versuchsreihe 3 auf 0.01 gesetzt. Alle Lernverläufe in der beiden Versuchsreihen hatten im Gegensatz zur ersten Versuchsreihe, eine erheblich längere Episodenlänge im Durchschnitt. In der vierten Versuchsreihe wurden α und γ auf 0.9 gesetzt, mit dem Ergebnis, das Q-Learning mit PS nach ca 40min eine gute Policy entdeckte. Die Policy kann Bälle mit sieben Aktionen einsortieren, wenn die Bälle die gleiche Farbe haben. Wenn die Farbe der Bälle von Episode zu Episode wechselt, werden mehr Aktionen benötigt, da der Arm, der nicht gebraucht wird, an die Start-Position fährt.

10 Auswertung

10.1 Fazit

In dieser Arbeit wurde ein Programm entwickelt, dass es ermöglicht, verschiedene RL-Algorithmen gleichzeitig zu testen. Hierbei kann zur Programmlaufzeit individuell bestimmt werden, welche RL-Algorithmen von den Lernern verwendet werden sollen und welche Parameter diese verwenden.

Die Versuchsdurchführung ist sowohl in der Simulation, als auch mit Roboter in der realen Umgebung möglich. Da die Wissensbasen von Lernvorgängen, beispielsweise Q-Tabellen, abgespeichert und geladen werden können, gibt es nicht nur einen Modus, der das Verhalten des Agenten nach dem Lernvorgang zeigt. Die Agenten können in verschiedenen Lernstadien gezeigt werden[10]. Hierbei kann das Lernen nicht nur durch die Bewegungen des Nao gezeigt werden, sondern wird über eine grafische Benutzeroberfläche visualisiert. Durch das dynamische Ein- und Ausblenden von Optionen ist die GUI übersichtlich und intuitiv bedienbar. Da die Optionen mit Tool-Tips angereichert wurden, ist das Programm leicht verständlich. Des Weiteren werden die über die Oberfläche eingegebenen Parameter überprüft, was eine sichere Exploration des Nutzers ermöglicht. Das einfache Design der Oberfläche verleiht dem Programm ein professionelles Aussehen. Während des Lernvorgangs wird das Verhalten des Agenten, sowie seine Q-Werte und der erhaltene Reward, in einem Graphen dargestellt. Weitere Visualisierungen zeigen die Episodendauer und wichtige Daten über den Lernvorgang an. Dies ermöglicht dem Anwender, ein besseres Verständnis für den Lernvorgang zu erlangen. Um die Lernverläufe auswerten zu können, ist es möglich, diese durch das Programm abzuspeichern. Da das Programm auf einem Rechner des KI-Labors implementiert und getestet wurde, ist es auch auf den Rechnern in diesem Labor lauffähig. Durch den modularen Aufbau des Programms ist einfache Austauschbarkeit gewährleistet.

Durch das Kommentieren des Programmcodes, kann das Programm auch nach langer Nichtbenutzung weiterentwickelt werden. Des Weiteren lassen sich die Koordinaten der Naos über die GUI kalibrieren, dadurch ist es möglich das Programm auch nach Umpositionierung der Naos wieder problemlos zu nutzen.

Das Ziel, eine gute Policy zu finden, wurde erfüllt. Der Algorithmus Q-Learning mit Prioritized Sweeping findet in der Simulation eine gute Policy in unter einer Stunde, bei einer Aktionsdauer von einer Sekunde. Nach dem Lernvorgang kann der Agent mehrere gleichfarbige Bälle hintereinander, mit sieben Aktionen pro Episode, einsortieren. Bei einer Sequenz von verschiedenfarbigen Bällen braucht der Agent länger, da er den Arm, den er nicht benötigt, zur Start-Position zurückfährt.

10.2 Ausblick

Wie in Abschnitt 1.4 auf Seite 5 erwähnt ist das Projekt noch nicht abgeschlossen. Im Folgenden wird gezeigt, wie sich das Projekt weiter entwickeln könnte und welche Aspekte noch behandelt werden müssen.

[10] Es existieren bereits Q-Tabellen mit guter Policy, die geladen werden können um das Lernen zu verdeutlichen.

10.2.1 Grafische Benutzeroberfläche

Einstellungsmöglichkeiten Bislang hat beinhaltet das Programm viele Parameter, die noch nicht in der GUI zugewiesen werden können. Dies betrifft insbesondere das Lernen. In der Simulation muss es in Zukunft möglich sein die Aktionsdauer einzustellen. Die Aktionsdauer ist entscheidend für die Visualisierung. Wenn die Aktionsdauer zu kurz ist, wird die GUI mit Anfragen überlastet und bricht zusammen. Um jedoch schneller zu simulieren, sollten die Visualisierungen, die das Programm verlangsamen, abgeschaltet und die Aktionsgeschwindigkeit erhöht werden. Weitere Einstellungsmöglichkeiten, die bisher noch nicht implementiert wurden sind:

- Die *Trace Decay Rate* λ, bei $Q(\lambda)$,
- die *Temperatur* von Softmax,
- der Schwellwert `priority_treshold` von PS,
- und die Tilings von `QFeature`.

Des Weiteren sollte es möglich sein, über die GUI Modelle abzuspeichern.

Visualisierung des Lernens In dieser Arbeit konnte das Lernen auf drei verschiedene Art und Weisen visualisiert werden. Dennoch fehlen wichtige Informationen, die zwar im Programm vorhanden sind, aber nicht in der GUI angezeigt werden. So zeigt die Lernkurve zwar deutlich den Lernverlauf an, aber nicht, ob eine Episode erfolgreich, mit einem positiven Reward, endete oder nicht. Außerdem wäre ein MSE-Kurve, um den Lernverlauf darzustellen, aussagekräftiger, da diese Ansicht auch oft in der Literatur, beispielsweise [SB12], verwendet wird. Des Weiteren ist eine Anzeige, die die gesamte Aktionsanzahl zeigt, nützlich. Die Aktionsanzahl kann auch der bestehenden Detail-Ansicht hinzugefügt werden.

Des Weiteren wäre es von Vorteil, wenn angezeigt werden würde, ob der Agent exploriert oder exploitiert. Dies könnte entweder im Graph oder in der Animation des Agenten farblich sichtbar gemacht werden.

Kalibrierung der Koordinaten Die Kalibrierung der Koordinaten ist unintuitiv. Es sollte so gestaltet sein, dass der Nutzer im Klaren ist, welche Koordinate er verändert. Die Koordinaten sollten sequenziell in einer sinnvollen Reihenfolge abgefragt werden.

Präsentation der Aktionen Um Zuschauern die *Kommunikation* zwischen Agent und Umgebung näher zu bringen, wäre ein Modus sinnvoll, in welchem ein Nutzer den Agenten *spielen* könnte. Der Nutzer hätte eine Oberfläche mit einem Graphen, wie in Abb. 25 vor sich. Der Nutzer fungiert hierbei als Policy und wählt über die Oberfläche die nächste Aktion aus.

10.2.2 Reinforcement Learning

Veränderung der Parameter über die Zeit Der einzige Zahlen-Parameter, der sich über die Zeit ändert ist `epsilon`. Andere Parameter, wie `alpha` oder `gamma`, ändern sich bisher nicht. Es ist sinnvoll die Werte über die Zeit anzupassen, um beispielsweise das Lernen zu Beginn zu beschleunigen und später zu entschleunigen, damit Gelerntes gefestigt und nicht verlernt wird.

Explorator Funktionen Bisher wurden simple Explorationsalgorithmen implementiert. *PUCB* ist ein Explorationsalgoritmus, der *UCB1* erweitert. *UCB* steht für *upper confidence bound*. PUCB steht für Predictor + UCB und wurde in [Ros11] vorgestellt. PUCB wurde in [SHM+16] genutzt. Im Gegensatz zu den bereits implementierten Explorationsstrategien, bezieht PUCB in die Exploration mit ein, wie oft eine (s,a)-Paar durchlaufen wurde.

Estimator Funktionen Neuronale Netze wurden bisher noch nicht betrachtet. In [MKS+] wurde Deep Q Learning genutzt, um einem Agenten das Spielen von Atari-Spielen beizubringen. In [SHM+16] wurde unter anderem ein Policy Network genutzt. Beide Typen von neuronalen Netzen sollten implementiert werden. Natural Actor Critic wird in [DCP+13] genutzt und sollte implementiert werden.

Temporal Difference Learning Funktionen Wie bereits in Abschnitt 9.3 auf Seite 65 erwähnt, sollte aufgrund der schlechten Performance von $Q(\lambda)$ eine andere Variante des Algorithmus gewählt werden. Da Waltkins $Q(\lambda)$, wie in Abschnitt 5.2.3 auf Seite 26 beschrieben, sehr kurze *Traces* entwickelt, sollte zunächst Pengs $Q(\lambda)$, siehe Zeile 14 auf Seite 27, implementiert werden. $Q(\lambda)$ sollte auch für andere Estimator als die Q-Tabelle implementiert werden. Da Q-Learning eine off-Policy Methode ist, siehe Abschnitt 5.2.1 auf Seite 24, wäre es interessant das unterschiedliche Lernverhalten einer on-Policy Methode zu Untersuchen. Hierbei würde sich SARSA, siehe Abschnitt 5.2.2 auf Seite 25, anbieten, da die Methode, abgesehen davon, dass sie on-Policy ist, Q-Learning ähnelt.

Planende Funktionen Weitere planende Funktionen sollen implementiert und getestet werden. Beispiele hierfür sind Dyna-Q+, siehe [SB12], und Prioritized Experience Preplay, siehe [SQAS15].

10.2.3 Überhitzungsproblem

Nach ca. 10 Min. im Einsatz überhitzt der Nao, da die Motoren in seinem Arm zu heiß werden. Die Folgen sind, dass die Koordinaten nicht richtig angefahren werden, wodurch Nao und der Versuchsaufbau beschädigt werden können und dass die Motoren langfristig kaputtgehen. Es gibt verschiedene Möglichkeiten dieses Problem zu lösen:

1. Der Nao begibt sich automatisch in eine sichere Position, wenn er überhitzt und schaltet seine Motoren für eine gewisse Zeit ab.

2. Überhitzte Motoren wirken sich negativ auf den Reward aus, der Agent kann sich selber in einen Zustand versetzen, in dem er die Motoren abschalten kann. In diesem Zustand bekommt der Agent keinen negativen Reward, solange die Motoren überhitzt sind.

3. Der Ansatz von Punkt 2 wird verbessert, indem der Zustand der Motoren Teil des Agenten-Zustands wird.

Der Lösungsansatz von Punkt 1 schützt den Nao zwar vor Schaden, der Agent lernt jedoch nicht, welche Aktionen zu einer Überhitzung führen. Wenn er dies wüsste, könnte er entsprechende Aktionen vermeiden. Bei Punkt 2 lernt der Agent, in einen sicheren Zustand zu gehen, wenn er überhitzt. Da es dadurch Eigenschaften gibt, die sich auf den Reward auswirken, jedoch nicht im Zustand erfasst sind, muss der Agent bei jedem Überhitzen

und Erhitzen wieder umlernen, weil sich die für ihn gestellte Aufgabe ändert. Mit dem Lösungsansatz von Punkt 3 wird der Zustandsraum erweitert. Der Agent ist dadurch in der Lage, eine Verbindung zwischen negativen Reward und Zustand herzustellen. Der Zustandsraum könnte beispielsweise über zweit Dimensionen mit jeweils zwei Dimensions-Zuständen verfügen. Die Zustandsdimensionen repräsentieren die Motoren des rechten und des linken Armes. Die Dimensions-Zuständen sind *Motor ist überhitzt* und *Motor ist nicht überhitzt*.

10.2.4 Änderung des Zustandaktionsraumes

Bisher ist der Zustandsaktionsraum in diesem Versuch diskret. Bei aktuellen Arbeiten, die sich mit RL in Verbindung mit Robotik beschäftigen, sind die Zustandsaktionsräume kontinuierlich. In kontinuierlichen Zustandsaktionsäumen ist RL schwerer umzusetzen, da Zustände und Aktionen generalisiert werden müssen. Der Nao hätte keine Aktionen, die die Arme zu festen Koordinaten bewegen, sondern Aktionen, die die Arme des Nao in bestimmte Richtungen bewegen. Wie in [GHLL17] könnte ein Zustand unter anderem durch die Gelenkwinkel in den Armen des Nao ausgedrückt werden. Um eine Zerstörung des Versuchsaufbaus und des Nao zu vermeiden, sollten wie in [SCSD17], Bereiche definiert werden in denen der Nao die Arme bewegen darf. Wenn er diesen Bereich verlässt, müsste er die Bewegung anhalten.

Literatur

[Alb75] Albus, J. S.: A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC). In: *Journal of Dynamic Systems, Measurement, and Control* 97 (1975), Sep, Nr. 3, 220-227. http://dx.doi.org/10.1115/1.3426922. – DOI 10.1115/1.3426922. – ISSN 0022–0434

[Ald17a] Aldebaran: *NAO - Technical Guide*. http://doc.aldebaran.com/2-1/family/index.html. Version: 2017

[Ald17b] Aldebaran: *NAO, IBM create new Hilton concierge*. https://developer.softbankrobotics.com/us-en/showcase/nao-ibm-create-new-hilton-concierge. Version: May 2017

[Bar06] Barto, Andrew: *Chapter 7: Eligibility Traces. 2nd ed.* http://www-anw.cs.umass.edu/~barto/courses/cs687/Chapter%207.pdf[Accessed23Nov.2017]. Version: 2006

[Chr17] Christ, Colin: Sortierung von verschiedenfarbigen Bällen mit dem Nao mittels Reinforcement Learning. In: *Studienarbeit* (2017)

[DCP+13] Du, T. ; Cox, M. T. ; Perlis, D. ; Shamwell, J. ; Oates, T.: From Robots to Reinforcement Learning. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013. – ISSN 1082–3409, S. 540–545

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2

[GHLL17] Gu, S. ; Holly, E. ; Lillicrap, T. ; Levine, S.: Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, S. 3389–3396

[GLSL16] Gu, Shixiang ; Lillicrap, Timothy P. ; Sutskever, Ilya ; Levine, Sergey: Continuous Deep Q-Learning with Model-based Acceleration. In: *Computing Research Repository* abs/1603.00748 (2016). http://arxiv.org/abs/1603.00748

[KLM96] Kaelbling, Leslie P. ; Littman, Michael L. ; Moore, Andrew W.: Reinforcement Learning: A Survey. (1996)

[KP12] Kober, Jens ; Peters, Jan ; Wiering, Marco (Hrsg.) ; Otterlo, Martijn van (Hrsg.): *Reinforcement Learning in Robotics: A Survey*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – 579–610 S. http://dx.doi.org/10.1007/978-3-642-27645-3_18. http://dx.doi.org/10.1007/978-3-642-27645-3_18. – ISBN 978–3–642–27645–3

[LLN17] Le, T. D. ; Le, A. T. ; Nguyen, D. T.: Model-based Q-learning for humanoid robots. In: *2017 18th International Conference on Advanced Robotics (ICAR)*, 2017, S. 608–613

[LPKQ16] Levine, Sergey ; Pastor, Peter ; Krizhevsky, Alex ; Quillen, Deirdre:

Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection. In: *Computing Research Repository* abs/1603.02199 (2016). http://arxiv.org/abs/1603.02199

[MA93] Moore, Andrew W. ; Atkeson, Christopher G.: Prioritized Sweeping Reinforcement Learning With Less Data and Less Time. Version: 1993. http://dx.doi.org/10.1007/978-94-011-1514-8-1. In: Sherwin, Keith (Hrsg.): *Introduction to Thermodynamics*. Dordrecht : Springer Netherlands, 1993. – DOI 10.1007/978–94–011–1514–8–1. – ISBN 978–0–412–47640–2, S. 1–20

[MKS+] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: *Playing Atari with Deep Reinforcement Learning*. http://arxiv.org/pdf/1312.5602v1

[MS97] McGovern ; Sutton: Towards a Better Q(λ). In: *Unbekannt* (1997)

[PW96] Peng, Jing ; Williams, Ronald J.: Incremental multi-step Q-learning. In: *Machine Learning* 22 (1996), Mar, Nr. 1, 283–290. http://dx.doi.org/10.1007/BF00114731. – DOI 10.1007/BF00114731. – ISSN 1573–0565

[Ros11] Rosin, Christopher D.: Multi-armed bandits with episode context. In: *Annals of Mathematics and Artificial Intelligence* 61 (2011), Mar, Nr. 3, 203–230. http://dx.doi.org/10.1007/s10472-011-9258-6. – DOI 10.1007/s10472–011–9258–6. – ISSN 1573–7470

[SB12] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. MIT Press, 2012

[SCSD17] Schwung, D. ; Csaplar, F. ; Schwung, A. ; Ding, S. X.: An application of reinforcement learning algorithms to industrial multi-robot stations for cooperative handling operation. In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, S. 194–199

[SHM+16] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent ; van den Driessche, George ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Panneershelvam, Veda ; Lanctot, Marc ; Dieleman, Sander ; Grewe, Dominik ; Nham, John ; Kalchbrenner, Nal ; Sutskever, Ilya ; Lillicrap, Timothy ; Leach, Madeleine ; Kavukcuoglu, Koray ; Graepel, Thore ; Hassabis, Demis: Mastering the game of Go with deep neural networks and tree search. In: *Nature* 529 (2016), Nr. 7587, S. 484–489. http://dx.doi.org/10.1038/nature16961. – DOI 10.1038/nature16961. – ISSN 1476–4687

[SQAS15] Schaul, Tom ; Quan, John ; Antonoglou, Ioannis ; Silver, David: Prioritized Experience Replay. In: *CoRR* abs/1511.05952 (2015). http://arxiv.org/abs/1511.05952

[TP17] Travnik, J. B. ; Pilarski, P. M.: Representing high-dimensional data to intelligent prostheses and other wearable assistive robots: A first comparison of tile coding and selective Kanerva coding. In: *2017 International Conference on Rehabilitation Robotics (ICORR)*, 2017, S. 1443–1450

[Wat89] Watkins, Christopher John Cornish Hellaby: *Learning from Delayed Rewards*. Cambridge, UK, King's College, Diss., May 1989. http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

Anhang

A Ausführen des Programms

Das Programm läuft unter Python 2.7.13 und wird durch das Ausführen der folgenden Datei gestartet:
`/NaoSortingReinforcement/GUI/main.py`

Folgende Pakete werden verwendet:

- `naoqi`, muss nicht verwendet werden. In diesem Fall ist es jedoch nicht möglich den Nao zu verwenden.
- `cv2`, muss nicht verwendet werden. In diesem Fall ist es jedoch nicht möglich den Nao zu verwenden.
- `HtmlPy`
- `csv`
- `ast`
- `numpy`
- `threading`
- `time`
- `sys`
- `thread`
- `PySide`
- `os`
- `sklearn`

B Argumentenübergabe von der GUI bis zum Agenten

Im Folgenden ist eine Beispiel für die Übergabe der Parameter von der GUI, bis zum Agenten gezeigt.

Der JavaScript-Teil der GUI übermittelt an das `BackEnd` folgenden `String`, der in ein `dictionary` geparsed wird:

```
{'model': 'true', 'epsilonfactor': '0.55', 'epsilon_limit': '0.1', 'name'
 : 'MOCK', 'epsilon': '1', 'planning_method': 'prioritized sweeping', '
model_name': 'new', 'estimator': 'QTable', 'alpha': '0.99', '
estimator_knowlage_source_name': 'new', 'exploration_function': '
epsilon_greedy_exploration', 'temporal_difference_learning_function': 'Q
-Learning', 'gamma': '0.99'}
```

Das `BackEnd` fügt Objektinstanzen hinzu, die von mehreren Agenten *gleichzeitig* genutzt werden können, und übergibt die Parameterliste an die Methode `startLearning` der Klasse `NaoSortingReinforcementAPI`, welche die Liste an eine `LearnerValidatorPair`-Instanz übergibt, die mit der Liste den Agenten, über die Methode `create_agent`, zusammenbaut:

```
{'planning_method': 'prioritized sweeping', 'epsilon_limit': 0.1, '
 threadLock': <thread.lock object at 0x05255570>, 'epsilon': 1.0, '
 epsilonImprovementValue': 0.55, 'validator': <Reinforcement.Agent.
 NaoMock.NaoMock instance at 0x0571D648>, 'model': <Reinforcement.Agent.
 Model.Model object at 0x0570DE70>, 'step_reward': -15, '
 temporal_difference_learning_function': 'Q-Learning'}
```

`create_agent` erstellt den Agenten. Die Parameterliste des Agenten `kwargs` kann folgendermaßen aussehen:

```
{'epsilon_impover': <Reinforcement.Agent.EpsilonImprover.EpsilonImprover
 instance at 0x0571D738>, 'epsilonImprovementValue': 0.55, 'epsilon_limit
 ': 0.1, 'name': 'MOCK', 'methods_after_action': [<bound method Model.
 update of <Reinforcement.Agent.Model.Model object at 0x0570DE70>>, <
 function add_sa_to_priority_queue at 0x05608A30>], '
 methods_if_episode_ends': [<function q_update_adapter at 0x03CBAE70>, <
 bound method EpsilonImprover.improve_epsilon_after_finish_turn of <
 Reinforcement.Agent.EpsilonImprover.EpsilonImprover instance at 0
 x0571D738>>, <bound method Logger.finishTurn of <Reinforcement.Logger.
 Logger instance at 0x0571D6C0>>], 'epsilon': 1.0, 'random': <random.
 Random object at 0x0B46E558>, 'rewarder': <Reinforcement.Agent.Rewarder.
 Rewarder instance at 0x0571D3F0>,
'ACTIONS': [<Reinforcement.Actions.Action.LHandOpen object at 0x0570DB90
 >, <Reinforcement.Actions.Action.RHandOpen object at 0x0570DB30>, <
 Reinforcement.Actions.Action.LHandClose object at 0x0570DB50>, <
 Reinforcement.Actions.Action.RHandClose object at 0x0570DB70>, <
 Reinforcement.Actions.Action.LArm2Start object at 0x0570DBF0>, <
 Reinforcement.Actions.Action.RArm2Start object at 0x0570DC10>, <
 Reinforcement.Actions.Action.LArm2Up object at 0x0570DC50>, <
 Reinforcement.Actions.Action.RArm2Up object at 0x0570DB10>, <
 Reinforcement.Actions.Action.LArmOverBall object at 0x0570D8F0>, <
 Reinforcement.Actions.Action.RArmOverBall object at 0x0570DC70>, <
 Reinforcement.Actions.Action.LArm2Ball object at 0x0570DC30>, <
 Reinforcement.Actions.Action.RArm2Ball object at 0x0570DA90>, <
 Reinforcement.Actions.Action.LArmOverDestination object at 0x0570DF50>,
 <Reinforcement.Actions.Action.RArmOverDestination object at 0x0570DFB0>,
  <Reinforcement.Actions.Action.SwapBallL2R object at 0x0570DFF0>, <
 Reinforcement.Actions.Action.SwapBallR2L object at 0x0570DFD0>, <
 Reinforcement.Actions.Action.LBallDetect object at 0x0570DD10>, <
 Reinforcement.Actions.Action.RBallDetect object at 0x0570DCF0>, <
 Reinforcement.Actions.Action.BallDetectAtSource object at 0x0570DCD0>],
 'update_estimator': <function q_update_adapter at 0x03CBAE70>, '
 methods_if_episode_goes_on': [<function q_update_adapter at 0x03CBAE70>,
  <bound method Logger.appendActionTuple of <Reinforcement.Logger.Logger
 instance at 0x0571D6C0>>], 'methods_before_action': [<function
 estimate_best_action at 0x05608930>, <function
 epsilon_greedy_exploration at 0x056087F0>], 'agent': <Reinforcement.
 Agent.Agent.Agent instance at 0x0571D6E8>, 'emergency_method': <bound
 method EpsilonImprover.set_epsilon_to_one of <Reinforcement.Agent.
 EpsilonImprover.EpsilonImprover instance at 0x0571D738>>, 'estimator':
```

```
    {}, 'alpha': 0.99, 'model': <Reinforcement.Agent.Model.Model object at 0
    x0570DE70>, 'step_reward': -15, 'logger': <Reinforcement.Logger.Logger
    instance at 0x0571D6C0>, 'gamma': 0.99}
3
```

C Erste Versuchsreihe

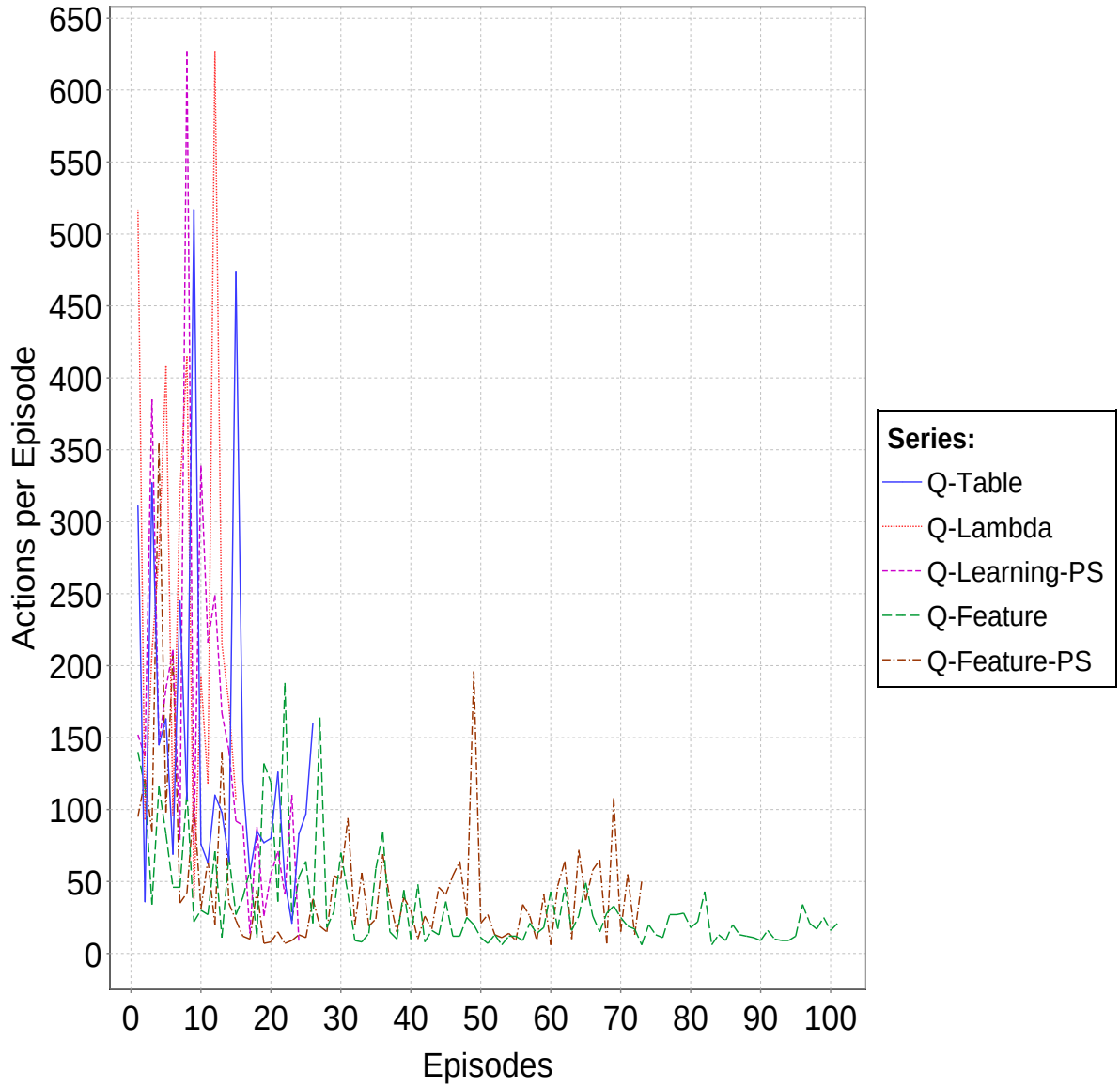


Abbildung 40: Lernverläufe der Algorithmen in der ersten Versuchsreihe

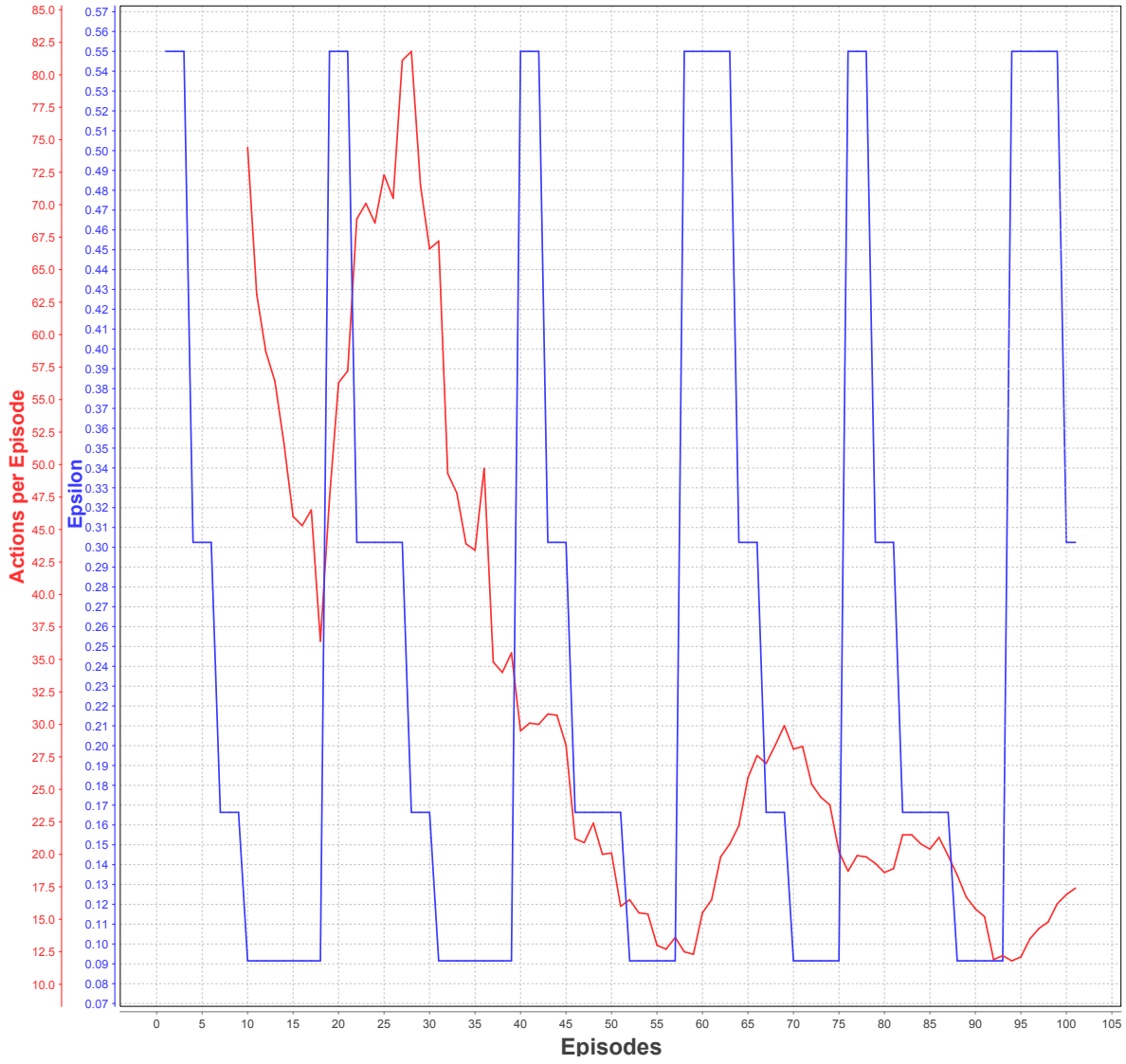


Abbildung 41: Q-Feature Lernverlauf in der ersten Versuchsreihe. Zyklisches Ansteigen und Abfallen von `epsilon`

Abbildung 42: Lernverlauf von Q-Learning in der ersten Versuchsreihe

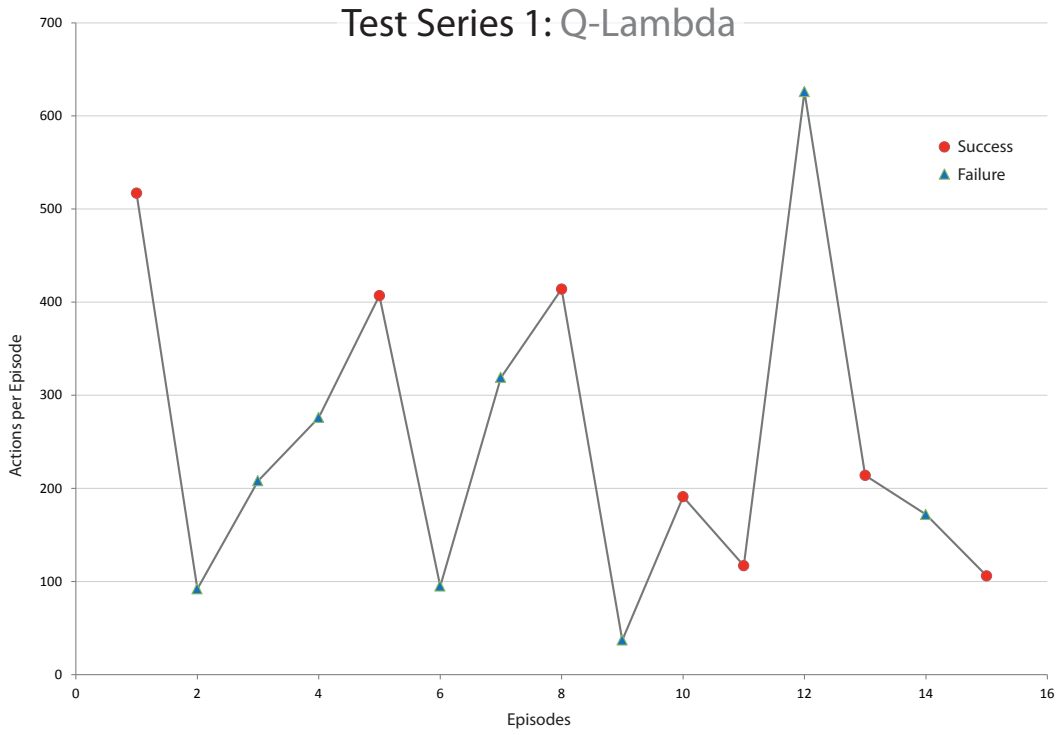


Abbildung 43: Lernverlauf von Q(λ) in der ersten Versuchsreihe

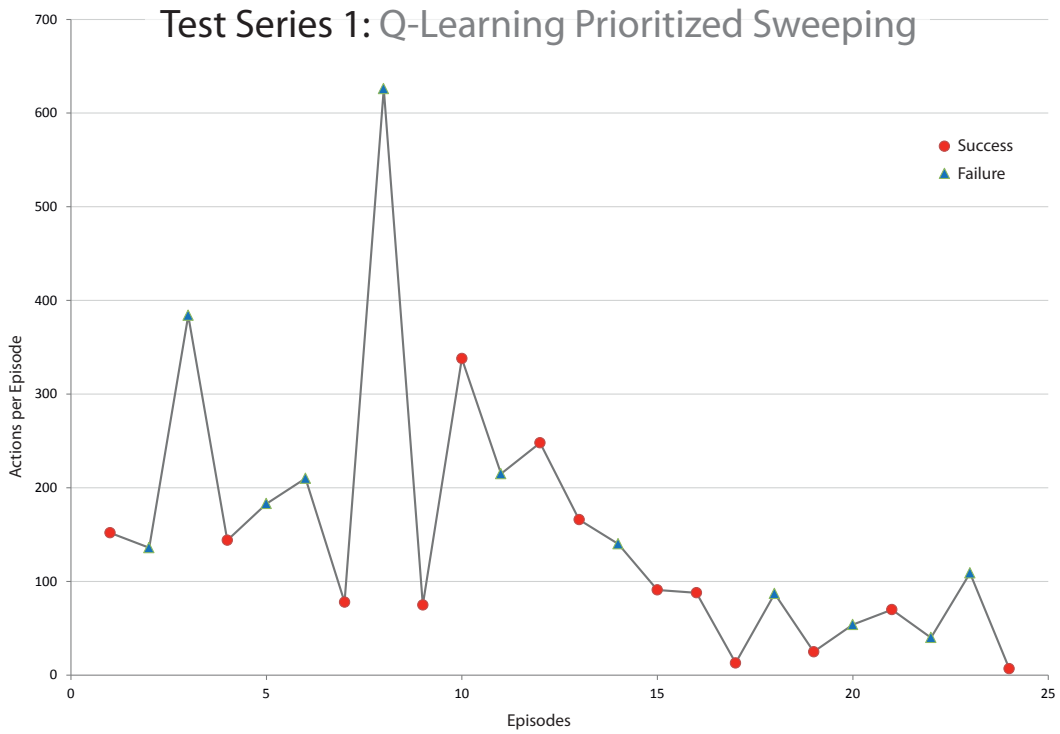


Abbildung 44: Lernverlauf von Q-Learning mit Prioritized Sweeping in der ersten Versuchsreihe

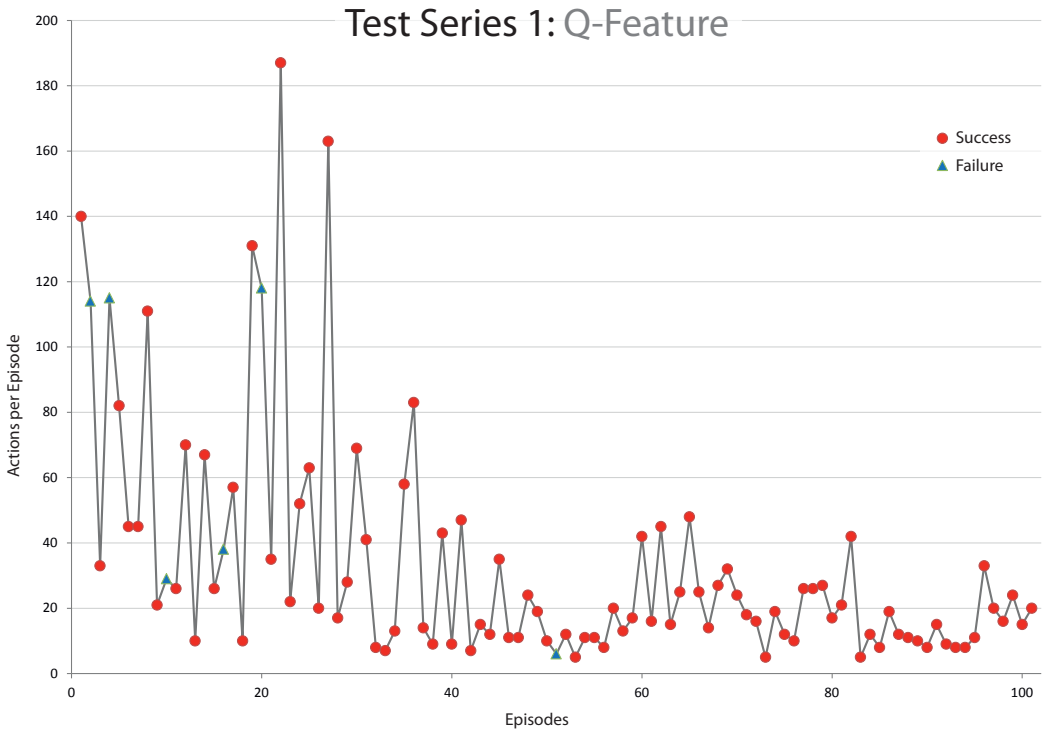


Abbildung 45: Lernverlauf von Q-Feature in der ersten Versuchsreihe

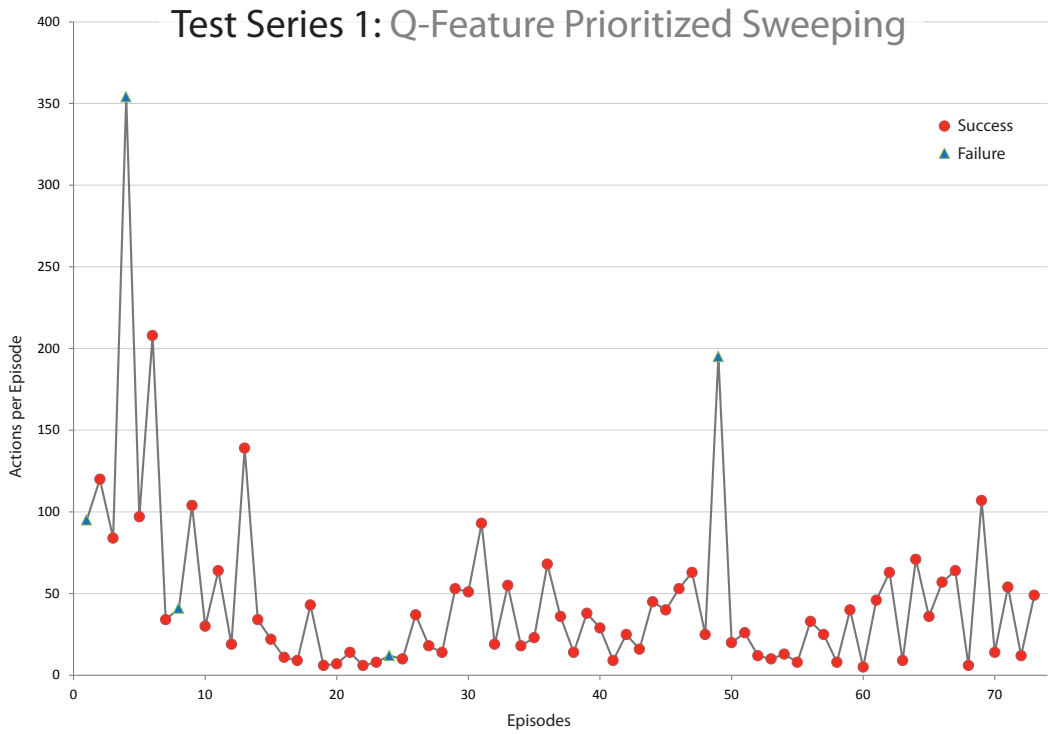


Abbildung 46: Lernverlauf von Q-Freature mit Prioritized Sweeping in der ersten Versuchsreihe

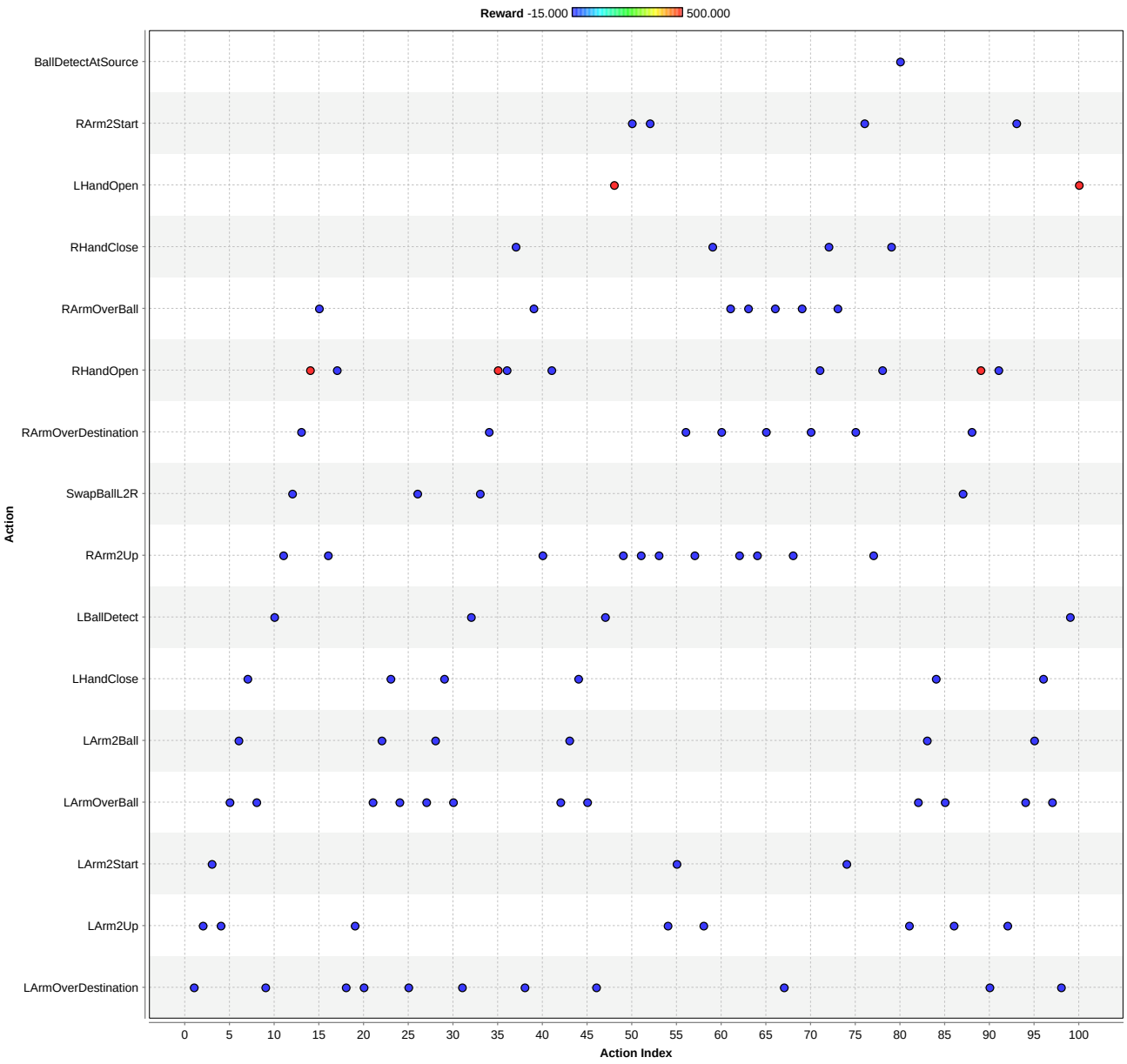


Abbildung 47: Q-Feature in der realen Umgebung mit dem Nao. Es wurden die ausgefüllten Feature-Werte von dem Q-Feature der ersten Versuchsreihe genutzt. `epsilon` wurde auf null gesetzt.

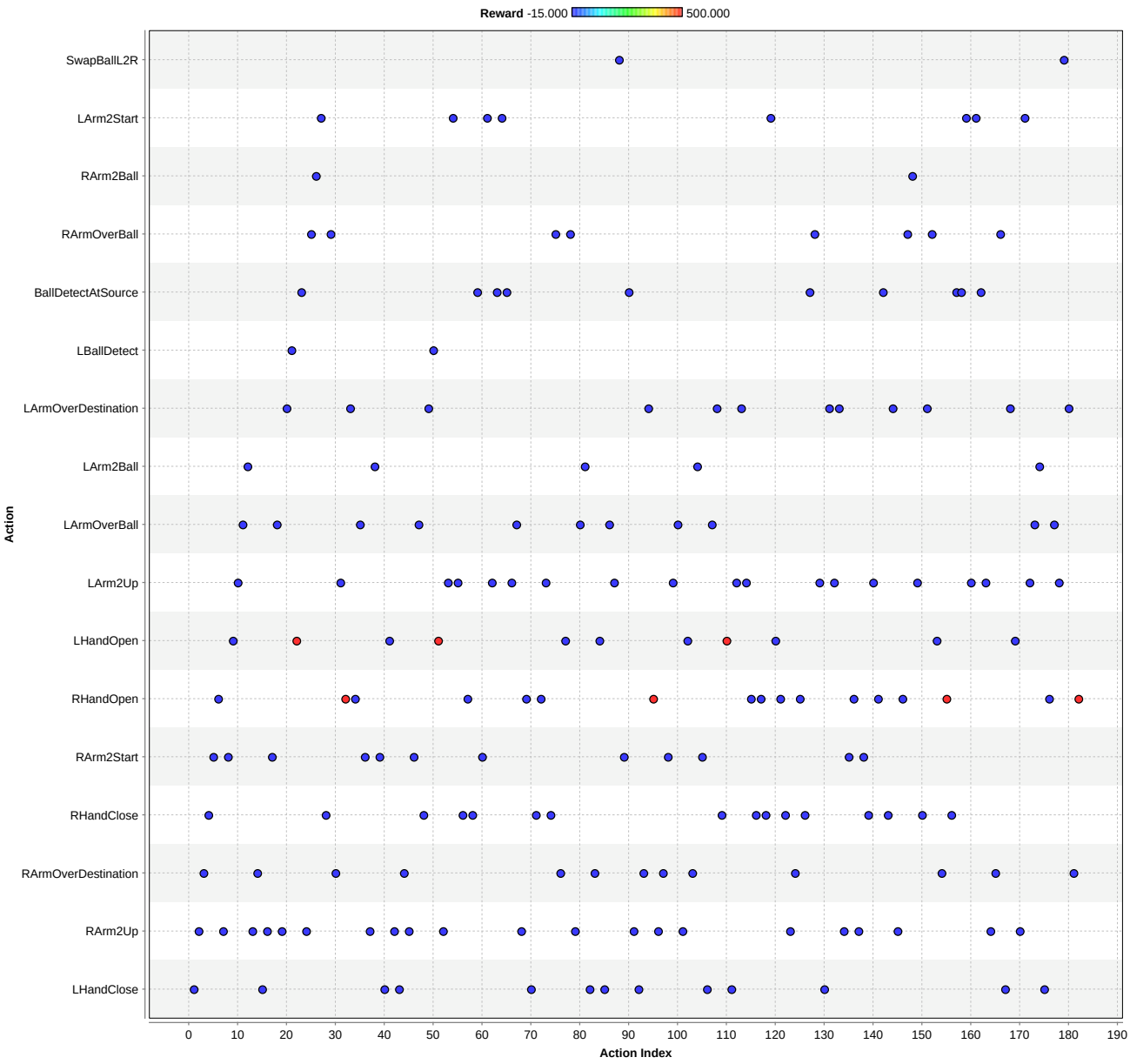


Abbildung 48: Q-Feature in der realen Umgebung mit dem Nao. Es wurden die ausgefüllten Feature-Werte von dem Q-Feature der ersten Versuchsreihe genutzt. `epsilon` wurde auf 0.55 gesetzt.

D Zweite Versuchsreihe

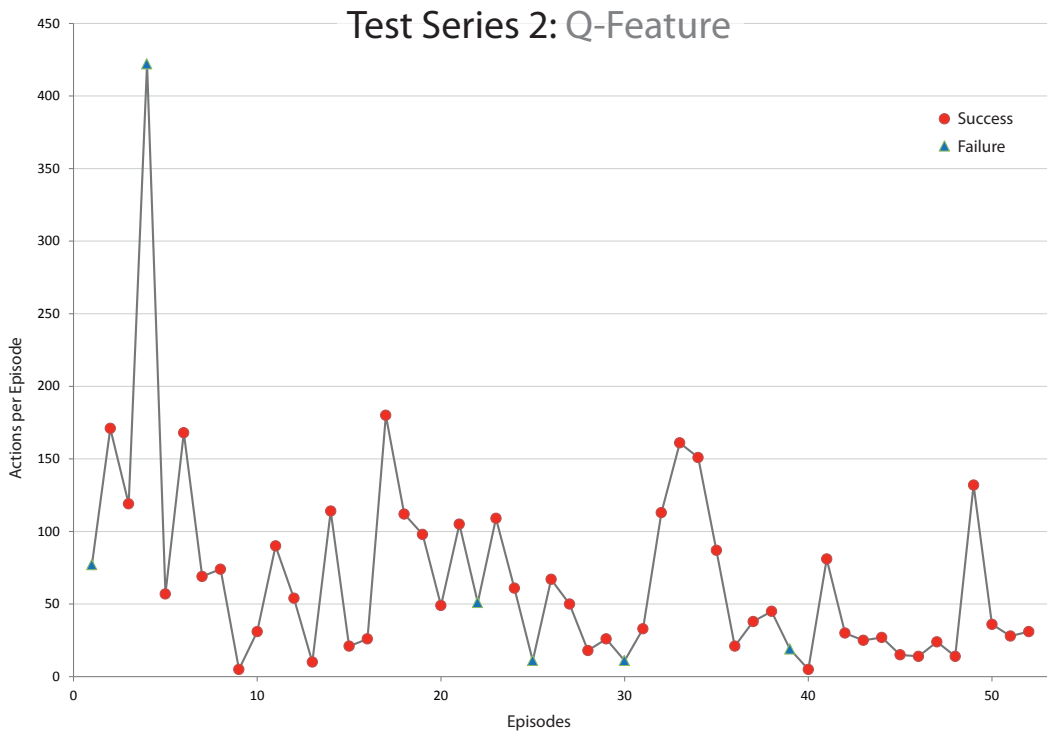


Abbildung 49: Lernverlauf von Q-Feature in der zweiten Versuchsreihe

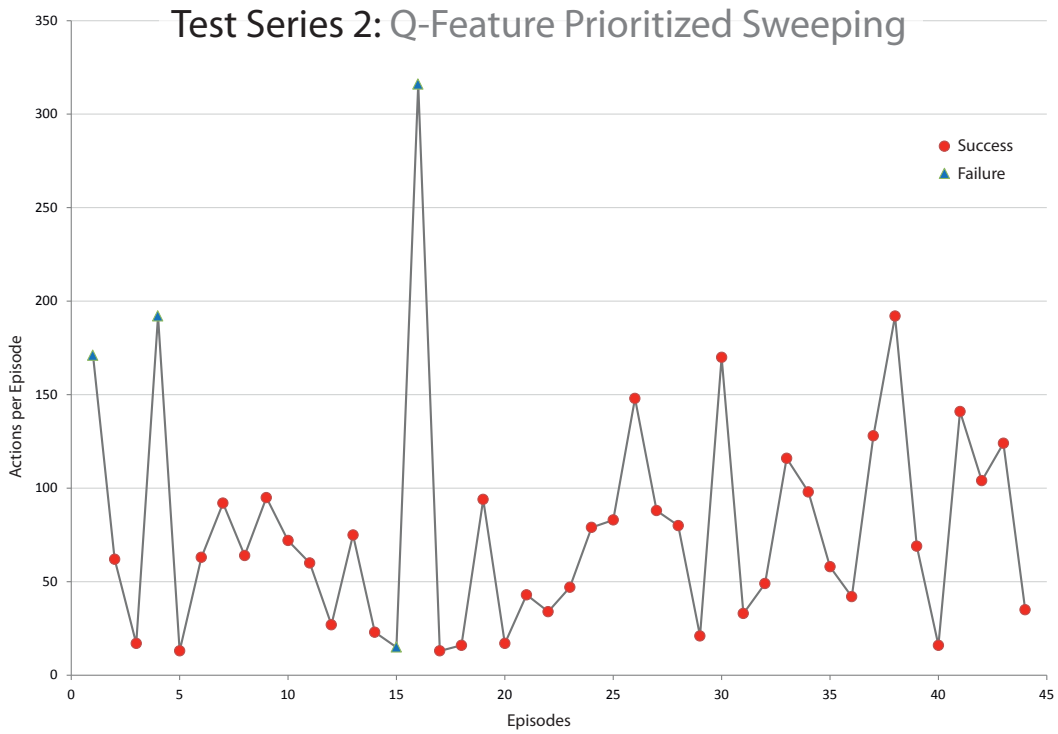


Abbildung 50: Lernverlauf von Q-Freature mit Prioritized Sweeping in der zweiten Versuchsreihe

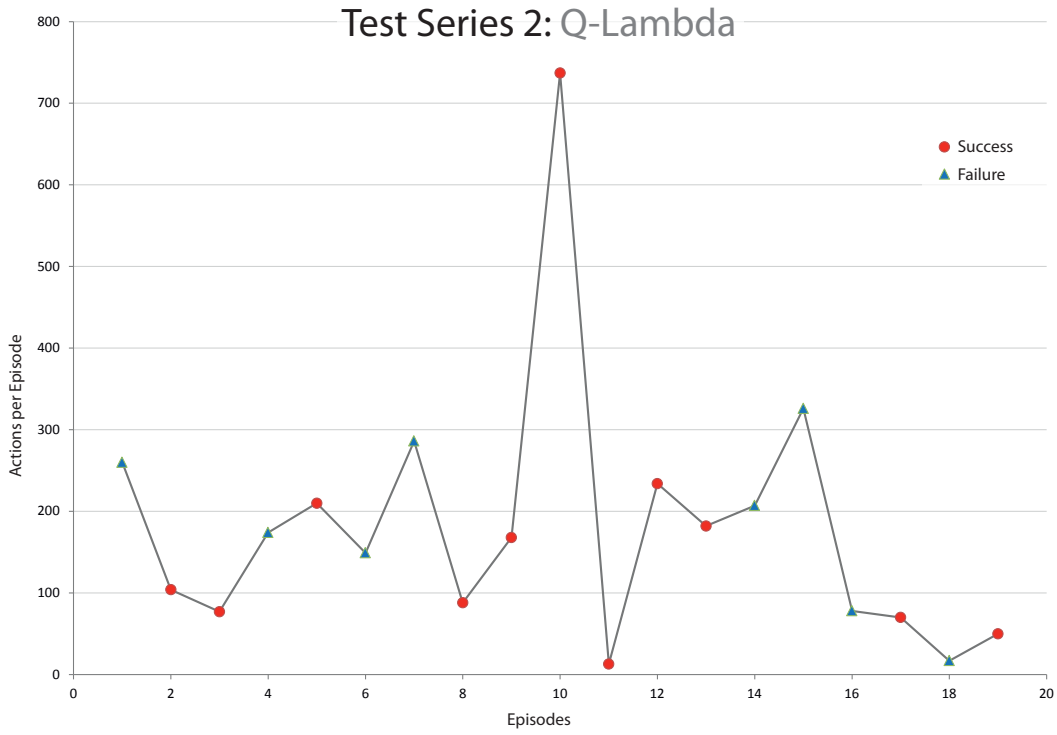


Abbildung 51: Lernverlauf von Q(λ) in der zweiten Versuchsreihe

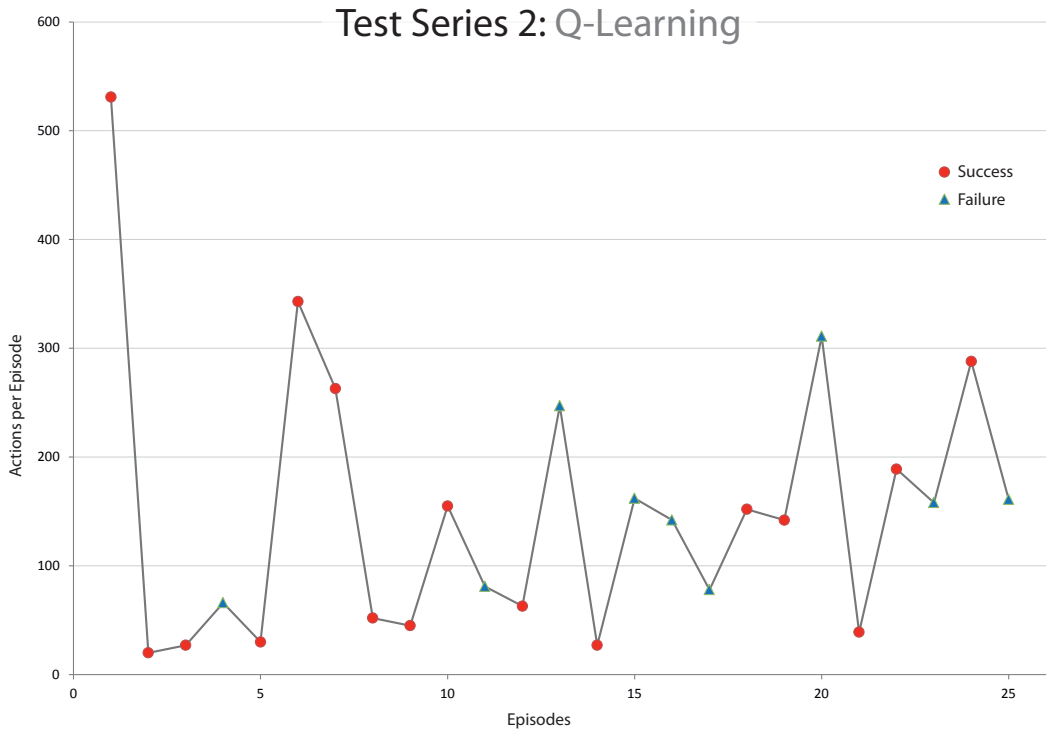


Abbildung 52: Lernverlauf von Q-Learning in der zweiten Versuchsreihe

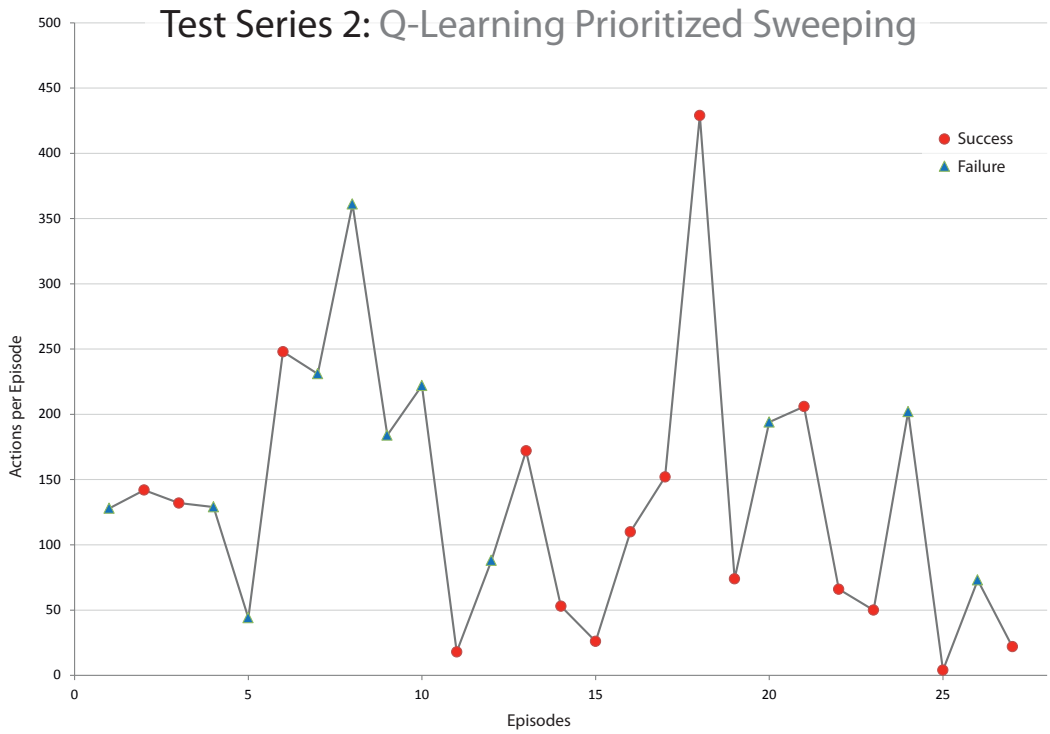


Abbildung 53: Lernverlauf von Q-Learning mit Prioritized Sweeping in der zweiten Versuchsreihe

E Dritte Versuchsreihe

Abbildung 54: Lernverlauf von Q-Feature in der dritten Versuchsreihe

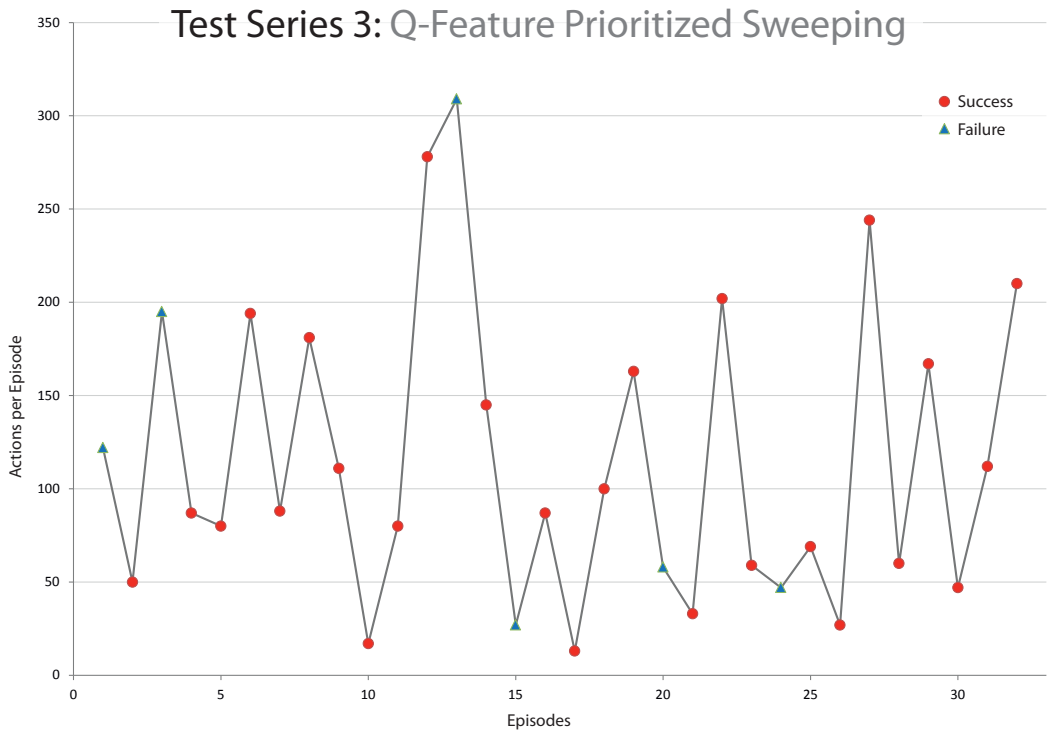


Abbildung 55: Lernverlauf von Q-Freature mit Prioritized Sweeping in der dritten Versuchsreihe

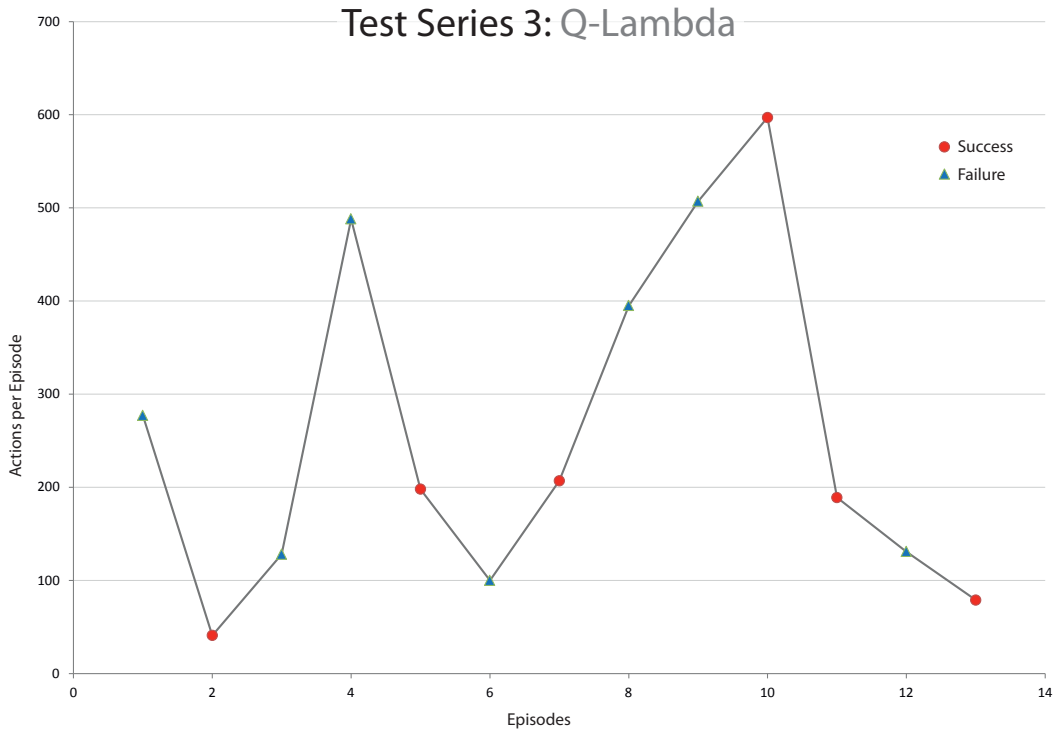


Abbildung 56: Lernverlauf von Q(λ) in der dritten Versuchsreihe

Abbildung 57: Lernverlauf von Q-Learning in der dritten Versuchsreihe

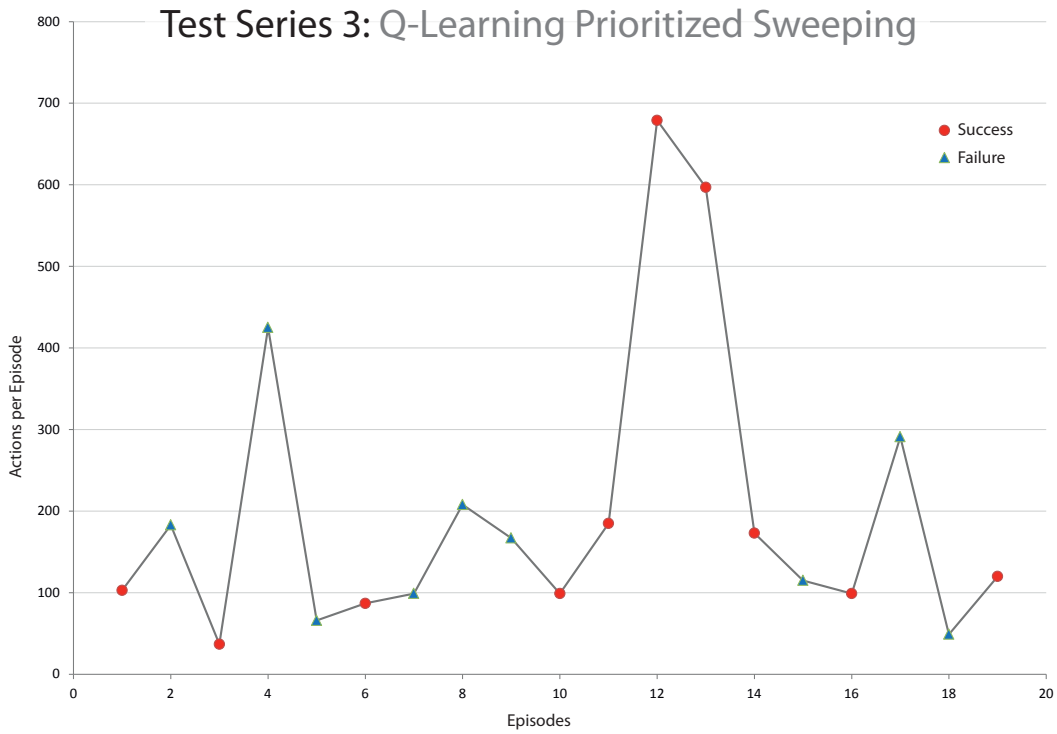


Abbildung 58: Lernverlauf von Q-Learning mit Prioritized Sweeping in der dritten Versuchsreihe

F Vierte Versuchsreihe

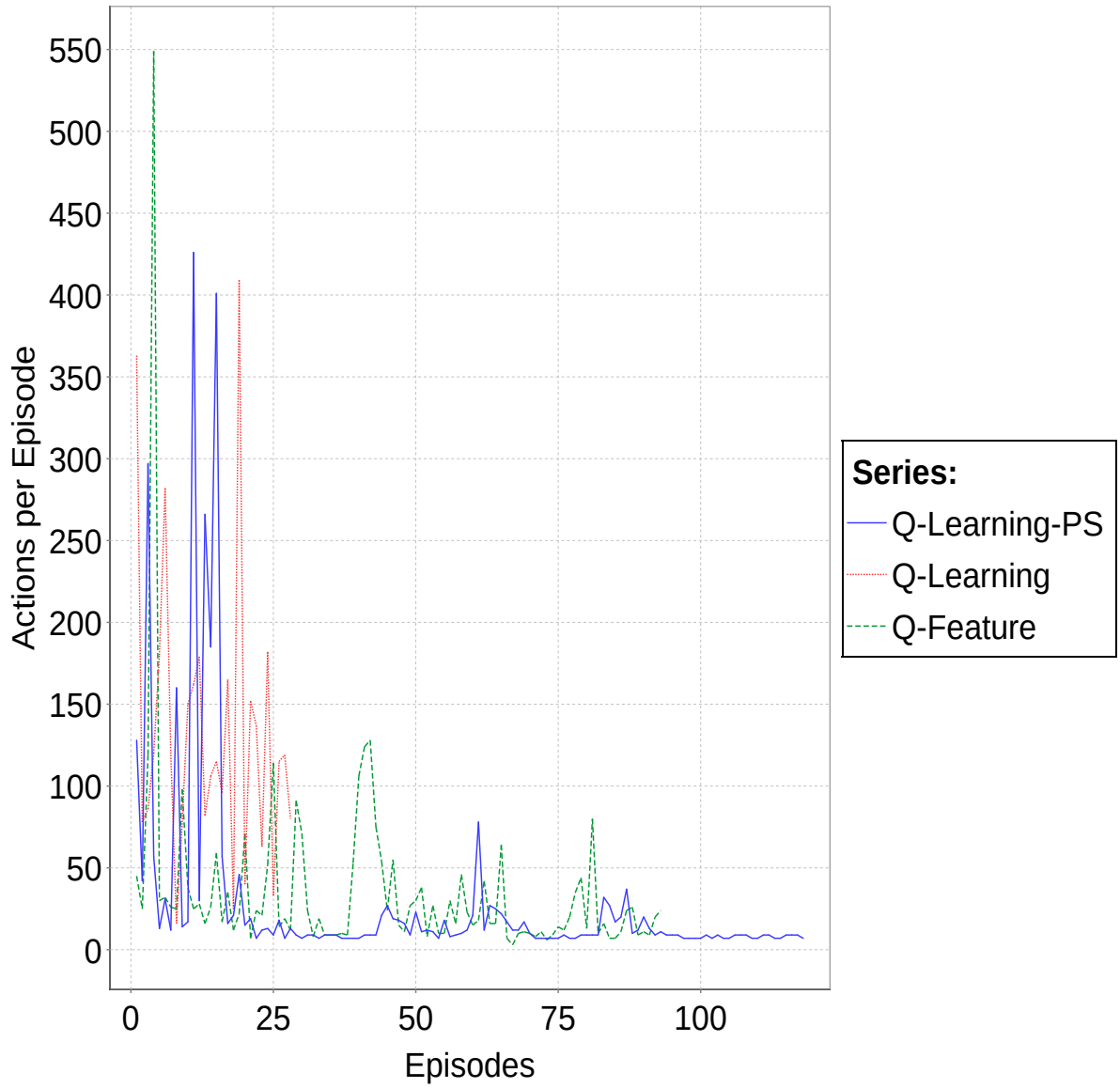


Abbildung 59: Lernverläufe der vierten Versuchsreihe

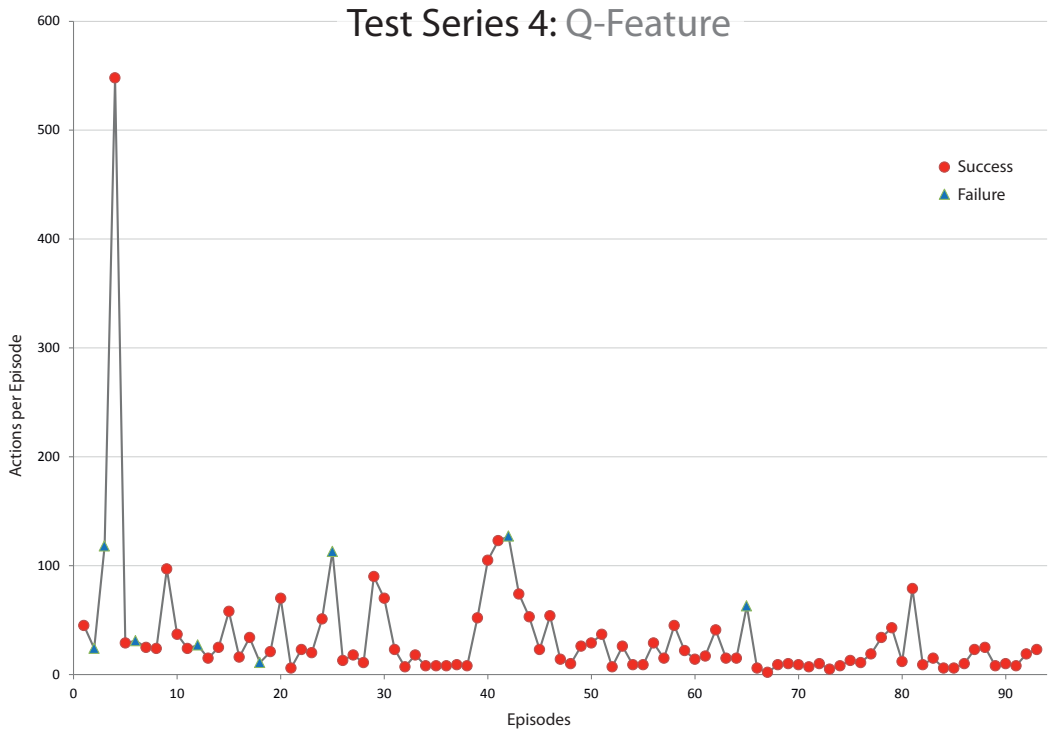


Abbildung 60: Lernverlauf von Q-Feature in der vierten Versuchsreihe

Abbildung 61: Lernverlauf von Q-Learning in der vierten Versuchsreihe

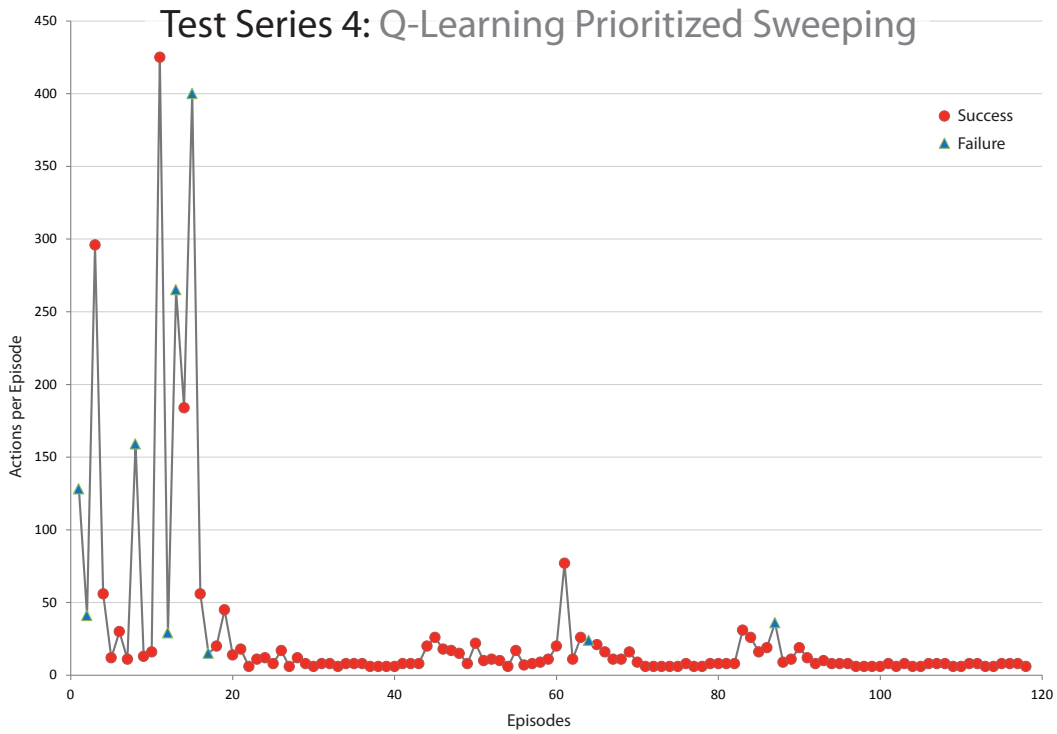


Abbildung 62: Lernverlauf von Q-Learning mit Prioritized Sweeping in der vierten Versuchsreihe

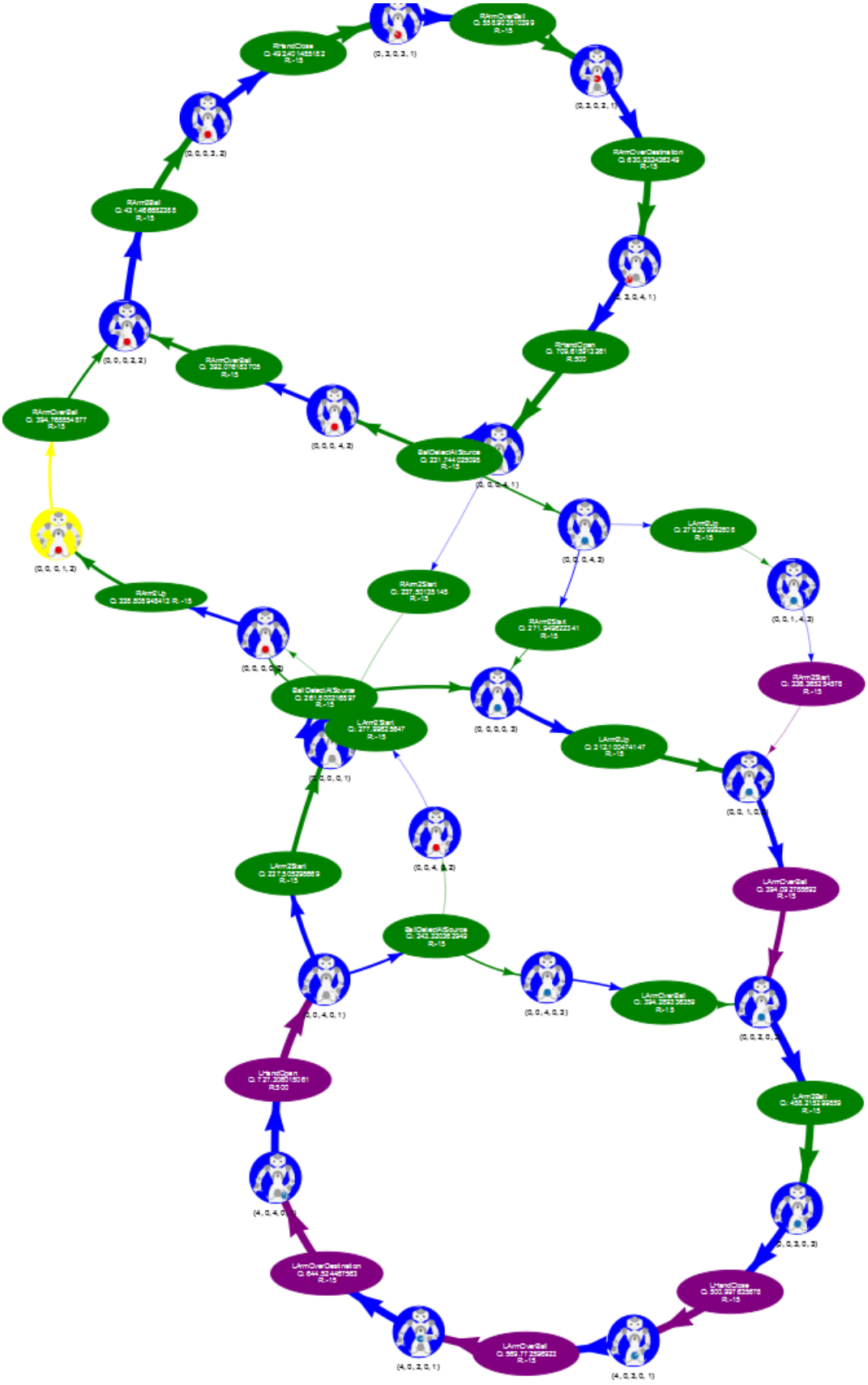


Abbildung 63: Vollständiger Graph von Q-Learning mit PS

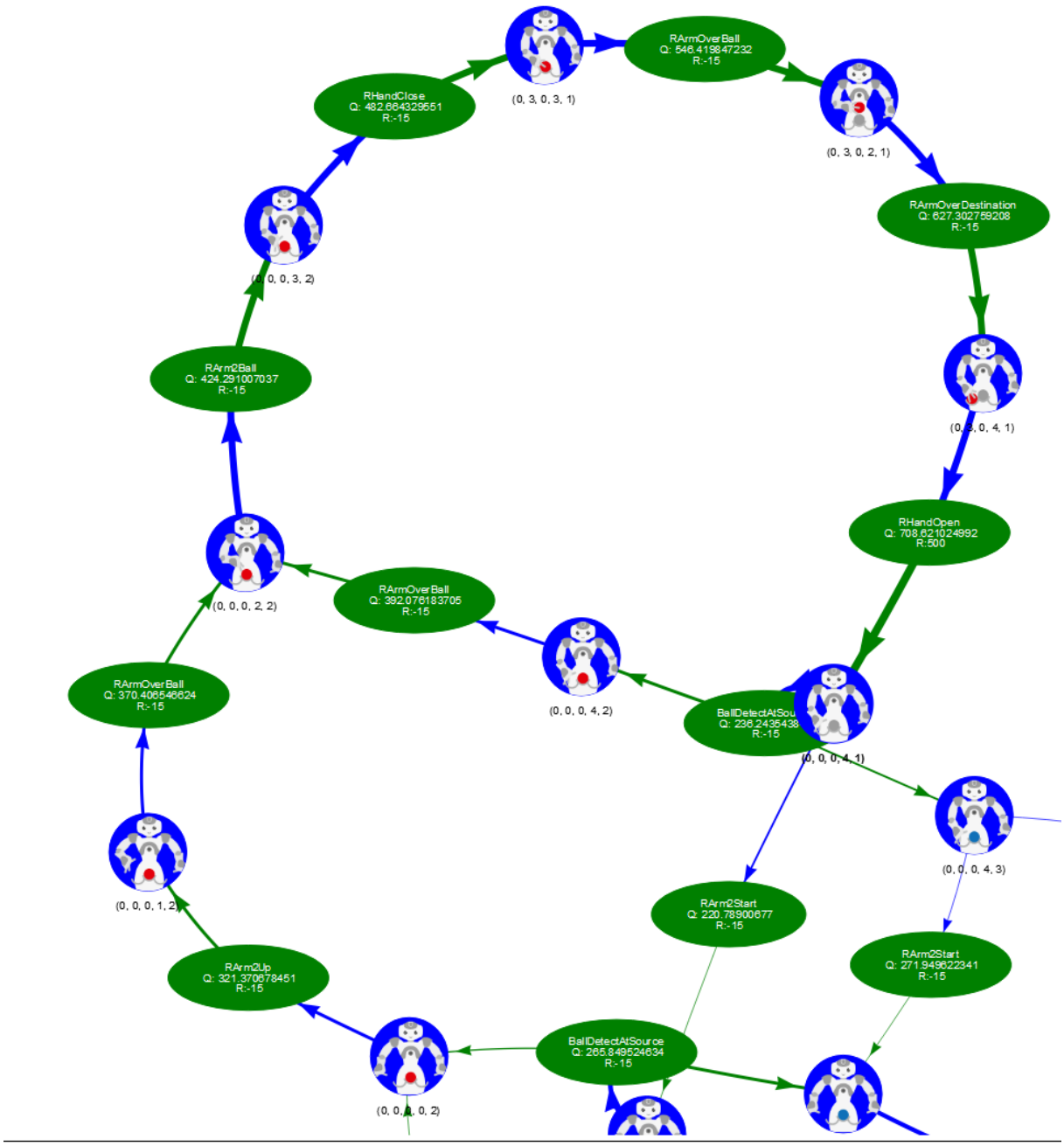


Abbildung 64: Oberer Graph von Q-Learning mit PS, roter Ball Zyklus

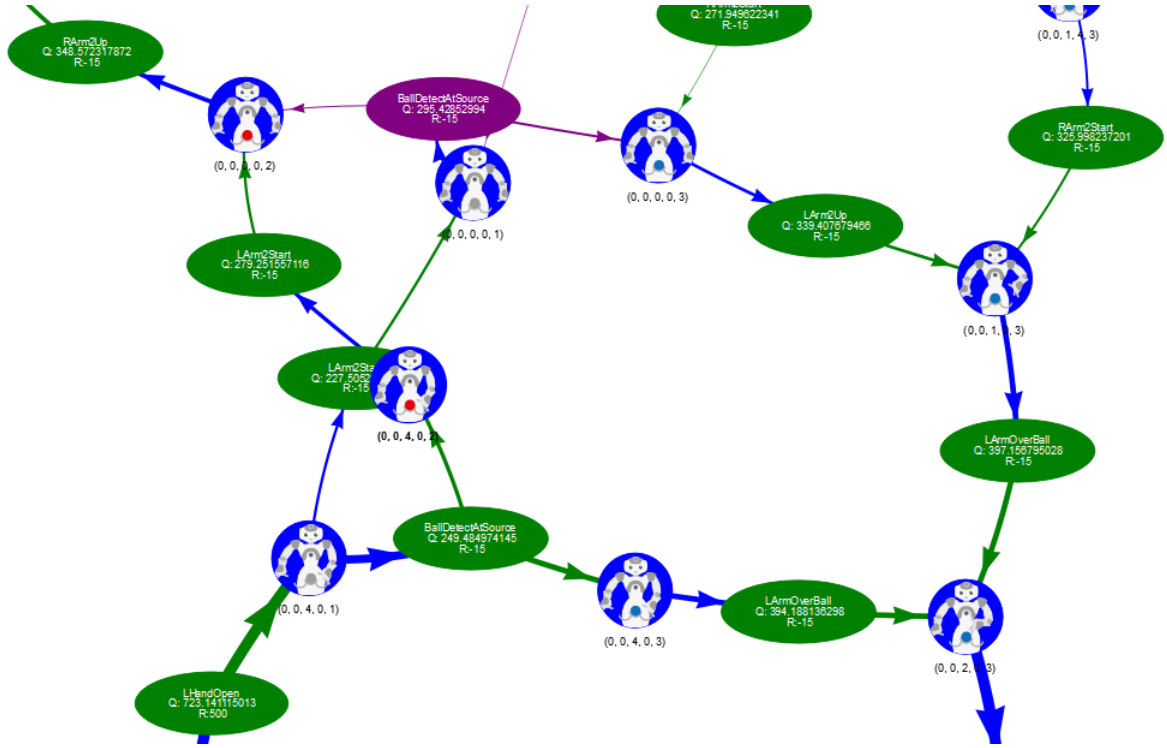


Abbildung 65: Mittlerer Graph von Q-Learning mit PS, Übergang zwischen rotem und blauem Ball

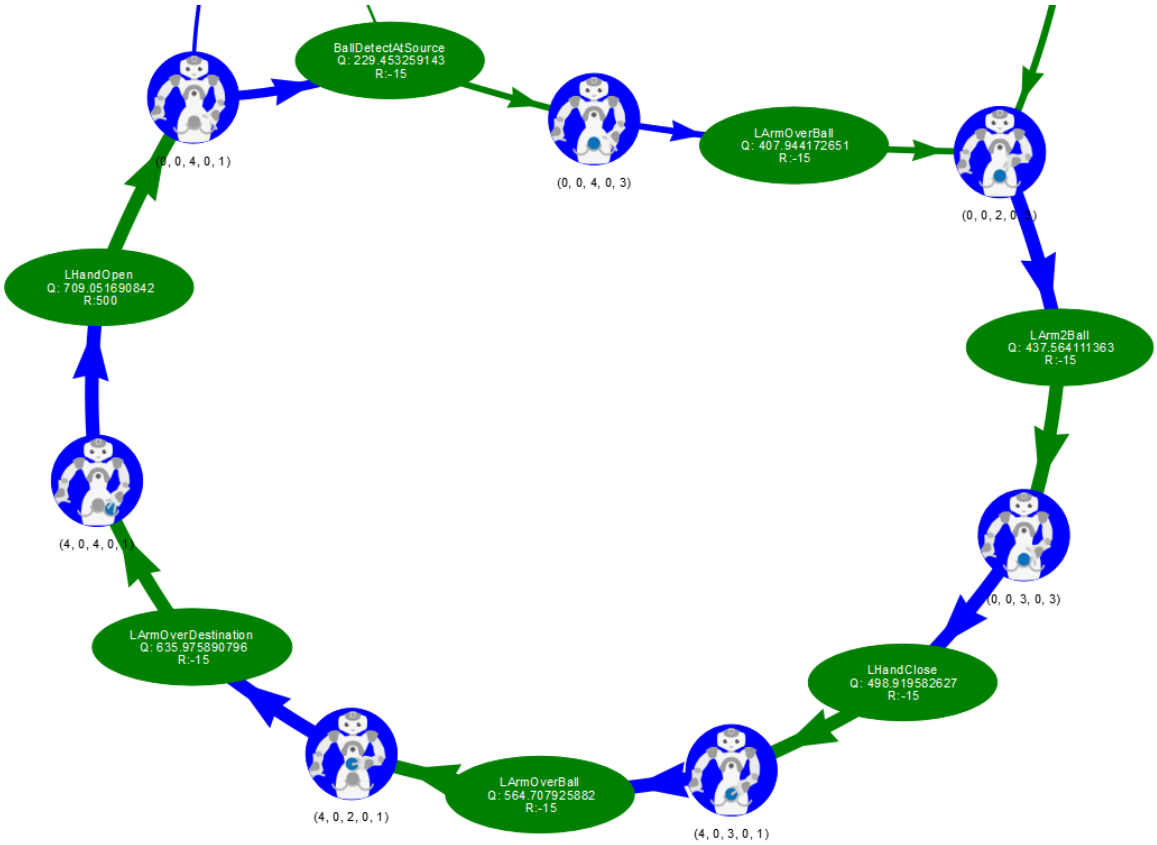


Abbildung 66: Unterer Graph von Q-Learning mit PS, blauer Ball Zyklus