ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Neuroinformatik
Fachgruppe Neurobotik

Adaptive Dynamic Programming mit Reservoir Computing zur interaktiven multikriteriellen Optimierung

Ein Ansatz für intrinsisch und extrinsisch motiviertes Lernen

Masterarbeit an der Universität Ulm

Vorgelegt von:
Christian Fischer
christian-1.fischer@uni-ulm.de

Gutachter:
Prof. Dr. Günther Palm
Dr. Mohamed Oubbati

Betreuer:
Dr. Mohamed Oubbati

2013

„Adaptive Dynamic Programming mit Reservoir Computing zur interaktiven multikriteriellen Optimierung“
Fassung vom 4. November 2013

© 2013 Christian Fischer

Adaptive Dynamic Programming mit Reservoir Computing zur interaktiven multikriteriellen Optimierung

Christian Fischer
christian-1.fischer@uni-ulm.de

Masterarbeit im Fach Informatik
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Neuroinformatik
Fachgruppe Neurobotik
Universität Ulm

4. November 2013

Zusammenfassung

In dieser Arbeit wird *Adaptive Dynamic Programming* mit einer auf *Echo State Networks* basierenden Kritik, zur Steuerung eines autonomen Agenten innerhalb einer kontinuierlichen Umgebung, untersucht. Der Agent approximiert mit der Kritik mehrere Kostenfunktionen und nutzt die Skalarisierung, um gleichzeitig mehrere Ziele zu optimieren. Über eine Entscheidungslogik wird, abhängig vom inneren Zustand des Agenten, eine optimale Zielgewichtung bestimmt. Der innere Zustand wird mit Trieben modelliert und die Entscheidungslogik nutzt *Q-learning*, um den Nutzen einer Zielgewichtung im gegebenen Zustand zu ermitteln.

Schlüsselwörter: Echo State Networks, Multiobjective Reinforcement Learning, Adaptive Dynamic Programming, Q-Learning, Decision Making, Goal Directed Agent, Intrinsic & Extrinsic Motivation

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielstellung 2

1.2 Aufbau der Arbeit 3

2 Grundlagen **5**

2.1 Maschinelles Lernen 5

2.1.1 Überwachtes Lernen 6

2.1.2 Unüberwachtes Lernen 6

2.1.3 Bestärkendes Lernen 7

2.2 Dynamisches Programmieren 8

2.2.1 Adaptive Dynamic Programming 9

2.3 Künstliche neuronale Netze 10

2.3.1 Das Perzeptron 11

2.3.2 Netzarchitekturen 12

2.4 Echo State Networks 13

2.4.1 Erzeugung eines ESN 14

2.4.2 Training eines ESN 17

2.5 Die RC-ADP-Architektur 19

2.5.1 Kritik 19

2.5.2 Aktor 20

3 Umsetzung **23**

3.1 Szenario 23

3.2 Bisherige Umsetzung 25

3.3 Neue Konzeption des Agenten 26
3.3.1 Architektur des Agenten 29
3.3.2 Kritik 30
3.3.3 Aktor 32

4 Multikriterielle Optimierung 35
4.1 Multikriterielle Optimierungsprobleme 35
4.1.1 Pareto-Optimierung 36
4.1.2 Skalarisierung 37
4.2 Anpassung der RC-ADP-Architektur an MOPs 38
4.2.1 Anpassung der Kritik 40

5 Entscheidungsfindung 43
5.1 Entscheidungsfindung auf Basis des inneren Zustands 43
5.1.1 Triebe als innerer Zustand 44
5.1.2 Bewertung des inneren Zustands 47
5.2 Finden optimaler Entscheidungen 48
5.2.1 Q-learning 48
5.2.2 Training der Entscheidungslogik 51

6 Ergebnisse und Diskussion 53
6.1 Versuchsdurchführung 53
6.2 Szenario 1 54
6.2.1 Versuch 1 55
6.2.2 Versuch 2 67
6.2.3 Versuch 3 71
6.2.4 Versuch 4 76
6.3 Szenario 2 80
6.3.1 Versuch 1 84
6.3.2 Versuch 2 88
6.3.3 Versuch 3 91
6.3.4 Versuch 4 93

7 Schlussbemerkungen **97**

7.1 Ausblick 99

Abbildungsverzeichnis **101**

Abkürzungsverzeichnis **103**

Literaturverzeichnis **105**

1 Einleitung

Autonomie bedeutet Selbständigkeit, Selbstbestimmung und Entscheidungsfreiheit. Ein autonomer Agent zeichnet sich durch diese Attribute aus. Er ist in der Lage, selbständig in seiner Umgebung zu agieren. Dazu muss er lernen, welche Konsequenzen seine Entscheidungen haben. Bestärkendes Lernen bietet eine Möglichkeit, um den Nutzen einer Aktion - abhängig vom gegebenen Zustand - zu bestimmen. Tritt zu einem späteren Zeitpunkt derselbe Zustand erneut ein, so kann der Agent auf seine Erfahrung zurückgreifen und bessere Entscheidungen treffen.

Ein kontinuierlicher Zustands- und Aktionsraum führt beim bestärkenden Lernen zu einem Komplexitätsproblem (*curse of dimensionality*). *Adaptive dynamic programming* (ADP) widmet sich dieser Problematik, indem der Nutzen einer Aktion von der sogenannten Kritik approximiert wird. Die Kritik wird üblicherweise mit einem künstlichen neuronalen Netz (KNN) realisiert. Rückgekoppelte neuronale Netze (RNNs) ermöglichen der Kritik, auch dynamische Probleme zu bewältigen. Der Agent und die Umgebung bilden in der Praxis häufig ein dynamisches System. Aus diesem Grund wurde vor wenigen Jahren die Kombination von ADP und *Reservoir Computing* (RC) untersucht [Koprinkova-Hristova et al., 2010, Koprinkova-Hristova et al., 2012, Oubbati et al., 2011, Oubbati et al., 2012]. RC erleichtert die praktische Anwendung rückgekoppelter neuronaler Netze. Die gesamte Dynamik befindet sich im Reservoir. Das Reservoir bildet die Netzeingaben auf einen höherdimensionalen Zustand ab, welcher von der Ausgabeschicht interpretiert wird. Es genügt die Ausgabeschicht zu trainieren um eine korrekte Abbildung von Reservoirzuständen auf Sollausgaben zu finden. Die Einfachheit und Mächtigkeit des RCs stellt in Kombination mit ADP ein vielversprechendes Werkzeug bereit. Bisher beziehen sich die Untersuchungen zu dieser Kombination auf das Lösen einfacher Optimierungsprobleme, bei denen der Agent nur ein Ziel verfolgt.

Viele Anwendungen erfordern jedoch die Optimierung mehrerer Ziele, die möglicherweise in Konflikt zueinander stehen. D.h. der Agent kann nicht alle Ziele gleichzeitig optimieren. Um diesen Konflikt zu lösen, ist der Agent gezwungen Kompromisse einzugehen. Es handelt sich um ein multikriterielles Optimierungsproblem (MOP). Die Lösung eines MOPs lässt sich als Vektor auffassen, dessen Elemente die Qualität der Lösung für jedes einzelne Ziel beschreibt. Der Agent identifiziert diejenigen Lösungen, die den besten Kompromiss darstellen. Diese Lösungen bilden die Pareto-Menge.

Die Pareto-Optimierung ist ein a posteriori Verfahren. Der Entscheidungsträger wählt im Nachhinein die für ihn beste Lösung unter allen pareto-optimalen Lösungen aus. Im Gegensatz dazu stehen a priori Verfahren, bei denen bereits im Vorfeld Wissen benötigt wird - wie z.B. die Prioritäten der einzelnen Ziele. In Zusammenhang mit bestärkendem Lernen sind a priori Verfahren geläufiger, da sie das MOP in ein Optimierungsproblem mit nur einem Ziel transformieren und anschließend auf herkömmliche Weise lösen.

1.1 Zielstellung

Das Ziel dieser Arbeit ist, die RC-ADP-Architektur [Koprinkova-Hristova et al., 2010, Koprinkova-Hristova et al., 2012, Oubbati et al., 2011, Oubbati et al., 2012] an multikriterielle Optimierungsprobleme anzupassen. Dies soll mit der Skalarisierung bzw. der Zielgewichtung erfolgen. Dazu werden dem Agent mehrere Zielgewichtungen vorgegeben, die die einzelnen Ziele unterschiedlich priorisieren. Abhängig von der Zielgewichtung bestimmt der Agent die nachfolgend auszuführenden Aktionen.

Darüberhinaus soll die Architektur um eine Entscheidungskomponente erweitert werden. Mit der Entscheidungskomponente wählt der Agent selbständig auf Basis seines inneren Zustands eine der vorgegebenen Zielgewichtungen aus. Der innere Zustand wird mit mehreren Trieben (z.B. Hunger und Durst) modelliert. Mit Hilfe von *Q-Learning* [Watkins and Dayan, 1992] soll der Agent lernen, welche Zielgewichtung einen bestimmten Trieb befriedigt oder weiter verstärkt. Außerdem soll er lernen, welche der Zielgewichtungen

im gegebenen Zustand den größten Nutzen verspricht, d.h. sein Wohlbefinden auf lange Sicht am stärksten optimiert.

1.2 Aufbau der Arbeit

Die Arbeit ist in sieben Kapitel gegliedert. Die Inhalte der einzelnen Kapitel werden nachfolgend zusammengefasst:

Kapitel 1 bietet eine kurze Übersicht, sowie die grundlegende Motivation hinter dieser Arbeit. Außerdem werden die Ziele genannt, die erreicht werden sollen.

Kapitel 2 stellt die relevanten Grundlagen kurz vor. Von zentraler Bedeutung sind dabei rückgekoppelte neuronale Netze, bestärkendes Lernen und dynamisches Programmieren

Kapitel 3 beschreibt das Szenario für diese Arbeit und den grundlegenden Aufbau des Agenten. Dies stellt den Ausgangspunkt für die Erweiterungen der RC-ADP-Architektur dar.

Kapitel 4 gibt eine Einführung zu multikriterieller Optimierung und zeigt die notwendigen Anpassungen des Agenten bzw. der RC-ADP-Architektur.

Kapitel 5 widmet sich der Entscheidungsfindung. Dort wird beschrieben, wie der Agent anhand seines inneren Zustands lernt optimale Aktionen auszuführen.

Kapitel 6 untersucht das Verhalten des Agenten anhand mehrerer Versuche. Die Ergebnisse werden zudem diskutiert.

Kapitel 7 fasst die Arbeit kurz zusammen und wiegt die Stärken und Schwächen dieses Ansatzes ab.

2 Grundlagen

In diesem Kapitel werden die für diese Arbeit relevanten Grundlagen vorgestellt. Zunächst wird das maschinelle Lernen dargestellt, sowie wichtige Begriffe und Methoden eingeführt. Von zentraler Bedeutung sind dabei rückgekoppelte neuronale Netze, bestärkendes Lernen und dynamisches Programmieren. Dies bietet eine Basis, um die Architektur und die Implementierung des Agenten besser nachvollziehen zu können. Dabei soll keine vollständige und umfassende Diskussion der einzelnen Themen erfolgen, sondern lediglich ein Überblick über die wichtigsten Aspekte gegeben werden.

2.1 Maschinelles Lernen

Allgemein versteht man unter Lernen den Erwerb neuer Fertigkeiten und Kenntnisse. Dabei können unterschiedlich komplexe Sachverhalte gelernt werden, wie einfache Tatsachen, motorische Abläufe und abstrakte Denkvorgänge.

Zum Beispiel muss der mobile Roboter, welcher in dieser Arbeit implementiert wird, lernen sich in seiner Umgebung zurecht zu finden. Darüber hinaus muss er verschiedene Aufgaben erfüllen. Über Sensoren kann er seine Umgebung wahrnehmen und seine aktuelle Situation einschätzen. Abhängig von seiner Situation kann er verschiedene Aktionen ausführen und damit auf seine Umwelt einwirken. Dabei lernt der Agent, welche Aktion in einer bestimmten Situation am vielversprechendsten ist.

Allgemein soll beim maschinellen Lernen ein Agent selbständig aus Beispielen lernen. Dabei steht nicht das exakte Auswendiglernen der Daten im Vordergrund, sondern es

müssen Zusammenhänge und Gesetzmäßigkeiten in den Beispielen erkannt werden. Ziel ist es, das gelernte Wissen so zu verallgemeinern, dass es vom Agenten auch in neuen, bisher unbekannten Situationen nutzbar ist.

Im Wesentlichen existieren drei Arten des maschinellen Lernens - das überwachte Lernen, das unüberwachte Lernen und bestärkendes Lernen. Im Folgenden werden die drei Arten kurz vorgestellt.

2.1.1 Überwachtes Lernen

Das überwachte Lernen erfolgt in zwei Phasen. In der ersten Phase - genannt Trainingsphase - werden dem Agenten Paare von Ein- und zugehörigen Ausgabedaten präsentiert. Über Änderungen seiner inneren Struktur versucht der Agent eine möglichst gute Abbildungsfunktion von Ein- auf Ausgaben zu finden. Anschließend wird das Gelernte in einer zweiten Phase auf neue Beispiele angewendet. Dabei lässt sich der Erfolg des Agenten über den Vergleich seiner Ausgaben mit den Sollausgaben messen. Der Lernerfolg ist stark von den Trainingsdaten abhängig. Zum einen müssen sie Gesetzmäßigkeiten enthalten, die der Agent erkennen kann und zum anderen sollten sie einen ausreichend großen Bereich des Problems umfassen. In der Praxis ist es oft schwierig, einen guten Satz von Trainingsdaten zu finden.

2.1.2 Unüberwachtes Lernen

Beim unüberwachten Lernen gibt es keine Paare von Ein- und Ausgabedaten, mit denen der Agent trainiert wird. Vielmehr muss dieser selbständig Zusammenhänge und Muster in vorhandenen Daten erkennen. Dies kann beispielsweise in der Vorverarbeitung größerer Datensätze genutzt werden, um ähnliche Daten automatisch zu erkennen und zusammenzufassen. Auf diese Weise lässt sich der Datensatz auf wenige repräsentative Prototypen reduzieren und somit einfacher verarbeiten.

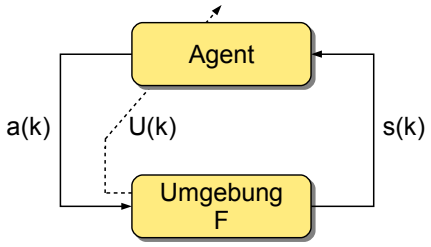


Abbildung 2.1: **Bestärkendes Lernen.** Der Agent nimmt den Weltzustands $s(k)$ wahr und führt daraufhin eine Aktion $a(k)$ aus. Es ergibt sich ein neuer Weltzustand $F(s(k), a(k))$. Ist der Folgezustand bekannt, erhält der Agent eine Bewertung $U(k)$ seiner Aktion $a(k)$ im Zustand $s(k)$.

2.1.3 Bestärkendes Lernen

Im Gegensatz zum überwachten Lernen handelt es sich beim bestärkenden Lernen um Lernen aus Erfahrung, anstatt Lernen aus Beispielen [Sutton and Barto, 1998]. Um Erfahrungen sammeln zu können, handelt der Agent nach dem Versuch-und-Irrtum-Prinzip. Er testet in jeder Situation unterschiedliche Aktionen, um anschließend deren Nutzen beurteilen zu können. Gerät der Agent zu einem späteren Zeitpunkt erneut in diese Situation, kann er auf seine früheren Erfahrungen zurückgreifen und dadurch bessere Entscheidungen treffen.

Abbildung 2.1 veranschaulicht das bestärkende Lernen. Der Agent erhält eine Repräsentation des Weltzustands $s(k)$ zum Zeitpunkt k und führt daraufhin eine Aktion $a(k)$ aus. Dadurch ergibt sich ein neuer Zustand $s(k+1)$:

$$s(k+1) = F(s(k), a(k)) \tag{2.1}$$

Der Folgezustand $s(k+1)$ wird durch die Systemfunktion F, den Zustand $s(k)$ und die Aktion $a(k)$ bestimmt, wobei in dieser Arbeit von einem dynamischen System ausgegangen wird. Jedem Zustand ist ein Bewertung zugeordnet. Ist der Folgezustand bekannt, erhält der Agent die Bewertung $U(k)$ mit der er seine Entscheidung beurteilen kann. Dadurch bereichert sich sein Wissen.

Bei der Aktionsauswahl kann sich der Agent entweder auf seine bisher gewonnenen Erfahrungen stützen, oder zufällig neue Aktionen austesten. Greift er auf seine Erfahrung zurück, erzielt er zunächst meist bessere Ergebnisse. Allerdings kann er sein Wissen nur erweitern, wenn er auch Aktionen wählt, deren Nutzen noch nicht bekannt ist. Dieser Konflikt wird als *exploration–exploitation dilemma* bezeichnet [Sutton and Barto, 1998].

Das Ziel des Agenten besteht darin Aktionen auszuwählen, die zu Zuständen führen, die den größtmöglichen Nutzen versprechen. Dabei verfolgt der Agent oftmals langfristige Ziele. Das bedeutet, dass es manchmal besser ist Aktionen auszuwählen die zunächst weniger vorteilhaft sind, aber von deren Folgezuständen aus ein besseres Endresultat erreichbar ist. Folglich sollte der Agent neben der direkten Bewertung auch künftige Bewertungen in seine Entscheidung mit einbeziehen. Die Methode des dynamischen Programmierens ermöglicht dem Agent eine Strategie zu finden, die langfristig die Summe aller Bewertungen maximiert.

2.2 Dynamisches Programmieren

Der Begriff dynamisches Programmieren wurde von Bellman [Bellman, 1957] eingeführt und steht für eine Sammlung von Methoden zur Lösung von Optimierungsproblemen. Der Grundgedanke zum dynamischen Programmieren beruht auf zwei Prinzipien [Harmon and Harmon, 1997]. Das erste Prinzip besagt, dass jede Aktion die zu einem schlechteren Zustand führt, als schlecht eingestuft werden sollte. Das zweite Prinzip wiederum besagt, dass jeder Zustand von dem aus nur schlechte Aktionen möglich sind, ebenfalls schlecht sein muss. Auf diese Weise lassen sich Zustände, für die der Agent per se keine schlechte Bewertung erhält, im Nachhinein korrekt klassifizieren. Mathematisch wird dieser Sachverhalt über die Bellman-Gleichung dargestellt:

$$J(k) = U(k) + \gamma \cdot J(k+1) \tag{2.2}$$

Dabei bezeichnet U die Bewertungsfunktion und J die Vorhersage bzw. Kostenfunktion (engl. *cost-to-go function*). Diese setzt sich aus der aktuellen Bewertung $U(k)$ und allen zukünftigen Bewertungen $J(k+1)$, die der Agent vom Folgezustand aus im besten Fall erreichen kann, zusammen. $J(k+1)$ steht für diejenigen Bewertungen, die der Agent erhält, sofern er vom Folgezustand aus stets optimale Aktionen wählt. Um die Vorhersage J zu ermitteln, muss der Agent eine optimale Strategie (engl. *policy*) finden, anhand derer er die Aktionen wählt.

Der *discount*-Faktor $\gamma \in [0, 1]$ gibt an, wie stark die künftigen Bewertungen gewichtet werden. Je kleiner der Wert, desto geringer ist ihr Einfluss auf die Vorhersage. Gleichung 2.2 lässt sich auch nicht-rekursiv darstellen:

$$J(k) = \sum_{i=k}^{\infty} \gamma^{i-k} \cdot U(i) \tag{2.3}$$

In dieser Darstellung ist deutlicher zu erkennen, wie die Vorhersage J alle zu erwartenden Bewertungen vereint. Außerdem ist zu sehen, dass die Einzelbewertungen - bedingt durch den *discount*-Faktor γ - weniger stark gewichtet werden, je weiter sie in die Zukunft reichen.

Ist die Vorhersage J bekannt, so weiß der Agent welche Aktion auf lange Sicht am vielversprechendsten ist. Die Schwierigkeit besteht jedoch darin, die Bellman-Gleichung zu lösen. Sie setzt per Definition bereits eine optimale Wahl von Aktionen - d.h. eine optimale Strategie - voraus.

2.2.1 Adaptive Dynamic Programming

Dynamisches Programmieren führt bei Problemen mit großem Zustandsraum oftmals zu einem Effizienz-Problem. Jede Aktion des Agenten beeinflusst den künftigen Zustand und oftmals zeigt sich erst nach zahlreichen Entscheidungen zu welchem Ergebnis die Aktionsfolge führt. Die Bestimmung der Vorhersage J ist mit enormen Aufwand verbunden [Vrabie and Lewis, 2009].

Aus diesem Grund wurde in [Prokhorov et al., 1997] eine Architektur vorgestellt, die den Agenten in zwei Komponenten unterteilt - die Kritik und den Aktor. Der Aktor ist für die Wahl der auszuführenden Aktion verantwortlich und stützt sich dabei auf die bisher gewonnenen Erfahrungen. Die Erfahrungen wiederum werden von der Kritik gesammelt. Daher arbeiten Aktor und Kritik eng zusammen. Während der Aktor direkten Zugriff auf die Ausgaben der Kritik hat, kann diese mit Hilfe der Bewertungen die vom Aktor getroffenen Entscheidungen beurteilen. Die Kritik nutzt die Bewertungen, um eine möglichst gute Approximation $\hat{J}(k)$ der Kostenfunktion zu finden:

$$\hat{J}(k) = U(k) + \gamma \cdot \hat{J}(k+1) \tag{2.4}$$

Zu Beginn verfügt die Approximation über keinerlei Vorhersagekraft und ist maßgeblich durch die Bewertungsfunktion U geprägt. Nachdem der Agent eine Aktion ausgeführt hat und der Folgezustand bekannt ist, wird die Approximation angepasst. Da die Anpassungen schrittweise erfolgen, verbessert sich die Approximation stetig.

Diese Architektur bezeichnet man als *Adaptive Critic Design* (ACD). Später haben sich zusätzlich Begriffe wie *Adaptive Dynamic Programming* (ADP) und *Approximate Dynamic Programming* etabliert [Wang et al., 2009]. In Abschnitt 2.5 wird eine konkrete Umsetzung von ADP anschaulich und detailiert beschrieben. Bei dieser Umsetzung wird die Kritik mittels eines künstlichen neuronalen Netzes (KNN) realisiert. Diese eignen sich zur Approximation kontinuierlicher Funktionen, weshalb sie im nächsten Abschnitt vorgestellt werden.

2.3 Künstliche neuronale Netze

Oftmals dient die Natur als Inspiration für technische Lösungen. Unser Gehirn ist in der Lage vielfältige Probleme zu bewältigen, die algorithmisch schwer umsetzbar sind. Daher versucht man die Architektur des Gehirns in einfacher Form nachzubilden. Künstliche neuronale Netze stellen dabei ein mathematisches Modell unseres Gehirns dar. Sie sind aus einzelnen Neuronen aufgebaut, die untereinander verbunden sind.

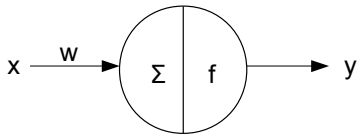


Abbildung 2.2: **Das Perzeptron.** Der Eingabevektor x wird gewichtet aufsummiert und mittels der Übertragungsfunktion f auf die Ausgabe y abgebildet. Die Klassifikation bzw. Approximation erfolgt über eine Adaption des Gewichtsvektors w.

2.3.1 Das Perzeptron

Das einfache Perzeptron modelliert die Funktion eines einzelnen Neurons innerhalb eines künstlichen neuronalen Netzes. Es beschreibt eine Abbildung von einem Eingabevektor x auf eine Ausgabe y. Abbildung 2.2 zeigt den schematischen Aufbau des Perzeptrons.

Die Eingabe x wird gewichtet aufsummiert und mittels der Übertragungsfunktion f auf die Ausgabe y abgebildet:

$$y = f\left(\sum_{i=1}^{n} x_i \cdot w_i\right) \tag{2.5}$$

Als Übertragungsfunktion wird häufig die lineare Funktion $f(x) = x$ und die nichtlineare Sigmoidfunktion (Tangens Hyperbolicus) $f(x) = tanh(x)$ verwendet. Sie beeinflussen maßgeblich das Verhalten des Neurons.

Durch schrittweises Adaptieren des Gewichtsvektors w kann das Perzeptron trainiert werden. Dazu wird der Fehler zwischen Ist- und Sollausgabe berechnet. Anschließend lassen sich anhand des Fehlervektors die einzelnen Gewichte in- bzw. dekrementieren, sodass der Fehler für das betrachtete Beispiel abnimmt.

Mit einfachen Perzeptronen lassen sich nur linear separierbare Probleme lösen [Minsky and Papert, 1969]. Der Anteil linear separierbarer Funktionen nimmt mit wachsender Dimensionalität ab. Deshalb werden viele Neuronen zu einem Netz verschaltet. Das so gewonnene KNN kann eine Vielzahl mächtiger Funktionen beliebig genau approximieren [Hornik, 1991].

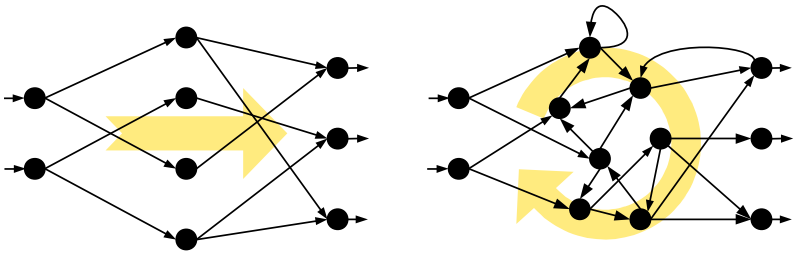


Abbildung 2.3: **Netzarchitekturen (nach [Jaeger, 2002]).** Links ist ein einfaches vorwärtsgerichtetes Netz (FFN) und rechts ein rückgekoppeltes Netz (RNN) dargestellt.

2.3.2 Netzarchitekturen

Werden die Neuronen in Schichten organisiert, so spricht man von einem mehrschichtigen oder mehrlagigen Netz. Hierbei handelt es sich um die einfachste Form von vorwärtsgerichteten Netzen (engl. *feed forward net* FFN). Die Ausgaben einer Schicht werden mit den Eingaben der folgenden Schicht verknüpft. Prinzipiell gehört jedoch jedes zyklenfreie Netz zur Gruppe der FFNs.

Dem gegenüber stehen die rückgekoppelten Neuronalen Netze (engl. *recurrent neural network* RNN). Bei diesen Netzen dürfen die Neuronen beliebig verschaltet sein und somit auch Zyklen enthalten. Abbildung 2.3 zeigt den Unterschied von FFNs und RNNs.

Der Vorteil von RNNs liegt darin, dass die Netzausgabe nicht nur durch die aktuelle Eingabe bestimmt ist, sondern auch von vorhergehenden Eingaben abhängig sein kann. Durch die Rückkopplungen wird der Effekt vorhergehender Eingaben im Netzwerk zumindest temporär erhalten. Dadurch lassen sich dynamische Funktionen approximieren.

Während für FFNs sehr effiziente Trainingsalgorithmen existieren, gestaltet sich das Training rekurrenter Netze erheblich schwieriger. Klassischerweise werden mehrlagige FFNs mittels *error backpropagation* [Rumelhart et al., 1986] trainiert. Dieses Verfahren stellt bis heute einen der wichtigsten Algorithmen zum Training neuronaler Netze bereit.

Er beruht auf einem Gradientenabstieg auf der Fehlerfläche und schrittweiser Adaption der Gewichte. Basierend auf *error backpropagation* wurden Methoden entwickelt, die durch dynamische Lernraten das Training weiter beschleunigen [Fahlman, 1988, Riedmiller and Braun, 1993]. Außerdem reduzieren sie die Wahrscheinlichkeit gegen ein lokales Optimum zu konvergieren, was ein häufiges Problem beim Gradientenabstieg ist.

Gängige Trainingsalgorithmen für RNNs, wie beispielsweise *Back Propagation Through Time* (BPTT) [Werbos, 1990], garantieren hingegen keinerlei Konvergenz. Falls sie überhaupt konvergieren, dann meistens nur recht langsam. Zudem benötigen sie deutlich mehr Rechenleistung und zeigen oftmals instabiles Verhalten [Jaeger, 2002].

2.4 Echo State Networks

Obwohl rekurrente neuronalen Netze aufgrund ihrer Architektur vielseitig einsetzbar sind und dynamisches Verhalten nachbilden können, erschweren die rechenintensiven Algorithmen ihren praktischen Nutzen. Darüber hinaus lässt sich ihr Verhalten schwer nachvollziehen und ein erfolgreiches Training stellt eine große Herausforderung dar. Aus diesen Gründen führten RNNs lange Zeit ein Schattendasein. Doch seit einigen Jahren stehen RNNs wieder verstärkt im Fokus der Forschungsgemeinde. Der Grund dafür liegt im Paradigma des *Reservoir Computing* (RC).

Etwa zur selben Zeit wurden zwei Methoden entwickelt die diesem Ansatz folgen. Auf der einen Seite *Echo State Networks* (ESNs) [Jaeger, 2001] und auf der anderen Seite *Liquid State Machines* (LSM) [Maass et al., 2002]. Während LSMs versuchen mit *spikenden* Neuronen neuronale Netze biologisch möglichst exakt nachzubilden, bieten ESNs einen einfacher zu verstehenden und weniger rechenintensiven Ansatz, was sie für viele praktische Probleme interessanter macht. Den allgemeinen Aufbau des KNN hingegen teilen sich die beiden Methoden. Sie gliedern das neuronale Netz in drei Teile: eine Eingabeschicht, eine Ausgabeschicht und dazwischen das Reservoir.

Die Eingabeschicht leitet die Eingaben zum Reservoir [1]. Für die Dynamik des Netzes ist einzig das Reservoir verantwortlich. Dieses beinhaltet eine beliebig große Anzahl von Neuronen ohne feste Strukturierung. Die Neuronen können innerhalb des Reservoirs beliebig miteinander verschaltet sein und dadurch Rückkopplungen enthalten. Die Idee dabei ist, jeden Eingabevektor auf einen höherdimensionalen Reservoirzustand abzubilden. Der Reservoirzustand wiederum hängt nicht nur von der aktuellen Eingabe, sondern auch von vorhergehenden Eingabevektoren ab. Um die gewünschte Ausgabe zu approximieren, muss der aktuelle Reservoirzustand interpretiert werden. Dies geschieht über die Ausgabeschicht. Sie bildet die Reservoirzustände auf die Sollausgaben ab. Bei ESNs besteht die Ausgabeschicht in der Regel aus exakt einer Schicht von Neuronen. Prinzipiell kann der Reservoirzustand aber auch von einem mehrlagigen Netz interpretiert werden.

Einmal initialisiert, wird das Reservoir während des Trainings nicht weiter verändert. Lediglich die Ausgabeschicht muss trainiert werden. Dies ist einer der größten Vorteile des RC. Andere Methoden, wie beispielsweise BPTT, trainieren nicht nur die Neuronen der Ausgabeschicht, sondern alle Neuronen im Netz. Dadurch entsteht ein deutlicher Mehraufwand gegenüber RC, welcher sich mit zunehmender Anzahl von Neuronen weiter verstärkt [Lukoševičius et al., 2012].

2.4.1 Erzeugung eines ESN

In Abbildung 2.4 ist der detailierte Aufbau eines ESN zu erkennen. Folgende Parameter müssen dabei berücksichtigt werden:

- Der Eingabevektor $u(k) \in \mathbb{R}^K$ der die K-dimensionale Eingabe u zur Zeit k beschreibt.

- Der Reservoirzustand $x(k) \in \mathbb{R}^N$. Dieser umfasst die Ausgaben aller N Reservoirneuronen.

[1] Die Eingaben können auch direkt zur Ausgabeschicht weitergeleitet werden.

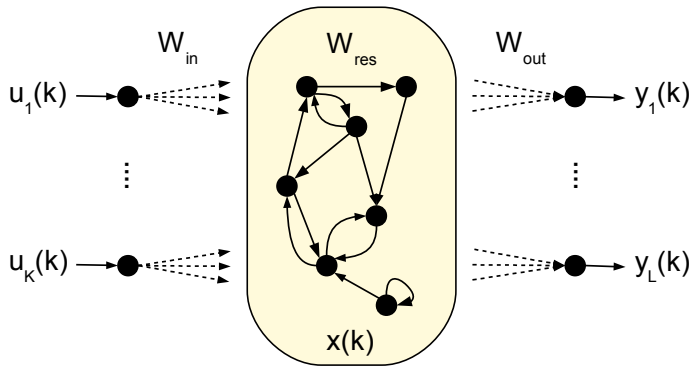


Abbildung 2.4: **Aufbau eines ESN (nach [Jaeger, 2002]).** Die Eingabe $u(k)$ wird ins Reservoir geleitet. Durch die Rückkopplungen bleibt der Effekt vorhergehender Eingaben $u(k-1),\ u(k-2),\ \ldots$ im Reservoir erhalten. Die Ausgabeschicht interpretiert den aktuellen Reservoirzustand $x(k)$ und erzeugt die Netzausgabe $y(k)$.

- Der Ausgabevektor $y(k) \in \mathbb{R}^L$ der die Ausgaben aller L Ausgabeneuronen zusammenfasst.

- Die Gewichtsmatrizen $W_{in} \in \mathbb{R}^{N \times K}$, $W_{res} \in \mathbb{R}^{N \times N}$ und $W_{out} \in \mathbb{R}^{L \times N}$, welche die Verbindungsstärke zwischen den Neuronen definieren.

- Die Übertragungsfunktion $f_{res} : \mathbb{R} \to \mathbb{R}$ der Reservoirneuronen und $f_{out} : \mathbb{R} \to \mathbb{R}$ der Ausgabeneuronen.

Nach der Erzeugung des ESNs wird der Zustand des Reservoirs und die Netzausgabe über die folgenden Gleichungen berechnet:

$$x(k) = f_{res}(W_{in} \cdot u(k) + W_{res} \cdot x(k-1)) \tag{2.6}$$

$$y(k) = f_{out}(W_{out}(k) \cdot x(k)) \tag{2.7}$$

Das hier beschriebene ESN zeigt einen minimalistischen Aufbau, der jedoch mächtig genug für die Problemstellung dieser Arbeit ist. Die folgenden Erweiterungen wurden nicht genutzt, sollen der Vollständigkeit halber dennoch erwähnt werden. Die Eingaben

können bei Bedarf am Reservoir vorbei und direkt zur Ausgabeschicht geleitet werden. Zudem ist es möglich die Ausgaben über eine *Feedback*-Matrix zurück zum Reservoir zu leiten. Näheres ist in [Jaeger, 2001, Jaeger, 2002, Lukoševičius, 2012] nachzulesen.

Echo State Eigenschaft

Die Eingabematrix W_{in} und die Ausgabematrix W_{out} werden üblicherweise mit einer Gleichverteilung im Bereich $[-1,\ 1]$ zufällig initialisiert. Für das Reservoir W_{res} gilt dies nicht. Dieses muss die *Echo State* Eigenschaft erfüllen [Jaeger, 2001]. Anschaulich bedeutet das, dass der Reservoirzustand $x(k)$ nur von den vorhergehenden Eingaben u abhängt und bei Unterbindung weiterer Eingaben in seinen initialen Zustand zurückkehrt. Je weiter die Eingaben zurückliegen, desto geringer ist ihr Einfluss. Das Reservoir hat also keine sich selbsthaltenden Zustände.

Für ein Reservoir mit sigmoider Übertragungsfunktion wurde herausgefunden, dass die *Echo State* Eigenschaft erfüllt ist, wenn der betragsmäßig größte Eigenwert von W_{res} kleiner 1 ist [Jaeger, 2002]. Allerdings ist die Aussage unbewiesen. Es handelt sich nur um eine Beobachtung, für die bisher kein Gegenbeispiel gefunden wurde. Um ein ESN mit *Echo State* Eigenschaft zu erzeugen, wird folgendes Vorgehen empfohlen [Jaeger, 2002, Lukoševičius, 2012] :

1. Wähle die Anzahl N an Reservoirneuronen, die Verbindungsdichte c_{res} des Reservoirs, einen spektralen Radius $\alpha\ \in\ (0,\ 1)$, die Skalierung r_{in} der Eingabematrix und die Übertragungsfunktionen f_{res} und f_{out}

2. Initialisiere W_{in} mit gleichverteilten Zufallswerten im Bereich [-r_{in}, r_{in}]

3. Initialisiere W_{res} als dünnbesetzte Matrix (engl. *sparse matrix*) mit Verbindungsdichte c_{res} und gleichverteilten Zufallswerten im Bereich [-1, 1]

4. Berechne den betragsmäßig größten Eigenwert λ_{max} von W_{res}

5. Skaliere W_{res} mit λ_{max} : $W_{res} = \frac{1}{|\lambda_{max}|}\ \cdot\ W_{res}$

6. Skaliere W_{res} mit dem spektralen Radius α : $W_{res} = \alpha \cdot W_{res}$

Die Initialisierung von W_{out} hängt vom Trainingsverfahren ab. Im nächsten Abschnitt wird dies erläutert. Für den initialen Reservoirzustand $x(0)$ kann ein Nullvektor gewählt werden, da sich das Reservoir aufgrund der *Echo State* Eigenschaft nach wenigen konstanten Eingaben u ohnehin stabilisiert.

2.4.2 Training eines ESN

Wie bereits erwähnt wurde, liegt einer der größten Vorteile des RC im einfachen Training. Im Gegensatz zu BPTT muss nur die Ausgabeschicht trainiert werden. Diese interpretiert lediglich den Reservoirzustand und bildet diesen auf die Sollausgabe ab.

Das Training kann auf zwei Arten erfolgen. Beim offline Training wird ein im Vorfeld bekannter Datensatz auf einmal verarbeitet, während beim online Training der Reihe nach die Daten gelernt werden. Die Trainingsmethode ist anwendungsabhängig und ergibt sich anhand der Rahmenbedingungen. Im Folgenden werden beide Methoden kurz vorgestellt.

Offline Training

Das offline Training ermöglicht das Lernen eines gesamten Datensatzes auf einmal. Dazu wird der Datensatz (mit P Elementen) ins Netz eingespeist und die Reservoirzustände nach Gleichung 2.6 berechnet [Jaeger, 2002]. Alle P Zustände werden dabei in einer Matrix $M \in \mathbb{R}^{P \times N}$ gesammelt. Eine weitere Matrix $T \in \mathbb{R}^{P \times L}$ enthält die Lehrsignale bzw. Sollausgaben y_{target}. Gesucht ist eine Ausgabematrix W_{out}, die den mittleren quadratischen Fehler E_{MSE} minimiert.

$$E_{MSE} = \frac{1}{P} \cdot \sum_{i=1}^{P} (y_{target}(i) - y(i))^2 \tag{2.8}$$

W_{out} kann mit einem beliebigen linearen Regressionsalgorithmus berechnet werden [Jaeger, 2001]. In [Jaeger, 2002, Lukoševičius, 2012] geschieht dies über das Pseudo-Inverse der Matrix M:

$$W_{out} = M^{-1} \cdot f_{out}^{-1}(T) \tag{2.9}$$

Da beim Offline Training E_{MSE} minimiert wird, werden die einzelnen Daten sehr exakt approximiert. Problematisch wird dies, wenn es zu einer Überanpassung kommt, zu wenige Daten vorhanden sind, oder die Daten nicht den gesamten Wertebereich abdecken. Ein weiteres Problem entsteht, wenn sich der Datensatz verändert - beispielsweise durch das Aufkommen neuer Daten. In diesem Fall muss der gesamte Datensatz erneut gelernt werden, was aufgrund der aufwendigen Berechnung der Pseudo-Inversen sehr kostspielig ist. Da in dieser Arbeit der Agent während seiner gesamten Lebenszeit neue Daten sammelt, empfiehlt sich für ihn das Online Training.

Online Training

Das online Training ermöglicht es, die Daten der Reihe nach zu lernen, indem W_{out} schrittweise adaptiert wird. Dazu wird W_{out} als Nullmatrix initialisiert. Anschließend wird für jede Eingabe $u(k)$ die Netzausgabe $y(k)$ berechnet und mit der Sollausgabe $y_{target}(k)$ verglichen. Abhängig vom Fehler $E(k) = y_{target}(k) - y(k)$ werden die einzelnen Gewichte von W_{out} angepasst:

$$W_{out}(k+1) = W_{out}(k) + \gamma \cdot \frac{\partial E(k)}{\partial W_{out}(k)} \tag{2.10}$$

Wegen der schnelleren Konvergenz wird anstatt dem klassischen Gradientenabstieg, das *Recursive Least Squares* (RLS) Verfahren benutzt [Vahidi et al., 2005]:

$$W_{out}(k+1) = W_{out}(k) + L(k) \cdot (y_{target}(k) - y(k)) \tag{2.11}$$

Der Korrekturfaktor $L(k)$ berechnet sich über die Kovarianzmatrix $P(k) \in \mathbb{R}^N$:

$$L(k) = P(k) \cdot x(k) = P(k-1) \cdot x(k) \cdot (1 + x^t(k) \cdot P(k-1) \cdot x(k))^{-1} \tag{2.12}$$

$$P(k) = (I - L(k) \cdot x^t(k)) \cdot P(k-1) \tag{2.13}$$

Dabei bezeichnet I die Einheitsmatrix und x den Reservoirzustand. Die Kovarianzmatrix P wird mit einer skalierten Einheitsmatrix initialisiert:

$$P(0) = I \cdot \delta \tag{2.14}$$

2.5 Die RC-ADP-Architektur

Die Kombination von *Adaptive Dynamic Programming* und *Reservoir Computing* wurde vor wenigen Jahren in [Koprinkova-Hristova et al., 2010, Oubbati et al., 2011, Koprinkova-Hristova et al., 2012, Oubbati et al., 2012, Oubbati et al., 2013] untersucht und stellt den Ausgangspunkt dieser Arbeit dar. Die Idee ist, die Kritik mit einem ESN zu realisieren, um dynamische Probleme mit kontinuierlichem Zustandsraum einfacher lösen zu können. Da außerdem ein autonomer Agent oftmals einem lebenslangen Lernprozess unterliegt, eignet sich der einfache online Trainingsalgorithmus zur Bestimmung der Approxiation $\hat{J}$.

2.5.1 Kritik

Die Kritik erhält zu jedem Zeitpunkt eine Repräsentation des Weltzustandes $s(k)$, sowie die Aktion $a(k)$ und bildet dies auf den Reservoirzustand $x(k)$ ab[2]:

$$x(k) = f_{res}(W_{in} \cdot [s(k),\ a(k)] + W_{res} \cdot x(k-1)) \tag{2.15}$$

Über die Ausgabematrix W_{out} erfolgt die Approximation:

$$\hat{J}(k) = f_{out}(W_{out}(k) \cdot x(k)) \tag{2.16}$$

[2] siehe Gleichung 2.6 mit $u(k) = [s(k)\ a(k)]$.

Das online Training wird mit dem RLS-Verfahren umgesetzt. Da

$$J(k) = U(k) + \gamma \cdot J(k+1)$$

gelten soll, ergibt sich der Fehler zu:

$$E(k) = \hat{J}(k) - U(k) - \gamma \cdot \hat{J}(k+1) \tag{2.17}$$

Anschließend wird die Ausgabematrix W_{out} an den Fehler angepasst:

$$W_{out}(k+1) = W_{out}(k) + L(k) \cdot E(k) \tag{2.18}$$

2.5.2 Aktor

Der Aktor erhält von der Kritik die Ableitung $\frac{\partial \hat{J}(k)}{\partial a(k)}$ und leitet daraus die nächste Aktion $a(k+1)$ ab:

$$a(k+1) = a(k) + \delta \cdot \frac{\partial \hat{J}(k)}{\partial a(k)} \tag{2.19}$$

Ist die Aktion $a(k)$ auch im aktuellen Zustand optimal, so hat die Ableitung $\frac{\partial \hat{J}(k)}{\partial a(k)}$ den Betrag 0. Andernfalls muss die Aktion angepasst werden. Der Faktor $\delta \in [0,\ 1]$ bezeichnet die Anpassungsrate und ist anwendungsabhängig. Näheres dazu findet sich in Kapitel 3.

Da sich der Reservoirzustand $x(k)$ aus den zuvor eingegebenen Zuständen $[s(k),\ a(k)]$, $[s(k-1),\ a(k-1)]$, ... zusammensetzt, ergibt sich die Ableitung $\frac{\partial \hat{J}(k)}{\partial a(k)}$ über die Kettenregel:

$$\frac{\partial \hat{J}(k)}{\partial a(k)} = \frac{\partial \hat{J}(k)}{\partial x(k)} \cdot \frac{\partial x(k)}{\partial a(k)} \tag{2.20}$$

Die genaue Herleitung der mathematischen Gleichungen ist in [Koprinkova-Hristova et al., 2012, Oubbati et al., 2013] beschrieben. Der Vollständigkeit halber wird sie kurz zusammengefasst.

Wird in Gleichung 2.16 die Identität als Übertragungsfunktion benutzt, so berechnet sich deren Ableitung nach:

$$\frac{\partial \hat{J}(k)}{\partial x(k)} = W_{out}(k) \tag{2.21}$$

In Gleichung 2.15 wird als Übertragungsfunktion f_{res} die Sigmoidfunktion verwendet. Mittels der Ableitung des Tangens Hyperbolicus[3] folgt:

$$\begin{aligned} \frac{\partial x(k)}{\partial a(k)} &= \left(I - tanh^2\left(W_{in} \cdot [s(k),\ a(k)] + W_{res} \cdot x(k-1)\right)\right) \cdot W_{in}^a \\ &= (I - x^2(k)) \cdot W_{in}^a \end{aligned} \tag{2.22}$$

Die Matrix W_{in}^a bezeichnet den Teil von W_{in}, der die Gewichte für die Eingabe a definiert, also $W_{in} = [W_{in}^s,\ W_{in}^a]$. Setzt man die Gleichungen 2.21 und 2.22 in Gleichung 2.20 ein, erhält man:

$$\frac{\partial \hat{J}(k)}{\partial a(k)} = W_{out}(k) \cdot (I - x^2(k)) \cdot W_{in}^a \tag{2.23}$$

[3] Ableitung des Tangens Hyperbolicus: $\frac{\partial}{\partial x} tanh(x) = 1 - tanh^2(x)$.

3 Umsetzung

Im Rahmen der Diplomarbeit [Uhlemann, 2011] wurde ein Agent entwickelt, der *Actor-Critic Design* mit ESNs kombiniert. Der Agent dient dazu, diese Architektur auf ihre Eignung zur Lösung von Problemen mit einem großen und kontinuierlichen Zustandsraum zu untersuchen. Der Agent wurde sowohl simulativ, als auch auf einem realen Roboter implementiert und dient als Inspiration für die Umsetzung in dieser Arbeit. Der interne Aufbau des Agenten wurde jedoch deutlich verändert indem die Kritik und der Aktor neu konzipiert wurden.

3.1 Szenario

An dieser Stelle soll zunächst das Szenario vorgestellt werden. Abbildung 3.1 zeigt eine virtuelle Welt, in der sich ein mobiler Agent befindet. Die Welt hat eine Breite von 100 Einheiten und eine Höhe von 160 Einheiten. Der Agent bewegt sich mit einer festen Geschwindigkeit von 4 Einheiten pro Zeitschritt. Um die Welt horizontal zu durchqueren, benötigt er bei reiner Translation 25 Zeitschritte.

Innerhalb der Welt können beliebige Bewertungsfunktionen U definiert werden, die jeder Position $p = [x,\ y]$ einen reellen Wert zuweisen:

$$U\colon\ \mathbb{R}^2 \rightarrow \mathbb{R}$$

Das Ziel des Agenten besteht darin, über die Änderung seiner Position eine optimale Bewertung zu erhalten. Dabei ist dem Agent seine wahre Position innerhalb der virtuellen

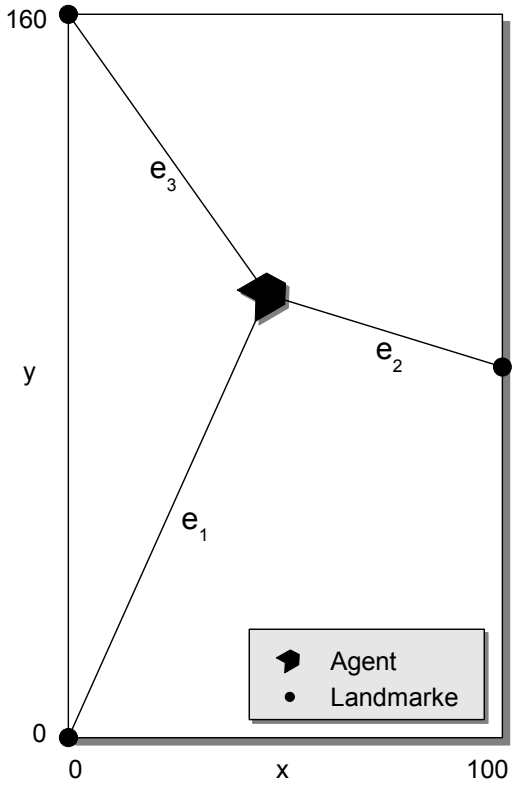


Abbildung 3.1: **Virtuelle Welt.** Der Agent bewegt sich in einer virtuellen Welt. Über die Distanzen e_i zu drei Landmarken erhält er eine abstrakte Beschreibung seiner Position. Seine wahre Position $p = [x,\ y]$ ist dem Agent dagegen unbekannt. Innerhalb der virtuellen Welt wird jeder Position eine bestimmte Bewertung zugeordnet. Dazu können beliebige Bewertungsfunktionen definiert werden.

Welt unbekannt. Er erhält lediglich ein abstrakte Beschreibung seiner Position in Form eines Entfernungsvektors:

$$e(k) = [e_1(k),\ e_2(k),\ e_3(k)]$$

Die Vektorelemente beschreiben den Abstand zu den Landmarken l_i und berechnen sich auf folgende Weise:

$$e_i(k) = d_l \cdot \|l_i - p(k)\| \tag{3.1}$$

wobei d_l als Skalierungsfaktor dient. Dem Agenten ist die Definition der Gleichung 3.1 unbekannt, d.h. er kann seine Position $p(k)$ nicht aus $e(k)$ herleiten. Dieser Punkt stellt einen wichtigen Unterschied zu der Umsetzung aus [Uhlemann, 2011] dar.

3.2 Bisherige Umsetzung

Das soeben beschriebene Szenario entspricht bis auf einige Details dem Szenario aus [Uhlemann, 2011]. Der wesentliche Unterschied zwischen der Umsetzung in dieser Arbeit und der Umsetzung aus [Uhlemann, 2011] liegt in der Architektur des Agenten. Der Agent aus [Uhlemann, 2011] leitet nur den Entfernungsvektor e in die Kritik. Damit lernt er eine Ein-Schritt-Vorhersage der Bewertungsfunktion U. Der Aktor führt einen Gradientenaufstieg auf der Vorhersage durch.

Obwohl die theoretischen Überlegungen zu der Architektur des Agenten aus [Uhlemann, 2011] richtig sind, zeigten sich bei einer Nachimplementierung einige Probleme. Das Hauptproblem liegt in der Schrittweite des Agenten. Bei einer sehr geringen Schrittweite gilt:

$$U(p(k)) \approx U(p(k+1)) \tag{3.2}$$

Der Verdacht liegt nahe, dass der Agent möglicherweise nur eine Approximation der Bewertungsfunktion U lernt:

$$\hat{J}(e(k)) \approx U(p(k)) \tag{3.3}$$

Dadurch ließe sich auch der schnelle Lernerfolg des Agenten erklären. Ob dies tatsächlich in [Uhlemann, 2011] der Fall ist, kann nicht mit Sicherheit gesagt werden.

Ein anderes Problem besteht in der Berechnung der jeweils nachfolgenden Aktion. Dazu wird der Gradient ∇ von $\hat{J}$ berechnet [Uhlemann, 2011]:

$$\frac{\partial \hat{J}(k)}{\partial \omega(k)} = \frac{\partial \hat{J}(k)}{\partial e(k)} \cdot \frac{\partial e(k)}{\partial p(k)} \cdot \frac{\partial p(k)}{\partial \omega(k)} \tag{3.4}$$

wobei der Gradient durch folgende Ableitung beschrieben wird:

$$\nabla(k) = \frac{\partial \hat{J}(k)}{\partial e(k)} \cdot \frac{\partial e(k)}{\partial p(k)} \tag{3.5}$$

Für die Ableitung $\frac{\partial e(k)}{\partial p(k)}$ muss dem Agent die Ableitungsfunktion der Distanzfunktion[1] bekannt sein. Interessant wäre jedoch, wenn dieses Wissen dem Agent vorenthalten wird. Der in dieser Arbeit realisierte Agent umgeht diese Problematik, indem er - zusätzlich zum Zustand $e(k)$ - auch die Aktion $a(k)$ in die Kritik leitet.

3.3 Neue Konzeption des Agenten

Der in dieser Arbeit realisierte Agent leitet nicht nur den Zustand $e(k)$ in die Kritik, sondern auch die Aktion $a(k)$. Die Idee dazu stammt aus [Koprinkova-Hristova et al., 2010, Oubbati et al., 2011, Koprinkova-Hristova et al., 2012]. Die Kritik lernt dadurch den Effekt der Aktion $a(k)$ auf die nachfolgenden Bewertungen.

Zunächst muss definiert werden, was als Aktion des Agenten in Frage kommt. Die folgenden zwei Möglichkeiten wurden in dieser Arbeit implementiert:

[1] Diese berechnet die Elemente des Entfernungsvektors e.

Aktionsdefinition φ

Ein einfacher Ansatz ist $a(k)$ als die Richtung, in die sich der Agent bewegt, aufzufassen. Formal bedeutet dies:

$$a(k) := \varphi(k) \qquad 0 \leq \varphi(k) \leq 2\pi \tag{3.6}$$

In diesem Fall ergibt sich der Folgezustand eindeutig aus $e(k)$ und $\varphi(k)$.

Aktionsdefinition ω

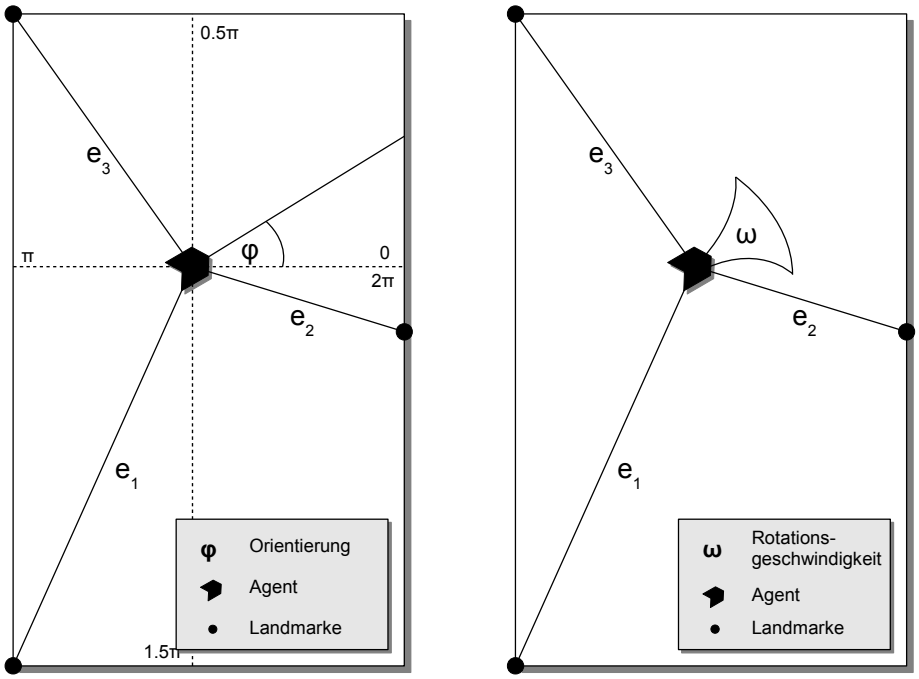
Die zweite Möglichkeit besteht darin, die Aktion $a(k)$ als eine Art Rotationsgeschwindigkeit $\omega(k)$ zu definieren:

$$a(k) := \omega(k) \qquad -\frac{\pi}{4} \leq \omega(k) \leq \frac{\pi}{4} \tag{3.7}$$

Dabei beschreibt $\omega(k)$ die Stärke der Richtungsänderung des Agenten innerhalb eines Zeitschritts. Ein Wert von $\frac{\pi}{4}$ bedeutet eine Drehung von $45°$ nach links, während ein Wert von $-\frac{\pi}{4}$ einer Drehung von $45°$ nach rechts entspricht. Der Agent kann seine Richtung innerhalb eines Zeitschrittes beliebig zwischen $0°$ und $45°$ ändern.

Bei dieser Definition ist dem Agenten seine wahre Orientierung innerhalb der Welt unbekannt. Allein mit Hilfe des Entfernungsvektors $e(k)$ kann er seinen Zustand nicht eindeutig identifizieren, da er nur seine Position und Rotationsgeschwindigkeit, nicht jedoch seine Orientierung kennt. Somit bewirkt die Aktion $\omega(k)$ im Zustand $e(k)$ - abhängig von der aktuellen Orientierung - verschiedene Folgezustände.

Mit den vorhergehenden Zustände $e(k-1)$, $e(k-2)$, ... kann der Agent dennoch abschätzen, welche Orientierung er zum Zeitpunkt k hat. Der Folgezustand ergibt sich somit über den Verlauf der Entfernungsvektoren und seiner Aktion $\omega(k)$. Abbildung 3.2 zeigt die beiden Alternativen zur Aktionsdefinition.



(a) Aktionsdefinition φ (b) Aktionsdefinition ω

Abbildung 3.2: **Alternativen zur Aktionsdefinition.** Links ist Alternative 1 dargestellt. Als Aktion wird die Richtung φ, in die der Agent fährt, gewählt. Der Agent kennt somit seine Position - in Form des Entfernungsvektors $e(k)$ - und seine exakte Orientierung $\varphi(k)$. Rechts ist Alternative 2 zu sehen, bei der die Rotationsgeschwindigkeit ω die Aktion definiert. Hier kennt der Agent seine wahre Orientierung nicht. Er weiß lediglich, wie stark er nach links oder rechts fährt. Nur anhand vorhergehende Entfernungsvektoren $e(k-1)$, $e(k-2)$, ... kann er seine Orientierung abschätzen.

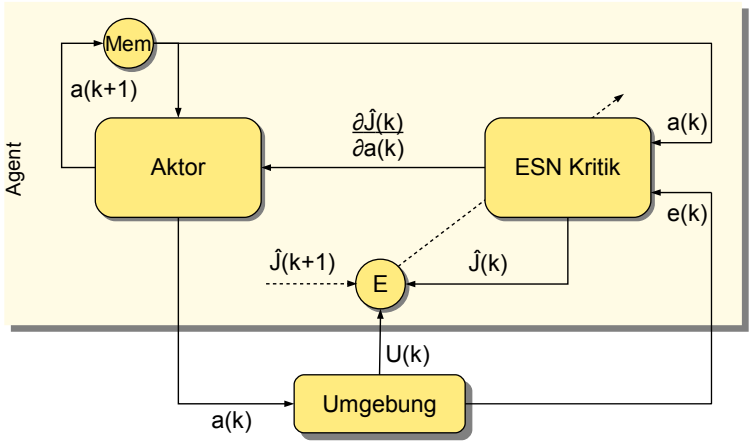


Abbildung 3.3: **Architektur des Agenten.** Die Kritik erhält als Eingabe den Zustand $e(k)$ und die Aktion $a(k)$ und bestimmt mit einem ESN die Approximation $\hat{J}(k)$ der zu erwartenden Bewertungen. Der Aktor führt die Aktion $a(k)$ aus. Dadurch ergibt sich der Folgezustand $e(k+1)$. Sobald der Folgezustand bekannt ist, erhält der Agent eine Bewertung $U(k)$ seiner Aktion $a(k)$. Über den Fehler $E(k)$ wird das ESN trainiert. Der Aktor nutzt die Ableitung von $\hat{J}(k)$ um die nächste Aktion $a(k+1)$ zu bestimmen. Diese Aktion wird im Puffer Mem zwischengespeichert und im Folgezustand $e(k+1)$ ausgeführt.

3.3.1 Architektur des Agenten

Der interne Aufbau des neu konzipierten Agenten entspricht der RC-ADP-Architektur, d.h. die Kritik wird mit einem ESN realisiert und der Aktor bestimmt die nächste Aktion mittels der Ableitung von $\hat{J}$. Der genaue Aufbau ist in Abbildung 3.3 zu sehen.

In jedem Zeitschritt k wird der Zustand $e(k)$ und die Aktion $a(k)$ in die Kritik geleitet. Die Kritik kann damit die Approximation $\hat{J}(k)$ berechnen. Mit der Ableitung von $\hat{J}(k)$ bestimmt der Aktor die Aktion $a(k+1)$, welche im nächsten Schritt ausgeführt wird. Zum Zeitpunkt k wird die zuvor berechnete Aktion $a(k)$ ausgeführt. Der Aktor handelt mit einer Verzögerung von einem Schritt. Dies stellt aufgrund des kontinuierlichen Zustandsraumes kein Problem dar[2]. Nach der Aktionsausführung ergibt sich der Folge-

[2]Der Aktor adaptiert die Aktion ohnehin nur geringfügig.

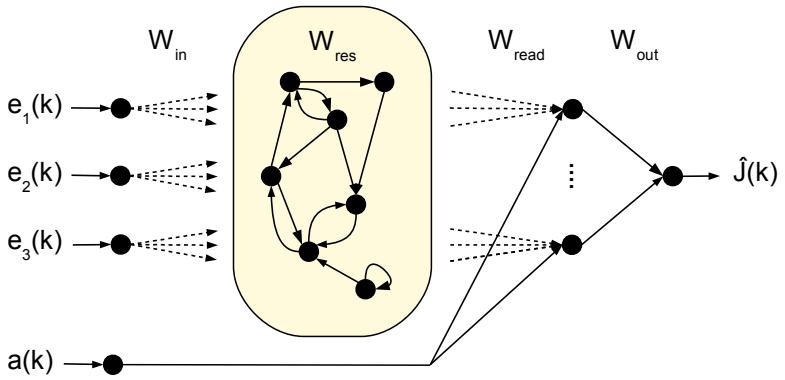


Abbildung 3.4: **Das ESN der Kritik.** Die Kritik approximiert die Vorhersage $\hat{J}(k)$ der zu erwartenden Bewertungen mit einem ESN, das den Reservoirzustand mit einem zweischichtigen Auslesenetz interpretiert. Der Zustand $e(k)$ wird ins Reservoir geleitet, während die Aktion $a(k)$ am Reservoir vorbeigeführt wird. Dadurch reduziert sich die Anzahl an Reservoirzuständen.

zustand $e(k+1)$. Die Bewertung $U(k)$ der Folgeposition ist nun bekannt. Mittels $U(k)$, $\hat{J}(k)$ und $\hat{J}(k+1)$ wird schließlich der Fehler $E(k)$ ermittelt und die Kritik angepasst. Die Anpassung und die Berechnung des Fehlers kann erst im darauffolgenden Zeitschritt erfolgen.

Nachdem der Aufbau und das Verhalten des Agenten kurz skizziert wurden, folgt nun eine detaillierte Beschreibung der Kritik und des Aktors.

3.3.2 Kritik

Für die Kritik wird ein ESN benutzt, welches den Reservoirzustand mit zwei Schichten interpretiert. Zwischen Reservoir und Ausgabeschicht wird eine weitere Schicht von Neuronen eingefügt - wie in Abbildung 3.4 dargestellt ist. Auf diese Weise können mächtige Funktionen approximiert werden ohne das Reservoir größer zu dimensionieren als erforderlich.

Die Aufgabe des Reservoirs ist es, eine eindeutige Zustandsbeschreibung aus den Eingaben e zu gewinnen. Die Rückkopplungen des Reservoirs sind nur notwendig, wenn als Aktion die Rotationsgeschwindigkeit ω gewählt wird. Durch den Effekt der früheren Eingaben $e(k-1)$, $e(k-2)$, ..., kann die Kritik den aktuellen Zustand eindeutig identifizieren. Da die vorangegangenen Aktionen $a(k-1)$, $a(k-2)$, ... bei der Wahl der nächsten Aktion $a(k+1)$ nicht von Interesse sind, werden sie am Reservoir vorbeigeleitet.

Die beiden Ausleseschichten haben die Aufgabe, abhängig von der Zustandsbeschreibung des Reservoirs und der Aktion $a(k)$ die Approximation $\hat{J}(k)$ zu erlernen.

Alle M Neuronen der zusätzlichen Ausleseschicht benutzen als Übertragungsfunktion den Tangens Hyperbolicus. Die Gewichtsmatrix dieser Schicht wird mit $W_{read} \in \mathbb{R}^{M \times N+1}$ bezeichnet. Die mathematischen Gleichungen der RC-ADP-Architektur bleiben weitestgehend erhalten. Der Reservoirzustand $x(k)$ berechnet sich zu:

$$x(k) = tanh(W_{in} \cdot e(k) + W_{res} \cdot x(k-1)) \tag{3.8}$$

Für den Zustand $z(k)$ der Ausleseschicht gilt:

$$z(k) = tanh(W_{read} \cdot [x(k),\ a(k)]) \tag{3.9}$$

und für die Ausgabe:

$$\hat{J}(k) = W_{out}(k) \cdot z(k) \tag{3.10}$$

Wie bisher wird nur die Ausgabeschicht W_{out} trainiert. Allerdings wird zur Berechnung des Korrekturfaktors $L(k)$ und der Kovarianzmatrix $P(k)$ (RLS-Verfahren) die Neuronenausgabe $z(k)$ anstatt des Reservoirzustands $x(k)$ benutzt.

Die Auslesematrix W_{read} wird zu Beginn mit gleichverteilten Zufallswerten im Bereich $[-1,\ 1]$ initialisiert und anschließend nicht mehr verändert.

Der Vorteil des zweischichtigen Auslesenetzes liegt in einer effizienteren Berechnung. Ohne die zweite Schicht, müsste das Reservoir für komplizierte Bewertungsfunktionen sehr groß gewählt werden. Dadurch entstehen selbst bei kleinem spektralem Radius

und niedriger Verbindungsdichte sehr viele mögliche Reservoirzustände, die wiederum korrekt interpretiert werden müssen. Außerdem müsste die Aktion $a(k)$ ebenfalls ins Reservoir geleitet werden, was die Anzahl an möglichen Zuständen weiter erhöht. Bereits bei der einfachen Umsetzung aus [Uhlemann, 2011] - wo nur der Zustand $e(k)$ ins Reservoir geleitet wird - sind ca. 500 Reservoirneuronen notwendig, um gute Resultate zu erzielen [Uhlemann, 2011].

3.3.3 Aktor

Der Aktor benötigt zur Bestimmung der nächsten Aktion $a(k+1)$ die Ableitung von $\hat{J}(k)$. Da $a(k)$ direkt in die Ausleseschicht geleitet wird, berechnet sich die Ableitung analog zu der Beschreibung aus Abschnitt 2.5. Über die Kettenregel erhält man:

$$\frac{\partial \hat{J}(k)}{\partial a(k)} = \frac{\partial \hat{J}(k)}{\partial z(k)} \cdot \frac{\partial z(k)}{\partial a(k)} \tag{3.11}$$

Mit Gleichung 3.10 folgt:

$$\frac{\partial \hat{J}(k)}{\partial z(k)} = W_{out}(k) \tag{3.12}$$

Wegen der sigmoiden Neuronen der Ausleseschicht gilt analog zu Gleichung 2.22:

$$\begin{aligned} \frac{\partial z(k)}{\partial a(k)} &= \left(I - tanh^2\left(W_{read} \cdot [x(k),\ a(k)]\right)\right) \cdot W^a_{read} \\ &= (I - z^2(k)) \cdot W^a_{read} \end{aligned} \tag{3.13}$$

wobei W^a_{read} denjenigen Teil von W_{read} bezeichnet, der die Gewichte für die Eingabe $a(k)$ definiert, d.h. $W_{read} = [W^x_{read},\ W^a_{read}]$. Setzt man die Gleichungen 3.12 und 3.13 in Gleichung 3.11 ein, erhält man:

$$\frac{\partial \hat{J}(k)}{\partial a(k)} = W_{out}(k) \cdot (I - z^2(k)) \cdot W^a_{read} \tag{3.14}$$

Dynamische Anpassungsrate

Nachdem der Aktor die Ableitung $\frac{\partial \hat{J}(k)}{\partial a(k)}$ von der Kritik erhalten hat, könnte er die nächste Aktion $a(k+1)$ direkt nach Gleichung 2.19 berechnen. Da die Bewertungsfunktionen in dieser Arbeit nicht linear sein müssen, wäre bei dieser Methode keine korrekte Navigation möglich. Dies liegt daran, dass der Betrag der Ableitung in denjenigen Bereichen sehr groß wird, wo die Bewertungsfunktion starken Änderungen unterliegt. In Bereichen mit geringen Änderungen hingegen nimmt der Betrag der Ableitung sehr kleine Werte an. Für eine korrekte Navigation wird statt der statischen Anpassungsrate δ eine dynamische Anpassungsrate benötigt.

Die Idee für eine dynamische Anpassungsrate ist angelehnt an den Momentumterm [Rumelhart et al., 1986]. Umgesetzt wird sie mit dem Term η:

$$\eta(k) = \mu \cdot \eta(k-1) + sgn\left(\frac{\partial \hat{J}(k)}{\partial a(k)}\right) \tag{3.15}$$

Die Funktion $sgn()$ bezeichnet die Signum-Funktion[3], für μ wird der Wert 0.6 benutzt und initialisiert wird η mit 0. Damit beschränkt sich der Wertebereich für η auf:

$$-\frac{5}{2} \leq \eta \leq \frac{5}{2}$$

Der Aktor berechnet damit die nächste Aktion:

$$a(k+1) = a(k) + \frac{\pi}{10} \cdot \eta(k) \tag{3.16}$$

was abhängig von $\eta(k)$ eine Richtungsänderung zwischen $0°$ und $45°$ bewirkt.

[3] Die Signum-Funktion liefert für negative Zahlen -1 und für positive Zahlen 1.

4 Multikriterielle Optimierung

Das erste Ziel dieser Arbeit besteht darin, die RC-ADP-Architektur so zu erweitern, dass multikriterielle Optimierungsprobleme (MOPs) gelöst werden können. Die bisherigen Arbeiten zu RC-ADP befassen sich ausschließlich mit der Optimierung eines einzigen Ziels. Da in der Robotik oftmals Problemstellungen auftreten, bei denen der Agent mehrere Ziele verfolgt, stellt diese Erweiterung einen interessanten Ansatz dar und wird in diesem Kapitel erläutert. Erste Ergebnisse dazu wurden in [Oubbati et al., 2013] veröffentlicht[1].

4.1 Multikriterielle Optimierungsprobleme

Bei dem in Kapitel 3 beschriebenen Szenario handelt es sich um ein einfaches Optimierungsproblem. Der Agent muss lediglich die Bewertungsfunktion U optimieren. Mathematisch lässt sich das Optimierungsproblem auf folgende Art beschreiben:

$$\begin{aligned} &\text{gegeben:} && U\colon\ \mathbb{R}^2 \rightarrow \mathbb{R} \\ &\text{gesucht:} && \max_{p \in \mathbb{R}^2} U(p) \end{aligned}$$

Ein schwierigeres Optimierungsproblem entsteht, wenn der Agent nicht nur ein Ziel verfolgt, sondern gleichzeitig mehrere Ziele. In diesem Fall wird die Bewertung des Zustands nicht durch einen skalaren Wert definiert, sondern durch einen Vektor. Jedes Vektorelement beschreibt die Bewertung des Zustands für ein Ziel. Abhängig davon,

[1] Die Ergebnisse wurden nicht mit dem in dieser Arbeit konzipierten Agenten erzielt.

wie die einzelnen Ziele priorisiert werden, können verschiedene Vektoren optimal sein. Formal lässt sich das MOP folgendermaßen darstellen [Marler and Arora, 2004]:

$$\begin{aligned} &\text{gegeben:} \quad U_1,\ U_2, ...,\ U_L\colon\ \mathbb{R}^2 \to \mathbb{R} \\ &\text{gesucht:} \quad \max_{p \in \mathbb{R}^2} U(p) = [U_1(p),\ U_2(p), ...,\ U_L(p)] \end{aligned}$$

wobei L die Anzahl an Zielen beschreibt. Die Anzahl der Entscheidungsvariablen ist durch die beiden Weltkoordinaten $[x,\ y]$ festgelegt und beträgt somit zwei. $U(p)$ bezeichnet den Vektor, der die Bewertungen $U_i(p)$ beinhaltet. Alle gültigen Position $p \in \mathbb{R}^2$ bilden den Entscheidungsraum (engl. *feasible decision space*). Dieser umfasst alle Position innerhalb der virtuellen Welt. Alle gültigen Lösungen $U(p) \in \mathbb{R}^L$ bilden den Lösungsraum (engl. *feasible criterion space*).

Um eine solches multikriterielles Optimierunsproblem lösen zu können, bietet sich die Pareto-Optimierung an [Marler and Arora, 2004, Linder and Lindkvist, 2011].

4.1.1 Pareto-Optimierung

Da für MOPs keine eindeutig beste Lösung existiert, ist es hilfreich diejenigen Lösungen zu finden, die den besten Kompromiss zwischen den einzelnen Zielen bieten. Dazu werden folgende Begriffe definiert:

Definition 1. *Ein Punkt p^* aus dem Entscheidungsraum ist ein Pareto-Optimum, falls kein weiterer Punkt p im Entscheidungsraum existiert, für den $U(p) \geq U(p^*)$ und zudem $U_i(p) > U_i(p^*)$ für mindestens eine Bewertungsfunktion U_i gilt [Marler and Arora, 2004].*

Bei einem Pareto-Optimum kann also kein Ziel weiter verbessert werden, ohne ein anderes zu verschlechtern. Punkte die Pareto-optimal sind werden in der Pareto-Menge vereint. Daneben gibt es den Begriff der nicht-dominierten Lösung, welcher sehr ähnlich definiert ist. Die Begriffe werden häufig synonym verwendet, jedoch bezieht sich der Begriff „nicht-dominiert“ auf den Lösungsraum. Schließlich werden noch schwache

Pareto-Optima definiert. Dies sind Punkte, bei denen sich nicht alle Ziele gleichzeitig verbessern lassen:

Definition 2. *Ein Punkt p^* aus dem Entscheidungsraum ist ein schwaches Pareto-Optimum, falls kein weiterer Punkt p im Entscheidungsraum existiert, für den $U(p) > U(p^*)$ gilt [Marler and Arora, 2004].*

Liegt die Pareto-Menge vor, so kann der Entscheidungsträger - in unserem Fall der Agent - die für ihn optimale Lösung auswählen. Ein Überblick über Methoden zur Bestimmung von Pareto-Optima ist in [Marler and Arora, 2004] gegeben. Die einfachste Möglichkeit bietet die Skalarisierung.

4.1.2 Skalarisierung

Die Skalarisierung ist ein Oberbegriff für Methoden, die das MOP in ein einfaches Optimierungsproblem transformieren. Am häufigsten wird dazu die Zielgewichtung (engl. *weighted sum method*) gewählt [Marler and Arora, 2004]. Dabei werden für alle L Ziele Prioritäten gesetzt. Die Bewertungen werden anschließend gewichtet aufsummiert, wobei sich die Gewichte w_i aus den Prioritäten ergeben:

$$U = \sum_{i=1}^{L} w_i \cdot U_i \tag{4.1}$$

Anstelle eines Lösungsvektors ergibt sich ein skalarer Wert. Das MOP reduziert sich auf ein einfaches Optimierungsproblem, welches mit den bekannten Verfahren gelöst werden kann. Problematisch an dieser Vorgehensweise ist, dass die Gewichte a priori festgelegt werden müssen. Bereits kleine Veränderungen in der Gewichtung könnten zu einem besseren Ergebnis führen. Um dieses Problem zu umgehen, können mehrere Durchläufe mit unterschiedlichen Gewichten ausgeführt werden. Doch auch dieses Vorgehen ist problematisch. Selbst wenn die Gewichte gleichmäßig und kontinuierlich verteilt werden, ist nicht garantiert, dass die Resultate gleichverteilt den Lösungsraum abdecken, d.h. dass eine vollständige Repräsentation der Pareto-Menge gefunden wird [Marler and Arora, 2004].

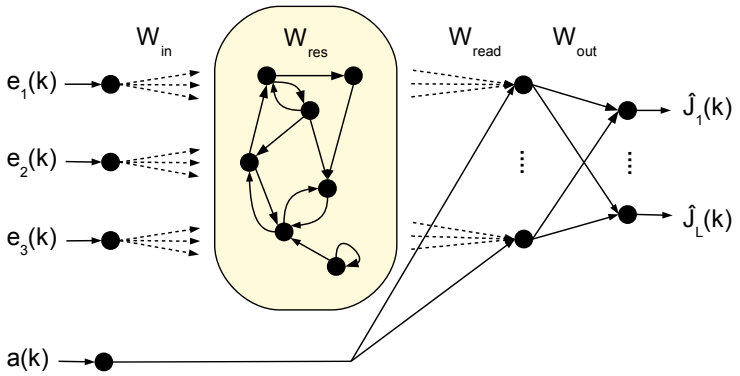


Abbildung 4.1: **ESN zur Approximation mehrerer Vorhersagen.** Bei multikriteriellen Optimierungsproblemen ist die Vorhersage $\hat{J}$ der Bewertungen vektoriell. Jedes Element $\hat{J}_i$ beschreibt die Vorhersage für eine Bewertungsfunktion.

4.2 Anpassung der RC-ADP-Architektur an MOPs

Nachdem im vorangegangenen Abschnitt eine kurze Einführung zu multikriterieller Optimierung gegeben wurde, kann die Anpassung der RC-ADP-Architektur an MOPs erläutert werden. Die Idee ist, mit Hilfe der Kritik für jede Bewertungsfunktion U_i eine Approximation $\hat{J}_i$ zu lernen. Dazu wird die Ausgabeschicht erweitert, d.h. aus $W_{out} \in \mathbb{R}^{N\times 1}$ wird $W_{out} \in \mathbb{R}^{N\times L}$ mit:

$$W_{out} = [W_{out_1},\ W_{out_2},\ \ldots,\ W_{out_L}]$$

und aus der skalaren Netzausgabe $\hat{J} \in \mathbb{R}$ wird ein Vektor $\hat{J} \in \mathbb{R}^L$. Abbildung 4.1 zeigt die Änderungen am ESN.

Mit Hilfe der Skalarisierung wird anschließend das multikriterielle Optimierungsproblem in ein einfaches Optimierungsproblem transformiert. Abbildung 4.2 zeigt den schematischen Aufbau der erweiterten RC-ADP-Architektur. Wegen der Erweiterung sind einige Anpassungen an der Kritik notwendig, welche nachfolgend beschrieben werden.

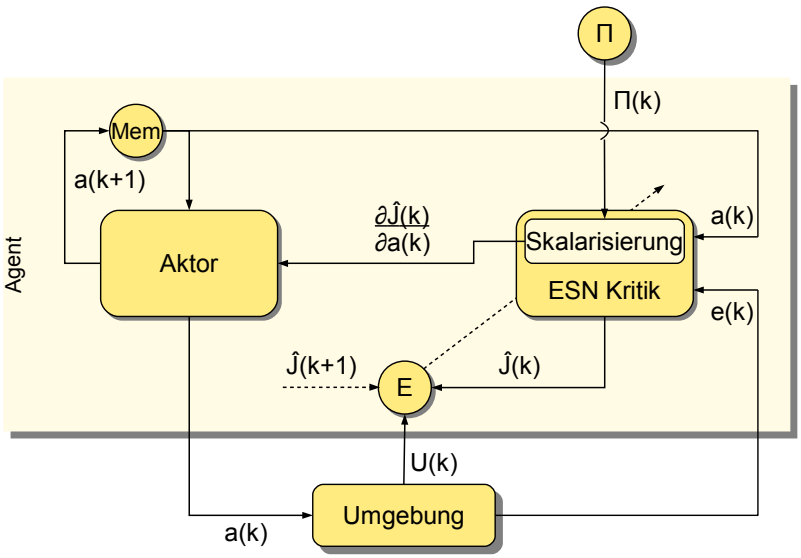


Abbildung 4.2: **Erweiterte Architektur des Agenten.** Die Approximation $\hat{J}(k)$ wird mit einer von außen vorgegebenen Zielgewichtung $\Pi(k)$ skalarisiert. Dadurch reduziert sich das MOP auf ein einfaches Optimierungsproblem. Der Aktor nutzt die Ableitung der skalarisierten Approximation und kann damit die nächste Aktion $a(k+1)$ ermitteln.

4.2.1 Anpassung der Kritik

Der Agent kann zu verschiedenen Zeitpunkten unterschiedliche Ziele verfolgen, d.h. die Zielgewichtung kann sich ändern. Die Zielgewichtung $\Pi(k)$ beschreibt die Prioritäten des Agenten zu einem bestimmten Zeitpunkt k. Sie lässt sich als Gewichtsvektor auffassen und wird bei der Skalarisierung genutzt:

$$\Pi(k) = [w_1(k),\ w_2(k),\ \ldots,\ w_L(k)]$$

Mit Hilfe von $\Pi(k)$ kann die Kritik an das MOP angepasst werden. Die skalarisierte Approximation lautet:

$$\hat{J}(k) = \sum_{i=1}^{L} w_i(k) \cdot \hat{J}_i(k) \tag{4.2}$$

Die Ableitung $\frac{\partial \hat{J}(k)}{\partial z(k)}$ ergibt sich mit der Summenregel zu:

$$\frac{\partial \hat{J}(k)}{\partial z(k)} = \underbrace{\sum_{i=1}^{L} w_i(k) \cdot W_{out_i}(k)}_{=:W_\Pi(k)} \tag{4.3}$$

Ersetzt man in Gleichung 3.14 $W_{out}(k)$ durch $W_\Pi(k)$ folgt schließlich:

$$\frac{\partial \hat{J}(k)}{\partial a(k)} = W_\Pi(k) \cdot (I - z^2(k)) \cdot W^a_{read} \tag{4.4}$$

Der Aktor kann damit die dynamische Anpassungsrate (siehe Gleichung 3.15) und die nächste Aktion (siehe Gleichung 3.16) bestimmen.

Training

Bei der Kritik ist zu beachten, dass beim Training mit dem *discount*-Faktor $\gamma > 0$ nicht jede Approximation $\hat{J}_i$ mit derselben Zielgewichtung trainiert werden kann. Die

Bellman-Gleichung beschreibt den zu erwartenden Nutzen unter der Voraussetzung, dass ausgehend vom Folgezustand stets optimale Aktionen ausgeführt werden[2]. Würde man $\hat{J}_i$ trainieren, aber vom Folgezustand aus die Aktionen nach einer beliebigen Zielgewichtung wählen, so wäre die Approximation für das Ziel i nicht gültig.

Aus diesem Grund wird für jedes Ziel eine eigene Prognose $\hat{J}_i(k+1)$ berechnet. Jede Prognose benötigt dazu eine eigene Zielgewichtung Π_i mit:

$$\begin{aligned}\Pi_1 &= [1,\ 0,\ \ldots,\ 0]\\ \Pi_2 &= [0,\ 1,\ \ldots,\ 0]\\ &\vdots\\ \Pi_L &= [0,\ 0,\ \ldots,\ 1]\end{aligned}$$

Mit den einzelnen Prognosen $\hat{J}_i(k+1)$ wird anschließend jeder Teil der Ausgabenmatrix separat trainiert:

$$W_{out_i}(k+1) = W_{out_i}(k) + L(k) \cdot (\hat{J}_i(k) - U_i(k) - \gamma \cdot \hat{J}_i(k+1)) \tag{4.5}$$

[2] siehe Kapitel 2.2.

5 Entscheidungsfindung

Ein Nachteil der bisher beschriebenen Architektur (zur Lösung von MOPs) ist, dass dem Agent von außen vorgegeben werden muss, zu welchem Zeitpunkt ein bestimmtes Ziel verfolgt werden soll. D.h. die Zielgewichtung $\Pi(k)$ muss für jeden Zeitschritt festgelegt oder zufällig generiert werden.

Die Idee ist nun, den Agent um eine Komponente zu erweitern, mit der er die Zielgewichtung $\Pi(k)$ selbst festlegen kann. Dazu benötigt der Agent neben dem äußeren Zustand $e(k)$ einen inneren Zustand. Der innere Zustand beschreibt die Bedürfnisse des Agenten und bietet eine Grundlage auf der entschieden werden kann, zu welchem Zeitpunkt ein bestimmtes Ziel verfolgt werden soll.

5.1 Entscheidungsfindung auf Basis des inneren Zustands

In [Malfaz and Salichs, 2010, Malfaz and Salichs, 2011, Salichs and Malfaz, 2006, Salichs and Malfaz, 2012] wird ein autonomer Agent beschrieben, der Entscheidungen auf Basis seines inneren und äußeren Zustands trifft. Der innere Zustand setzt sich aus Trieben, Motivationen und Emotionen zusammen. Abhängig von der Stärke der einzelnen Triebe entsteht eine bestimmte Motivation. Geleitet von der Motivation trifft der Agent wiederum Entscheidungen, die seine Triebe befriedigen oder weiter verstärken. Dies wirkt sich auf den emotionalen Zustand des Agenten aus.

Das Ziel des Agenten besteht darin, in einer einfachen diskreten Blockwelt zu überleben. Dazu muss er lernen, wie seine Entscheidungen den inneren Zustand beeinflussen. Die Umsetzung in dieser Arbeit orientiert sich an dem in [Malfaz and Salichs, 2010, Malfaz and Salichs, 2011, Salichs and Malfaz, 2006, Salichs and Malfaz, 2012] beschriebenen Agent. Die PSI Theorie [Dörner et al., 2002] dient ebenfalls als Inspiration. Allerdings steht weniger eine biologisch exakte Nachbildung von Trieben und Motivationen im Vordergrund, als vielmehr die technische Umsetzung der Entscheidungslogik innerhalb der RC-ADP-Architektur. Wegen des kontinuierlichen Zustandsraumes bedarf es anderer technischer Lösungen um die Entscheidungskomponente nachzubilden. Abbildung 5.1 zeigt die Architektur des Agenten inklusive der Entscheidungslogik.

5.1.1 Triebe als innerer Zustand

Der Agent trifft seine Entscheidungen auf Basis seines äußeren und inneren Zustands. Der innere Zustand wird durch eine bestimmte Anzahl von Trieben D_i und durch das Wohlbefinden Wb des Agenten festgelegt. In dieser Arbeit beschränkt sich die Anzahl an Trieben auf drei. Ein Trieb lässt sich als Vektor darstellen:

$$D_i = [D_i^s,\ D_i^p,\ D_i^g,\ D_i^l,\ D_i^t]$$

Der Vektoreintrag D_i^s beschreibt die Stärke des Triebs. Da ein Trieb nicht negativ oder beliebig groß werden kann gilt:

$$0 \leq D_i^s \leq 1$$

Die Priorität, die der Agent einem Trieb zuordnet, wird mit D_i^p bezeichnet. Der Einfachheit halber hat jeder Trieb dieselbe Priorität. Es gilt:

$$D_i^p = 1 \qquad \forall i.$$

Der Eintrag D_i^g (engl. *gain factor*) definiert, wie stark ein Trieb innerhalb eines Zeitschritts zunimmt, sofern er nicht befriedigt wird.

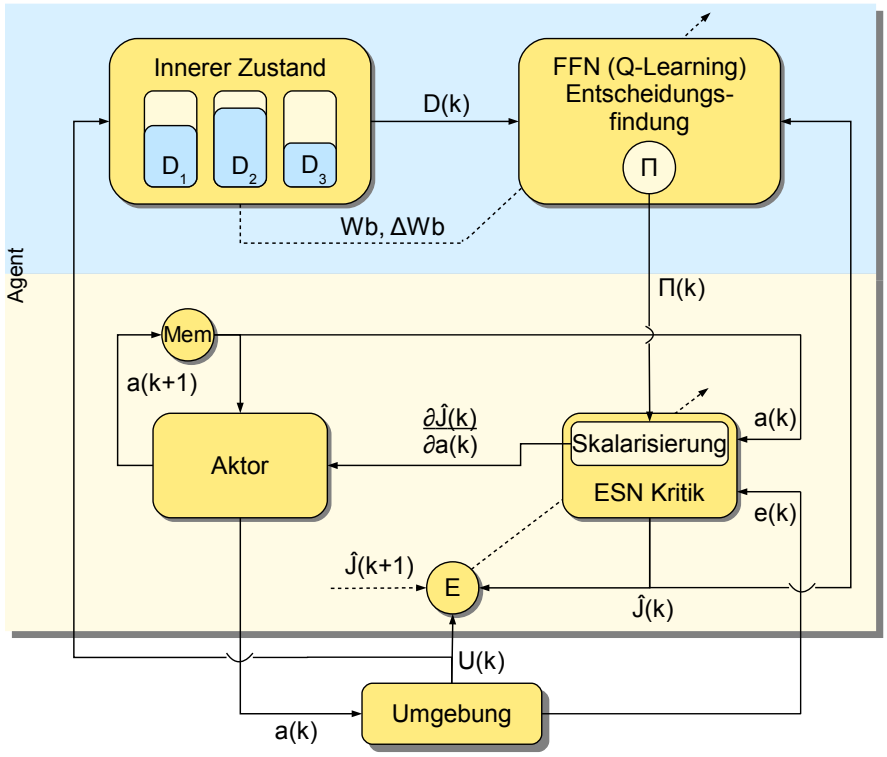


Abbildung 5.1: **Architektur des Agenten inklusive der Entscheidungslogik.** Um die Zielgewichtung nicht von außen vorgeben zu müssen, wird die Architektur um eine Entscheidungslogik erweitert. Diese ist im blau hinterlegten Bereich dargestellt. Die Entscheidungslogik bestimmt die Zielgewichtung in Abhängigkeit des äußeren und inneren Zustands. Der innere Zustand wird durch drei Triebe D_1, D_2 und D_3 beschrieben. Diese drücken die Bedürfnisse des Agenten zu einem bestimmten Zeitpunkt aus. Das Wohlbefinden Wb ergibt sich somit direkt aus den einzelnen Trieben. Mit *Q-learning* [Watkins and Dayan, 1992] ordnet der Agent jeder Zielgewichtung einen Wert zu. Diejenige Zielgewichtung, die im gegebenen Zustand den größten Wert erzielt, ist zudem am vielversprechendsten um das Wohlbefinden des Agenten zu optimieren.

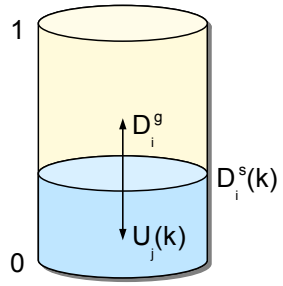


Abbildung 5.2: **Dynamik eines Triebs.** In jedem Zeitschritt k wächst der Trieb um den Zuwachsfaktor D_i^g. Die Stärke des Triebs $D_i^s(k)$ liegt im Intervall $[0,\ 1]$. Eine Bewertung $U_j(k)$ kann den Trieb beeinflussen. Welche Bewertung auf welchen Trieb wirkt, ist dem Agent zunächst unbekannt und muss von ihm gelernt werden.

Sobald ein Trieb vollständig befriedigt wurde, bleibt er eine bestimmte Zeit befriedigt. Diese Zeitspanne wird mit D_i^t (engl. *satisfaction time*) bezeichnet. Die Anzahl an Zeitschritten, die seit der letzten vollständigen Befriedigung vergangen sind, wird im Vektoreintrag D_i^l gespeichert.

Dynamik eines Triebs

In diesem Abschnitt wird die Dynamik eines Triebs erklärt. Abbildung 5.2 veranschaulicht wie sich ein Trieb in jedem Zeitschritt verändern kann. Formal lässt sich der Zuwachs eines Triebs auf folgende Art ausdrücken:

$$D_i^s(k+1) = \begin{cases} D_i^s(k) + D_i^g - u_j \cdot U_j(k) & \text{falls } D_i^l > D_i^t \\ 0 & \text{falls } D_i^l \leq D_i^t \end{cases} \tag{5.1}$$

Auf der einen Seite nimmt der Trieb pro Zeitschritt um den Zuwachsfaktor D_i^g zu. Auf der anderen Seite erhält der Agent nach jedem Zeitschritt eine Belohnung bzw. Bewertung. Abhängig von der Qualität der Bewertung wird der Trieb mehr oder weniger stark befriedigt. Die Bewertung $U_j(k)$ wird deshalb gewichtet mit u_j von $D_i^s(k)$ abgezogen.

Ein Trieb kann von mehreren Bewertungsfunktionen beeinflusst werden. Desweiteren kann eine Bewertungsfunktion auch auf mehrere Triebe wirken. Den genauen Zusammenhang muss der Agent lernen. Ihm ist zu Beginn unbekannt, welche Bewertung auf welche Weise einen bestimmten Trieb beeinflusst.

Da jeder Trieb im Intervall $[0,\ 1]$ liegt, muss die Stärke des Triebs eingegrenzt werden:

$$D_i^s(k+1) = \min\left(\max(D_i^s(k+1),\ 0),\ 1\right) \tag{5.2}$$

Führt dies dazu, dass die Stärke des Triebs auf 0 reduziert wird, so tritt eine vollständige Befriedigung ein. In diesem Fall werden die Anzahl an Zeitschritten seit der letzten vollständigen Befriedigung D_i^l des Triebs zurückgesetzt:

$$D_i^l(k+1) = \begin{cases} 0 & \text{falls } D_i^l(k) > D_i^t \ \wedge\ D_i^s(k+1) = 0 \\ D_i^l(k) + 1 & \text{sonst} \end{cases} \tag{5.3}$$

5.1.2 Bewertung des inneren Zustands

Die Bewertung des inneren Zustands leitet sich aus dem Wohlbefinden des Agenten ab. Das Wohlbefinden $Wb(k)$ zum Zeitpunkt k ergibt sich aus der Stärke der einzelnen Triebe:

$$\begin{aligned} Wb(k) &= Wb_{ideal} - \sum_i D_i^p \cdot D_i^s(k) \\ &= Wb_{ideal} - \sum_{i=1}^{3} D_i^s(k) \end{aligned} \tag{5.4}$$

Das optimale Wohlbefinden Wb_{ideal} hat der Agent, wenn alle seine Triebe vollständig befriedigt sind, d.h. wenn

$$D_i^s(k) = 0 \ \ \forall i$$

gilt. In dieser Arbeit wird das optimale Wohlbefinden folgendermaßen definiert:

$$Wb_{ideal} = 3$$

Ist mindestens einer der Triebe nicht vollständig befriedigt, so ist das Wohlbefinden $Wb(k)$ kleiner als Wb_{ideal}. In diesem Fall muss der Agent eine Entscheidung treffen, die auf den unbefriedigten Trieb optimal wirkt. Im nächsten Abschnitt wird erklärt, wie der Agent optimale Entscheidungen finden kann.

5.2 Finden optimaler Entscheidungen

Um einen Trieb befriedigen zu können, muss der Agent wissen, welche Belohnung bzw. Bewertungsfunktion U_j den Trieb beeinflusst. Außerdem muss er wissen, auf welche Art die jeweilige Bewertung den Trieb beeinflusst, d.h. ob sie den Trieb verstärkt oder befriedigt.

Erst mit diesem Wissen kann der Agent eine geeignete Zielgewichtung $\Pi(k)$ wählen. Um diesen Zusammenhang zu lernen, nutzt der Agent *Q-learning* [Watkins and Dayan, 1992] - eine Methode des bestärkenden Lernens.

5.2.1 Q-learning

Beim *Q-learning* wird jeder Aktion ein Wert (*Q-value*) zugeordnet. Der *Q-value* sagt aus, wie nützlich die Aktion im gegebenen Zustand auf lange Sicht ist. Er wird in dieser Arbeit folgendermaßen beschrieben:

$$Q_{\Pi_i}(k) = R(k) + \gamma \cdot \max_{\Pi_j \in \Pi}(Q_{\Pi}(k + t_{Episode})) \tag{5.5}$$

Der *Q-value* setzt sich aus dem direkten Nutzen $R(k)$ und dem bestmöglichen *Q-value* des Folgezustands zusammen. Der direkte Nutzen $R(k)$ ergibt sich (in dieser Arbeit) aus dem Wohlbefinden des Agenten. Darauf wird am Ende dieses Kapitels näher eingegangen.

Zielgewichtung als Aktion einer Episode

Als mögliche Aktionen fürs *Q-learning* dienen die Zielgewichtungen. Die Anzahl P an Zielgewichtungen wird dazu fest vorgegeben. D.h. der Agent erhält eine Menge von Zielgewichtungen:

$$\begin{aligned}
\Pi_1 &= [w_{11},\ w_{12},\ \ldots,\ w_{1L}] \\
\Pi_2 &= [w_{21},\ w_{22},\ \ldots,\ w_{2L}] \\
&\vdots \\
\Pi_P &= [w_{P1},\ w_{P2},\ \ldots,\ w_{PL}]
\end{aligned}$$

Anschließend wählt er eine der vorgegebenen Zielgewichtungen und bestimmt deren *Q-value*. Dazu teilt der Agent die Zeit in Episoden ein. Jede Episode besteht aus einer bestimmten Anzahl $t_{Episode}$ von Zeitschritten. Innerhalb einer Episode wird die gewählte Zielgewichtung beibehalten. Erst zu Beginn einer neuen Episode kann der Agent eine andere Zielgewichtung wählen.

Der Vorteil der Episoden liegt darin, dass sich aussagekräftigere *Q-values* ergeben. Innerhalb eines einzelnen Zeitschritts ändert sich der innere und äußere Zustand nur geringfügig. Innerhalb mehrerer Zeitschritte sind jedoch größere Unterschiede bemerkbar.

Q-learning mit neuronalem Netz

Um die *Q-values* zu lernen, wird ein zweischichtiges FFN genutzt. Abbildung 5.3 zeigt das in dieser Arbeit verwendete FFN.

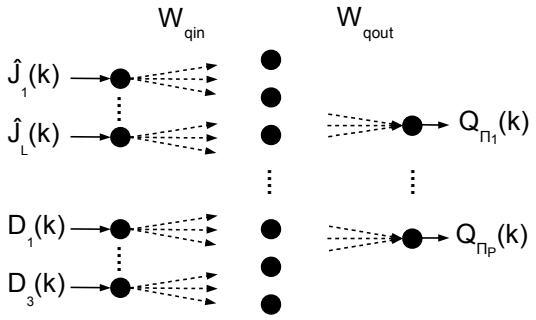


Abbildung 5.3: **FFN zur Approximation der *Q-values*.** Der Nutzen einer Zielgewichtung Π_i zum Zeitpunkt k wird über den *Q-value* $Q_{\Pi_i}(k)$ beschrieben. Ein zweischichtiges FFN approximiert die *Q-values* aller gegebenen Zielgewichtungen - abhängig vom inneren Zustand $D(k)$ und vom äußeren Zustand (repräsentiert durch $\hat{J}(k)$).

Der äußere Zustand ergibt sich aus den Vorhersagen $\hat{J}_i(k)$, während der innere Zustand durch die Triebe $D_i(k)$ beschrieben wird. Der Grund, weshalb der äußere Zustand ebenfalls berücksichtigt wird, ist dass der Agent bei mehreren unbefriedigten Trieben zusätzlich die Vorhersage $\hat{J}$ in seine Entscheidung mit einbeziehen kann. Befindet sich beispielsweise der Agent in der Nähe einer Belohnung, so kann es vorteilhaft sein, zuerst diese Belohnung anzusteuern und erst im Anschluss die restlichen.

Nachdem das Netz trainiert wurde, kann der Agent die Zustandsbeschreibung in das Netz leiten. Anhand der Netzausgabe kann er diejenige Zielgewichtung Π_i auswählen, die den größten Nutzen verspricht. Auf diese Weise optimiert er sein Wohlbefinden. Zuvor muss jedoch das Netz trainiert werden.

5.2.2 Training der Entscheidungslogik

Das Training erfolgt - wie bisher - online nach dem RLS-Verfahren[1]. Die *Q-values* werden nach folgender Gleichung adaptiert

$$Q_{\Pi_i}(k) = (1-\beta) \cdot Q_{\Pi_i}(k) + \beta \cdot \left(R(k) + \gamma \cdot \max_{\Pi_j \in \Pi} \left(Q_{\Pi}(k + t_{Episode}) \right) \right) \tag{5.6}$$

wobei $\beta \in [0,\ 1]$ die Lernrate bzw. Vergessensrate bezeichnet.

Exploration-Exploitation Dilemma

Beim Training der Entscheidungslogik muss das in Kapitel 2.1 beschriebene *exploration-exploitation dilemma* berücksichtigt werden. Der Agent sollte deshalb nicht in jeder Episode die bestmögliche Zielgewichtung wählen. Stattdessen wählt er die Zielgewichtung nach folgender Gleichung:

$$\Pi(k) = \begin{cases} \arg\max_{\Pi_i \in \Pi} \left(Q_{\Pi}(k) \right) \\ \Pi_r \end{cases}$$

wobei Π_r eine zufällig gewählte Zielgewichtung aus der Menge Π bezeichnet. Wenn der Agent Π_r wählt, so exploriert er. Andernfalls nutzt er das bisher gewonnene Wissen aus. Das Verhältnis zwischen dem Explorieren und dem Ausnutzen des Wissens wird über einen Faktor geregelt. Dieser Faktor kann im Verlauf des Trainings angepasst werden. Näheres dazu findet sich in Kapitel 6.

[1] siehe Kapitel 2.4.

Wohlbefinden als direkter Nutzen

Der direkte Nutzen $R(k)$ einer Zielgewichtung leitet sich aus dem Wohlbefinden Wb ab. Es gibt zwei Möglichkeiten $R(k)$ zu definieren:

1. $R(k) := Wb(k + t_{Episode})$
2. $R(k) := \Delta Wb(k + t_{Episode})$
   $= Wb(k + t_{Episode}) - Wb(k)$

Die erste Möglichkeit betrachtet das neue Wohlbefinden als direkten Nutzen. Die zweite Möglichkeit basiert auf dem Unterschied zwischen dem alten und dem neuen Wohlbefinden. D.h. der direkte Nutzen kann im zweiten Fall sowohl positive als auch negative Werte annehmen. Wie sich die beiden Möglichkeiten auf den Lernerfolg des Agenten auswirken, wird in Kapitel 6 untersucht.

6 Ergebnisse und Diskussion

In diesem Kapitel wird das Verhalten des Agenten untersucht. Dazu werden mehrere Versuche durchgeführt. Außerdem werden die Ergebnisse der einzelnen Versuche diskutiert. Zunächst folgt eine Beschreibung der Versuchsdurchführung.

6.1 Versuchsdurchführung

Der Agent beginnt jeden Versuch von einer beliebig definierbaren Startposition innerhalb der virtuellen Welt. Nach einer kurzen Initialisierungsphase[1] startet der Agent das Training. Dabei wird die Zeit in Episoden unterteilt. Jede Episode besteht aus einer frei wählbaren Anzahl an Zeitschritten. Dies wird durch die Episodenlänge $t_{Episode}$ festgelegt. Der Agent behält während einer Episode eine festgelegte Zielgewichtung bei. Nach jeder Episode kann er eine neue Zielgewichtung wählen.

Verlässt der Agent während einer Episode die virtuelle Welt, muss von außen eingegriffen werden. Es bieten sich drei Behandlungsmöglichkeiten an:

1. Die Episode wird abgebrochen.

2. Der Agent darf bis zum Ende der Episode außerhalb der Welt weiterfahren.

3. Die Orientierung des Agenten wird neu ausgerichtet (auf den Mittelpunkt der Welt).

[1] Das Reservoir benötigt - abhängig vom spektralen Radius - einige Zeitschritte um sich einzuschwingen.

Bei der ersten Möglichkeit muss für die nächste Episode eine neue Startposition festgelegt werden. Bei der zweiten Möglichkeit ist dies nur erforderlich, falls der Agent nicht von selbst in die virtuelle Welt zurückfindet. Die dritte Möglichkeit hat den Nachteil, dass sich die Richtungsänderung negativ auf den Reservoirzustand auswirken kann (bei Aktionsdefinition ω). Für das Training wird deshalb die Behandlungsmöglichkeit 1 gewählt. Bei den Testläufen wird - abhängig davon was gezeigt werden soll - entweder Behandlungsmöglichkeit 1 oder 2 gewählt.

6.2 Szenario 1

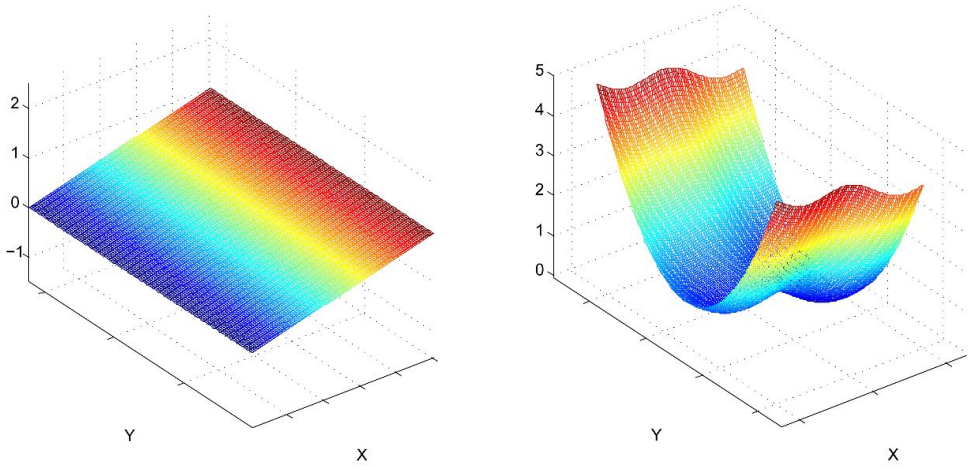
In diesem Szenario soll das Verhalten des Agenten anhand eines aus der Literatur bekannten Beispiels gezeigt werden. In [Yun et al., 2004, Linder and Lindkvist, 2011] wird folgendes MOP untersucht:

$$
\begin{aligned}
&Z_1(x_1, x_2) = x_1 \\
&Z_2(x_1, x_2) = 1 + x_2^2 - x_1 - 0.2 \cdot sin(3\pi \cdot x_1) \\
&\text{mit } 0 \leq x_1 \leq 1 \text{ und } -2 \leq x_2 \leq 2.
\end{aligned}
$$

Dabei sollen die beiden Ziele Z_1 und Z_2 minimiert werden. Skaliert auf die Dimensionen der virtuellen Welt ergeben sich die beiden Bewertungsfunktionen

$$
\begin{aligned}
U_1(p) &= \frac{1}{100} \cdot x \\
U_2(p) &= 1 + \left(\frac{4}{160} \cdot y - 2 \right)^2 - \frac{1}{100} \cdot x - 0.2 \cdot sin\left(3\pi \cdot \frac{1}{100} \cdot x \right)
\end{aligned}
$$

wobei $p = [x,\ y]$ die Position beschreibt. Das Ziel des Agenten besteht darin, die beiden Bewertungsfunktionen U_1 und U_2 zu minimieren. Diese sind in Abbildung 6.1 dreidimensional dargestellt. Bei U_1 handelt es sich um eine lineare Funktion, während U_2 nicht-linear ist. Die Varianz in den Steigungen von U_1 und U_2 erschwert die Navigation des Agenten. Wegen der dynamischen Anpassungsrate η gelingt ihm dennoch eine korrekte Navigation, was unter anderem im folgenden Versuch gezeigt wird.



(a) Bewertungsfunktion U_1 (b) Bewertungsfunktion U_2

Abbildung 6.1: **Bewertungsfunktionen aus Szenario 1.** Dargestellt sind die beiden Bewertungsfunktionen dieses Szenarios. Sie ordnen jeder Position $p = [x,\ y]$ innerhalb der virtuellen Welt eine Bewertung zu.

6.2.1 Versuch 1

Bei diesem Versuch soll die Fähigkeit des Agenten zur Lösung multikriterieller Optimierungsprobleme untersucht werden. Der Versuch findet für beide Aktionsdefinitionen φ und ω statt[2]. Um einen möglichst guten Vergleich zu erhalten, werden dieselben Rahmenbedingungen gesetzt. Neben den Netzparametern und den Trainingsparametern betrifft dies auch den *random-seed*. Dieser wird mit 1 initialisiert. Die Wahl der Parameter dieses Versuchs zeigen die Tabellen 6.1 und 6.2.

Tabelle 6.1: Netzparameter Szenario 1 Versuch 1

Parameter	**Bezeichnung**	**Wert**
Anzahl Reservoirneuronen	N	50
Anzahl Ausleseneuronen	M	400
Spektraler Radius	α	0.4
Reservoir-Verbindungsdichte	c_{res}	0.3
Eingabe-Skalierung	r_{in}	0.65

[2] siehe Kapitel 3.

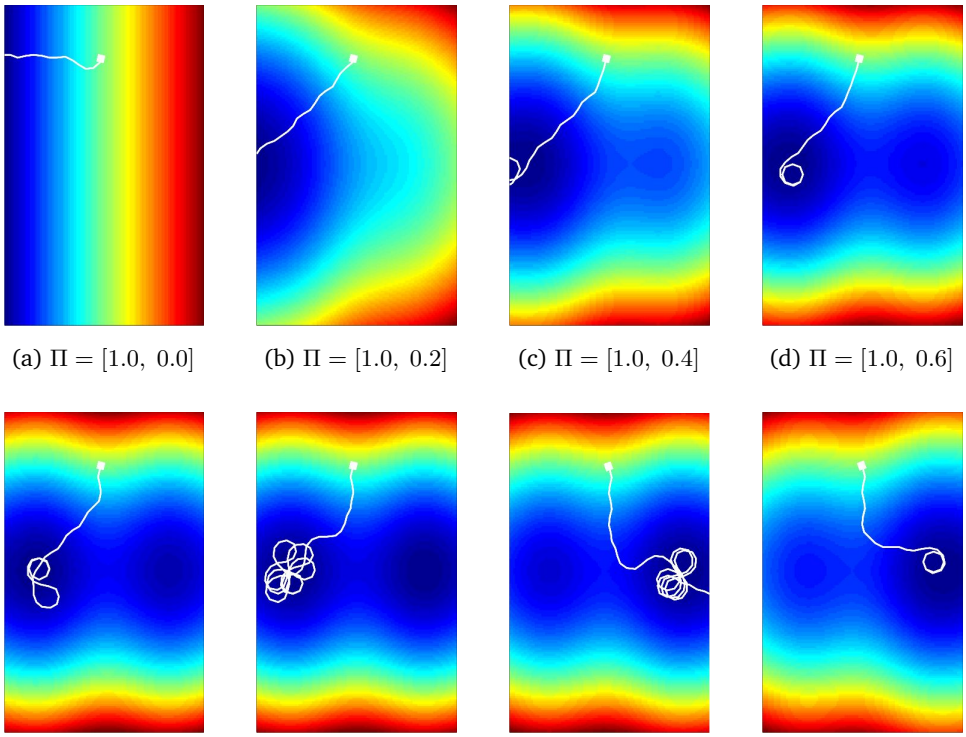
Tabelle 6.2: Trainingsparameter Szenario 1 Versuch 1

Parameter	**Bezeichnung**	**Wert**
Episodenlänge	$t_{Episode}$	100
Anzahl Trainingsepisoden	-	500
Initialisierungslänge	-	3
Discount-Faktor	γ	0.0

Da nur eine geringe Rückkopplung notwendig ist, um eine eindeutige Zustandsbeschreibung zu erhalten, wird der spektrale Radius sehr klein gewählt. Außerdem hat dies den Vorteil, dass die Initialisierungsphase kürzer ausfallen kann. Die Trainingsdauer wird mit 500 Episoden hoch angesetzt. Das unterschiedliche Verhalten des Agenten bei den beiden Aktionsdefinitionen φ und ω soll nicht durch ein unausreichendes Training bedingt sein. Da die Bewertungsfunktionen U_1 und U_2 überall in der Welt einen Gradienten definieren, kann bei diesem Szenario als *discount*-Faktor 0 gewählt werden. Die restlichen Parameter wurden zuvor experimentell bestimmt.

Die Abbildung 6.2 zeigt 8 Testläufe mit unterschiedlichen Zielgewichtungen Π und Aktionsdefinition φ (Orientierung). Die einzelnen Testläufe finden nach den 500 Trainingsepisoden statt. Der Agent startet von Position $[50,\ 136]$ und ist nach unten (1.5π) ausgerichtet. Es ist gut zu erkennen, dass es dem Agent gelingt, die skalarisierte Bewertungsfunktion zu minimieren. Da bei den Testläufen 1-3 das Minimum links außerhalb der virtuellen Welt liegt, muss der Agent die Welt verlassen. Warum er dies auch am Ende des 7. Testlaufs tut, kann nicht genau erklärt werden. Eine mögliche Ursache ist, dass die Approximation $\hat{J}$ für den gegebenen Reservoirzustand noch nicht ausreichend exakt ist.

Besonders interessant ist der 6. Testlauf. Bei diesem wird die Zielgewichtung so gewählt, dass die skalarisierte Bewertungsfunktion zwei gleichwertige Optima aufweist. Exakt in der Mitte der virtuellen Welt wird die Bewertungsfunktion gespiegelt. Da der Agent in der Mitte startet und nach unten ausgerichtet ist, sind die beiden Alternativen nach links oder rechts zu fahren gleich vielversprechend. Die Ableitung $\frac{\partial \hat{J}}{\partial a}$ sollte im Idealfall 0 ergeben und der Agent folglich geradeaus fahren. Der Agent entscheidet sich jedoch, nach links zu fahren. Drei Faktoren spielen dabei eine Rolle: Erstens könnte



(a) $\Pi = [1.0,\ 0.0]$ (b) $\Pi = [1.0,\ 0.2]$ (c) $\Pi = [1.0,\ 0.4]$ (d) $\Pi = [1.0,\ 0.6]$

(e) $\Pi = [1.0,\ 0.8]$ (f) $\Pi = [1.0,\ 1.0]$ (g) $\Pi = [0.5,\ 1.0]$ (h) $\Pi = [0.0,\ 1.0]$

Abbildung 6.2: **Testläufe mit Aktionsdefinition φ.** Zu sehen ist das Verhalten des Agenten bei unterschiedlicher Zielgewichtung Π. Als Aktionsdefinition wurde die Orientierung φ gewählt. Der weisse Punkt zeigt die Startposition des Agenten. Die weisse Linie stellt den Pfad des Agenten während des Testlaufs dar. Zuvor hat der Agent 500 Trainingsepisoden absolviert. Farbig hinterlegt ist die mit Π skalarisierte Bewertungsfunktion U.

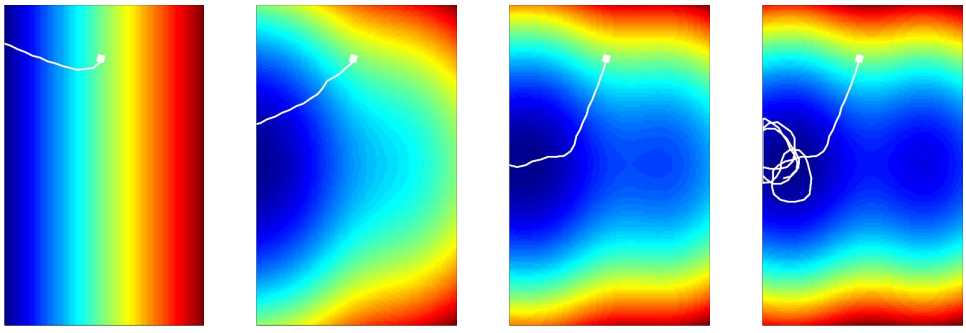
die Approximation nicht ausreichend gut sein. Zweitens könnten die Reservoirzustände der Initialisierungsphase nicht trainiert worden sein. Der dritte Faktor ist besonders interessant zu erwähnen und wird in Abbildung 6.7 veranschaulicht. Die Signum-Funktion, welche bei der Berechnung der dynamischen Anpassungsrate beteiligt ist, liefert für $\frac{\partial \hat{J}}{\partial a} < 0$ den Wert -1. Da die Ableitung $\frac{\partial \hat{J}}{\partial a}$ im ersten Zeitschritt tatsächlich minimal kleiner als 0 ist, ergibt sich eine negative Anpassungsrate η. Bei $\eta < 0$ fährt der Agent - aus seiner Sicht - nach rechts, wie in Kapitel 3 erklärt wurde. Dies ist ein weiterer Vorteil der dynamischen Anpassungsrate. Bei einer statischen Anpassungsrate würde der Agent im Idealfall (bei $\frac{\partial \hat{J}}{\partial a} = 0$) geradeaus fahren und keines der beiden Optima finden.

In Abbildung 6.3 sind die gleichen Testläufe bei Aktionsdefinition ω (Rotationsgeschwindigkeit) zu sehen. Hier hat es der Agent erheblich schwerer das Optimum zu finden. Dennoch gelingt es ihm ansatzweise. Der Agent verlässt jedoch häufig die Welt. Die Zustandsbeschreibungen bei Aktionsdefinition ω sind nicht so eindeutig wie bei Aktionsdefinition φ. Da die skalarisierten Bewertungsfunktionen stets das Optimum am Rand der virtuellen Welt haben und dort die Unterschiede zwischen den Zustandsbewertungen sehr gering sind, ist es nachvollziehbar, dass der Agent die Welt verlässt.

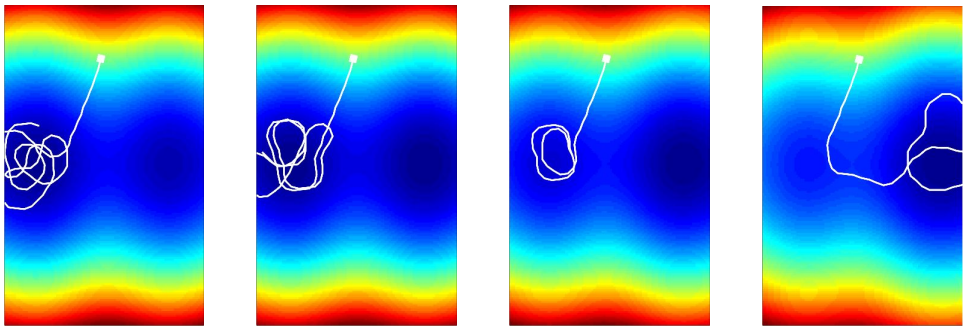
Die Wahl der Netzparameter hat einen großen Einfluss auf die Zustände im Reservoir und damit auf das Verhalten des Agenten. Ein bessere Wahl der Parameter könnte auch bei Aktionsdefinition ω zu einem besseren Ergebnis führen.

In den Abbildungen 6.4, 6.5 und 6.6 sind die einzelnen Bewertungen zu sehen, die der Agent während der Testläufe erhalten hat. Da die Bewertungen außerhalb der virtuellen Welt nicht gültig sind, sind sie nur bis zu dem Zeitpunkt, an dem der Agent erstmals die Welt verlässt, eingetragen.

Es ist gut sichtbar, dass der Agent die Bewertungen so optimiert, wie es die entsprechende Zielgewichtung vorgibt. Bei Testlauf 1 optimiert der Agent nur das 1. Ziel und bei Testlauf 8 nur das 2. Ziel. Bei den restlichen Testläufen versucht er beide Ziele bzw. Bewertungen zu optimieren. Da bei Testlauf 3 die Bewertungsfunktion U_2 stärker gewichtet wird als bei Testlauf 2, optimiert der Agent das 2. Ziel auch stärker.

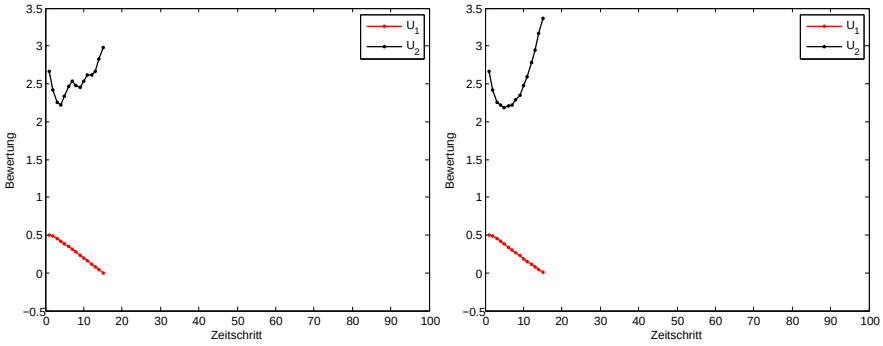


(a) $\Pi = [1.0,\ 0.0]$ (b) $\Pi = [1.0,\ 0.2]$ (c) $\Pi = [1.0,\ 0.4]$ (d) $\Pi = [1.0,\ 0.6]$

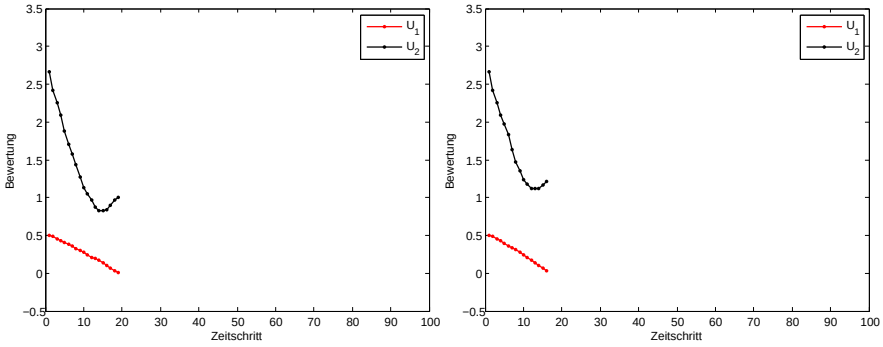


(e) $\Pi = [1.0,\ 0.8]$ (f) $\Pi = [1.0,\ 1.0]$ (g) $\Pi = [0.5,\ 1.0]$ (h) $\Pi = [0.0,\ 1.0]$

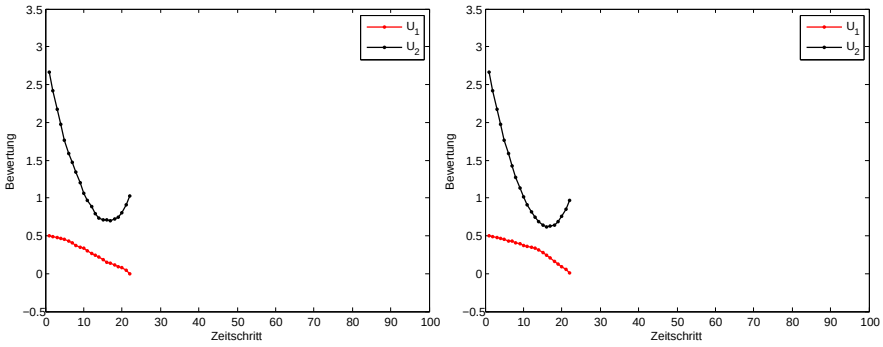
Abbildung 6.3: **Testläufe mit Aktionsdefinition ω.** Bei diesen Testläufen wird als Aktionsdefinition die Rotationsgeschwindigkeit ω benutzt. Es ist deutlich zu erkennen, dass die Navigation des Agenten weniger präzise ist als bei Aktionsdefinition φ.



(a) Testlauf 1 mit Zielgewichtung $\Pi = [1.0,\ 0.0]$

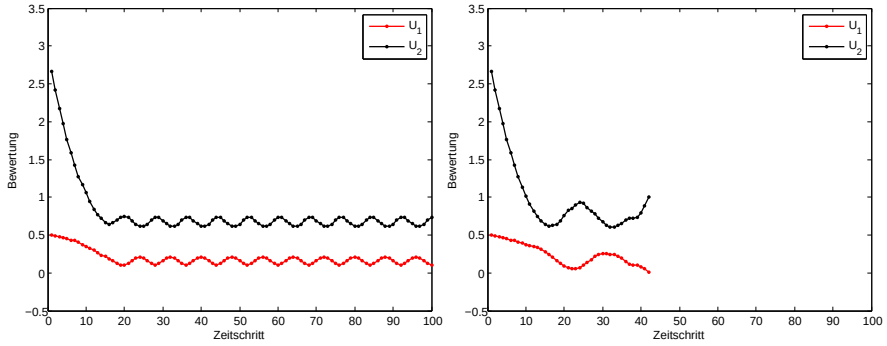


(b) Testlauf 2 mit Zielgewichtung $\Pi = [1.0,\ 0.2]$

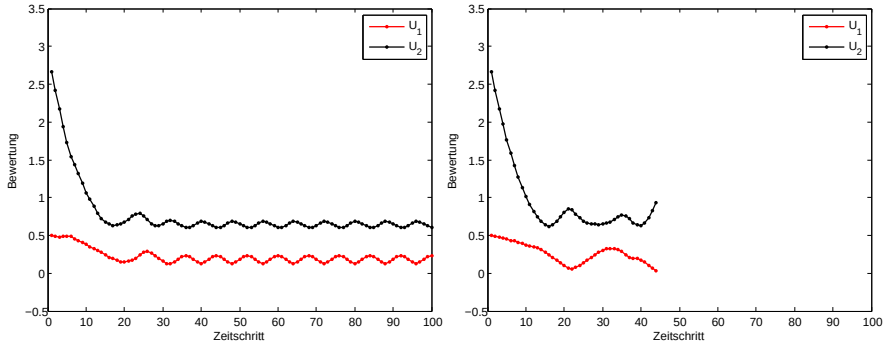


(c) Testlauf 3 mit Zielgewichtung $\Pi = [1.0,\ 0.4]$

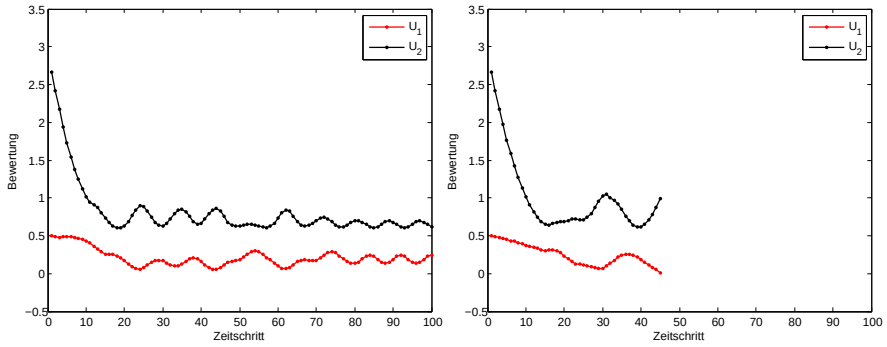
Abbildung 6.4: **Bewertungen während der Testläufe 1, 2 und 3.** Dargestellt sind die Bewertungen, die der Agent während jedes Testlaufs erhalten hat - bis zu dem Zeitpunkt wo er die Welt verlässt. Links sind die Bewertungen bei Aktionsdefinition φ zu sehen und rechts die Bewertungen bei Aktionsdefinition ω.



(a) Testlauf 4 mit Zielgewichtung $\Pi = [1.0,\ 0.6]$

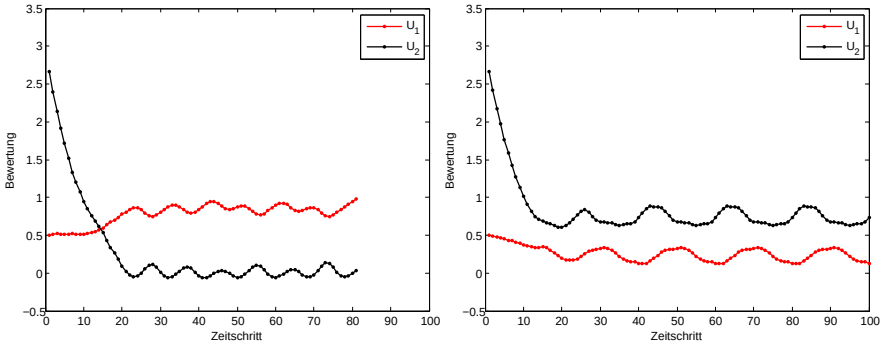


(b) Testlauf 5 mit Zielgewichtung $\Pi = [1.0,\ 0.8]$

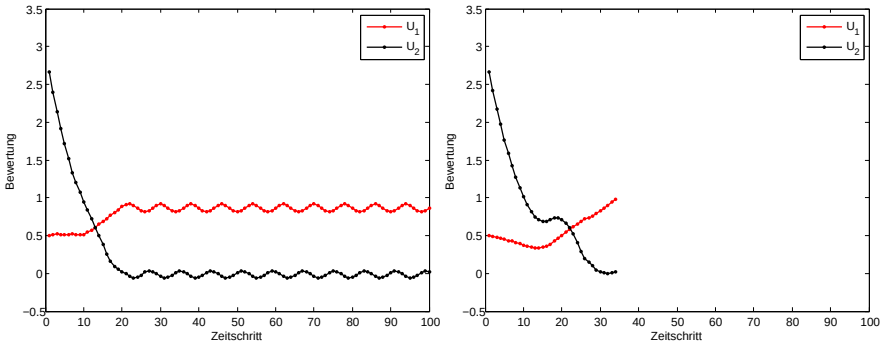


(c) Testlauf 6 mit Zielgewichtung $\Pi = [1.0,\ 1.0]$

Abbildung 6.5: **Bewertungen während der Testläufe 4, 5 und 6.** Dargestellt sind die Bewertungen, die der Agent während jedes Testlaufs erhalten hat - bis zu dem Zeitpunkt wo er die Welt verlässt. Links sind die Bewertungen bei Aktionsdefinition φ zu sehen und rechts die Bewertungen bei Aktionsdefinition ω.



(a) Testlauf 7 mit Zielgewichtung $\Pi = [0.5,\ 1.0]$



(b) Testlauf 8 mit Zielgewichtung $\Pi = [0.0,\ 1.0]$

Abbildung 6.6: **Bewertungen während der Testläufe 7 und 8.** Dargestellt sind die Bewertungen, die der Agent während jedes Testlaufs erhalten hat - bis zu dem Zeitpunkt wo er die Welt verlässt. Links sind die Bewertungen bei Aktionsdefinition φ zu sehen und rechts die Bewertungen bei Aktionsdefinition ω.

Beim 7. Testlauf ist gut zu erkennen, dass der Agent bei Aktionsdefinition φ das globale Optimum findet, während er bei Aktionsdefinition ω nur ein lokales Optimum findet. Hier würde ein Training mit *discount*-Faktor > 0 hilfreich sein.

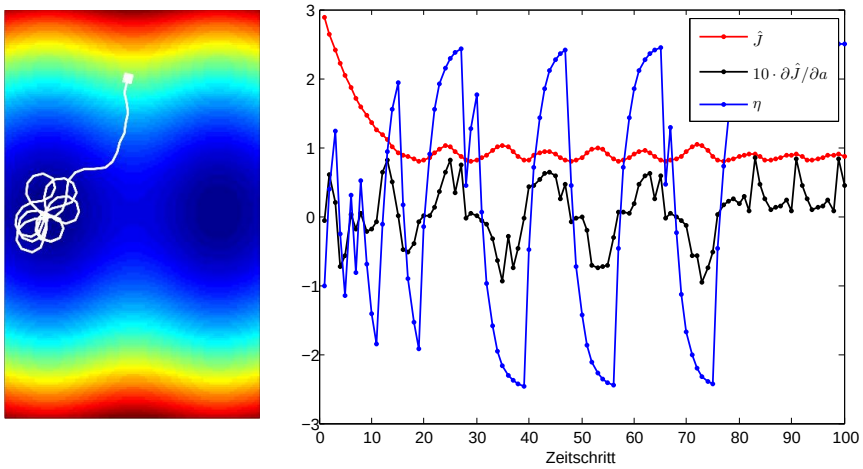
Außerdem sind bei dieser Darstellung die Kreisfahrten des Agenten zu sehen. Sobald die Bewertungen sinusförmige Schwingungen aufweisen, kreist der Agent um ein Optimum.

Die beiden Abbildungen 6.7 und 6.8 veranschaulichen, wie der Agent die auszuführenden Aktionen bestimmt. Er nutzt dazu die Ableitung $\frac{\partial J(\hat{k})}{\partial a(k)}$ (welche zur besseren Darstellung mit 10 multipliziert wird) und berechnet damit die dynamische Anpassungsrate[3] $\eta(k)$. Wie bei Aktionsdefinition ω zu sehen ist, kann die Ableitung stark unterschiedliche Werte annehmen. Dies zeigt ein Vergleich zwischen Zeitschritt 1 und den restlichen Zeitschritten. Der Agent minimiert durch die Wahl der Aktionen die Ableitung $\frac{\partial J(\hat{k})}{\partial a(k)}$, weshalb sie ab Zeitschritt 2 stets sehr gering ist.

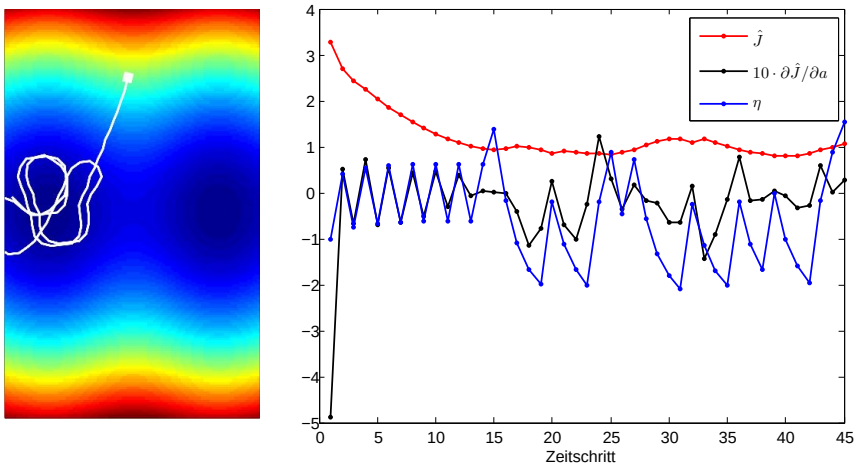
Außerdem ist in den beiden Abbildungen 6.7 und 6.8 zu erkennen, dass der Agent für $\eta < 0$ - aus seiner Sicht - nach rechts fährt, während er für $\eta > 0$ nach links fährt. Oszilliert die dynamische Anpassungsrate η um den Wert 0, so fährt der Agent geradeaus.

Abbildung 6.9 zeigt die Einzelbewertungen U_1 und U_2 beider Ziele bzw. Bewertungsfunktionen während des 7. Testlaufs. Sie beziehen sich jeweils auf die aktuelle Position des Agenten. Zum Vergleich sind zudem die Approximationen $\hat{J}_1$ und $\hat{J}_2$ dargestellt. Da als *discount*-Faktor 0 gewählt wurde, handelt es sich um eine Ein-Schritt-Vorhersage. Dies ist in der Abbildung gut zu erkennen, denn $\hat{J}_1$ und $\hat{J}_2$ haben eine hohe Übereinstimmung mit den Bewertungen U_1 und U_2 der Folgeposition.

[3] siehe Kapitel 3.

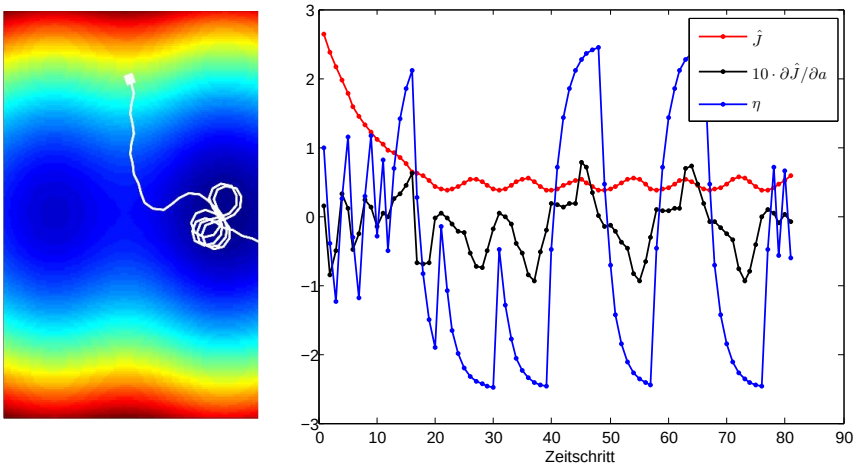


(a) Testlauf 6 mit Aktionsdefinition φ.

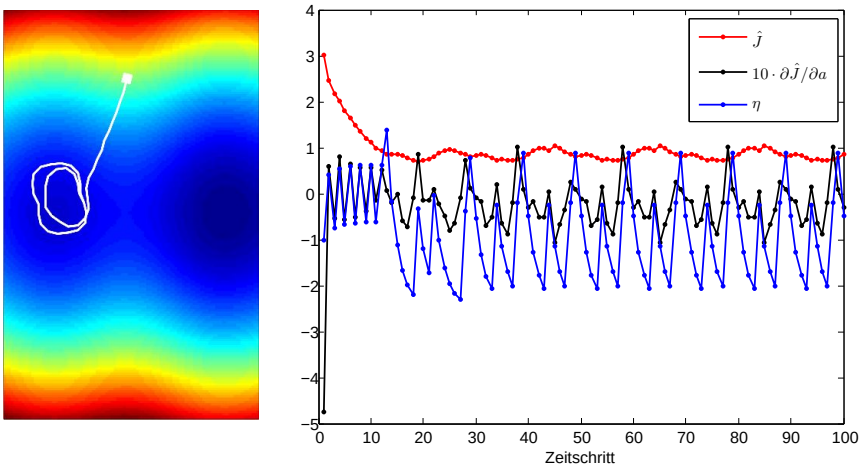


(b) Testlauf 6 mit Aktionsdefinition ω.

Abbildung 6.7: **Aktionsberechnung mit dynamischer Anpassungsrate (Testlauf 6).** Links ist Testlauf 6 bei beiden Aktionsdefinitionen dargestellt. Auf der rechten Seite erkennt man, wie der Aktor mit Hilfe der dynamischen Anpassungsrate η erfolgreich die Ableitung $\partial\hat{J}/\partial a$ (zwecks besserer Darstellung mit Faktor 10 multipliziert) minimieren kann. Die Approximation $\hat{J}$ zeigt bis zum 10. Zeitschritt starke Änderungen, danach sind die Änderungen pro Zeitschritt gering. Unten hat die Ableitung im 1. Zeitschritt einen sehr großen Betrag obwohl er 0 sein sollte. Dies lässt sich auf einen Approximationsfehler des Initialzustands zurückführen, da die Initial-Aktion nicht von der Soll-Aktion abweicht.

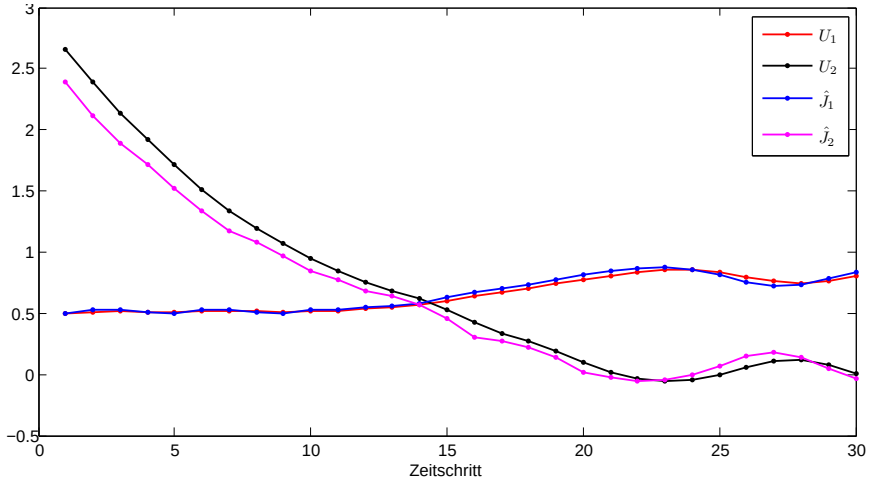


(a) Testlauf 7 mit Aktionsdefinition φ.

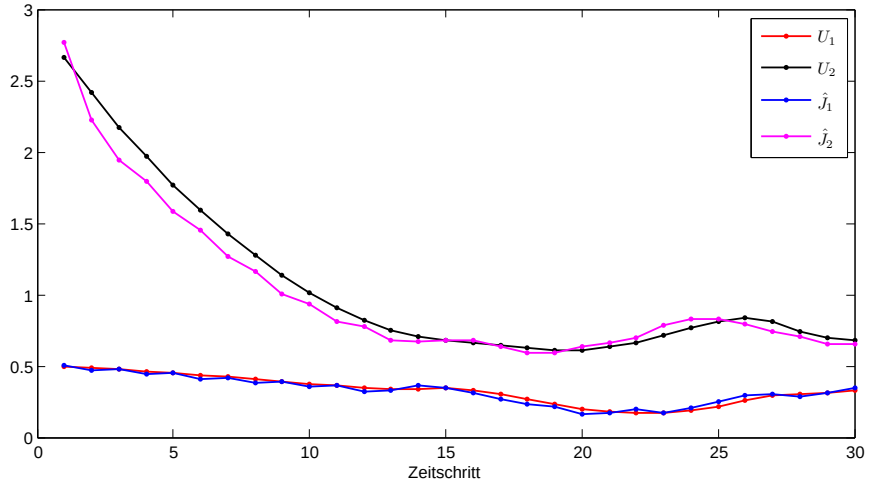


(b) Testlauf 7 mit Aktionsdefinition ω.

Abbildung 6.8: **Aktionsberechnung mit dynamischer Anpassungsrate (Testlauf 7).** Bei Testlauf 7 zeigt die Approximation $\hat{J}$ bis zum 13. Zeitschritt starke Änderungen, danach sind die Änderungen pro Zeitschritt gering. Unten ist η meist negativ, d.h. der Agent fährt Rechtskurven. Oben fährt der Agent sowohl Links- als auch Rechtskurven. Bis Zeitschritt 13 fährt er geradeaus, was sich in einer Oszillation um den Wert 0 bemerkbar macht.



(a) Aktionsdefinition φ.



(b) Aktionsdefinition ω.

Abbildung 6.9: **Vergleich der aktuellen Bewertung mit der Vorhersage (Testlauf 7).** Dargestellt sind die Bewertungen U_1 und U_2 während des 7. Testlaufs. Sie beziehen sich auf die aktuelle Position des Agenten. Zum Vergleich sind außerdem die Vorhersagen $\hat{J}_1$ und $\hat{J}_2$ zu sehen. Da als *discount*-Faktor 0 gewählt wurde, fließt nur die Bewertung des direkten Folgezustands in $\hat{J}$ ein. Es handelt sich um eine Ein-Schritt-Vorhersage. Dies ist nicht ausreichend, um stets das globale Optimum zu finden. Bei Aktionsdefinition φ findet der Agent das globale Optimum, bei Aktionsdefinition ω hingegen nur ein lokales Optimum.

6.2.2 Versuch 2

In diesem Versuch soll gezeigt werden, wie der Agent mittels der Skalarisierung (Zielgewichtung) eine Pareto-Optimierung durchführt. Dabei kann nicht garantiert werden, dass Pareto-optimale Lösungen gefunden werden. Jedoch können die gefundenen, nichtdominierten Lösungen mit den Pareto-optimalen Lösungen aus [Yun et al., 2004, Linder and Lindkvist, 2011] verglichen werden.

Tabelle 6.3: Netzparameter Szenario 1 Versuch 2

Parameter	**Bezeichnung**	**Wert**
Anzahl Reservoirneuronen	N	50
Anzahl Ausleseneuronen	M	400
Spektraler Radius	α	0.4
Reservoir-Verbindungsdichte	c_{res}	0.3
Eingabe-Skalierung	r_{in}	0.65

Die Parameter dieses Versuchs sind identisch mit den Parametern aus Versuch 1 und in den Tabellen 6.3 und 6.4 zusammengefasst.

Tabelle 6.4: Trainingsparameter Szenario 1 Versuch 2

Parameter	**Bezeichnung**	**Wert**
Episodenlänge	$t_{Episode}$	100
Anzahl Trainingsepisoden	-	500
Initialisierungslänge	-	3
Discount-Faktor	γ	0.0

Vor der Pareto-Optimierung absolviert der Agent die 500 Trainingsepisoden. Anschließend finden die Testepisoden der Pareto-Optimierung statt. Jede Testepisode beginnt auf demselben Wissensstand. Der Agent startet von zufällig generierten Startpositionen $[x,\ y]$ wobei $35 \leq x \leq 65$ und $y = 112$ gilt. Mit $y = 112$ soll verhindert werden, dass der Agent bereits auf einem Pareto-Optimum startet. Mit $35 \leq x \leq 65$ wird bezweckt, dass der Agent bei gleicher Zielgewichtung verschiedene Optima erreichen kann.

Für jede Testepisode wird eine zufällige Zielgewichtung $\Pi = [r_1,\ r_2]$ generiert. Dabei bezeichnen r_1 und r_2 gleichverteilte Zufallszahlen im Bereich $[0,\ 1]$. Wenn der

Agent die virtuelle Welt verlässt, wird die Testepisode nach Behandlungsmöglichkeit 1 abgebrochen.

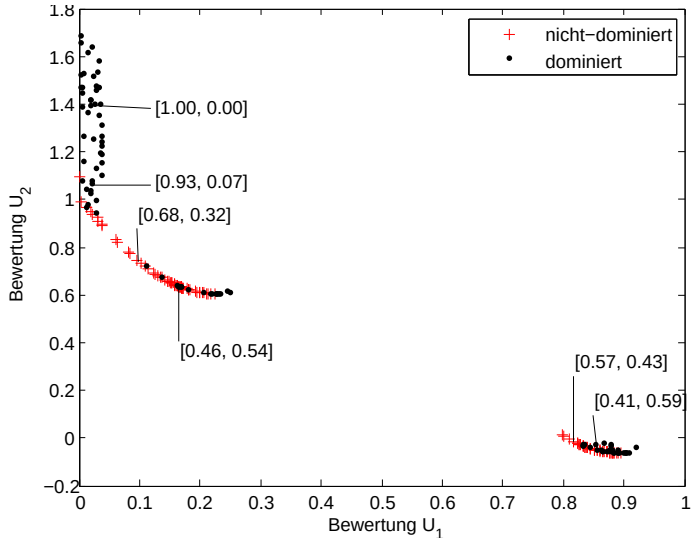
Abbildung 6.10 zeigt das Ergebnis der Pareto-Optimierung für beide Aktionsdefinitionen φ und ω. Dabei wird als Ergebnis jeweils die beste Bewertung während einer Episode dargestellt. Die beste Bewertung ist diejenige, die skalarisiert mit der Zielgewichtung den kleinsten Wert erzielt.

Das Ergebnis deckt die Erwartungen. Für Zielgewichtungen bei denen Ziel 2 kaum Priorität hat, fährt der Agent nach links und auf direktem Weg aus der virtuellen Welt. Dadurch ergeben sich viele dominierte Lösungen, die sich links oben im Diagramm wiederfinden. Für Zielgewichtungen, bei denen Ziel 2 deutlich mehr Priorität als Ziel 1 hat, fährt der Agent meist nach rechts. Bei allen anderen Zielgewichtungen erreicht der Agent eines der Optima.

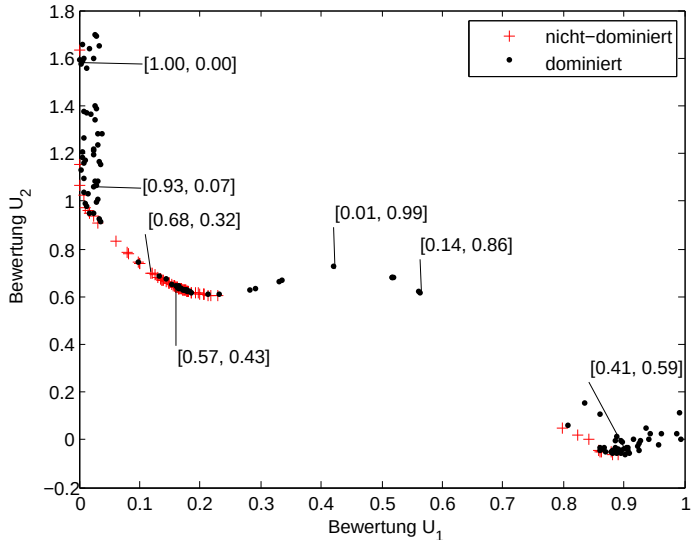
Bei Aktionsdefinition ω zeigt sich desweiteren, dass der Agent die virtuelle Welt häufig nach rechts verlässt. Dieses Verhalten wurde bereits in Versuch 1 beobachtet und diskutiert. Außerdem findet er dominierte Lösungen im mittleren Bereich. Dies lässt sich ebenfalls auf eine ungenauere Zustandsbeschreibung - und damit eine schlechtere Navigation - zurückführen.

Problematisch an der soeben beschriebenen Pareto-Optimierung ist, dass der Agent während der 100 Zeitschritte viele Positionen in der virtuellen Welt anfährt. D.h. er könnte auch per Zufall einige der besten Lösungen gefunden haben. Aus diesem Grund wird eine weitere Pareto-Optimierung durchgeführt. Sie zeigt statt der besten Bewertung einer jeden Episode, die jeweils letzte Bewertung der Episode. Dadurch ist besser erkennbar, wohin der Agent bei der entsprechenden Zielgewichtung fährt.

Die Ergebnisse fallen nun schlechter aus. Das Ende einer Episode tritt häufig durch das Verlassen der Welt auf. Dies erklärt die dominierten Lösungen am rechten und am linken Rand. Sie halten sich bei Aktionsdefinition φ im Rahmen. Bei Aktionsdefinition ω scheint ein zufriedenstellendes Ergebnis nur durch eine bessere Wahl der Netzparameter erreichbar zu sein.

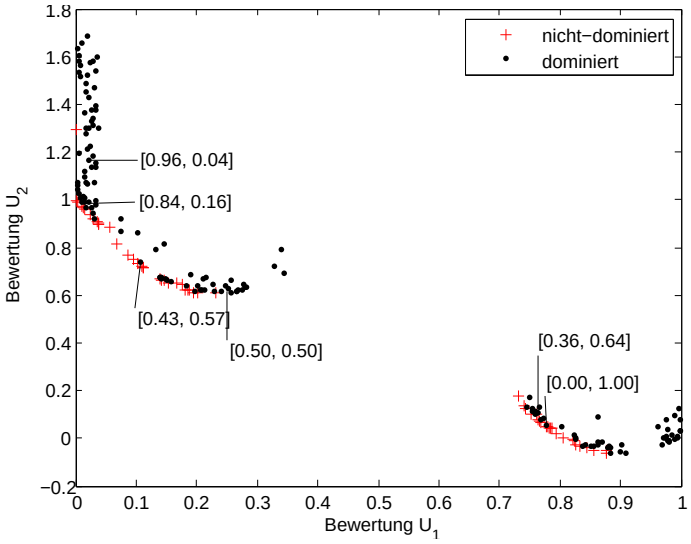


(a) Beste Bewertungen bei Aktionsdefinition φ.

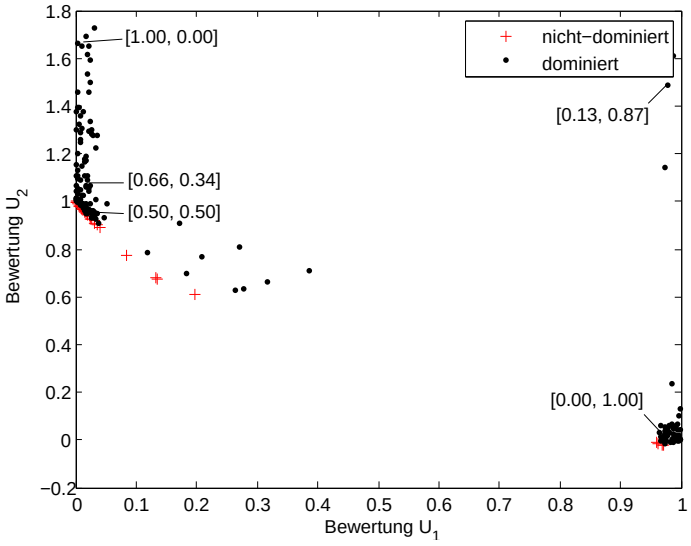


(b) Beste Bewertungen bei Aktionsdefinition ω.

Abbildung 6.10: **Ergebnis der Pareto-Optimierung (beste Bewertung).** Für die Pareto-Optimierung wurden 200 Episoden durchgeführt (nach vorangegangenem Training). Zu sehen sind die besten Bewertungen, die der Agent während der einzelnen Episoden erhalten hat. Für jede Episode wurde eine Zielgewichtung $\Pi = [r_1,\ r_2]$ mit gleichverteilten Zufallszahlen ($0 \leq r \leq 1$) erzeugt. Im Diagramm sind die Zielgewichtungen einzelner Episoden annotiert. Der Agent beginnt die Episoden von verschiedenen Startpositionen $[x,\ y]$ mit $35 \leq x \leq 65$ und $y = 112$.



(a) Letzte Bewertungen bei Aktionsdefinition φ.



(b) Letzte Bewertungen bei Aktionsdefinition ω.

Abbildung 6.11: **Ergebnis der Pareto-Optimierung (letzte Bewertung).** Bei dieser Pareto-Optimierung wird anstatt der besten Bewertung einer Episode die jeweils letzte Bewertung der Episode gewählt. Der Grund liegt darin, dass der Agent während einer Episode die Bewertungen von 100 Positionen erhält und dabei per Zufall eine für seine Zielgewichtung ideale Bewertung enthalten sein kann. Die letzte Bewertung ist aussagekräftiger, die sie die Zielposition des Agenten relativ gut beschreibt.

6.2.3 Versuch 3

Die Versuche 1 und 2 wurden jeweils mit dem spektralen Radius 0.4 durchgeführt. In diesem Versuch soll der Einfluss des spektralen Radius bei Aktionsdefinition ω untersucht werden. Wie in Kapitel 2.4 beschrieben, definiert der spektrale Radius neben der Reservoir-Verbindungsdichte die Stärke der Rückkopplung [Jaeger, 2002]. Je größer der spektrale Radius gewählt wird, desto größer wird der Einfluss vorhergehender Eingaben auf den Reservoirzustand. Außerdem beschreibt der spektrale Radius, wie lange sich der Effekt einer Eingabe im Reservoir hält [Jaeger, 2002]. Mit zunehmendem spektralen Radius machen sich immer weiter zurückliegende Eingaben bemerkbar.

Eine Besonderheit stellt der spektrale Radius 0 dar. In diesem Fall wird aus der Gewichtsmatrix W_{res} eine Nullmatrix. Es verschwinden sämtliche Rückkopplungen und aus dem ESN wird ein FFN, bei dem nur die Ausgabeschicht trainiert wird. Diese Eigenschaft wird bei diesem Versuch genutzt, um zu zeigen, dass bei Aktionsdefinition ω tatsächlich die Rückkopplung des Reservoirs notwendig ist.

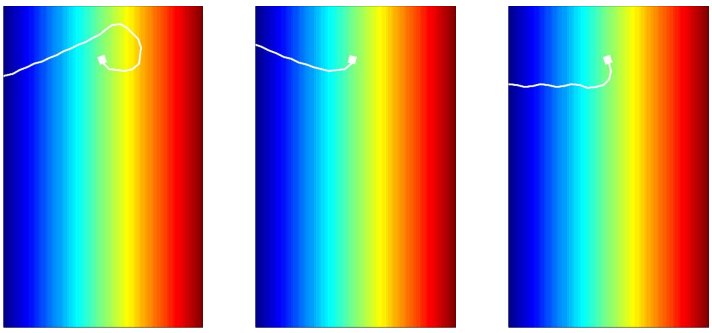
Die Parameter dieses Versuchs sind bis auf den spektralen Radius identisch mit den Parametern aus Versuch 1. Sie sind den Tabellen 6.5 und 6.6 zu entnehmen.

Tabelle 6.5: Netzparameter Szenario 1 Versuch 3

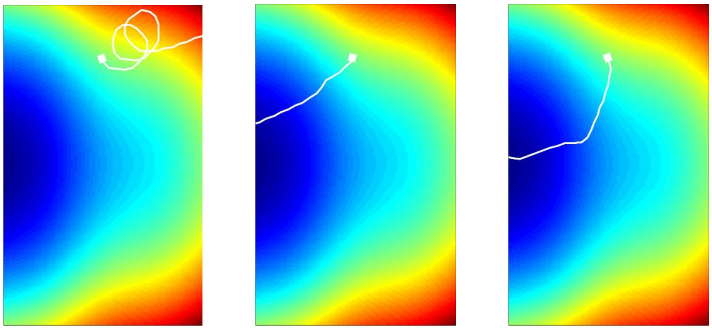
Parameter	**Bezeichnung**	**Wert**
Anzahl Reservoirneuronen	N	50
Anzahl Ausleseneuronen	M	400
Spektraler Radius	α	0.0, 0.4, 0.6
Reservoir-Verbindungsdichte	c_{res}	0.3
Eingabe-Skalierung	r_{in}	0.65

Tabelle 6.6: Trainingsparameter Szenario 1 Versuch 3

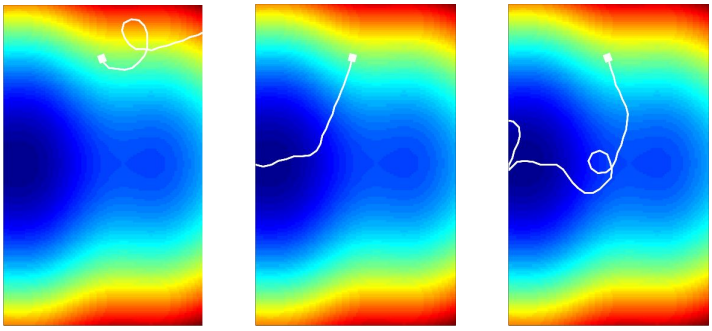
Parameter	**Bezeichnung**	**Wert**
Episodenlänge	$t_{Episode}$	100
Anzahl Trainingsepisoden	-	500
Initialisierungslänge	-	3
Discount-Faktor	γ	0.0



(a) $\Pi = [1.0,\ 0.0]$

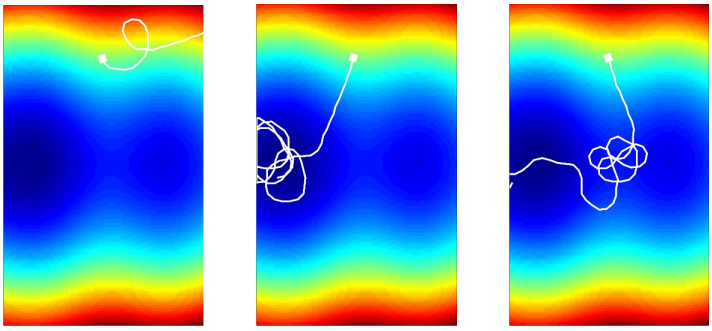


(b) $\Pi = [1.0,\ 0.2]$

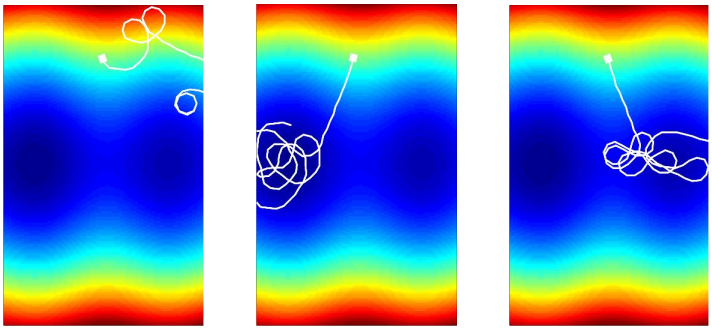


(c) $\Pi = [1.0,\ 0.4]$

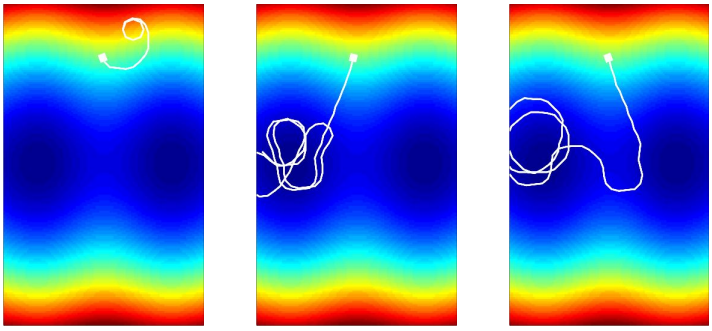
Abbildung 6.12: **Vergleich des Verhaltens bei unterschiedlichem spektralem Radius und Aktionsdefinition ω (Testlauf 1-3).** Auf der linken Seite sind die Pfade des Agenten bei spektralem Radius 0.0 zu sehen. Zum Vergleich sind in der Mitte die aus Versuch 1 bekannten Testläufe (spektraler Radius 0.4) nochmals dargestellt. Die rechte Seite zeigt das Verhalten bei spektralem Radius 0.6.



(a) $\Pi = [1.0,\ 0.6]$

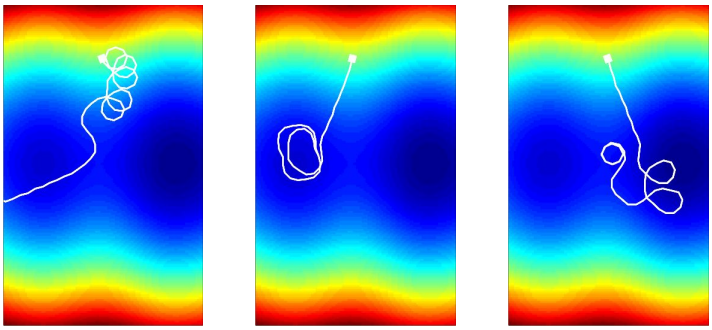


(b) $\Pi = [1.0,\ 0.8]$

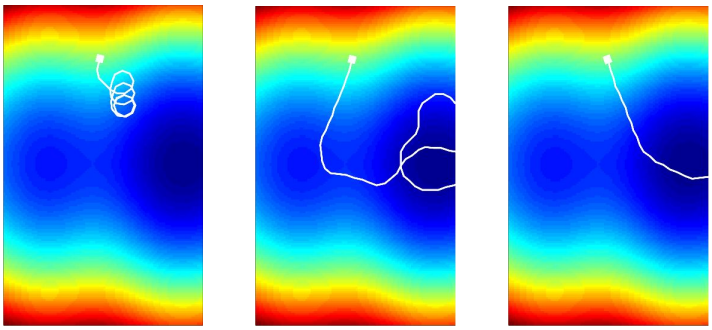


(c) $\Pi = [1.0,\ 1.0]$

Abbildung 6.13: **Vergleich des Verhaltens bei unterschiedlichem spektralem Radius und Aktionsdefinition ω (Testlauf 4-6).** Auf der linken Seite sind die Pfade des Agenten bei spektralem Radius 0.0 zu sehen. Zum Vergleich sind in der Mitte die aus Versuch 1 bekannten Testläufe (spektraler Radius 0.4) nochmals dargestellt. Die rechte Seite zeigt das Verhalten bei spektralem Radius 0.6.



(a) $\Pi = [0.5,\ 1.0]$



(b) $\Pi = [0.0,\ 1.0]$

Abbildung 6.14: **Vergleich des Verhaltens bei unterschiedlichem spektralem Radius und Aktionsdefinition ω (Testlauf 7-8).** Auf der linken Seite sind die Pfade des Agenten bei spektralem Radius 0.0 zu sehen. Zum Vergleich sind in der Mitte die aus Versuch 1 bekannten Testläufe (spektraler Radius 0.4) nochmals dargestellt. Die rechte Seite zeigt das Verhalten bei spektralem Radius 0.6.

Es finden dieselben 8 Testläufe wie in Versuch 1 statt, jedoch unter Verwendung verschiedener spektraler Radien. Als Aktionsdefinition dient die Rotationsgeschwindigkeit ω. Da nur wenig vorhergehende Eingaben notwendig sind, um eine eindeutige Zustandsbeschreibung zu erhalten, werden die spektralen Radien 0.0, 0.4 und 0.6 benutzt. Die Abbildungen 6.12, 6.13 und 6.14 zeigen das Verhalten des Agenten während der 8 Testläufe.

Wie man anhand der Testläufe erkennen kann, scheint eine korrekte Navigation ohne die Rückkopplung des Reservoirs unmöglich zu sein. Bei spektralem Radius 0.0 fährt der Agent ohne Orientierung durch die virtuelle Welt. Das beste Verhalten zeigt der Agent bei spektralem Radius 0.4. Bereits bei einer Erhöhung des spektralen Radius auf 0.6 verschlechtert sich die Navigation. Offensichtlich entstehen dadurch zuviele Reservoirzustände, die der Agent nicht mehr zuordnen kann.

Eine Veränderung der restlichen Parameter - wie beispielsweise der Reservoir- Verbindungsdichte und der Anzahl an Reservoirneuronen - könnte das Verhalten des Agenten wieder verbessern. Deshalb kann nicht pauschal gesagt werden, dass ein spektraler Radius von 0.6 ungeeignet ist. Hierzu bedarf es der Untersuchung zahlreicher Parameter-Kombinationen.

Auch eine Vergrößerung der Ausleseschicht W_{read} und zusätzliche Trainingsepisoden können das Ergebnis positiv beeinflussen. Denn dadurch können mehr Reservoirzustände gelernt und interpretiert werden.

6.2.4 Versuch 4

Bei diesem Versuch handelt es sich um eine Wiederholung von Versuch 3. Er findet unter den gleichen Rahmenbedingungen statt, jedoch wird als Aktionsdefinition die Orientierung φ gewählt. Die Parameter sind in den Tabellen 6.7 und 6.8 aufgelistet.

Tabelle 6.7: Netzparameter Szenario 1 Versuch 4

Parameter	**Bezeichnung**	**Wert**
Anzahl Reservoirneuronen	N	50
Anzahl Ausleseneuronen	M	400
Spektraler Radius	α	0.0, 0.4, 0.6
Reservoir-Verbindungsdichte	c_{res}	0.3
Eingabe-Skalierung	r_{in}	0.65

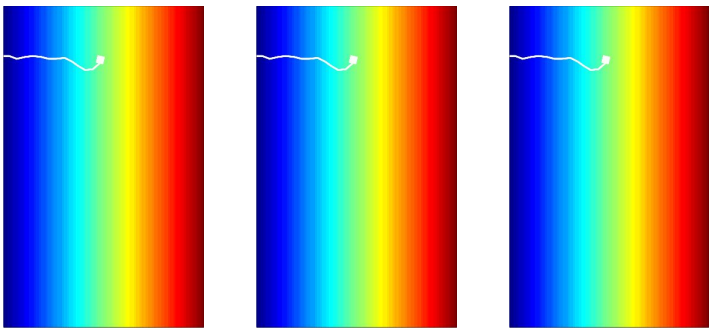
Tabelle 6.8: Trainingsparameter Szenario 1 Versuch 4

Parameter	**Bezeichnung**	**Wert**
Episodenlänge	$t_{Episode}$	100
Anzahl Trainingsepisoden	-	500
Initialisierungslänge	-	3
Discount-Faktor	γ	0.0

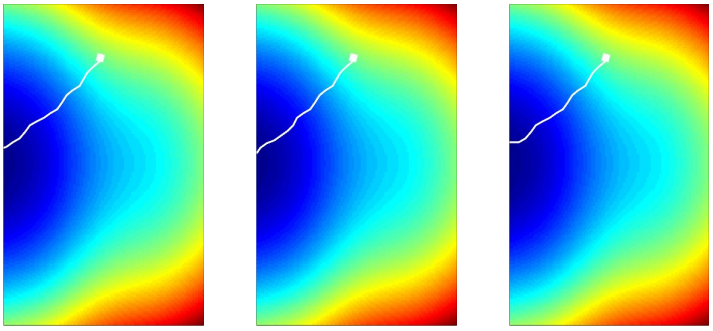
Es ist sofort ersichtlich, dass der Agent bei Aktionsdefinition φ keine Rückkopplung benötigt. Die Navigation gelingt mit einem spektralen Radius von 0.0 äußerst gut. Dennoch scheint der Agent von einer geringen Rückkopplung zu profitieren. Bei spektralem Radius 0.4 sieht es so aus, als würde der Agent ein noch besseres Verhalten zeigen. Dies ist vor allem bei den Testläufen 3,7 und 8 ersichtlich.

Vermutlich führt eine geringe Rückkopplung zu robusteren bzw. stabileren Zustandsbeschreibungen. Auf der anderen Seite muss beachtet werden, dass ohne Rückkopplung die Neuronenaktivierung im Reservoir geringer ist. D.h. eine größere Eingabeskalierung r_{in} könnte das Ergebnis bei spektralem Radius 0.0 verbessern.

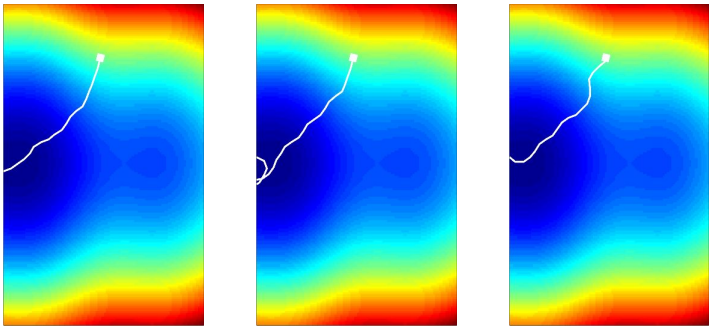
Auch bei diesem Versuch zeigt sich, dass ein spektraler Radius von 0.6 zu einem schlechteren Verhalten führt. Die Gründe dafür sind dieselben wie bei Versuch 3. Gleichermaßen gilt, dass es auf die Kombination der Parameter ankommt und keine pauschalen Aussagen bzgl. der Änderung eines einzelnen Parameters getroffen werden können.



(a) $\Pi = [1.0,\ 0.0]$

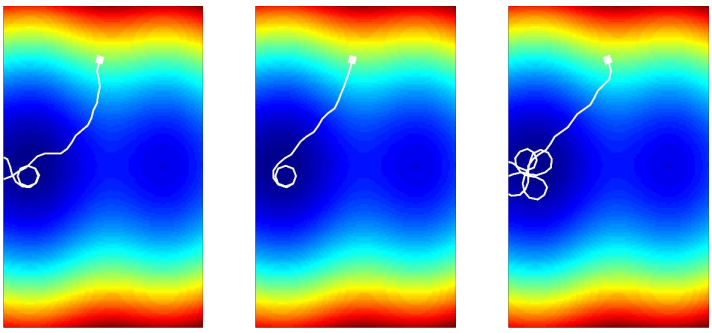


(b) $\Pi = [1.0,\ 0.2]$

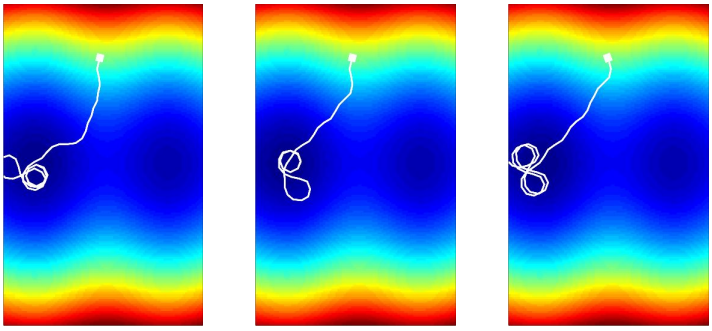


(c) $\Pi = [1.0,\ 0.4]$

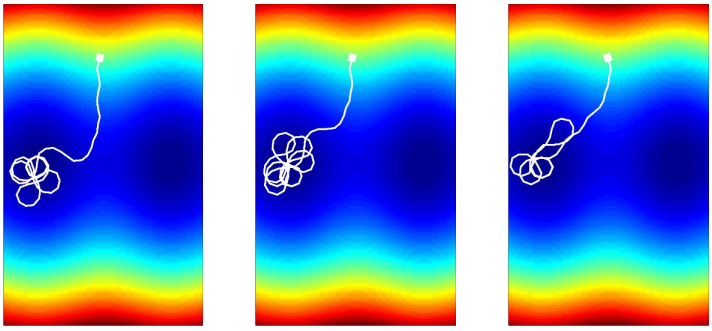
Abbildung 6.15: **Vergleich des Verhaltens bei unterschiedlichem spektralem Radius und Aktionsdefinition φ (Testlauf 1-3).** Auf der linken Seite sind die Pfade des Agenten bei spektralem Radius 0.0 zu sehen. Zum Vergleich sind in der Mitte die aus Versuch 1 bekannten Testläufe (spektraler Radius 0.4) nochmals dargestellt. Die rechte Seite zeigt das Verhalten bei spektralem Radius 0.6.



(a) $\Pi = [1.0,\ 0.6]$

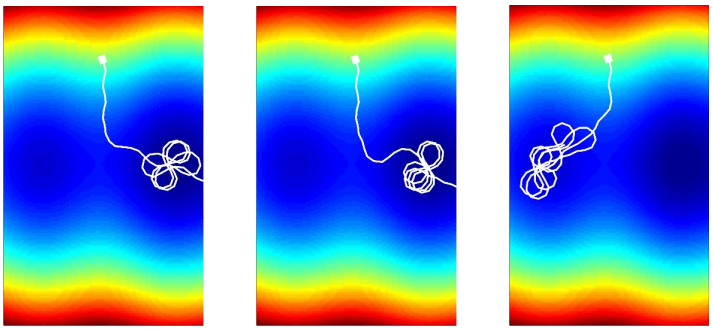


(b) $\Pi = [1.0,\ 0.8]$

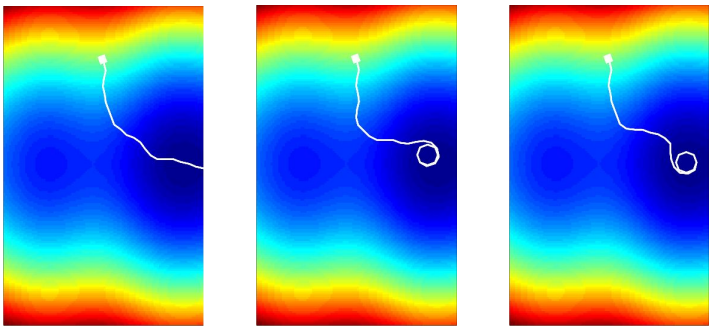


(c) $\Pi = [1.0,\ 1.0]$

Abbildung 6.16: **Vergleich des Verhaltens bei unterschiedlichem spektralem Radius und Aktionsdefinition φ (Testlauf 4-6).** Auf der linken Seite sind die Pfade des Agenten bei spektralem Radius 0.0 zu sehen. Zum Vergleich sind in der Mitte die aus Versuch 1 bekannten Testläufe (spektraler Radius 0.4) nochmals dargestellt. Die rechte Seite zeigt das Verhalten bei spektralem Radius 0.6.



(a) $\Pi = [0.5,\ 1.0]$



(b) $\Pi = [0.0,\ 1.0]$

Abbildung 6.17: **Vergleich des Verhaltens bei unterschiedlichem spektralem Radius und Aktionsdefinition φ (Testlauf 7-8).** Auf der linken Seite sind die Pfade des Agenten bei spektralem Radius 0.0 zu sehen. Zum Vergleich sind in der Mitte die aus Versuch 1 bekannten Testläufe (spektraler Radius 0.4) nochmals dargestellt. Die rechte Seite zeigt das Verhalten bei spektralem Radius 0.6.

6.3 Szenario 2

In Szenario 1 wurde die Fähigkeit des Agenten zur Optimierung multikriterieller Probleme untersucht. Es wurde gezeigt, wie das Verhalten bzw. der Lernerfolg durch die Wahl der Netzparameter beeinflusst wird. Außerdem konnte gezeigt werden, dass der Agent abhängig von der vorgegebenen Zielgewichtung, die Bewertungen erfolgreich optimiert.

In diesem Szenario steht die Entscheidungslogik im Vordergrund. Es wird untersucht, ob der Agent seinen inneren Zustand und damit sein Wohlbefinden optimieren kann. Da der innere Zustand mit drei Trieben modelliert wird, empfiehlt es sich drei Bewertungsfunktionen zu definieren. Die drei Bewertungsfunktionen sind:

$$U_1(p) = 2 \cdot \exp\left(-\left(\frac{(x-50)^2}{2 \cdot 25^2} + \frac{(y-104)^2}{2 \cdot 25^2}\right)\right)$$

$$U_2(p) = -2 \cdot \exp\left(-\left(\frac{(x-50)^2}{2 \cdot 13^2} + \frac{(y-80)^2}{2 \cdot 13^2}\right)\right)$$

$$U_3(p) = 2 \cdot \exp\left(-\left(\frac{(x-50)^2}{2 \cdot 25^2} + \frac{(y-56)^2}{2 \cdot 25^2}\right)\right)$$

wobei $p = [x,\ y]$ die Position beschreibt. Jede Bewertungsfunktion definiert eine Gauß-Glocke. Die Zentren sind an den Positionen $[50,\ 104]$, $[50,\ 80]$ und $[50,\ 56]$. Im Gegensatz zu Szenario 1 handelt es sich hier um ein Maximierungsproblem. D.h. der Agent optimiert die Bewertungen indem er diejenigen Position ansteuert, an denen die Bewertungsfunktionen den größten Betrag haben. Bei U_2 handelt es sich um eine negative Bewertungsfunktion. Je weiter sich der Agent dem Zentrum von U_2 nähert, desto schlechter werden die Bewertungen. Abbildung 6.18 stellt U_1, U_2 und U_3, sowie die mit der Zielgewichtung $\Pi = [1,\ 1,\ 1]$ skalarisierte Bewertungsfunktion, graphisch

dar. Anhand der skalarisierte Bewertungsfunktion lassen sich die Überlagerungen von U_1, U_2 und U_3 erkennen.

Tabelle 6.9: Zusammenhang zwischen Trieben und Bewertungsfunktionen

Bewertungsfunktion	**Wirkung**	**Trieb**
U_1	befriedigt	D_1
U_2	verstärkt	D_2
U_3	befriedigt	D_3

Jede Bewertungsfunktion beeinflusst einen Trieb. Der Zusammenhang zwischen Trieben und Bewertungsfunktionen ist Tabelle 6.9 zu entnehmen. Der Trieb D_1 wird durch positive Bewertungen von U_1 befriedigt. Ebenso wird der Trieb D_3 durch positive Bewertungen von U_3 befriedigt. Im Gegensatz dazu nimmt der Trieb D_2 im Laufe der Zeit von selbst ab, solange der Agent keine negativen Bewertungen von U_2 erhält. Befindet sich der Agent in einer Region, in der U_2 die Positionen negativ bewertet, so nimmt D_2 zu.

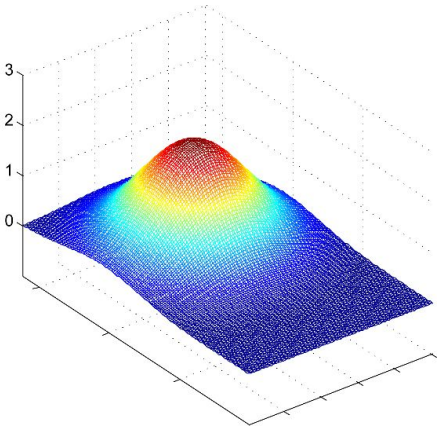
Tabelle 6.10: Netzparameter Szenario 2

Parameter	**Bezeichnung**	**Wert**
Anzahl Reservoirneuronen	N	40
Anzahl Ausleseneuronen	M	300
Spektraler Radius	α	0.4
Reservoir-Verbindungsdichte	c_{res}	0.1
Eingabe-Skalierung	r_{in}	0.65

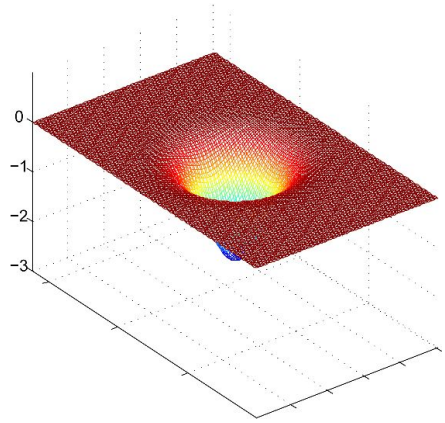
Tabelle 6.11: Trainingsparameter Szenario 2

Parameter	**Bezeichnung**	**Wert**
Episodenlänge	$t_{Episode}$	100
Anzahl Trainingsepisoden	-	500
Initialisierungslänge	-	3
Discount-Faktor	γ	0.0

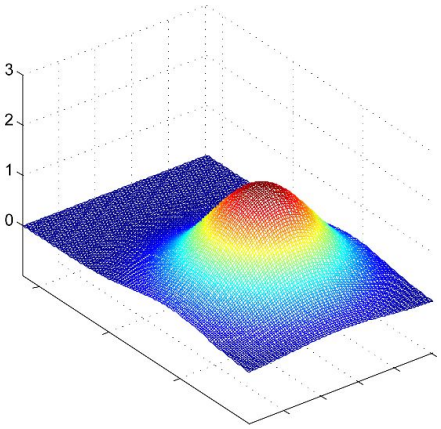
Wegen der präzisieren Navigation und der einfacheren Parameterwahl werden die Versuche ausschließlich nach Aktionsdefinition φ durchgeführt. Alle Versuche finden mit denselben Netz- und Trainingsparametern statt. Die Parameter sind in den Tabellen 6.10 und 6.11 aufgelistet. Sowohl die Anzahl an Reservoir- und Ausleseneuronen als auch



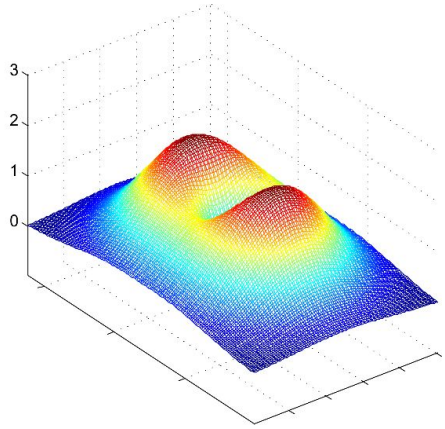
(a) Bewertungsfunktion U_1



(b) Bewertungsfunktion U_2



(c) Bewertungsfunktion U_3



(d) Skalarisierte Bewertungsfunktion (Zielgewichtung $\Pi = [1,\ 1,\ 1]$)

Abbildung 6.18: **Bewertungsfunktionen aus Szenario 2.** Dargestellt sind die drei Bewertungsfunktionen des 2. Szenarios. Sie ordnen jeder Position $p = [x,\ y]$ innerhalb der virtuellen Welt eine Bewertung zu. Zusätzlich ist in (d) die mit der Zielgewichtung $\Pi = [1,\ 1,\ 1]$ skalarisierte Bewertungsfunktion dargestellt.

die Reservoir-Verbindungsdichte konnten reduziert werden. Die restlichen Parameter sind identisch zu den Parametern aus Szenario 1.

Für die Entscheidungsfindung müssen mehrere Zielgewichtungen vorgegeben werden[4]. In allen Versuchen werden dieselben Zielgewichtungen benutzt. Die Wahl der Menge Π aller vorgegebenen Zielgewichtungen Π_i wird in Versuch 1 diskutiert. Sie lautet:

$$\Pi = [\Pi_1,\ \Pi_2,\ \Pi_3,\ \Pi_4,\ \Pi_5]$$

mit

$$\Pi_1 = [1.0,\ 0.0,\ 0.0]$$
$$\Pi_2 = [0.0,\ 0.0,\ 1.0]$$
$$\Pi_3 = [1.0,\ 0.5,\ 0.0]$$
$$\Pi_4 = [0.0,\ 0.5,\ 1.0]$$
$$\Pi_5 = [1.0,\ 0.0,\ 1.0]$$

Außerdem wird in allen Versuchen dasselbe FFN zur Approximation der *Q-values* benutzt. Die Parameter der Entscheidungslogik sind Tabelle 6.12 zu entnehmen. Das FFN enthält 100 Neuronen in der mittleren Schicht. Für das Training wird die Episodenlänge auf 15 Zeitschritte reduziert. Zum einen ist dies ausreichend um zwischen den Episoden deutliche Unterschiede des Wohlbefindens festzustellen und zum anderen kann der Agent innerhalb kurzer Zeit eine neue Zielgewichtung wählen. Der *discount*-Faktor beträgt 0.7. Dadurch kann der Agent Entscheidungen treffen, die auf lange Sicht das Wohlbefinden optimieren. Die Vergessensrate wird während dem Training in gleichmäßigen Intervallen von 0.35 auf 0.15 reduziert. Die *exploitationrate* definiert, wie stark der Agent sein Wissen bei der Wahl der nächsten Zielgewichtung miteinbezieht. Bei einem Wert von 1.0 wählt der Agent stets die Aktion mit dem größten *Q-value*, während bei einem Wert von 0.5 die Zielgewichtung in 50 Prozent der Fälle zufällig aus der Menge

[4] siehe Kapitel 5.2.1.

Π entnommen wird. Die *exploitationrate* wird während dem Training in gleichmäßigen Intervallen erhöht. Dadurch exploriert der Agent zu Beginn relativ stark, während er am Ende sein Wissen immer mehr in die Entscheidung miteinbezieht.

Tabelle 6.12: *Q-Learning* Parameter Szenario 2

Parameter	**Bezeichnung**	**Wert**
Anzahl Neuronen	-	100
Episodenlänge	$t_{Episode}$	15
Discount-Faktor	γ	0.7
Vergessensrate	β	0.35-0.15
Exploitationrate	-	0.65-1.0

6.3.1 Versuch 1

In diesem Versuch soll gezeigt werden, wie die vorgegebenen Zielgewichtungen den inneren Zustand des Agenten beeinflussen. Dazu werden fünf Testläufe mit einer Episodenlänge von jeweils 100 Zeitschritten durchgeführt. Der Agent startet dabei von verschiedenen Positionen innerhalb der virtuellen Welt und folgt während eines jeden Testlaufs der entsprechenden Zielgewichtung. Vor den Testläufen wurden 500 Trainingsepisoden absolviert.

Die Dynamik der drei Triebe ergibt sich anhand der Parameter aus Tabelle 6.13. Zu Beginn ist jeder Trieb zur Hälfte befriedigt. Das Wachstum pro Zeitschritt ist bei Trieb D_3 am größten. Der Trieb D_2 befriedigt sich von selbst, solange der Agent keine negative Bewertung von U_2 erhält. Eine vollständige Befriedigung hemmt den Zuwachs von D_1 45 Zeitschritte lang und den Zuwachs von D_3 30 Zeitschritte lang.

Tabelle 6.13: Triebdynamik Versuch 1

Trieb	$\boldsymbol{D_i^s}$	$\boldsymbol{D_i^g}$	$\boldsymbol{D_i^l}$	$\boldsymbol{D_i^t}$
D_1	0.5	0.008	-	45
D_2	0.5	-0.005	-	0
D_3	0.5	0.012	-	30

Die beiden Abbildungen 6.19 und 6.20 zeigen das Ergebnis der Testläufe. Auf der linken Seite ist der Pfad des Agenten während jedes einzelnen Testlaufs dargestellt. Das

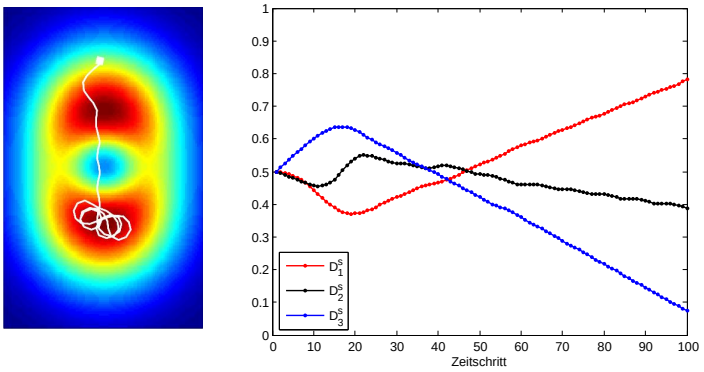
Diagramm auf der rechten Seite zeigt, wie sich die drei Triebe während des Testlaufs verändern.

Bei Testlauf 1 fährt der Agent entsprechend der Zielgewichtung $\Pi_2 = [0.0,\ 0.0,\ 1.0]$ direkt in Richtung des Zentrums von U_3. Zu Beginn befindet er sich in einer Region, die von U_1 positiv bewertet wird. Dadurch wird der Trieb D_1 kurzeitig befriedigt. Zwischen dem 10. und 20. Zeitschritt passiert der Agent das Zentrum von U_2. Folglich verstärkt sich der Trieb D_2. Sobald er sich in Nähe des Zentrums von U_3 befindet nimmt D_2 wieder ab, da er keine negativen Bewertungen mehr erhält. Zudem wird der Trieb D_3 befriedigt.

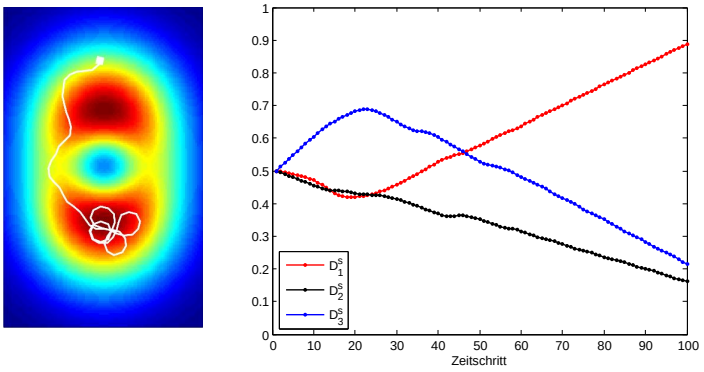
Im Gegensatz zum 1. Testlauf meidet der Agent beim 2. Testlauf das Zentrum von U_2. Dadurch erhält er keine negativen Bewertungen, was von Beginn an zu einer Befriedung des Triebs D_2 führt. Die Dynamik der restlichen Triebe ist vergleichbar mit dem 1. Testlauf.

Bei Testlauf 3 ist gut zu erkennen, dass der Trieb D_1 im Gegensatz zu D_3 sehr schnell befriedigt werden kann. Im 73. Zeitschritt tritt eine vollständige Befriedung von D_1 ein. Die Ursache liegt im Vergleich zwischen D_1^g und D_2^g. Der Zuwachs von D_1 ist relativ zur Bewertung U_1 deutlich geringer als bei D_3.

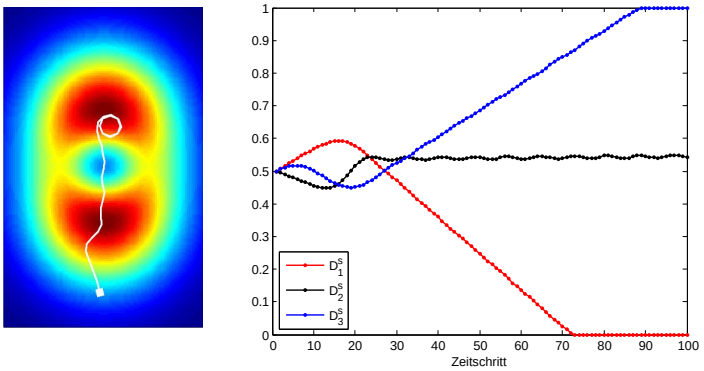
Interessant ist auch der 5. Testlauf. Entsprechend der Zielgewichtung $\Pi_5 = [1.0,\ 0.0,\ 1.0]$ bewegt sich der Agent zwischen den Zentren von U_1 und U_2. Da dort die Bewertungen von U_1 und U_2 etwa den Betrag 1 haben, werden die Triebe D_1 und D_2 nur langsam befriedigt. Gleichzeitig wird D_2 deutlich verstärkt, da sich der Agent dicht beim Zentrum von U_2 aufhält. Offensichtlich ist die Zielgewichtung Π_5 bei dieser Triebdynamik ungeeignet um das Wohlbefinden des Agenten zu optimieren.



(a) Testlauf 1 mit Zielgewichtung $\Pi_2 = [0.0,\ 0.0,\ 1.0]$

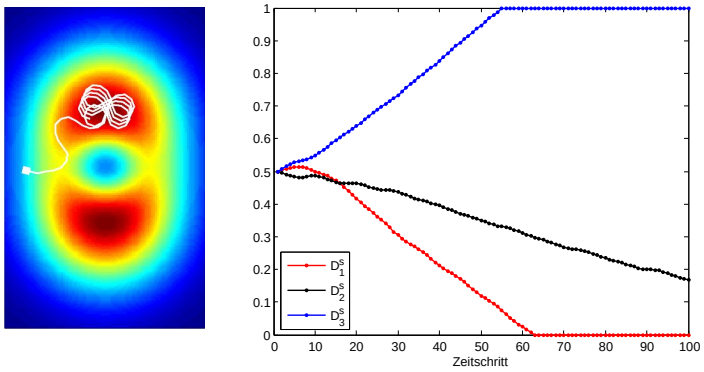


(b) Testlauf 2 mit Zielgewichtung $\Pi_4 = [0.0,\ 0.5,\ 1.0]$

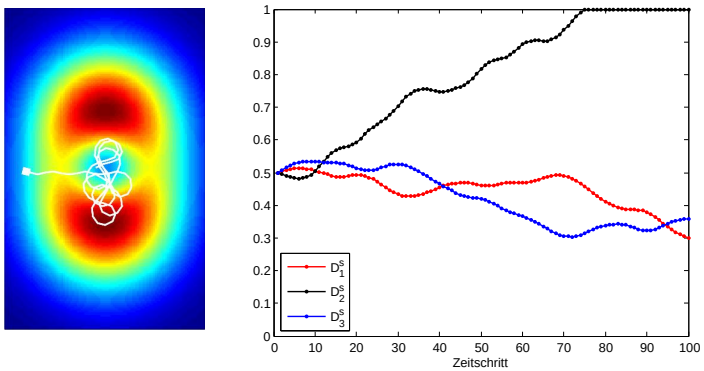


(c) Testlauf 3 mit Zielgewichtung $\Pi_1 = [1.0,\ 0.0,\ 0.0]$

Abbildung 6.19: **Veränderung des inneren Zustands während der Testläufe 1, 2 und 3.** Links ist der Pfad des Agenten (weiße Linie) während jedes einzelnen Testlaufs dargestellt. Farbig hinterlegt ist die mit $[1,\ 1,\ 1]$ skalarisierte Bewertungsfunktion. Die rechte Seite zeigt, wie sich die Bewertungen auf die Stärke der einzelnen Triebe auswirken.



(a) Testlauf 4 mit Zielgewichtung $\Pi_3 = [1.0,\ 0.5,\ 0.0]$



(b) Testlauf 5 mit Zielgewichtung $\Pi_5 = [1.0,\ 0.0,\ 1.0]$

Abbildung 6.20: **Veränderung des inneren Zustands während der Testläufe 4 und 5.** Links ist der Pfad des Agenten (weiße Linie) während jedes einzelnen Testlaufs dargestellt. Farbig hinterlegt ist die mit $[1,\ 1,\ 1]$ skalarisierte Bewertungsfunktion. Die rechte Seite zeigt, wie sich die Bewertungen auf die Stärke der einzelnen Triebe auswirken.

6.3.2 Versuch 2

Bei diesem Versuch wird die Leistung der Entscheidungslogik untersucht. Dazu absolviert der Agent 10000 Episoden mit jeweils 15 Zeitschritten. Mit dem FFN werden dabei die *Q-values* für jede Zielgewichtung gelernt. Es werden drei Durchgänge ausgeführt:

Im 1. Durchgang findet eine reine Exploration statt, d.h. der Agent wählt ausschließlich per Zufall die Zielgewichtungen aus der vorgegebenen Menge Π aus.

Beim 2. Durchgang greift der Agent auf sein gesammeltes Wissen zurück. Als direkter Nutzen R (*Q-learning reinforcement*) wird das Wohlbefinden Wb verwendet.

Hingegen wird im 3. Durchgang die Änderung des Wohlbefindens ΔWb zwischen der aktuellen und der nächsten Episode als direkter Nutzen verwendet.

Tabelle 6.14: Triebdynamik Versuch 2

Trieb	$\boldsymbol{D_i^s}$	$\boldsymbol{D_i^g}$	$\boldsymbol{D_i^l}$	$\boldsymbol{D_i^t}$
D_1	0.5	0.008	-	45
D_2	0.5	-0.005	-	0
D_3	0.5	0.012	-	30

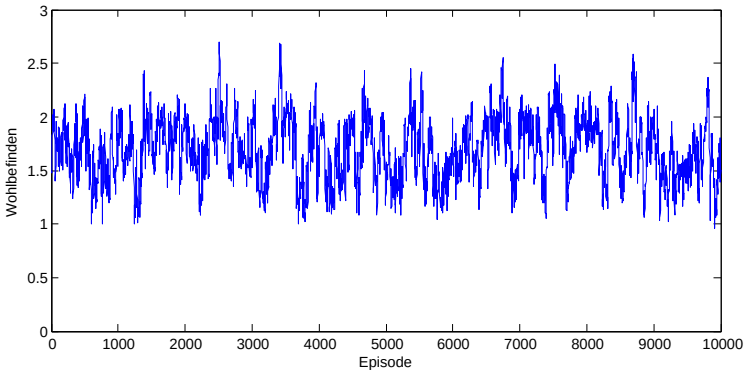
Die Beschreibung der Triebdynamik ist Tabelle 6.14 zu entnehmen. Es handelt sich um dieselbe Dynamik wie bei Versuch 1. Alle weiteren Parameter wurden bereits bei der Beschreibung des 2. Szenarios genannt und sind in den Tabellen 6.10, 6.11 und 6.12 aufgelistet.

Abbildung 6.21 zeigt das Ergebnis der drei Durchgänge. Bei reiner Exploration bewegt sich der Agent entsprechend der möglichen Zielgewichtungen stets zwischen den Zentren der Bewertungsfunktionen. Dadurch kann er die Triebe teilweise befriedigen. Das durchschnittliche Wohlbefinden während der 10000 Episoden beträgt 1,7011.

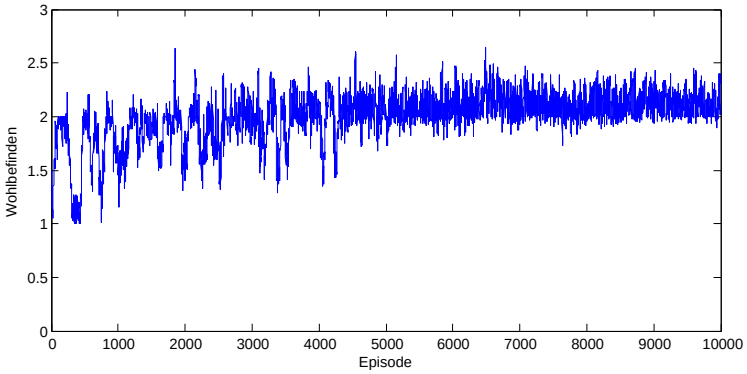
Beim 2. Durchgang beträgt das durchschnittliche Wohlbefinden 2,0061. Etwa ab der 5000. Episode gelingt es dem Agent sein Wohlbefinden zu stabilisieren. Zu dieser Zeit nutzt er überwiegend sein Wissen und exploriert nur noch wenig, weshalb das Wohlbefinden keine auffälligen negativen Peaks mehr aufweist.

Beim 3. Durchgang gelingt es dem Agent sein Wohlbefinden ab der 6000. Episode auf einem hohen Niveau zu halten. Als direkter Nutzen scheint ΔWb besser geeignet zu sein. Das durchschnittliche Wohlbefinden beträgt während der 10000 Episoden 2,6437. Auffällig ist auch, dass der Agent bereits ab der 100. Episode ein höheres Wohlbefinden hat als im 2. Durchgang.

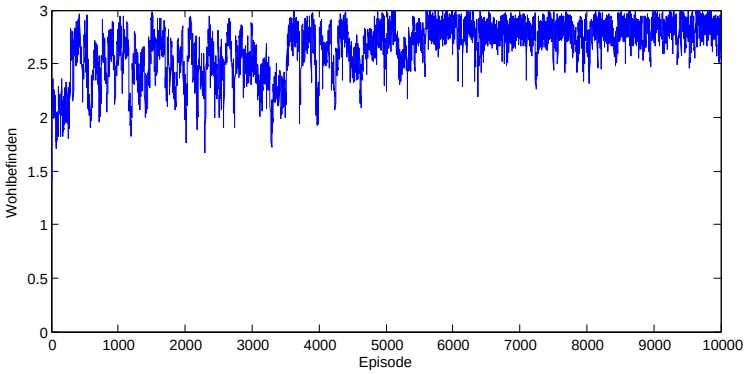
Der Nachteil von Wb als direkter Nutzen ist, dass das Wohlbefinden stets positive Werte aufweist und sich innerhalb einer Episode nur leicht verändern kann. Dies wirkt sich auf die *Q-values* aus. Sie stabilisieren sich nicht so schnell wie bei ΔWb und weisen auch geringere Unterschiede auf. Beim nächsten Versuch tritt dies deutlicher in Erscheinung.



(a) Wohlbefinden bei reiner Exploration



(b) Wohlbefinden bei direktem Nutzen $R := Wb$



(c) Wohlbefinden bei direktem Nutzen $R := \Delta Wb$

Abbildung 6.21: **Verlauf des Wohlbefindens.** Oben ist der Verlauf des Wohlbefindens bei reiner Exploration dargestellt. D.h. der Agent wählt die Zielgewichtungen zufällig aus der vorgegebenen Menge Π aus. In der Mitte nutzt der Agent sein erworbenes Wissen. Als direkter Nutzen R (*Q-learning reinforcement*) wird das Wohlbefinden Wb verwendet. Unten wird die Änderung des Wohlbefindens ΔWb verwendet.

6.3.3 Versuch 3

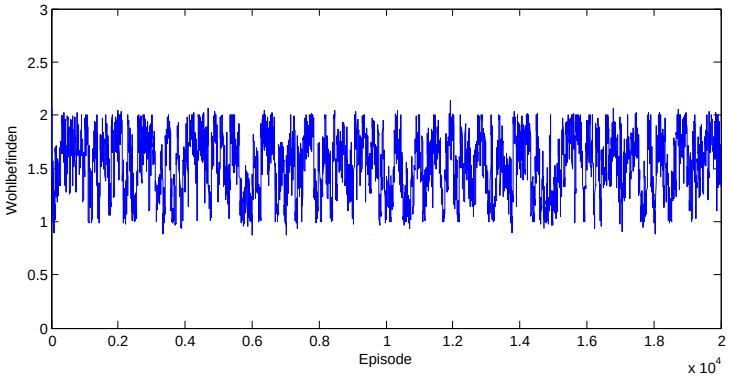
Bei diesem Versuch handelt es sich um eine Wiederholung von Versuch 2, jedoch wird die Triebdynamik verändert. Tabelle 6.15 zeigt die verwendeten Parameter. Der Trieb D_1 zeichnet sich durch einen geringen Zuwachsfaktor D_1^g aus. Dafür wächst er kontinuierlich und wird nach einer vollständigen Befriedigung nicht gehemmt. Im Gegensatz dazu bleibt der Trieb D_3 nach einer vollständigen Befriedigung für 60 Zeitschritte konstant 0. Der Zuwachsfaktor ist dafür mit 0.015 deutlich ausgeprägt. Dies führt dazu, dass der Agent vergleichsweise viel Zeit benötigt, um D_3 zu befriedigen.

Tabelle 6.15: Triebdynamik Versuch 3

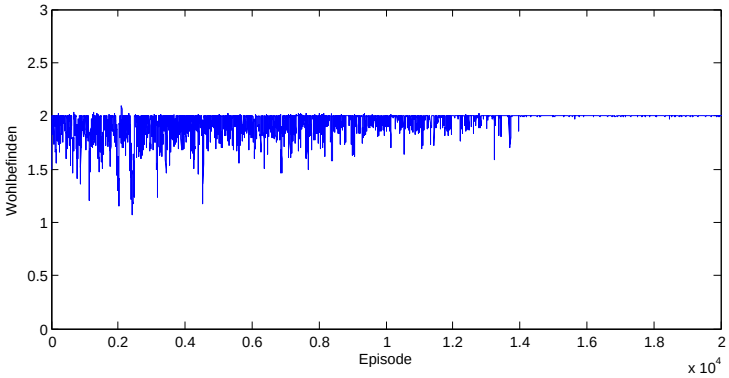
Trieb	$\boldsymbol{D_i^s}$	$\boldsymbol{D_i^g}$	$\boldsymbol{D_i^l}$	$\boldsymbol{D_i^t}$
D_1	0.5	0.006	-	0
D_2	0.5	-0.005	-	0
D_3	0.5	0.015	-	60

Das Ergebnis dieses Versuchs zeigt Abbildung 6.22. Bei reiner Exploration beträgt das durchschnittliche Wohlbefinden 1,5421. Auffällig ist der Verlauf des Wohlbefindens im 2. Durchgang. Dort beträgt das durchschnittliche Wohlbefinden 1.9520. Der Agent optimiert im letzten Drittel nur noch einen Trieb, weshalb das Wohlbefinden ab diesem Zeitpunkt konstant 2 ist. Zuvor sind negative Peaks zu erkennen. Diese entstehen durch die Exploration. Da die *exploitationrate* im Verlauf der Zeit zunimmt, nutzt der Agent vermehrt sein Wissen. Offensichtlich ordnet er einer der Zielgewichtungen einen dominanten *Q-value* zu und kann deshalb nur noch einen Trieb befriedigen.

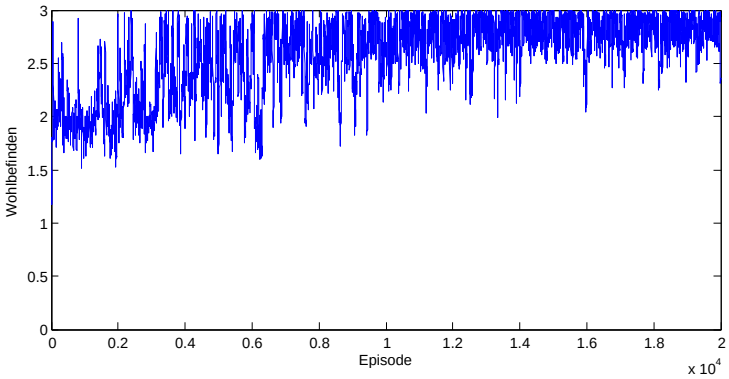
Das beste durchschnittliche Wohlbefinden hat der Agent im 3. Durchgang - bei ΔWb als direktem Nutzen. Es beträgt 2,5665. Im Vergleich zu Versuch 2 stabilisiert sich das Wohlbefinden deutlich später und die negativen Peaks sind ebenfalls stärker ausgeprägt. Dies liegt daran, dass der Trieb D_3 unverzüglich und stark anwächst, sofern der Agent ein anderes Ziel verfolgt und den Trieb zuvor nicht vollständig befriedigt hat.



(a) Wohlbefinden bei reiner Exploration



(b) Wohlbefinden bei direktem Nutzen $R := Wb$



(c) Wohlbefinden bei direktem Nutzen $R := \Delta Wb$

Abbildung 6.22: **Verlauf des Wohlbefindens.** Bei diesem Versuch handelt es sich um eine Wiederholung von Versuch 2, jedoch mit einer anderen Triebdynamik. Dies beeinflusst sowohl den Verlauf des Wohlbefindens, als auch den Lernerfolg des Agenten. Bei Wb als direktem Nutzen scheint eine Zielgewichtung zu dominieren. Der Agent optimiert lediglich ein Ziel bzw. eine Bewertungsfunktion und kann dadurch nur einen der Triebe befriedigen.

6.3.4 Versuch 4

Dieser Versuch zeichnet sich durch eine besonders anspruchsvolle Triebdynamik aus. Wie Tabelle 6.16 zu entnehmen ist, sind die Zeitintervalle (*satisfaction times*), in der die Triebe nach einer vollständigen Befriedigung gehemmt werden, mit 60 - 150 Zeitschritten sehr großzügig dimensioniert. Dadurch wird der vollständigen Befriedigung eine hohe Bedeutung zugeschrieben. Da beide Zuwachsfaktoren D_1^g und D_3^g ebenfalls einen hohen Betrag haben, kann der Agent sein Wohlbefinden nur optimieren, wenn er die *satisfaction time* beider Triebe ausnutzt. Außerdem nimmt D_2 im Laufe der Zeit nur sehr langsam ab. Dies hat zur Folge, dass sich negative Bewertungen von U_2 deutlich und langfristig bemerkbar machen. Der Agent ist gezwungen, dem Zentrum der Gauß-Glocke von U_2 auszuweichen.

Tabelle 6.16: Triebdynamik Versuch 4

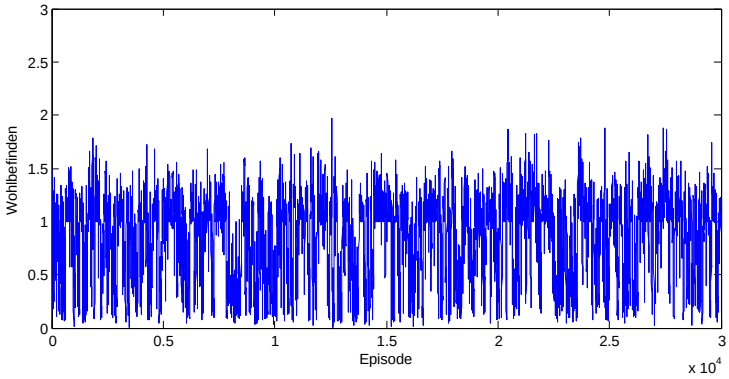
Trieb	$\boldsymbol{D_i^s}$	$\boldsymbol{D_i^g}$	$\boldsymbol{D_i^l}$	$\boldsymbol{D_i^t}$
D_1	0.5	0.013	-	60
D_2	0.5	-0.004	-	0
D_3	0.5	0.012	-	150

Abbildung 6.23 zeigt das Ergebnis dieses Versuchs. Bereits bei der reinen Exploration im 1. Durchgang zeigt sich, wie anspruchsvoll die Triebdynamik ist. Das durchschnittliche Wohlbefinden des Agenten beträgt während der 30000 Episoden 0,8453. Da die Zielgewichtungen in jeder Episode zufällig gewählt werden, ist es unwahrscheinlich dass der Agent eine vollständige Befriedigung eines Triebs erreicht. Folglich kann er sein Wohlbefinden nicht optimieren.

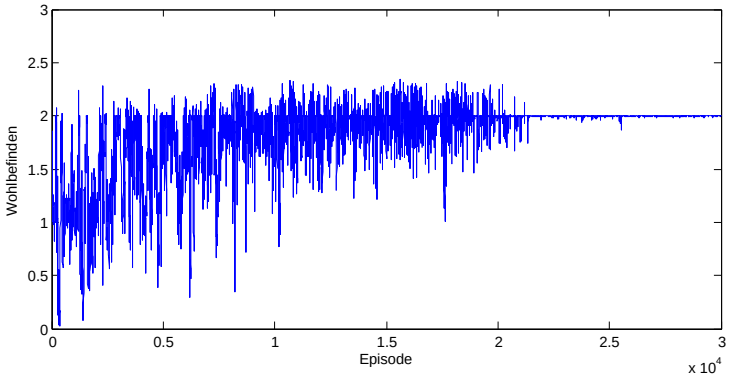
Im 2. Durchgang zeigt sich ein ähnliches Bild wie bei Versuch 3. Der Agent nutzt nur eine Zielgewichtung und kann deshalb nur einen der Triebe befriedigen. Das Wohlbefinden beträgt im letzten Drittel etwa 2. Das durchschnittliche Wohlbefinden liegt mit 1.8346 leicht unterhalb davon.

Bei ΔWb als direktem Nutzen gelingt es dem Agent sein Wohlbefinden zu optimieren. Das durchschnittliche Wohlbefinden beträgt 2,3428. Der Lernprozess fällt dabei deutlich länger aus als bei den vorhergegangenen Versuchen. Auch am Ende der 30000 Episoden

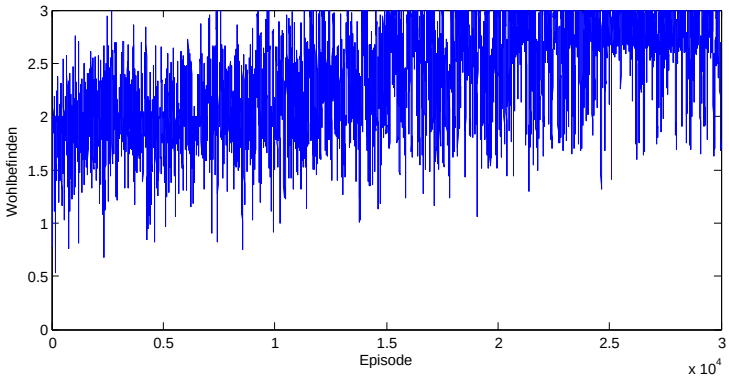
zeigen sich negative Peaks des Wohlbefindens. Dies liegt daran, dass der Agent in der Zeit, wo er einen der beiden Triebe D_1 und D_3 befriedigt, den jeweils anderen Trieb vernachlässigen muss. Außerdem benötigt er etwa 10 - 20 Zeitschritte um seine Position zwischen den Zentren der Gauß-Glocken von U_1 und U_3 zu wechseln. Kommt er dabei in eine Region, die von U_2 negativ bewertet wird, so wächst der Trieb D_2 an. Dieser benötigt wegen dem kleinen Betrag von D_2^g relativ viele Zeitschritte, bis er wieder befriedigt ist.



(a) Wohlbefinden bei reiner Exploration



(b) Wohlbefinden bei direktem Nutzen $R := Wb$



(c) Wohlbefinden bei direktem Nutzen $R := \Delta Wb$

Abbildung 6.23: **Verlauf des Wohlbefindens.** Bei diesem Versuch wachsen die Triebe D_1 und D_3 pro Zeitschritt stark an. Um das Wohlbefinden zu optimieren, muss der Agent die Zeitspanne in der das Wachstum gehemmt ist (*satisfaction time*) ausnutzen. D.h. er muss zuerst einen Trieb vollständig befriedigen, bevor er den nächsten Trieb befriedigen kann. Dies gelingt dem Agent nur bei ΔWb (unteres Diagramm).

7 Schlussbemerkungen

Im Rahmen dieser Arbeit wurde *Adaptive Dynamic Programming* mit einer auf *Echo State Networks* basierenden Kritik untersucht. In einem ersten Schritt wurde dazu der in [Uhlemann, 2011] umgesetzte Agent verbessert. Obwohl die theoretischen Betrachtungen zu der Architektur des Agenten in [Uhlemann, 2011] korrekt sind, zeigten sich bei der Umsetzung einige Probleme. Um diese zu umgehen, wurde der interne Aufbau des Agenten neu konzipiert. Neben dem Zustand wird nun auch die Aktion in die Kritik geleitet. Dies erforderte Anpassungen am ESN. Zwischen dem Reservoir und der Ausgabeschicht wurde eine weitere Schicht von Neuronen eingefügt. Auf diese Weise konnte das Reservoir auf eine geeignete Größe reduziert werden und gleichzeitig blieb die Mächtigkeit des Netzes erhalten. Erst durch diese Änderung zeigte der Agent ein brauchbares Verhalten. Außerdem wurde die RC-ADP-Architektur um eine dynamische Anpassungsrate erweitert, um nichtlineare Bewertungsfunktionen handhaben zu können.

In Kapitel 6 konnte unter anderem gezeigt werden, dass der Agent die Rückkopplungen des Reservoirs benötigt. Dazu wurden Versuche mit unterschiedlichen spektralen Radien durchgeführt. Unter anderem wurde der spektrale Radius 0 verwendet, welcher sämtliche Rückkopplung im Reservoir deaktiviert und dadurch das ESN in ein FFN transformiert. In diesem Fall gelang dem Agent keine korrekte Navigation.

Das erste Ziel dieser Arbeit bestand darin, die RC-ADP-Architektur an multikriterielle Optimierungsprobleme anzupassen. Dazu wurde die Skalarisierung bzw. Zielgewichtung genutzt. Auf diese Weise waren nur geringfüge Anpassungen an der Kritik notwendig. In Kapitel 6 zeigte sich, dass der Agent entsprechend seiner Zielgewichtung erfolgreich die Bewertungen optimiert. Ein weiterer Vorteil der Anpassungen zeigte sich beim Training.

In [Uhlemann, 2011] wurde festgestellt, „dass sich der Agent in einem bestimmten Verhalten festfährt“ [Uhlemann, 2011]. Eine ungenügende Exploration könnte die Ursache dafür sein. Bei den Untersuchungen zu MOPs konnte dieses Verhalten nicht festgestellt werden, da während dem Training verschiedene Zielgewichtungen vom Agent genutzt werden. Die unterschiedlichen Zielgewichtungen bewirken indirekt eine stetige Exploration. Allerdings führt dies dazu, dass der Agent häufiger die virtuelle Welt verlässt, was ein Eingriff von außen erforderlich macht. Dieses Problem ließe sich beispielsweise beheben, indem eine zusätzliche Bewertungsfunktion alle Positionen außerhalb der virtuellen Welt negativ bewertet.

Obwohl in dieser Arbeit das RLS-Verfahren genutzt wurde, fiel der Trainingsaufwand erheblich größer als in [Uhlemann, 2011] aus. Der Grund dafür liegt unter anderem in der verstärkten Exploration. Daneben gestaltete sich das Training mit *discount*-Faktor $\gamma > 0$ als äußerst aufwendig und war nur unter zahlreichen Randbedingungen möglich. Deshalb wurde auch in dieser Arbeit auf eine Auswertung und Analyse der Ergebnisse bei $\gamma > 0$ verzichtet.

Bei der ersten Erweiterung der Architektur muss dem Agent vorgegeben werden, welcher Zielgewichtung er zu einem bestimmten Zeitpunkt folgen soll. Das zweite Ziel dieser Arbeit bestand daher in der Entwicklung einer Entscheidungslogik, mit der der Agent selbständig eine Zielgewichtung wählen kann.

Als Entscheidungsbasis dient neben dem äußeren Zustand der innere Zustand des Agenten. Dieser wurde mit drei Trieben modelliert. Mit *Q-learning* lernt der Agent, welche Zielgewichtung in einem bestimmten Zustand den größten Nutzen hat, d.h. die einzelnen Triebe am besten befriedigt. Bei der Umsetzung stand weniger die biologisch plausible Nachbildung von Trieben im Vordergrund, als vielmehr ein erster Versuch dieses Konzept in die RC-ADP-Architektur zu integrieren.

Wie sich in Kapitel 6 zeigte, konnte der Agent erfolgreich sein Wohlbefinden bzw. die einzelnen Triebe optimieren. Allerdings war dazu ein enormer Trainingsaufwand notwendig, obwohl das Szenario sehr einfach gehalten wurde.

7.1 Ausblick

Die Anpassung der RC-ADP-Architektur an multikriterielle Optimierungsprobleme, sowie die Erweiterung um eine Entscheidungslogik lassen in Anbetracht der erzielten Ergebnisse darauf schließen, dass es sich hierbei um eine vielversprechende Grundlage für die Realisierung autonomer Agenten handelt. Das in dieser Arbeit und in [Uhlemann, 2011] verwendete Szenario, bei dem die virtuelle Welt vom Agent kartiert wird, scheint jedoch wegen des Trainingsaufwands eher von theoretischem Interesse zu sein. Sobald sich die virtuelle Welt verändert, muss ein Großteil des erworbenen Wissens neu gesammelt werden.

Für eine praktische Anwendung auf einem realen Roboter scheint das in [Koprinkova-Hristova et al., 2010,Koprinkova-Hristova et al., 2012,Oubbati et al., 2011] verwendeten Szenario - bzw. ein vergleichbares Szenario - besser geeignet zu sein. In [Koprinkova-Hristova et al., 2010, Koprinkova-Hristova et al., 2012, Oubbati et al., 2011] lernt der Agent die Vorhersage der Bewertungen relativ zu einem Objekt. Verändert sich beispielsweise die Position eines Objekts innerhalb der Umgebung des Agenten, so muss der Agent sein Wissen nicht von Grund auf neu erwerben. Der relative Bezug zum Objekt bleibt bei diesem Szenario erhalten.

Weitere Verbesserungsmöglichkeiten bestehen beispielsweise bei der Wahl der Netzparameter, sowie bei der Modellierung des inneren Zustands des Agenten. In [Salichs and Malfaz, 2012] werden neben den Trieben und der daraus resultierenden Motivation auch Emotionen nachgebildet. Auch das Konzept der riskanten Aktionen (engl. *risky actions*) und den damit verbunden Q_{worst}-*values* [Malfaz and Salichs, 2011] würden zu einem biologisch plausibleren Verhalten und einer mächtigeren Entscheidungslogik führen.

Im Rahmen weiterer Arbeiten könnten die genannten Verbesserungsvorschläge untersucht werden. Desweiteren sind zahlreiche Versuche auf Basis der erweiterten RC-ADP-Architektur möglich, um das Verhalten des Agenten detaillierter nachvollziehen zu können.

Abbildungsverzeichnis

2.1 Bestärkendes Lernen 7
2.2 Das Perzeptron 11
2.3 Netzarchitekturen 12
2.4 Aufbau eines ESN 15

3.1 Virtuelle Welt 24
3.2 Alternativen zur Aktionsdefinition 28
3.3 Architektur des Agenten 29
3.4 ESN der Kritik 30

4.1 ESN für MOPs 38
4.2 Erweiterte Architektur des Agenten 39

5.1 Architektur des Agenten inklusive Entscheidungslogik 45
5.2 Dynamik eines Triebs 46
5.3 Q-Learning FFN 50

6.1 Szenario 1 Bewertungsfunktionen 55
6.2 Szenario 1 V1 Testläufe Aktionsdefinition φ 57
6.3 Szenario 1 V1 Testläufe Aktionsdefinition ω 59
6.4 Szenario 1 V1 Bewertungen Testläufe 1, 2 und 3 60
6.5 Szenario 1 V1 Bewertungen Testläufe 4, 5 und 6 61
6.6 Szenario 1 V1 Bewertungen Testläufe 7 und 8 62
6.7 Szenario 1 V1 Aktionsberechnung Testlauf 6 64
6.8 Szenario 1 V1 Aktionsberechnung Testlauf 7 65
6.9 Szenario 1 V1 Vergleich Bewertung mit Vorhersage 66

6.10 Szenario 1 V2 Pareto-Optimierung 1 69
6.11 Szenario 1 V2 Pareto-Optimierung 2 70
6.12 Szenario 1 V3 Einfluss des spektralen Radius - ω (1) 72
6.13 Szenario 1 V3 Einfluss des spektralen Radius - ω (2) 73
6.14 Szenario 1 V3 Einfluss des spektralen Radius - ω (3) 74
6.15 Szenario 1 V4 Einfluss des spektralen Radius - φ (1) 77
6.16 Szenario 1 V4 Einfluss des spektralen Radius - φ (2) 78
6.17 Szenario 1 V4 Einfluss des spektralen Radius - φ (3) 79
6.18 Szenario 2 Bewertungsfunktionen 82
6.19 Szenario 2 V1 Testlauf 1 bis 3 86
6.20 Szenario 2 V1 Testlauf 4 und 5 87
6.21 Szenario 2 V2 Verlauf des Wohlbefindens 90
6.22 Szenario 2 V3 Verlauf des Wohlbefindens 92
6.23 Szenario 2 V4 Verlauf des Wohlbefindens 95

Abkürzungsverzeichnis

KNN Künstliches Neuronales Netz
FFN Feed Forward Network
RNN Recurrent Neural Network
RC Reservoir Computing
ESN Echo State Network
LSM Liquid State Machines
BPTT Back Propagation Through Time
RLS Recursive Least Squares
ADP Adaptive Dynamic Programming
ACD Adaptive Critic Design
MOP Multikriterielles Optimierungsproblem

Literaturverzeichnis

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

[Dörner et al., 2002] Dörner, D., Bartl, C., Detje, F., Gerdes, J., Halcour, D., Schaub, H., and Starker, U. (2002). Die Mechanik des Seelenwagens. Eine neuronale Theorie der Handlungsregulation.

[Fahlman, 1988] Fahlman, S. E. (1988). An empirical study of learning speed in back-propagation networks.

[Harmon and Harmon, 1997] Harmon, M. E. and Harmon, S. S. (1997). Reinforcement learning: A tutorial. Technical report, DTIC Document.

[Hornik, 1991] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

[Jaeger, 2001] Jaeger, H. (2001). The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148.

[Jaeger, 2002] Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach*. GMD-Forschungszentrum Informationstechnik.

[Koprinkova-Hristova et al., 2010] Koprinkova-Hristova, P., Oubbati, M., and Palm, G. (2010). Adaptive critic design with echo state network. In *Systems Man and Cyberne-*

tics (SMC), 2010 IEEE International Conference on, pages 1010–1015. IEEE.

[Koprinkova-Hristova et al., 2012] Koprinkova-Hristova, P., Oubbati, M., and Palm, G. (2012). Heuristic dynamic programming using echo state network as online trainable adaptive critic. *International Journal of Adaptive Control and Signal Processing*.

[Linder and Lindkvist, 2011] Linder, J. and Lindkvist, S. (2011). Interactive multiobjective optimization with application to hot rolling mills. Master's thesis, Chalmers University of Technology.

[Lukoševičius, 2012] Lukoševičius, M. (2012). A practical guide to applying echo state networks. In *Neural Networks: Tricks of the Trade*, pages 659–686. Springer.

[Lukoševičius et al., 2012] Lukoševičius, M., Jaeger, H., and Schrauwen, B. (2012). Reservoir computing trends. *KI-Künstliche Intelligenz*, 26(4):365–371.

[Maass et al., 2002] Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560.

[Malfaz and Salichs, 2010] Malfaz, M. and Salichs, M. (2010). Using muds as an experimental platform for testing a decision making system for self-motivated autonomous agents. *AISB J*, 2(1):21–44.

[Malfaz and Salichs, 2011] Malfaz, M. and Salichs, M. A. (2011). Learning to avoid risky actions. *Cybernetics and Systems*, 42(8):636–658.

[Marler and Arora, 2004] Marler, R. T. and Arora, J. S. (2004). Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. (1969). Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19:88.

[Oubbati et al., 2011] Oubbati, M., Kächele, M., Koprinkova, P., and Palm, G. (2011). Anticipating rewards in continuous time and space with echo state networks and actor-critic design. In *ESANN*, page 117–122.

[Oubbati et al., 2013] Oubbati, M., Oess, T., Fischer, C., and Palm, G. (2013). Multiobjective reinforcement learning using adaptive dynamic programming and reservoir computing.

[Oubbati et al., 2012] Oubbati, M., Uhlemann, J., and Palm, G. (2012). Adaptive learning in continuous environment using actor-critic design and echo-state networks. In *From Animals to Animats 12*, pages 320–329. Springer.

[Prokhorov et al., 1997] Prokhorov, D. V., Wunsch, D. C., et al. (1997). Adaptive critic designs. *Neural Networks, IEEE Transactions on*, 8(5):997–1007.

[Riedmiller and Braun, 1993] Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE.

[Rumelhart et al., 1986] Rumelhart, D. E., Hintont, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[Salichs and Malfaz, 2006] Salichs, M. A. and Malfaz, M. (2006). Using emotions on autonomous agents. the role of happiness, sadness and fear. In *Proceedings of the AISB'06 Symposium on Integrative Approaches to Machine Consciousness*, pages 157–164.

[Salichs and Malfaz, 2012] Salichs, M. A. and Malfaz, M. (2012). A new approach to modeling emotions and their use on a decision-making system for artificial agents. *Affective Computing, IEEE Transactions on*, 3(1):56–68.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT Press.

[Uhlemann, 2011] Uhlemann, J. (2011). Actor-Critic Design mit Echo State Networks zur Steuerung eines mobilen Roboters. Master's thesis, Universität Ulm.

[Vahidi et al., 2005] Vahidi, A., Stefanopoulou, A., and Peng, H. (2005). Recursive least squares with forgetting for online estimation of vehicle mass and road grade: theory and experiments. *Vehicle System Dynamics*, 43(1):31–55.

[Vrabie and Lewis, 2009] Vrabie, D. and Lewis, F. (2009). Neural network approach to continuous-time direct adaptive optimal control for partially unknown nonlinear systems. *Neural Networks*, 22(3):237–246.

[Wang et al., 2009] Wang, F.-Y., Zhang, H., and Liu, D. (2009). Adaptive dynamic programming: an introduction. *Computational Intelligence Magazine, IEEE*, 4(2):39–47.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

[Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

[Yun et al., 2004] Yun, Y., Nakayama, H., and Arakava, M. (2004). Generation of pareto frontiers using support vector machine. *MCDM'04*.

Name: Christian Fischer Matrikelnummer: 621904

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ulm, den .....

Christian Fischer