Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Dieter Büchler

Optimale Trajektorien mit Reinforcement Learning

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Dieter Büchler

Optimale Trajektorien mit Reinforcement Learning

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Ing Andreas Meisel
Zweitgutachter : Prof. Dr. rer. nat. Annabella Rauscher-Scheibe

Abgegeben am 12. April 2012

Dieter Büchler

Thema der Bachelorthesis

Optimale Trajektorien mit Reinforcement Learning

Stichworte

Optimale Trajektorien, autonomes Fahren, Q-Learning, Neural Fitted Q Iteration, künstliche neuronale Netze

Kurzzusammenfassung

Diese Arbeit umfasst die Modellierung, die Umsetzung und den Test eines Zustandssignals für einen Reinforcement Learning Agenten. Ziel ist es, so schnell, wie möglich, über eine Rennstrecke zu fahren, was mit der Suche nach einer optimalen Fahrspur verbunden ist. Mittel des Neural Fitted Q Iteration Algorithmus wird in einem kontinuierlichen Zustand- und Aktionsraum und ohne Modell der Umwelt Daten für die Q-Funktion gesammelt. Die Approximation dieser Funktion wird mit einem künstlichen neuronalen Netz umgesetzt.

Dieter Büchler

Title of the paper

Optimal Racing Lines with Reinforcement Learning

Keywords

Optimal Racing Lines, autonomous driving, Q-Learning, Neural Fitted Q Iteration, artificial neural network

Abstract

This work covers the development, the implementation and the test of a state signal for a Reinforcement Learning agent. The aim is to drive as fast as possible over a race circuit. That involves searching for an optimal racing line. Data for the Q-function is collected in a continuous action-state-space without a model of the environment using the Neural Fitted Q Iteration algorithm. The function approximation of the Q-function is done by an artificial neural network.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Andere Ansätze zur Suche nach optimalen Trajektorien 2

1.2 Ziele 2

1.3 Gliederung der Arbeit 3

2 Theoretische Grundlagen **4**

2.1 Reinforcement Learning 4

2.1.1 Der Agent und seine Umwelt 5

2.1.2 Ziel und Belohnungen 7

2.1.3 Markov Eingenschaft 9

2.1.4 Werte-Funktion und Bellmann-Gleichung 11

2.1.5 Q-Learning 15

2.1.6 Zustand- & Aktionsräume 18

2.2 Funktionsapproximation 19

2.2.1 Supervised Learning 19

2.2.2 Neuronale Netze 20

2.3 Neural Fitted Q-Iteration 22

3 Umsetzung/Implementierung **24**

3.1 Zustandsdarstellung 24

3.1.1 Leitfaden/Grundidee 25

3.1.2 Streckenabschnitte 27

3.1.3 Geschwindigkeit und „Distsum“ 35

3.1.4 Anpassungen an Torcs 37

3.2 Aktionsdarstellung 40

3.3 „Unstuck“-Modus 43

3.4 Neuronales Netzwerk 44

3.4.1 Topologie 45
3.4.2 Skalierung der Eingangswerte 46
3.4.3 Skalierung der Ausgangswerte 47
3.5 Gangschaltung 48

4 Kostenfunktion 49
4.1 Grundidee/Leitfaden 49
4.2 Variante 1 51
4.3 Variante 2 55
4.4 Variante 3 57
4.5 Variante 4 61

5 Zusammenfassung 62

6 Ausblick 63
6.1 ABS und Trajektionskontrolle 63
6.2 Teststrecke 63
6.3 Online- anstatt Batch-Lernen 64

A APPENDIX 71
A.1 Parallelisierung 71
A.2 selbstgeschriebene Simfunktion für Matlab 71
A.3 Restartbefehl in Torcs einfügen 73
A.4 Kommunikation 73
A.5 Tank und Schaden 74

Eigenständigkeitserklärung 75

1 Einleitung

Für erfahrene Autofahrer stellt die Suche nach einer optimalen Bahn zumindest theoretisch kein großes Problem dar. Man fährt die Kurve aussen an und schneidet sie möglichst so, dass man am Kurvenausgang wieder an der Außenseite ist. Man lenkt nur einmal ein und bremst nur so viel ab, wie unbedingt nötig. Am Ende einer Geraden sollte man wieder in einer guten Ausgangslage für die nächsten Abschnitte sein. Außerdem ist die ideale Fahrspur auf einer Rennstrecke durch den Gummiabrieb der vielen Reifen eindeutig eingezeichnet. Der Grund hierfür ist, dass viele Rennfahrer in vielen Versuchen Erfahrungen darüber gesammelt haben. So sind sie zum Entschluss gekommen einen bestimmten Streckenabschnitt in einer bestimmten Weise zu fahren, damit am Ende die Rundenzeit minimiert wird. Die Erfahrung besteht aus Belohnung und Bestrafung. Mal kommt das Auto von der Strecke ab, mal fährt man zu langsam durch eine Kurve und hat das Gefühl schneller fahren zu können. Es ist jedoch auch möglich eine Kurve so optimal zu treffen, dass die Sicherung genau dieser Erfahrung sich lohnt und nachahmungswürdig ist.

Wendet man künstliche Intelligenz zum Lernen des „schnellen Fahrens“ an, muss ein Computer, der das Auto steuert, den oben beschriebenen Vorgang nachahmen. Dieser steuernde Computer wird Agent genannt und spielt eine zentrale Rolle in dieser Arbeit. Der Lernaufwand für den Agenten wird noch durch das nicht vorhandene Vorwissen des Menschens erweitert. Ein Computer weiß nicht, was „außerhalb der Strecke“ bedeutet und konnte noch nie beobachten, wie ein Rennfahrer Gaspedal und Lenkung bedient.

Im nächsten Abschnitt werden andere Möglichkeiten genannt, um die oben erwähnten Aufgaben für den Agenten zu bewerkstelligen. Diese sollen helfen die Ziele dieser Bachelorarbeit im darauf folgenden Abschnitt zu formulieren. Danach wird ein kleiner Überblick über die Arbeit gegeben.

1.1 Andere Ansätze zur Suche nach optimalen Trajektorien

Die Suche nach idealen Fahrspuren ist verbunden mit dem Ziel die Rundenzeit zu minimieren. Seit 2007 findet jährlich das Torcs[1] Simulated Car Racing Championship[Loiacono u. a., 2009] statt. Torcs ist eine realistische Rennsimulation, die eine gute Schnittstelle für das Programmieren eines Agenten hat. Es sind viele Informationen über die Strecke und Zustand des Autos verfügbar. Somit bietet Torcs eine gute Umgebung für diesen virtuellen Wettbewerb. Jedes Team kann einen eigenen autonomen Agenten entwickeln und tritt mit diesem gegen andere Teams in verschiedenen Kategorien an. Viele Ansätze für das schnelle Fahren eines kompletten Rennens wurden dabei hervorgebracht. Es wurden sowohl Reinforcement Learning Algorithmen([Cardamone u. a., 2010],[Cardamone u. a., 2009]), Fuzzy-Logik [Onieva u. a., 2009] oder handcodierte Controller, bei denen fest einprogrammierte Regeln vorgegeben werden, verwendet. Andere Ansätze imitieren menschliche Fahrer([Van Hoorn u. a., 2009]). Die Gemeinsamkeit aller obigen Ansätze besteht darin, dass sie nur einen gewissen Radius der Strecke vor sich sehen und darauf basierend ihre Fahrweise optimieren. So ist es nicht möglich die Fahrspur über die komplette Strecke zu verbessern.
In dieser Bachelorarbeit kann der Agent weiter vorausschauen. Er erhält ähnliche Informationen über den weiteren Verlauf der Strecke, wie ein Rallye Fahrer. So hat der Agent die Möglichkeit eine optimale Trajektorie zu finden.

1.2 Ziele

- Es soll eine Umgebung geschaffen werden, die es ermöglicht mithilfe von Reinforcement Learning das „schnelle Fahren“ zu erlernen.
- Die dazu nötigen Schritte sollen mit der Theorie verständlich gemacht werden.

[1] The Open Racing Car Simulator

- Die entstandene Umgebung soll getestet und ausgewertet werden. Die Ergebnisse sollen kritisch hinterfragt und Probleme diskutiert werden.

1.3 Gliederung der Arbeit

Im zweiten Kapitel sollen Begriffe des Reinforcement Learnings, speziell das Q-Learning, sowie neuronaler Netze hergeleitet werden. Dieses Wissen ist erforderlich, um danach den NFQ[2] -Algorithmus[Riedmiller u. a., 2007] zu erläutern, der in dieser Bachelorarbeit verwendet wird.
Kapitel drei handelt von der Modellierung des Ziels „schnelles Fahren“ . Zuletzt findet eine Auswertung der Versuche statt, um zu zeigen welche Ziele erreicht werden konnten und welche weiteren Wege zielführend sein können.

[2] Neural Fitted Q Iteration

2

Theoretische Grundlagen

In dieser Bachelorarbeit soll ein Auto autonom das „schnelle Fahren“ über eine Rennstrecke unter der Bedingung, dass es nicht von der selbigen abkommt, erlernen. Eine gute Umsetzungsmöglichkeit hierfür bietet Reinforcement Learning(Bestärkendes Lernen), worauf im ersten Abschnitt dieses Kapitels eingegangen werden soll. Zunächst wird erläutert, wie man ein Problem formuliert, sodass es mit Reinforcement Learning gelöst werden kann.
Es wird möglichst nah am Beispiel erklärt und versucht die Ideen verdeutlicht als jeden mathematisch möglichen Fall abzudecken. Für weitergehende Informationen über RL wird [Sutton und Barto, 1998] empfohlen.
Im weiteren Verlauf wird eine mögliche Lösung mittels des NFQ-Algorithmus vorgestellt. Dazu werden im Vorfeld notwendige theoretische Aspekte neuronaler Netze beleuchtet, die Teil des NFQ-Algorithmus sind.

2.1 Reinforcement Learning

Reinforcement Learning(RL) ist der Überbegriff für eine Reihe von Methoden des Maschinellen Lernens. Der große Vorteil dieser Verfahren ist die Fähigkeit „künstlich“ Wissen aus Erfahrung zu generieren. Es wird von Beispielen gelernt und nach Beendigung der Lernphase verallgemeinert. Auf diese Weise werden Gesetzmäßigkeiten in den Lerndaten erkannt. Beispielsweise soll das Auto lernen, dass Positionen,die zwischen zwei benachbarten Punkten außerhalb der Strecke liegen, ebenfalls nicht auf der Fahrbahn sind und es sich nicht lohnt diese Positionen anzufahren. Die Verwendung von Vorwissen ist hierbei mit größter Vorsicht zu behandeln, da so die Wahrscheinlichkeit sinkt neue, evtl. bessere Wege zum Ziel zu finden. Im Gegenteil, es können sogar falsche Wege gefunden werden, falls Teilziele erreicht werden ohne dem Endziel näher zu

kommen.
Im Gegensatz dazu wird bei der klassischen Regelungstechnik Vorwissen für die Auswahl von Entscheidungen mit einbezogen. Bei vorhandenem und fundiertem Wissen über ein System muss nicht mehr gelernt werden. Ein passendes Beispiel wäre die Regelung der Wasserhöhe in einem Behälter mit Zu- und Abfluss. Will man jedoch das Schach-Spiel automatisieren, ist es schwer alle Fälle im Vorfeld mit Regeln abzudecken. Abhilfe schafft dabei ein lernender Agent, der u.a. Aktionen auswählt, die bisher noch nicht verwendet wurden, aber für ihn potentiell gut erscheinen. So wird das Wissen Schritt für Schritt vergrößert, sodass daraus sich Gesetzmäßigkeiten herauskristallisieren.

2.1.1 Der Agent und seine Umwelt

Der Lernvorgang basiert auf der Interaktion des Agenten mit seiner Umwelt. In Bild 2.1 wird der Vorgang grafisch dargestellt. Die Pfeile definieren dabei

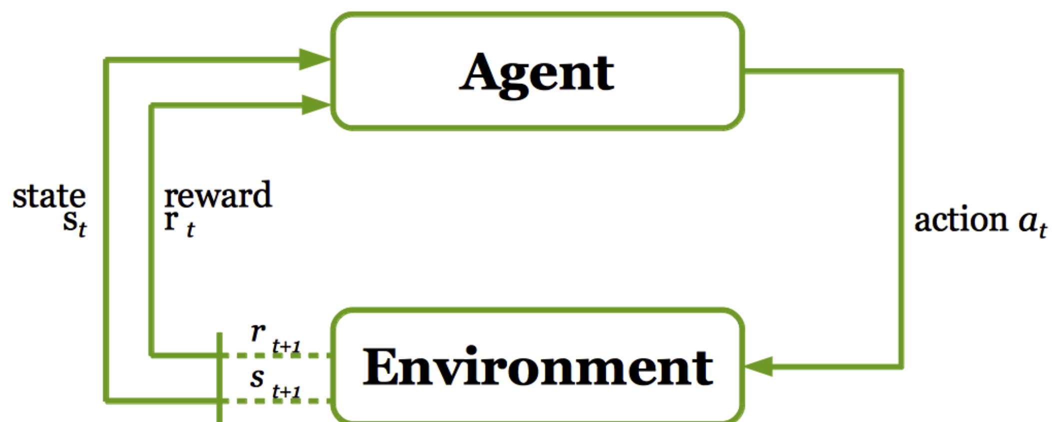


Bild 2.1: Interaktion zwischen Agent und Umwelt bei RL

die Kommunikation und ihre Richtung. Die gestrichelte Linie stellt den Zeitübergang dar.
Der Zustand

$$s_t = (s_{1t}, s_{2t} \dots s_{nt}) \in \mathcal{S} \tag{2.1}$$

für einen diskreten Zeitschritt $t \in [1, 2, \dots, T]$(engl. state) ist dabei ein Abbild der Umwelt, das dem Agenten für ihn relevante Aspekte anzeigt. Gibt es keine

natürliche Diskretisierung der Zeit, wie beim Schachspielen - ein Zug, ein Zeitschritt -, muss in konstanten Schritten abgetastet werden.

Beispiel Ein Zustand kann dabei die Umwelt nicht zu 100% abbilden, da z.B. ein Laufroboter nur eine begrenzte Zahl von Sensoren hat. Dies ist jedoch kein Nachteil, da der Laufroboter nur Kenntnisse über eigene Position, die Hindernisse vor sich und den Status seiner Aktoren benötigt. Wissen über die Anzahl der Personen, die sich im selben Raum befinden, sind überflüssig. Sind irrelevante Informationen gegeben, müssen diese trotzdem verarbeitet und ausgewertet werden. Der Lernvorgang wird dadurch verlangsamt.

episodische/kontinuierliche Aufgaben

Handelt es sich um ein Problem, das kein natürlich gegebenes Ende hat, wie beispielsweise das Balancieren einer Stange auf einem Finger, läuft der finale Zeitabschnitt T gegen unendlich. Man spricht von einer kontinuierlichen Aufgabe(engl. continuing task). Im Gegensatz dazu weisen beispielsweise Brettspiele ein natürliches Ende auf. Bei Sieg oder Niederlage eines Spielers wird neu angefangen. Dann spricht man von einer episodische Aufgabe(engl. episodic task). Oft ist es nützlich eine kontinuierliche zu einer episodischen Aufgabe umzuformulieren. Dies kann erreicht werden, wenn man beispielsweise jedes mal abbricht, wenn die Stange ihre optimale Position auf dem Finger für drei Sekunden hält. Die Art der Modellierung hat Einfluss auf die Lerngeschwindigkeit und Parametrierung. Dies wird in Abschnitt 2.1.2 auf Seite 7 näher erläutert.
Seinen eigenen Zustand in der Umwelt kann der Agent durch Aktionen(engl. action) verändern. Er beeinflusst damit den Folgezustand s_{t+1}.

Genauso wie ein Zustand hat auch eine Aktion

$$a_t = (a_{1t}, a_{2t} \dots a_{mt}) \in \mathcal{A} \tag{2.2}$$

eine Ordnung $m = n \vee m \neq n$ und wird aus einer Menge von möglichen Aktionen $\mathcal{A}$ analog zu den Zuständen(Menge $\mathcal{S}$) gewählt. Worauf es bei der Definition des Zustandes und der Aktion für ein bestimmtes Problem in RL

ankommt, wird in Abschnitt 2.1.3 auf Seite 9 erläutert.
Jede Aktion a_t im Zustand s_t impliziert eine Kritik[Alpaydin, 2008] $r(s_t, a_t, s_{t+1}) = r_{t+1}$ bzw. $c(s_t, a_t, s_{t+1}) = c_{t+1}$[1] und einen Folgezustand s_{t+1}. Diese kann positiv, Belohnung(engl. reward), oder negativ, Kosten(engl. cost),ausgelegt werden, wobei beide im Prinzip das selbe aussagen. Die Belohnungsfunktion $r(s_t, a_t, s_{t+1})$ sagt dem Agenten, was die Ziele sind, aber nicht wie er sie erreichen soll. Auf diesen Zusammenhang wird in Abschnitt 4.1 auf Seite 49 eingegangen.

2.1.2 Ziel und Belohnungen

Das Ziel des Agenten ist es die Summe aller zukünftigen Belohnungen zu maximieren. D.h. der Agent muss in jedem Zeitschritt die zu erwartende zukünftige kumulative Belohnung[2] (engl. return)

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T = \sum_{k=0}^{T} r_{t+k+1} \tag{2.3}$$

vom aktuellen bis zum terminalen Zustand im Zeitschritt T abschätzen können. Hier wird schon klar, dass eine Funktion benötigt wird, die als Argument den Zustand entgegen nimmt und den obigen Wert ausgibt. Dann kann der Agent sich den Folgezustand s_{t+1} mit dem höchsten Gewinn aussuchen, den er mit einer Aktion a_t erreichen kann. Formal wählt er a_t aus der Menge im Zustand s_t möglicher Aktionen $\mathcal{A}(s_t)$ aus, die ihn in den Folgezustand mit dem höchsten R_t-Wert bringt. Im Verlauf dieser Arbeit werden Aktionen als gut bzw. schlecht bezeichnet. Dies geschieht immer im Bezug auf den Gewinn. Führen verschiedene Aktion den Agenten in verschiedene Folgezustände, deren Gewinn sich unterscheidet, so ist die Aktion „besser“, die den Agenten zum Folgezustand mit dem höheren Gewinn führt. An dieser Stelle soll R_t eingeführt werden, um im weiteren Verlauf dieses Kapitels zu zeigen, wie dieser Wert abgeschätzt wird.

[1] In der betreffenden Literatur wird die Theorie mit Belohnungen hergeleitet. Daran soll sich hier gehalten werden. Im praktischen Teil dieser Arbeit werden jedoch Kosten modelliert.
[2] Im weiteren Verlauf kurz Gewinn genannt.

verzögerte Belohnung

RL wird oft auf Probleme angewendet, bei denen eine aussagekräftige Kritik r erst sehr spät erhalten wird. Bei einem Spiel wird als Ziel des RL-Problems der Gewinn ausgegeben. Aussagen über Sieg oder Niederlage können jedoch erst am Ende des Spiels getroffen werden, sodass alle vorherigen Kritiken neutral sind. Man kann sich vorstellen, dass der Agent viele Durchgänge benötigt, bevor er weiß, welche Entscheidungen zielführend sind. In solch einem Fall spricht man von einer verzögerten Belohnung(engl. delayed reward).

Diskontierung

Im Falle eines kontinuierlichen Problems läuft der Gewinn leicht gegen unendlich, da $T \rightarrow \infty$. So kann jedoch keine Funktion gebaut werden, die diesen Wert abschätzt. Um diesem Problem entgegen zu treten, wird die Diskontierungsrate γ wie folgt verwendet:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.4}$$

Sie fügt zukünftigen Belohnungen ein Gewicht hinzu. Ist $\gamma < 1$ wird die Belohnung r_{t+1}, die weniger weit in der Zukunft liegt als r_{t+3}, höher gewichtet. Gilt beispielsweise $\gamma = 0.9$,

$$\begin{aligned} R_t = \sum_{k=0}^{\infty} 0.8^k r_{t+k+1} &= 0.9^0 r_{t+1} + 0.9^1 r_{t+2} + 0.9^2 r_{t+3} \dots \\ &= r_{t+1} + 0.9 r_{t+2} + 0.81 r_{t+3} \dots \end{aligned}$$

wird r_{t+3} mit 0.81 und r_{t+1} mit 1 gewichtet. Also kann man sagen, dass mit der Höhe der Diskontierungsrate der „Weitblick“ eingestellt wird. Bei $\gamma = 0$ würde der Gewinn nur die nächste Belohnung beschreiben. Verwendet der Agent dies als Grundlage für die Wahl der nächsten Aktion, wird nur die aktuelle Belohnung maximiert. Durch die Verwendung der Diskontierungsrate erhält R_t die Form einer geometrischen Reihe. Also konvergiert R_t unter der Bedingung, dass $\gamma < 1$ ist und die einzelnen Belohnungen r_t begrenzt sind.
Nun können sowohl episodische ($\gamma = 1$)als auch kontinuierliche Probleme($T = \infty$) formal beschrieben werden. Dabei können jedoch nicht beide Fälle gleich-

zeitig auftreten($\gamma = 1$ und $T = \infty$). Natürlich weiß der Agent nicht, welche Schritte er später macht, und hat somit kein Wissen über zukünftige Belohnungen. Abhilfe schafft dabei die Verwendung einer Strategie π, die in Abschnitt 2.1.4 auf Seite 12 erläutert wird.

2.1.3 Markov Eingenschaft

Wie Bild 2.1 auf Seite 5 zeigt, erhält der Agent seine Belohnung aufgrund der Aktion a_t, die er im Zustand s_t ausgeführt hat. Diese führt ihn in den Folgezustand s_{t+1}, wo er die Belohnung r_{t+1} erhält. Dabei ist es wichtig, dass alle relevanten Information, für die der Agent kritisiert wird, im Zustandssignal vorhanden sind. Das folgende Beispiel soll wichtige Punkte bei der Zusammenstellung des Zustandsingals verdeutlichen.

Beispiel Nehmen wir an, dass ein Roboterarm eine Stange balancieren soll. Als Zustand werden zunächst die Winkel $s_{1t} = \alpha(t)$ und $s_{2t} = \beta(t)$ definiert(siehe Bild 2.2). Der Roboter kann den Ursprung des Koordinatensystems verschieben. Zwei Szenarien sollen nun durchgespielt werden. Zunächst wird die Stange

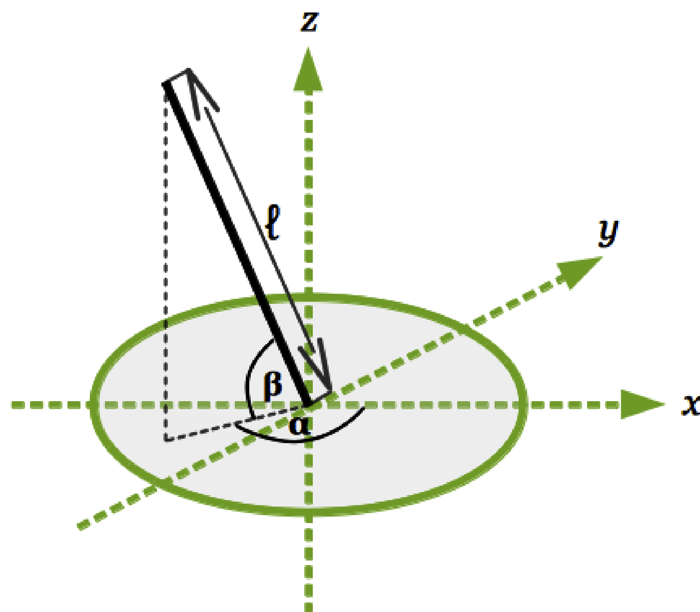


Bild 2.2: Skizze zum Beispiel „Roboter balanciert eine Stange"

in die Position $\beta_1(0) = 89°$ und $\alpha = \alpha_1$ gebracht und dann losgelassen. Sie

fällt in die x-y-Ebene. Danach wird die Stange in die Position $\beta_2(0) < \beta_1(0)$ und $\alpha = \alpha_1$ gebracht und fällt, wie im ersten Szenario, in die x-y-Ebene. Es sei angenommen, dass beide Szenarien ohne Einwirken des Roboters ablaufen. Für den Agenten, der seine Umwelt nur durch das Zustandssignal sieht, sind beide Daten im Bereich $\beta = 0°$ bis β_2 identisch. Würde der Roboter das Fallen verhindern wollen, müsste er jedoch mit zwei verschiedenen Geschwindigkeiten den Koordinatenursprung in Richtung α_1 verschieben, um die beiden Fallbewegungen abzufangen. Dies kann er jedoch nicht lernen, da beide Zustände der Fallbewegung aus seiner Sicht identisch sind. Er benötigt zusätzlich die Winkelgeschwindigkeiten $s_{3t} = \dot{\alpha}(t)$ und $s_{4t} = \dot{\beta}(t)$ im Zustandssignal.
Das obige Beispiel zeigt, dass ein Zustandssignal die Historie zusammenfassen muss(verschiedene Startpositionen führen zu verschiedenen Winkelgeschwindigkeiten). Dies hat zur Folge, dass die Belohnung r_{t+1} und der Folgezustand s_{t+1} nur vom aktuellen Zustand s_t und der dort gewählten Aktion a_t abhängen. Nur so können aufgrund des aktuellen Zustandes und der gewählten Aktion, der Folgezustand und die Belohnung vorhergesagt werden. Dann besitzt die Zustandsmodellierung die Markov-Eigenschaft.

Markov'sches Entscheidungsproblem

So wird ein RL-Problem zu einem Markov'schen Entscheidungsprozess[3]. Ein MDP mit endlichen Zustands- und Aktionsräumen $\mathcal{S}$ und $\mathcal{A}$ entspricht einem finiten Markov'schen Entscheidungsprozess[4]. Dann gilt im allgemeinen Fall für die Transitionswahrscheinlichkeit

$$\mathcal{P}^a_{ss'} = p\left[s_{t+1} = s | s_t = s, a_t = a\right]. \tag{2.5}$$

In Worten: die Wahrscheinlichkeit, dass durch die Wahl von Aktion $a_t = a$ in Zustand $s_t = s$ im Zeitschritt t der Folgezustand $s_{t+1} = s'$ eintritt, beträgt $\mathcal{P}_{ss'}$ und ist ausschließlich abhängig von s und a. Genauso ist im Allgemeinen die

[3] engl. Markov Decission Process. Im weiteren Verlauf MDP genannt.
[4] Im weiteren Verlauf fMDP genannt.

Belohnung nicht deterministisch:

$$\mathcal{R}^a_{ss'} = E\left[r_{t+1} = s | s_t = s, a_t = a\right]. \tag{2.6}$$

Das bedeutet, bei Ausführung der selben Aktion im selben Zustand zu verschiedenen Zeitpunkten kann der Agent verschiedene Belohnungen erhalten. Diese folgen jedoch einer stochastischen Verteilung, sodass ein Erwartungswert gebildet werden kann. Ein solches Verhalten tritt z.B. bei verrauschten Sensorsignalen auf. Es wird vorausgesetzt, dass die Umwelt stationär ist. D.h. der Erwartungswert dieser Verteilung verändert sich nicht mit der Zeit.
Das System in dieser Arbeit ist sogar ein deterministisches. Nach [Röttger, 2009] ist dann die Belohnung $r(s, a, s')$ sowie der Folgezustand s' eindeutig und in jedem Zeitschritt gleich, in dem a in s ausgeführt wird.

$$\mathcal{P}^a_{ss'} = \begin{cases} 1 & \text{für } s \xrightarrow{a} s' \\ 0 & \text{sonst} \end{cases} \tag{2.7}$$

$$\mathcal{R}^a_{ss'} = r(s, a, s') \tag{2.8}$$

$\mathcal{P}^a_{ss'}$ und $\mathcal{R}^a_{ss'}$ werden Umweltmodellparameter genannt.

2.1.4 Werte-Funktion und Bellmann-Gleichung

In Abschnitt 2.1.2 wurde von der Notwendigkeit einer Funktion gesprochen, die den Gewinn R_t abschätzt. Dabei ist es zunächst einleuchtend diesen Wert in Abhängigkeit eines Zustandes s_t anzugeben. Im Falle der Kenntnis über die Umweltmodellparamter $\mathcal{P}^a_{ss'}$ und $\mathcal{R}^a_{ss'}$ kann der Agent im Zustand s_t alle möglichen Folgezustände mit dieser Funktion „abtasten“ und die Aktion wählen, die ihn zu dem am höchsten abgetasteten Folgezustand führt. Eine Schätzung kann durch den Erwartungswert ausgedrückt werden und führt zu der Zustand-Wert-Funktion(engl. state-value-function)

$$V^\pi(s) = E_\pi[R_t | s_t = s] = E_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} | s_t = s\right]. \tag{2.9}$$

Sie gibt an, wie günstig es für den Agenten ist sich im Zustand s_t zu befinden, wenn danach die Strategie $\pi(s)$ befolgt wird.

Strategie

Eine Strategie ist im Allgemeinen eine stochastische Funktion, die besagt, wie hoch die Wahrscheinlichkeit ist, dass der Agent in einem Zustand s eine ihm dort zur Verfügung stehende Aktionen der Menge $\mathcal{A}(s)$ wählt.

$$\pi : \mathcal{S} \times \mathcal{A}(s) \rightarrow \mathbb{R} \tag{2.10}$$

In dem hier vorliegenden deterministischen Fall sind diese Wahrscheinlichkeiten für eine Aktion 1 und für die restlichen 0. Also macht es mehr Sinn direkt die eine Aktion auszugeben, die in s mit der Wahrscheinlichkeit 1 gewählt wird.

$$\pi : \mathcal{S} \rightarrow \mathcal{A}(s) \tag{2.11}$$

Die Strategie soll am Ende des Lernvorgangs so angepasst sein, dass sie für jeden Zustand die optimale Aktion anzeigt. Dann ist sie die optimale Strategie π^*. Man beachte, dass eine optimale Aktion in einem bestimmten Zustand nicht die Aktion mit der höchsten kurzfristigen Belohnung ist. Vielmehr führt diese zu weiteren guten[5] Zuständen, sodass der Gewinn maximiert wird.
Nun wird deutlich, wie es möglich ist den Gewinn R_t abzuschätzen. Der Mangel an Wissen über zukünftige Aktionen wird durch eine zu Beginn zufällig gewählte Strategie ausgedrückt. Dadurch macht man das Unwissen greifbar und ermöglicht gesammelte Erfahrung mit einzubeziehen. Das selbe gilt für die Zustands-Wert-Funktion $V^\pi(s)$, denn sie wird ebenfalls zufällig oder mit Werten, die zumindest nicht den Endwerten entsprechen, initialisiert und im Laufe des Lernvorgangs verbessert.
Dabei haben π und $V^\pi(s)$ einen direkten Zusammenhang. Die Zustand-Wert-Funktion bewertet eine Strategie. Wählt man eine Strategie, die den Agenten in Zustände bringt, die schlechtere Belohnungen mit sich tragen, als eine andere, werden verschiedene Werte für den selben Zustand in $V^\pi(s)$ erzeugt. Wie eine Strategie in Hinblick auf $V^\pi(s)$ verbessert werden kann , wird später in diesem

[5] Die Güte wird durch die Zustand-Wert-Funktion ermittelt.

Abschnitt erläutert.

Bellman-Gleichung

Nun soll die Idee der Aktualisierung der Zustands-Wert-Funktion $V^\pi(s)$ aufgezeigt werden. Genauer gesagt, sollen gesammelte Daten bzw. Erfahrungen verwertet werden, indem diese in $V^\pi(s)$ einfließen.
In [Bellman, 1957] wurde für Gleichung 2.9 auf Seite 11 ein Rekursionsformel gefunden. Dabei wurden die allgemeinen Formen der Umweltmodellparamter angenommen, sodass auch der Fall einer stochastischen Umwelt damit behandelt werden kann. Später wird die Gleichung auf den deterministischen Fall, der hier vorliegt, angepasst.

$$\begin{aligned} V^\pi(s) &= E_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s\right] \\ &= E_\pi\left[r_{t+1}+\gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s\right] \\ &= \sum_a \pi(s,a)\sum_{s'}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'}+\gamma E_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s'\right]\right] \\ &= \sum_a \pi(s,a)\sum_{s'}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'}+\gamma V^\pi(s')\right] \end{aligned} \tag{2.12}$$

Es soll nicht haarklein jeder Umformungschritt begutachtet, sondern die letzte Umformung 2.12 ausgewertet werden.
Es wird mit der Strategie π mit der Umwelt interagiert. Im stochastischen Fall wird jede mögliche Aktion a in Zustand s mit der Transitionswahrscheinlichkeit gewichtet. Im deterministischen Fall ist die Wahrscheinlichkeit für genau eine Aktion 1 und die restlichen 0. Es wird also in einem Zustand nur eine Aktion gewählt. Zu notieren ist, dass dieser Term bekannt ist und genauso wie $V^\pi(s)$ aktualisiert werden muss.
Des Weiteren erscheinen die Umweltmodellparamter $\mathcal{P}^a_{ss'}$ und $\mathcal{R}^a_{ss'}$. Nehmen wir an, diese seien bekannt, dann können Werte aus den Verteilungen(stochastischer Fall) entnommen werden, sodass es keine Unbekannte Werte in dieser Gleichung gibt, denn auch $V^\pi(s')$ und $\pi(s,a)$ sind initialisiert. Mithilfe von dynamischer Programmierung, worauf hier nicht weiter eingegangen werden soll, kann auch ohne Interaktion mit der Umwelt(denn die Parameter sind bekannt, die an-

sonsten von der Umwelt geliefert werden würden) eine Lösung für die optimale Strategie π^* und Zustand-Wert-Funktion $V^*(s)$ gefunden werden.
In den meisten Fällen sind die Umweltmodellparameter jedoch nicht bekannt. Die obige Rekursionsformel eröffnet eine Möglichkeit $V^\pi(s')$ schrittartig zu aktualisieren. Dies soll kurz für ein deterministisches System angeschnitten werden. Wie in Gleichungen 2.7 und 2.8 auf Seite 11 schon erwähnt, können folgende Ersetzungen vorgenommen werden

$$\begin{aligned} \sum_{s'} \mathcal{P}^a_{ss'} &\rightarrow 1 \\ \mathcal{R}^a_{ss'} &\rightarrow r_{t+1} \\ \Rightarrow V^\pi(s) =& \pi(s)\left[r_{t+1} + \gamma V^\pi(s')\right]. \end{aligned} \tag{2.13}$$

Dadurch hängt die Zustand-Wert-Funktion nur noch von der Belohnung im nächsten Zeitschritt r_{t+1} und vom Wert der Zustands-Wert-Funktion des Folgezustandes $V^\pi(s')$ ab. Nun kann man sich für den angenommenen deterministischen Fall vorstellen Schritt für Schritt die Belohnung zu sammeln und die Zustands-Wert-Funktion zu aktualisieren. Dies ist keine mathematisch fundierte Aussage, aber zeigt die Idee und den Einstiegspunkt für die Aktualisierung vieler RL-Algorithmen und damit die Wichtigkeit der Bellman Gleichung.

Zusammenhang zwischen π & V^π

Analog dazu kann eine Strategie, wie folgt, verbessert werden:

$$\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')]. \tag{2.14}$$

Wurde dieser Vorgang für alle Zustände aus $\mathcal{S}$ ausgeführt, ist die Strategie „gierig"(engl. greedy) im Bezug auf die aktuelle Zustand-Wert-Funktion, da nun immer die aktuell beste Aktion für einen bestimmten Zustand ausgegeben wird(engl. policy improvement).
Wird nun die aktualisierte Strategie verwendet, um mit der Umwelt zu interagieren, erneuern sich die Werte $V^\pi(s)$ durch Anwendung der Rekursionsformel 2.12 in jedem Schritt(engl. policy evaluation). Nun kann man sich vorstellen, dass durch Iteration dieser beiden Vorgänge(engl. generalized policy iteration) die Strategie und die Zustand-Wert-Funktion sich ihren optimalen Werten

annähern. In Bild 2.3 wird dieser Vorgang dargestellt.

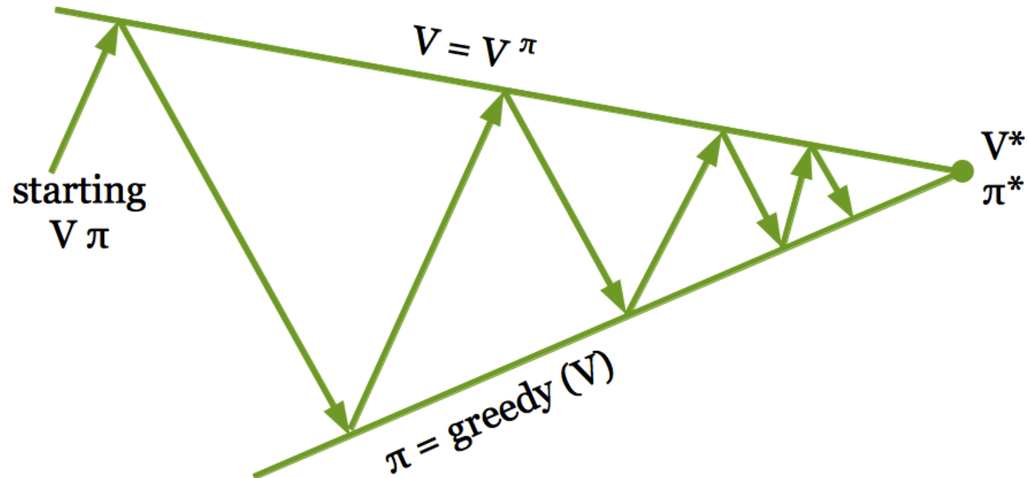


Bild 2.3: Iterative gegenseitige Verbesserung hinsichtlich des Gewinns von V^π und π

Exploration/Exploitation

Der oben erläuterte Effekt kann durch den Besuch neuer, bisher unerkundeter Zustände vergrößert werden. Im leichtesten Fall wird mit der Wahrscheinlichkeit ϵ nicht $\pi(s)$ befolgt, sondern eine zufällige Aktion gewählt(Exploration). Demzufolge wird mit der Wahrscheinlichkeit $1 - \epsilon$ die zum Zeitpunkt t als beste angenommene Aktion gewählt(Exploitation). Sinnvoll ist es zunächst mit einem hohen ϵ zu starten und dieses dann weiter zu senken, aber nie zu Null werden zu lassen. So wird das immer größer werdende Wissen des Agenten mit der Zeit mehr und mehr ausgenutzt und eine fortlaufende Exploration des Agenten sichergestellt. Solche Verfahren werden Explorationsstrategien genannt.
Für viele RL-Algorithmen ist die Konvergenz gegen V^* und π^* unter der Voraussetzung der fortwährenden Exploration bewiesen worden. Es müssen also alle Zustands-Aktionspaare unendlich oft besucht werden.

2.1.5 Q-Learning

Nun soll der implementierte RL-Algorithmus, Q-Learning[Watkins, 1989], vorgestellt werden. Das System dieser Arbeit ist ein deterministisches, da die

Belohnungsfunktion, die in Abschnitt 4.1 auf Seite 49 erläutert wird, für den selben Zustand s und Aktion a zu verschiedenen Zeitpunkten die selbe Belohnung r und den selben Folgezustand s' zur Folge hat. Q-Learning ist Teil des hier verwendeten NFQ-Algorithmus.

Nehmen wir erstmal den allgemeinen Fall an, dass kein Wissen über die Umweltmodellparameter $P^a_{ss'}$ und $R^a_{ss'}$ vorhanden ist. Zusätzlich ist das System stochastisch. Dann kann der Agent nicht wissen, in welchen Zustand ihn eine bestimmte Aktion führt. Bei der Wahl einer bestimmten Aktion a in einem bestimmten Zustand s führt diese den Agenten zu verschiedenen Zeitpunkten t möglicherweise in verschiedene Folgezustände s_{t+1}. Also ist die Zustand-Wert-Funktion $V^\pi(s)$ in diesem Fall nutzlos, da der Agent nicht alle möglichen Folgezustände s_{t+1} von s_t abtasten kann. Daher wird die Aktion-Wert-Funktion(engl. action-value function) eingeführt:

$$Q^\pi(s_t, a_t) = E_\pi[R_t | s_t = s, a_t = a] \tag{2.15}$$

Sie gibt an, wie gut es für den Agenten ist in Zustand s_t Aktion a_t auszuführen und dann der Strategie π zu folgen (Definition von $V^\pi(s)$: siehe Seite 11). Um den Unterschied bzw. den Vorteil zu $V^\pi(s)$ zu verdeutlichen, wurde $Q^\pi(s_t, a_t)$ erst jetzt eingeführt. Alle Schritte, die anhand von $V^\pi(s)$ erklärt wurden, können ebenfalls mit $Q^\pi(s_t, a_t)$ hergeleitet werden.

So gilt z.B. analog zu Gleichung 2.12 für die Bellman Gleichung

$$Q^\pi(s_t, a_t) = \sum_{s_{t+1}} \mathcal{P}^a_{s_t s_{t+1}} \left[\mathcal{R}^a_{s_t s_{t+1}} + \gamma \max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1}) \right]. \tag{2.16}$$

Im allgemeinen Fall gilt für die Aktualisierungsregel des Q-Learning

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_b Q(s_{t+1}, b) - Q(s_t, a_t)]. \tag{2.17}$$

Für den deterministischen Fall(siehe dazu Abschnitt 2.1.3 auf Seite 9) wird Gleichung 2.16 zu

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_b Q(s_{t+1}, b). \tag{2.18}$$

Auch im deterministischen Fall hat die Q-Funktion einen Vorteil gegenüber der V-Funktion. Man stelle sich vor, der Agent befindet sich in einem Zustand s, in dem er sich noch nie vorher befunden hat, wie z.B. im Zeitschritt $t = 0$. Da das System deterministisch ist, ist es eindeutig, dass jede gewählte Aktion $a \in \mathcal{A}(s)$ den Agenten in einen bestimmten Folgezustand s' bringt. Dies gilt auch für alle anderen Zeitpunkte. Aus folgenden Gründen ist hier $V(s)$ trotzdem nutzlos:

- Der Agent kennt die Menge der Folgezustände nicht und kann diese nicht in $V(s)$ einsetzen.
- Es ist nicht bekannt welche Aktion $a \in \mathcal{A}(s)$ der Agent ausführen muss, um in einem bestimmten Folgezustand s' zu gelangen.

Der Agent könnte nun in jedem Zustand jede mögliche Aktion ausprobieren und das Resultat aufzeichnen. Dies würde jedoch einem erarbeiten eines Modells der Umwelt gleich kommen.
Zusammengefasst kann man sagen, dass bei Fehlen von Wissen über die Umweltmodellparamter $\mathcal{R}^a_{ss'}$ und $\mathcal{P}^a_{ss'}$[6] immer die Q-Funktion gewählt werden muss.

strategiefrei/-basiert

Um einen Q-Wert zu aktualisieren wird eine sog. Transition

$$(s_t, a_t, r_{t+1}, s_{t+1})$$

aufgezeichnet. In dieser Transition ist die Aktion b, die für den Q-Wert des Folgezustands bei der Aktualisierung des aktuellen Q-Wertes benötigt wird, nicht enthalten(siehe Gleichung 2.18 auf Seite 16). Die Aktion $b \in \mathcal{A}(s_{t+1})$ wird so gewählt, dass der höchst mögliche Wert für $Q(s_{t+1}, b)$ eintrifft(max-Funktion). Im Grunde wird hier eine Strategie angewendet. Abhängig vom Zustand wurde eine Aktion gewählt, egal welche Gesetzmäßigkeit dahinter steht. Diese Strategie wird Schätzungsstrategie genannt(engl. evaluation policy).
Sie unterscheidet sich von der Verhaltensstrategie(engl. behaviour policy), anhand der der Agent die nächste Aktion wählt. Bisher wurde diese gemeint,

[6] Man spricht dann auch davon, dass kein Modell der Umwelt vorliegt.

wenn Strategien erwähnt wurden. Zusammengefasst heißt das, mit der Schätzungsstrategie wird der Q-Wert aktualisiert und mit der Verhaltensstrategie die nächste Aktion gewählt. Eine solche Methode wird strategiefrei(off-policy) genannt.

SARSA

Als Gegenbeispiel soll die SARSA-Methode$(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ als strategiebasierten Algorithmus vorgestellt werden. Die Aktualisierungsformel sieht der des Q-Learnings sehr ähnlich:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (2.19)$$

Elementar anders ist die Verwendung der Schätzungsstrategie. Der für die Aktualisierung benötigte Q-Wert des nächsten Zustandes, erhält für die Berechnung die Aktion a_{t+1}, die auch für den nächsten Schritt verwendet wird. Also stimmen hier Schätzungs- und Verhaltensstrategie überein. Solche Methoden werden strategiebasiert(engl. on-policy) genannt.
Welche Art von Methode ist besser? Fest steht, sucht sich der Agent seine Aktionen anhand der Q-Funktion, die mit Q-Learning aktualisiert wurde, aus, geht er immer den, für den aktuellen Zeitpunkt, am „besten“ erscheinenden Weg. Es kann sich jedoch herausstellen, dass die Anwendung der „besten“ Aktion eine trotzdem geringe Belohnung mit sich trägt(stochastisches System). Im Gegensatz dazu, wird bei SARSA die Strategie, die angegeben wurde, Schritt für Schritt aktualisiert. Die Aktualisierungsformel des Q-Learnings scheint das Ziel schneller zu finden, da sie, anstatt der aktuellen Strategie nachzugehen, den besten gefundenen Wert verwendet. Sie approximiert Q^* anstatt Q^π.

2.1.6 Zustand- & Aktionsräume

Bisher wurde davon ausgegangen, dass alle Werte einer Q- oder V-Funktion tabellarisch abgespeichert werden. Dies ist bei begrenzten und vor allem kleinen Zustands- und Aktionsräumen denkbar. Problematisch wird diese tabellarische Anordnung in folgenden Fällen:

- Es gibt viele mögliche diskrete Zustände und Aktionen. Bei hohen Dimensionen m und n[7] kann es dazu leicht kommen.
- Die Zustände und Aktionen sind kontinuierlich. Dann muss in jedem Zeitschritt abgetastet werden. Ein geringer Unterschied zweier abgetasteten kontinuierlichen Werte würde trotzdem zu unterschiedlichen Einträgen in der Tabelle führen, obwohl Zustand und Aktion fast gleich sind.

Sind die Räume deterministisch, bildet nicht nur die Größe des Speicherplatzes eine Herausforderung. Es bedarf sehr vieler Episoden, um alle Zustands-Aktionspaare zu besuchen. Im kontinuierlichen Fall ist dies sogar unmöglich, da es unendlich viele gibt.
Wünschenswert wäre eine Funktion, die als Argumente Zustand s und Aktion a entgegen nimmt und den jeweiligen Q-Wert ausgibt. Einem bisher unbesuchten Zustands-Aktionspaar soll ein Q-Wert, der im Verhältnis zu den benachbarten, bereits besuchten Zustands-Aktionspaaren steht, zugewiesen werden. Es soll also generalisiert werden. Dies ist ein klassisches Problem des „Supervised Learning(SL)[8]". Der Unterschied zu RL, sowie die Aspekte des SL sollen im nächsten Abschnitt verdeutlicht werden.

2.2 Funktionsapproximation

In diesem Abschnitt sollen theoretische Aspekte künstlicher neuronaler Netze aufgezeigt werden, um Anforderung an diese stellen zu können. In Abschnitt 3.4 auf Seite 44 werden konkrete Implementierungsdetails vorgestellt.

2.2.1 Supervised Learning

Anhand des hier behandelten Problems soll der Unterschied zwischen SL und RL erläutert werden. Wie in Bild 2.4 zu sehen ist, wird das komplette System als Blackbox angesehen, das über Eingänge sowie Ausgänge verfügt. In

[7] siehe Abschnitt 2.1.1 auf Seite 5
[8] Überwachtes Lernen

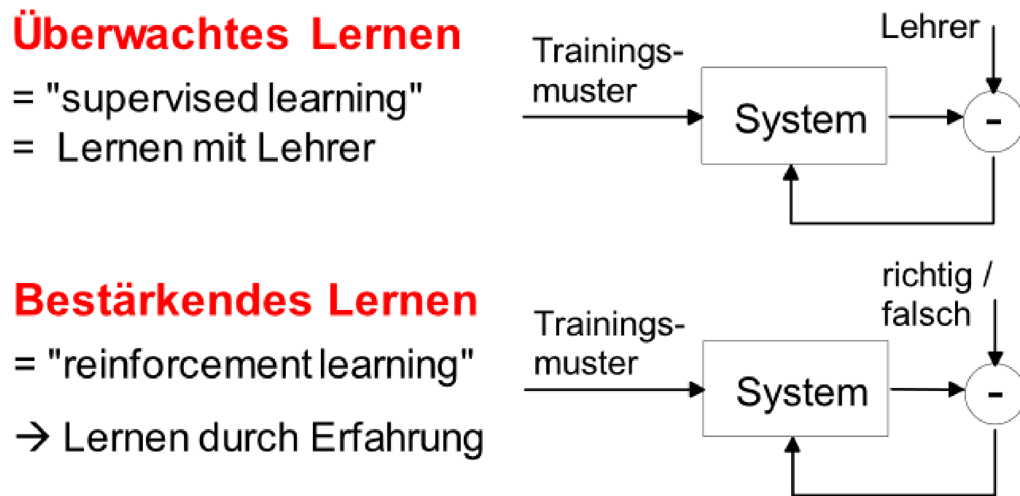


Bild 2.4: Unterschied zwischen Reinforcement Learning und Supervised Learning

beiden Fällen erhält das System eine Repräsentation seiner Position auf der Strecke(Trainingsmuster) und soll die Beschleunigung und Lenkung festlegen. Beim RL wird die Ausgabe mit richtig oder falsch bzw. einem numerischen Wert, der proportional zur Güte der Ausgabe ist, bewertet. In einer Rechtskurve würde die Aktion „links abbiegen“ schlechter belohnt werden als in einer Linkskurve. Darauf aufbauend wird das System angepasst, sodass in zukünftigen Zeitschritten bessere Kritiken erhalten werden.
Beim SL erhält das System die richtige Lösung und wird daraufhin angepasst(Linkskurve → “links abbiegen“). In dieser Aufgabe ist jedoch kein Lehrer vorhanden. Daher kann das vorliegende Problem, so wie es hier formuliert wird, nicht mit SL gelöst werden.

2.2.2 Neuronale Netze

Ein künstliches neuronales Netz besteht aus mehreren Neuronen. In Bild 2.5 ist die Funktionsweise dargestellt. Es wird folgende Rechenvorschrift gebildet:

$$\text{out} = f_{\text{act}}\left(\sum_{i}^{d} \text{in}_i \cdot w_i + \text{bias}\right) \tag{2.20}$$

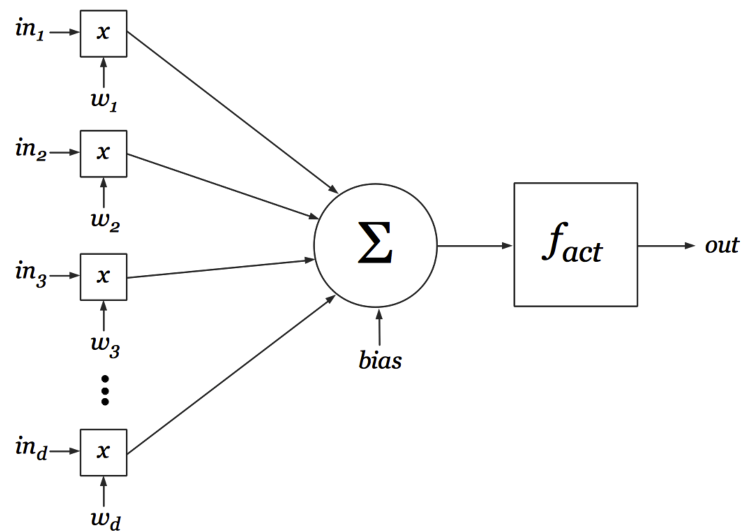


Bild 2.5: Neuron

Die Aktivierungsfunktion kann als lineare, Schritt- oder Sigmoidfunktion realisiert sein. Die Auswahl hängt von der Aufgabenstellung ab.
Durch Kombinationen von mehreren Neuronen in mehreren Schichten kann das entstehende Netz kompliziertere Funktionen darstellen. In Bild 2.6 wird ein Feedforward Netz gezeigt, wobei die einzelnen Kreise jeweils ein Neuron darstellen. Trainiert werden neuronale Netze, indem ein Trainingsmuster am

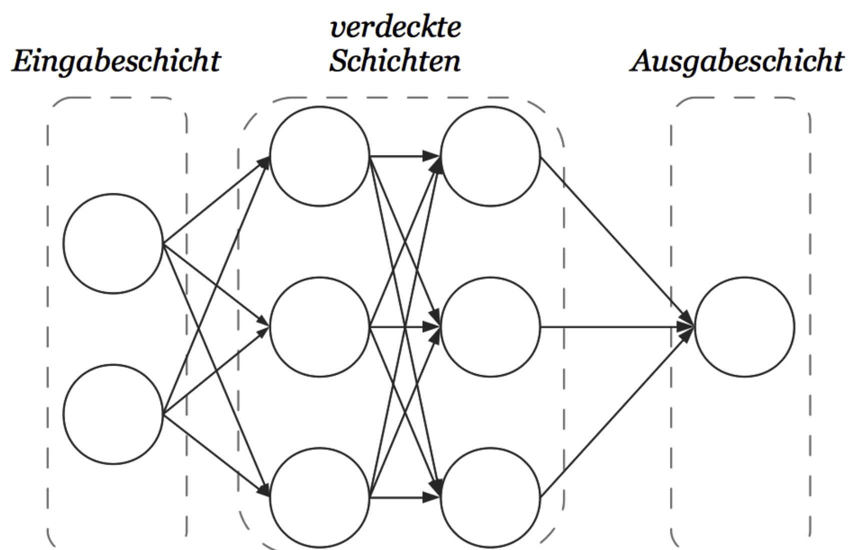


Bild 2.6: Feedforward Netzwerk

Eingang und die Sollwerte am Ausgang angelegt werden. Die Gewichte sowie der bias-Wert werden so verändert, dass die Differenz zwischen Soll- und

Ist-Wert möglichst gering ist. Dazu suchen viele Lernalgorithmen das Minimum im Fehlergebirge, wobei die anzupassenden Werte die Achsen bilden.
Grundsätzlich muss zwischen Batch- und Online-Lernen unterschieden werden. Im ersten Fall werden die Änderungen gesammelt, die bei der Suche nach dem Minimum für jedes angelegte Trainingsmuster gefunden werden. Die Änderung wird dann einmal durchgeführt. Beim Online-Lernen werden die Gewichte nach jedem einzelnen Muster aktualisiert.

Generalisierung

Voraussetzung für die Anwendung einer Funktionsapproximationsmethode beim RL ist, dass ähnliche Eingangswerte ähnliche Ausgangswerte erzeugen. D.h. bei Abfrage der Q-Funktion mit einem noch nicht bekannten Zustand-Aktions-Paar soll ein Q-Wert ausgegeben werden, der ähnlich zu de benachbarten bereits bekannten Q-Werten ist. Dann spricht man von einer guten Generalisierung.
Beeinflussen kann man dies mit der Wahl der Topologie, also die Anzahl der Schichten, sowie die Anzahl der Neuronen innerhalb der Schichten. Je mehr Neuronen und Schichten verwendet werden, umso kompliziertere Funktionen können abgebildet werden. Dabei kann folgendes Problem auftauchen: Werden zu viele Neuronen verwendet, kann zwischen nahe liegenden Abtastpunkten eine komplizierte Form hineingelegt werden. Befinden sich die Abtastpunkte in einem Bereich des Zustandsraums, der der Aufgabe entsprechend gut abgetastet wurde, spricht man von Überanpassung(engl. Overfitting). Auch bei Verwendung von zu wenig Neuronen können Probleme auftauchen, da zu viel generalisiert werden könnte. Daher muss die Topologie eines neuronalen Netzes je nach Trainingsdaten angepasst werden(siehe 3.4 auf Seite 44).

2.3 Neural Fitted Q-Iteration

In dieser Arbeit kommt der NFQ-Algorithmus aus [Gabel u. a., 2011] zum Einsatz. Bei konventioneller Verwendung von Q-Learning und künstlichen neuronalen Netzen wird dieses im Online-Lernverfahren angepasst. Nach jeder gesammelten Transition werden die Gewichte verändert. Ändert man jedoch

einen Bereich im Zustandsraum, werden die restlichen mit beeinflusst. Dies kann bisherige eingebundene Erfahrung löschen, was zu einem verlangsamten Lernverhalten führt. NFQ löst dieses Problem, indem anstatt Online-Lernen, Batch-Lernen angewendet wird. Die Transitionen werden bei der Interaktion mit der Umwelt gesammelt und nach einer gewissen Zeit oder Ende einer Episode simultan trainiert. Dazu wird der Fehler im Mittel über alle Trainingsmuster minimiert und dann der Gradientenabstieg durchgeführt. So wird die generalisierende Eigenschaft neuronaler Netze ausgenutzt und gleichzeitig die oben beschriebene negative Eigenschaft[9] vermieden.

[9] In [Wilson und Martinez, 2003] wird hingegen gezeigt, dass Online-Lernen in vielen Fällen dem Batch-Lernen überlegen ist. Diese Eigenschaft ist also umstritten.

3 Umsetzung/Implementierung

Nun soll konkret beschrieben werden, wie die vorliegende Aufgabe als RL-Problem formuliert wurde. Dafür werden zunächst die Schritte erläutert, die nötig waren, um den Zustandsraum u.a. im Hinblick auf einen MDP[1] anzupassen. Im Anschluss wird gezeigt, wie eine Aktion in einem Zustand aus dem kontinuierlichen Raum gewählt werden kann. Die Aktionen

- Stellung des Gas- bzw. Bremspedals und
- Stellung des Lenkrads

müssen in jedem Zeitschritt definiert werden. Zum Schluss wird ein Einblick über wichtige Schritte zur Verbesserung der Lerngeschwindigkeit des neuronalen Netzwerks gegeben.
Diese hier erarbeiteten Kenntnisse sind notwendig, um im nächsten Kapitel auf verschiedene Varianten der Kostenfunktion $c(s, a, s')$ eingehen zu können.

3.1 Zustandsdarstellung

Ein Zustand bei RL besteht aus mehreren Komponenten(siehe Abschnitt 2.1.1 auf Seite 5)

$$s_t = (s_{1t}, s_{2t} \dots s_{nt})$$

und dient dazu dem Agenten ausschließlich die Informationen zu geben, die für die Lernaufgabe elementar sind. Alle zusätzlichen Informationen verlangsamen die Lerngeschwindigkeit, da zusätzlicher Input auch ausgewertet werden muss.

[1] siehe Abschnitt 2.1.3 auf Seite 9

3.1.1 Leitfaden/Grundidee

Im Falle diskreter Zustände[2] sind, neben unbrauchbaren zusätzlichen Informationen, vor allem ein zu großer Zustandsraum für langsames Lernen verantwortlich. So können viele Kombinationen der Zustandskomponenten $s_{it} \; \forall \; i = 1, 2 \ldots n$ entstehen, was eine hohe Anzahl von möglichen Zuständen zur Folge hat. Man kann sich vorstellen, dass der Agent zumindest viele der möglichen Zustände gesehen haben muss, um eine gute(wenn auch nicht optimale) Strategie zu finden.

Auch bei Verwendung einer Generalisierungsmethode[3] muss der Zustandsraum möglichst in jeder Ecke abgetastet werden. Das bedeutete nicht, dass jeder Punkt im Zustandsraum besucht werden muss, sondern die Abtastpunkte sind so zu wählen, das dazwischenliegende Punkte ausreichend generalisiert werden können. Die Generalisierungsfunktion sollte an vakanten Punkten, bildlich gesprochen, festgenagelt werden, sodass sie nicht viele Möglichkeiten hat komplizierte Formen in die Zwischenräume reinzulegen. Im Bild 3.1 werden

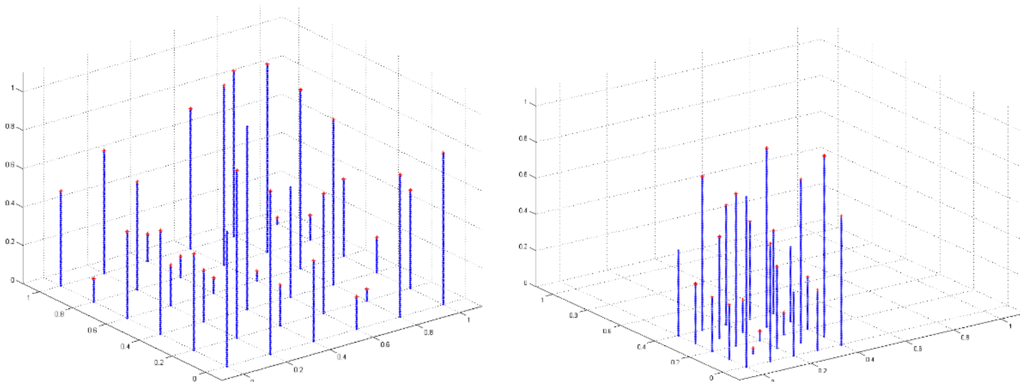


Bild 3.1: Gute und schlechte Abtastung eines Zustandsraums

eine gute(links) und eine schlechte Aufspannung(rechts) eines Zustandsraums dargestellt. Im nicht besuchten Bereich im rechten Teilbild hat die Generalisierungsmethode keine Anhaltspunkte, welche Form sie dort annehmen soll. Wird die Generalisierungsfunktion in so einem Bereich abgefragt, können beliebige Wert zurückgegeben werden. Dies führt zur Auswahl ungünstigerer oder zur

[2] wie z.B. bei Brettspielen

[3] siehe 2.1.6 auf Seite 18

Außerachtlassung guter Aktionen.

Im Falle kontinuierlicher Zustände muss ohnehin generalisiert werden. Dort nützt es jedoch nichts, die Größe des Intervalls einer Zustandskomponente zu verändern(z.B. von [-10,10] auf [-1,1]), da es trotzdem unendlich viele Zwischenwerte gibt. Daher muss die Zustandsdarstellung relativ zum Agenten formuliert werden. Dies gilt auch für diskrete Zustände.

Beispiel Der Begriff der Relativität einer Zustandskomponente soll anhand eines Beispiels mit kontinuierlichen Zuständen näher erläutern werden. Nehmen wir an, ein Modellflugzeug soll lernen einen gegebenen Punkt im dreidimensionalen Raum anzufliegen. In Bild 3.2 ist die Anordnung der Position des Flugzeuges

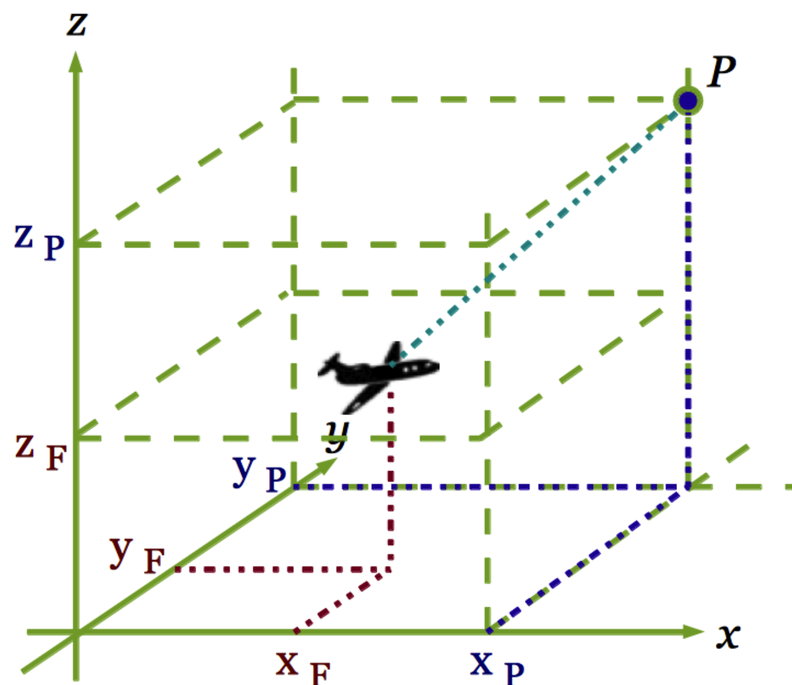


Bild 3.2: Beispiel: Flugzeug soll zu Punkt P fliegen lernen

$F = (x_F, y_F, z_F)$ und der des Zielpunkts $P = (x_P, y_P, z_P)$ verdeutlicht. Gibt

man dem Agenten als Zustandssignal den Zielpunkt in globalen Koordinaten

$$s_{1t} = x_P$$
$$s_{2t} = y_P$$
$$s_{3t} = z_P$$

an, erhält das Flugzeug jedes mal, wenn P sich ändert, andere Zahlenwerte. Gibt man stattdessen die Koordinaten des Zielpunkts in Abhängigkeit zu der Position des Flugzeuges

$$s_{1t} = x_P - x_F$$
$$s_{2t} = y_P - y_F$$
$$s_{3t} = z_P - z_F$$

an, treten gleiche Zahlenwerte öfter auf[4]. Also kommen ähnliche Zustände öfter vor. Befindet sich der Agent häufiger in einem bestimmten Bereich des Zustandsraums, ist es wahrscheinlicher diesen hinreichend für die Generalisierung aufzuspannen und damit die Lerngeschwindigkeit zu erhöhen.
Zusammengefasst sollten folgende wichtige Punkte bei der Zustandsdimensionierung beachtet werden:

- Nur die Informationen in das Zustandssignal einfließen lassen, die wirklich nötig sind, um das Lernziel zu erreichen.
- diskrete Räume: Das Intervall der verschiedenen Zustandskomponenten $s_{it} \; \forall \; i = 1, 2 \ldots n$ verringern.
- Relative Angaben.

3.1.2 Streckenabschnitte

In diesem Abschnitt soll die konkrete Implementierung des Zustandsignals vorgestellt werden. Dafür muss das Lernziel nochmals genauer definiert werden,

[4] In diesem Beispiel werden nur die Teile des Zustandsignals beachtet, die die Position des Zielpunkts P betreffen. Weiter Komponenten müssten bei einer Umsetzung hinzugefügt werden.

denn der Agent soll nur die Informationen erhalten, die er für die Erfüllung seines Lernziels benötigt. In der Einleitung wurde als Hauptziel das Lernen des „schnellen Fahrens“ über eine Strecke angegeben. Dies ist untrennbar mit der Suche nach einer optimalen Trajektorie über die gegebene Strecke verbunden. Dafür darf der Agent nicht von der Strecke abkommen und muss sich daher Wissen über die Streckengrenzen selbst erarbeiteten können. Neben der Suche nach der optimalen Trajektorie, muss der Agent seine Geschwindigkeit, je nach Streckenpassage, anpassen. In Abschnitt 3.1.3 auf Seite 35 wird darauf näher eingegangen. Viele andere Ansätze sehen die Strecke nur einen gewissen Radius im Voraus. Damit kann der Agent höchstens lernen, im Sichtfeld optimal die Linie zu halten. Ein Rennfahrer hat jedoch bereits Vorwissen über ganze Streckenpassagen, die vor ihm liegen. Nur so kann ein Fahrer lernen z.B. S-Kurven zu fahren. Also erhält der Agent Informationen über den weitere Verlauf der Strecke. Dabei wurde sich an dem Wissen eines Rallye-Fahrers orientiert. Sie haben immer einen Co-Piloten dabei, der Streckeninformationen, wie Kurvenradien, Längen und Form der nächsten Abschnitte, zum richtigen Zeitpunkt mitteilt.

Unterschied: Segment ↔ Abschnitt

Um alle diese Faktoren zu berücksichtigen, erhält der Agent Eckpunkte von Streckenabschnitten. Diese werden relativ zu seiner Position berechnet. Auf die Umrechnung von globalen in relative Streckenpunkte wird in Abschnitt 3.1.4 auf Seite 37 eingegangen. Die Streckenpunkte werden in Bild 3.3 verdeutlicht. Jede Strecke in Torcs besteht aus einzelnen Segmenten, die hier grau gestrichelt dargestellt sind. Als Streckenabschnitt sollen die schwarz eingezeichneten Blöcke definiert werden. Ihre Eckpunkte werden

$$SL \rightarrow \text{start left}$$
$$SR \rightarrow \text{start right}$$
$$EL \rightarrow \text{end left}$$
$$ER \rightarrow \text{end right}$$

Bild 3.3: Streckenpunkte,Abschnitte und Segmente

bezeichnet. Der nächste Abschnitt wird durch die Endpunkte des aktuellen Abschnitts(EL,ER) und nächsten Abschnitts(EL1,EL2) definiert. Sobald sich in Fahrtrichtung der Typ eines Segments (straight, left turn, right turn) oder der Radius einer Kurve ändert, wird ein neuer Abschnitt begonnen.

Veränderung bei Bewegung des Autos

Die Startpunkte des aktuellen Segments verändern sich, während das Auto fährt. Verlässt das Auto ein Segment des aktuellen Abschnitts, verkleinert sich der Abschnitt, um das verlassene Segment. Dies wird in Bild 3.4 visualisiert. Der aktuelle Abschnitt wird immer kleiner, wenn sich das Auto in Fahrtrichtung

Bild 3.4: Veränderung des aktuellen Abschnitts während der Fahrt

der Strecke fortbewegt. Davon wird sich versprochen, dass die Koordinaten der Startpunkte(SL,SR) sich nicht stark verändern. Mit größerer Entfernung der Streckenpunkte zum Auto werden die Zahlenwerte der Koordinaten größer, da diese relativ zum Auto gegeben werden.
Verlässt das Auto den aktuellen Abschnitt, also überschreitet es die Linie zwischen den Punkten EL und ER, wird der nächste Abschnitt zum aktuellen, und die nächsten k gegebenen Abschnitte rücken auf die Position des jeweils nächsten Abschnitts. Dieser Sachverhalt wird in Bild 3.5 verdeutlicht. Die

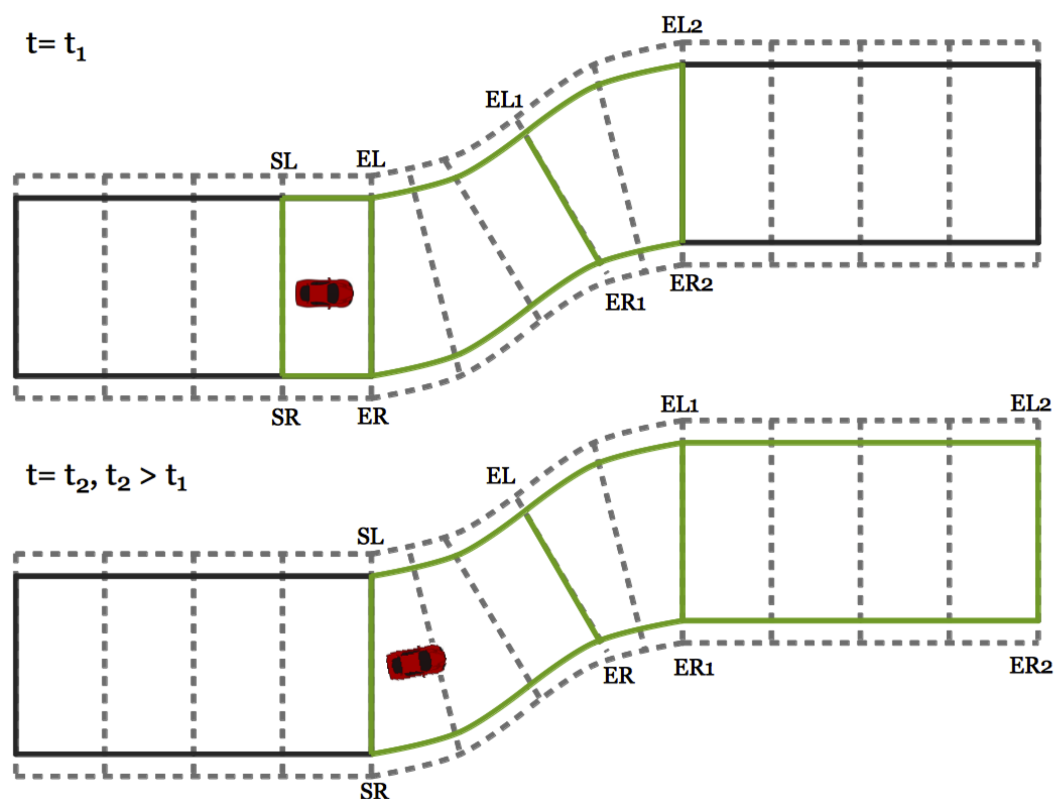


Bild 3.5: Wechsel des aktuellen Abschnitts

Abschnitte, die zu den Zeitpunkten $t = t_1$ bzw. $t = t_2$ gegeben sind, sind grün hervorgehoben.

Ziele dieser Zustandsrepresentation

Das Zustandssignal beinhaltet also die Eckpunkte des aktuellen Abschnitts und die von k weiteren Abschnitten. Erhält der Agent nur die Eckpunkte des aktuellen Abschnitts, hat er ausreichend Informationen, um zu lernen, wohin er fahren soll. Des Weiteren kann der Agent aus der Anordnung der Punkte

des aktuellen Abschnitts erkennen, um welchen Abschnittstyp es sich handelt. Bild 3.6 zeigt ein Geradenstück und zwei Kurven mit verschiedenen Radien.

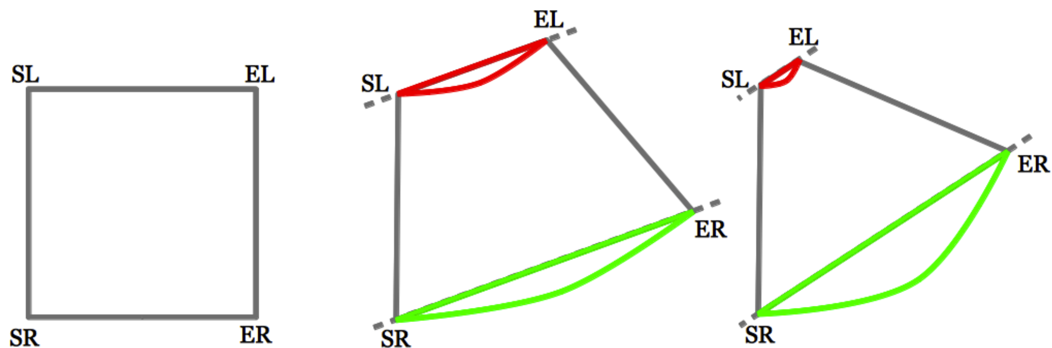


Bild 3.6: Form aus Anordnung der Streckenpunkte

Bestraft man den Agenten jedes mal, wenn er von der Strecke abkommt, kann dieser folgende Rückschlüsse aus der Anordnung ziehen:

- Rechteck → Linien $\overline{\mathrm{SL}, \mathrm{EL}}$ und $\overline{\mathrm{SR}, \mathrm{ER}}$ nicht überschreiten.
- kein Rechteck → roten Bereich zwischen SL und EL nicht anfahren und grüner Bereich zwischen SR und ER kann benutzt werden.

Solche Zusammenhänge kann er herausfinden, da eine fortschreitende Exploration vorausgesetzt wird. Also muss er irgendwann in diese Bereiche gelangen und erhält dort eine positive bzw. negative Belohnung r_t, aus der er die oben genannten Rückschlüsse ziehen kann.
Lernt der Agent die oben genannten Zusammenhänge aus der Anordnung der Streckenpunkte, ist er unabhängig von der Streckenbreite und den Radien der Kurven. Erlaubte und nicht erlaubte Bereiche kann er aus der Anordnung der Punkte identifizieren. Davon wird sich versprochen, das Wissen einer Strecke auf eine andere Strecke übertragen zu können. Aufgrund dieses *generischen* oder *relativen Charakters* der Formulierung wurde das Zustandssignal auf diese Weise definiert.

Bisher wurde ausgearbeitet, was der Agent mit dem Wissen über die Eckpunkte des aktuellen Abschnitts lernen kann. Das Zustandssignal wird jedoch um k weitere Abschnitte, also 2 zusätzliche Eckpunkte pro k, erweitert. Der Sinn dieser Information soll anhand eines Beispiels verdeutlicht werden.

Beispiel In den Bildern 3.7 bis 3.9 ist jeweils vier mal das selbe Streckenstück abgebildet. Es besteht aus 2 Kurven und 3 Geraden, die jeweils einen Abschnitt darstellen. Das Auto startet jedes mal von der selben Position in dem ersten Geradenstück. In jedem Bild fährt es von Abschnitt zu Abschnitt. Das Ziel dabei ist die Linie zwischen den letzten Punkten, des am weitesten im Voraus liegenden Abschnitts. Wenn das Auto am Ende eines Abschnittes angekommen ist, verändert sich die Situation, weil der Agent dann andere Abschnitte im Zustandssignal findet.
Auf dem vorherigen Bild aufbauend, wird die neue Situation erklärt. Grün illustriert sind dabei die Abschnitte über deren Eckpunkte der Agent in seinem Zustandssignal verfügt. Ziel ist es zu zeigen, welche Trajektorien der Agent bei verschiedenen k lernen kann. Dabei wird angenommen, dass der Agent schon viele Erfahrungen gesammelt hat und die Belohnungsfunktion korrekt formuliert ist, sodass er in vielen Situation schnell fährt. Die Trajektorie, die der Agent in diesem Beispiel fährt, ist blau eingezeichnet. Blau gestrichelt ist der Weg, den er in seiner dargestellten Situation anvisiert. Es sei darauf hingewiesen, dass dies *mögliche* Trajektorien sind. Es wird lediglich aufgezeigt, was der Agent mit dem gegebenen Wissen lernen kann.
Zunächst wird die Entstehung der einzelnen Trajektorien erläutert, damit danach ein Vergleich zwischen ihnen gezogen werden kann.
In Bild 3.7 erhält der Agent nur die Streckenpunkte des aktuellen Abschnitts. In Teilbild 1 beschleunigt er stark, um möglichst schnell die Linie $\overline{EL, ER}$ zu erreichen. Im nächsten Abschnitt(Teilbild 2) merkt er erst, dass er rechts abbiegen muss und schafft dies am linken Rand der Rechtskurve. Dann folgt wieder eine Gerade, in der er voll beschleunigt und geradeaus fährt. Teilbild 4 zeigt, dass die Beschleunigung zu hoch war und der Agent so von der Strecke abkommt, weil er in Teilbild 3 keine Information über die Linkskurve aus Teilbild 4 hatte.
Bild 3.8 zeigt den Agenten, ausgestattet mit den Informationen über den aktuellen sowie nächsten Abschnitt. Im Gegensatz zu Teilbild 1 des Bildes 3.7, weiß der Agent sofort dass er die Linie $\overline{EL1, ER1}$ erreichen soll, dabei aber nicht über den rechten Rand in der Rechtskurve hinaus kommen darf. Also entschliesst er sich den geraden und kürzesten Weg zu Linie $\overline{EL1, ER1}$ zu

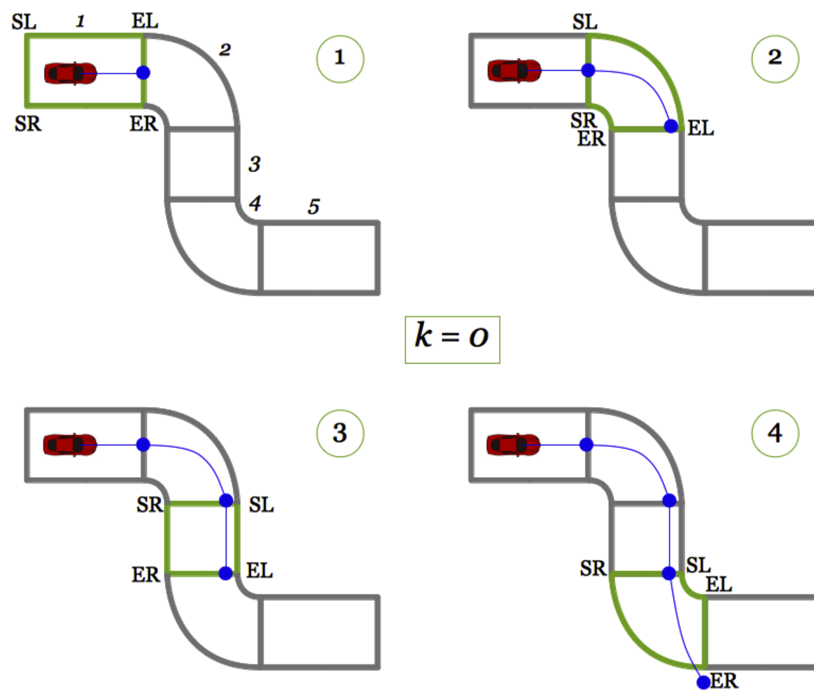


Bild 3.7: mögliche Trajektorie für $k = 0$

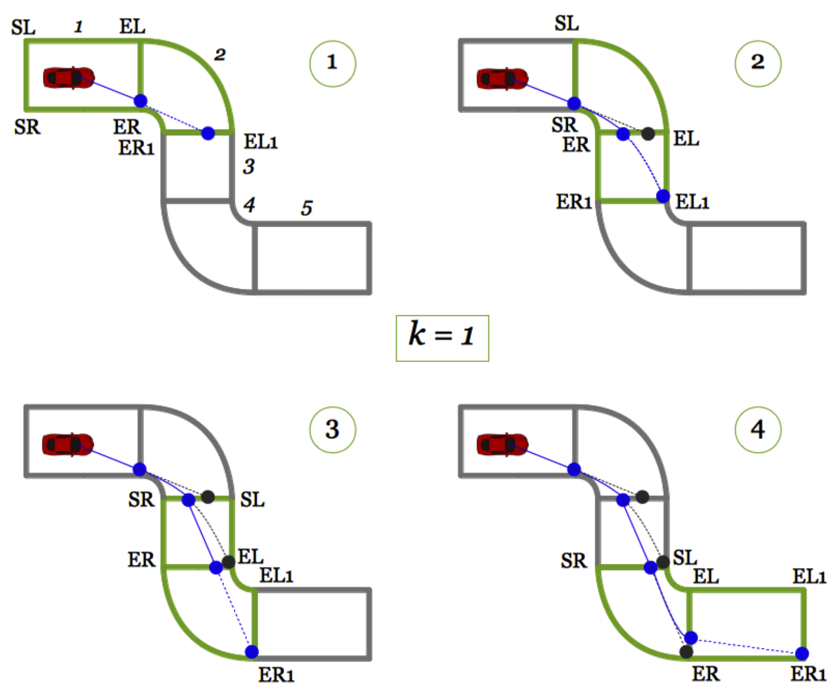


Bild 3.8: mögliche Trajektorie für $k = 1$

wählen. Der Agent beschleunigt stark, da er sein Ziel schnell erreichen soll. In Teilbild 2 bremst er ein wenig ab und stellt die Lenkung stärker nach rechts ein, damit er die Linie $\overline{EL1, ER1}$ am linken Rand noch erwischt. Dies validiert der Agent im nächsten Teilbild, da er sonst von über den linken Rand der Linkskurve hinaus fahren würde. Er lenkt etwas mehr nach rechts ein und kann so knapp am Rand vorbei fahren, direkt geradeaus auf den rechten Rand der Linie $\overline{EL1, ER1}$. In Teilbild 4 korrigiert der Agent seinen Kurs, indem er abbremst und die Ziellinie am Ende des letzten Geradenstücks anpeilt.

Zuletzt soll noch eine mögliche Trajektorie über die gegebene Strecke mit

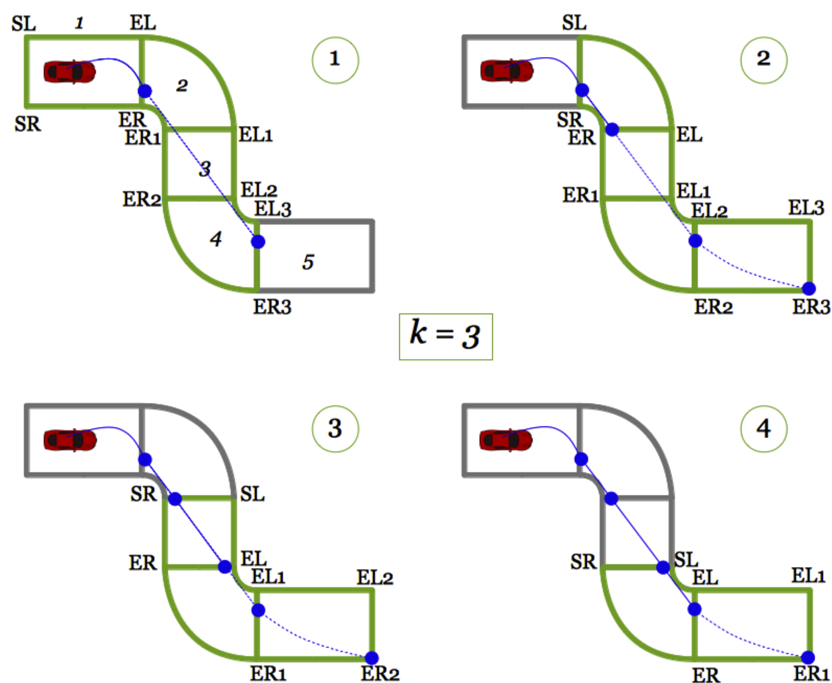


Bild 3.9: mögliche Trajektorie für $k = 3$

$k = 3$ extra Abschnitten im Zustandssignal gezeigt werden(Bild 3.9). Da der Agent hier bereits in Abschnitt 1 über Kenntnis der Kurvenkombination verfügt, plant er zwischen den Scheitelpunkten der beiden Kurven geradeaus zu fahren. Dafür fährt er die Kurve aussen an und beschleunigt dann stark. In Teilbild 2 verändert sich nichts an seinem Vorhaben aus Teilbild 1, bis auf das Verhalten nach dem Scheitelpunkt der Linkskurve. Nun weiß der Agent, dass auf die Linkskurve eine Gerade folgt und nutzt dieses Wissen, um die Lenkung nach dem Scheitelpunkt der Linkskurve so einzustellen, dass er den rechten Rand der Linie $\overline{EL3, ER3}$ erreicht. Es wird angenommen, dass nach der letzten Geraden

keine weiteren Abschnitte mehr kommen, sodass sich die Trajektorie nicht mehr ändert.

Vergleich der Trajektorien

Nun werden die Trajektorien der jeweiligen Teilbilder 4 aus den Bildern 3.7 bis 3.9 verglichen. Es erscheint logisch, dass je mehr Wissen der Agent über vorliegende Abschnitte hat, er umso bessere Aktionen wählen kann. Doch was macht der Agent konkret anders? Analysiert man die Trajektorie aus Teilbild 4 in Bild 3.7, so erkennt man, dass der Agent sich am Ende der Linkskurve außerhalb der Strecke befindet. Dies geschieht, weil er kein Wissen über nächste Abschnitte hat und so erst reagieren kann, wenn er sich in der besagten Linkskurve befindet. Also sinkt die Wahrscheinlichkeit von der Strecke abzukommen, je größer k gewählt wird. Die Trajektorie aus Teilbild 4 in Bild 3.8 zeigt eine mögliche Verbesserung.
Aus Bild 3.9 kann der Schluss gezogen werden, dass bei genügend hohem k für eine bestimmte Kurvenkombination, der Agent lernen könnte die Kurven aussen anzufahren und sie zu schneiden. Im Extremfall könnte der Agent sogar die optimale Trajektorie über die komplette Strecke finden, wenn er alle Abschnitt in seinem Zustandssignal erhält. Ausreichen würden jedoch auch so viele Abschnitte, dass durch hinzukommen neuer Informationen, die gedachte Trajektorie des aktuellen Abschnitts und des nächsten sich nicht ändert. Dies ist in Bild 3.9 der Fall. Dabei sei gesagt, dass der Agent sich keine Trajektorie „denkt“. Damit ist gemeint, dass die trainierte Strategie ihn so durch das Streckenstück führen würde, wenn seine Kenntnisse so blieben.

3.1.3 Geschwindigkeit und „Distsum“

Geschwindigkeit

Der Agent soll zum einen eine optimale Bahn über die Strecke finden. Zum anderen soll er die Geschwindigkeit anpassen, die er durch die Stellung des Gas- bzw. Bremspedals beeinflusst. Daher müssen die drei Komponenten der Geschwindigkeit(vx,vy und vz) im Zustandssignal vorhanden sein. Dies ist

auch im Hinblick auf die Markov-Eigenschaft wichtig[5]. Über seine Position wird der Agent durch die Streckenpunkte informiert. An einer bestimmten Position kann er verschiedene Geschwindigkeiten haben. Von dieser hängt ab, ob der Agent bremsen oder beschleunigen soll. Ist die Geschwindigkeit nicht im Zustandssignal vorhanden, basiert die Entscheidung über die folgende Aktion nicht nur auf dem gegebenen Zustand. So ist die Markov-Eigenschaft nicht gegebenen.

„Distsum“

Der „Distsum“-Wert wird in manchen Ansätzen im Zustandssignal verwendet und in manchen nicht. In Abhängigkeit dieses Wertes wird entschieden, ob das Auto in den „Unstuckmode“ gehen soll. Siehe hierzu Abschnitt 3.3 auf Seite 43. In Kapitel 4 auf Seite 49 wird auf die Verwendung näher eingegangen.
„Distsum“ ist die Summe der Elemente einer Warteschlange[6], die zurückgelegte Distanzen in Fahrtrichtung der Strecke enthält. Bild 3.10 macht den Unterschied zur zurückgelegten Strecke des Autos deutlich. In jedem Zeitschritt wird

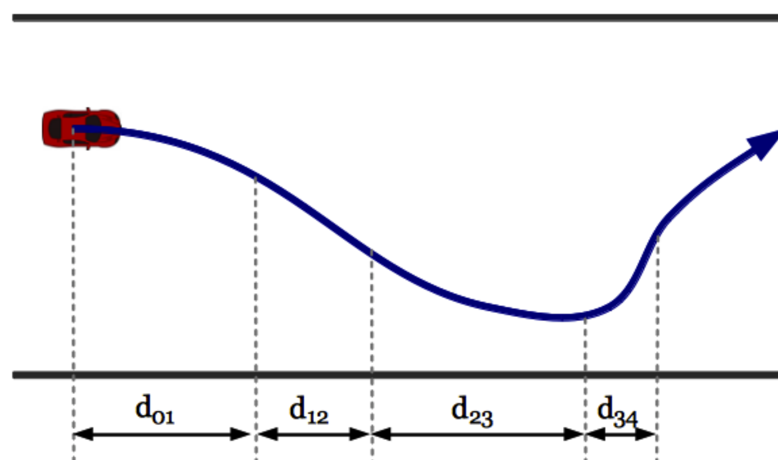


Bild 3.10: Unterschied zwischen zurückgelegter Strecke und Distanz in Fahrtrichtung der Strecke

gemessen, wie weit sich das Auto vom Ziel weg bewegt hat. Zwischen Zeitschritt

[5] siehe Abschnitt 2.1.3 auf Seite 9
[6] Engl. queue. Arbeitet nach dem First „In - First Out“-Prinzip. D.h. erfolgt ein Ausgabebefehle(dequeue), wird das Element zurückgegeben, das als erstes mit einem Eingabebefehl(enqueue) eingefügt wurde.

0 und 1 hat sich das Auto um d_{01} in Fahrtrichtung der Strecke bewegt. Dieser Wert wird in die Schlange eingefügt. Im gleichen Moment fliegt das älteste Element raus. Alle enthaltenen Werte werden aufsummiert. Im Grunde ist dies ein Mittelwert der in Fahrtrichtung der Strecke zurückgelegten Distanzen.

3.1.4 Anpassungen an Torcs

Markov Decision Process

Torcs ist eine realistische Rennsimulation. Daher verbraucht das Auto während es fährt Sprit und der Tank wird immer leerer. Ein leichteres Auto hat andere Kurven- und Beschleunigungseigenschaften. Das Auto kann also im selben Zustand s, die selbe Aktion a wählen und zu verschiedenen Zeitpunkten t in verschiedene Folgezustände s' kommen. Dann hängt der Folgezustand s' und somit auch die dort erhaltene Belohnung r nicht mehr vom aktuellen Zustand und der dort getroffenen Aktion ab. Dies widerspricht der Bedingung, dass das RL-Problem als MDP[7] formuliert sein muss, um korrekt gelöst werden zu können. Also muss der Verbrauch abgeschaltet werden.(siehe Appendix A.5auf Seite 74) Das selbe gilt für den Schaden, den das Auto nehmen kann, wenn es z.B. gegen die Bande fährt. Dann würde sich ebenfalls die Physik des Autos verändern.
Wählt man die Zustandsdarstellung so, wie oben beschrieben, kann das RL-Problem als fMPD angesehen werden. Zum einen weist diese die Markov-Eigenschaft auf und zum anderen ist der Zustandsraum $\mathcal{S}$ begrenzt. Dies ist gegeben, da das Welt-Koordinatensystem, in dem die Strecke aufgebaut ist, begrenzt ist. Zudem ist auch der Aktionsraum $\mathcal{A}$ begrenzt, worauf in Abschnitt 3.2 auf Seite 40 näher eingegangen wird.

Datenvorverarbeitung

Um das Zustandssignal, wie oben, formulieren zu können, müssen alle gegebenen Punkte erst umgerechnet werden. Sowohl die Streckenpunkte als auch die Position des Autos liegen in Weltkoordinaten vor. In seiner Position ist das

[7] siehe Abschnitt 2.1.3 auf Seite 9

Auto zusätzlich in gewisser Weise ausgerichtet. Diese Orientierung wird durch die Parameter

$$\begin{aligned} \text{roll} &\rightarrow \text{Drehung um x-Achse} \\ \text{pitch} &\rightarrow \text{Drehung um y-Achse} \\ \text{yaw} &\rightarrow \text{Drehung um z-Achse} \end{aligned}$$

beschrieben. Bild 3.11 zeigt die Winkel für eine beispielhafte Ausrichtung des Autos. Bild 3.12 zeigt die Anordnung des Autos im Raum(Punkt C) und einen

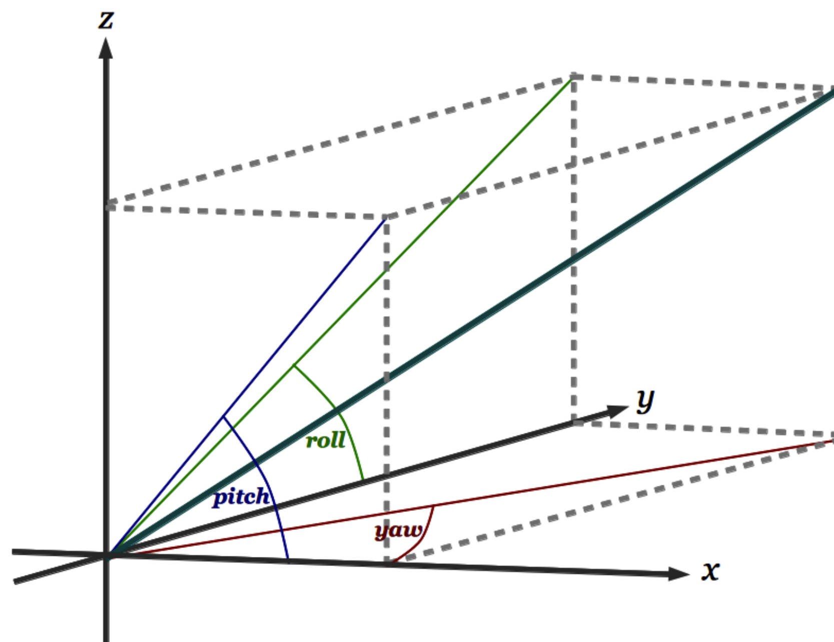


Bild 3.11: pitch, roll, yaw

beliebigen Punkt P, der z.B. ein Streckenpunkt sein kann. Die Koordinaten dieses Punktes müssen in das verdrehte Koordinatensystem des Autos umgerechnet werden. In diesem Beispiel ist der Winkel yaw rot eingezeichnet. Ansonsten ist das Auto-Koordinatensystem genauso, wie das Welt-Koordinatensystem, ausgerichtet.

Den Punkt P dreht man in das Autokoordinatensystem rein, indem man diesen um -yaw, -pitch und -roll um die jeweiligen Achsen rotiert. Hierfür kann man

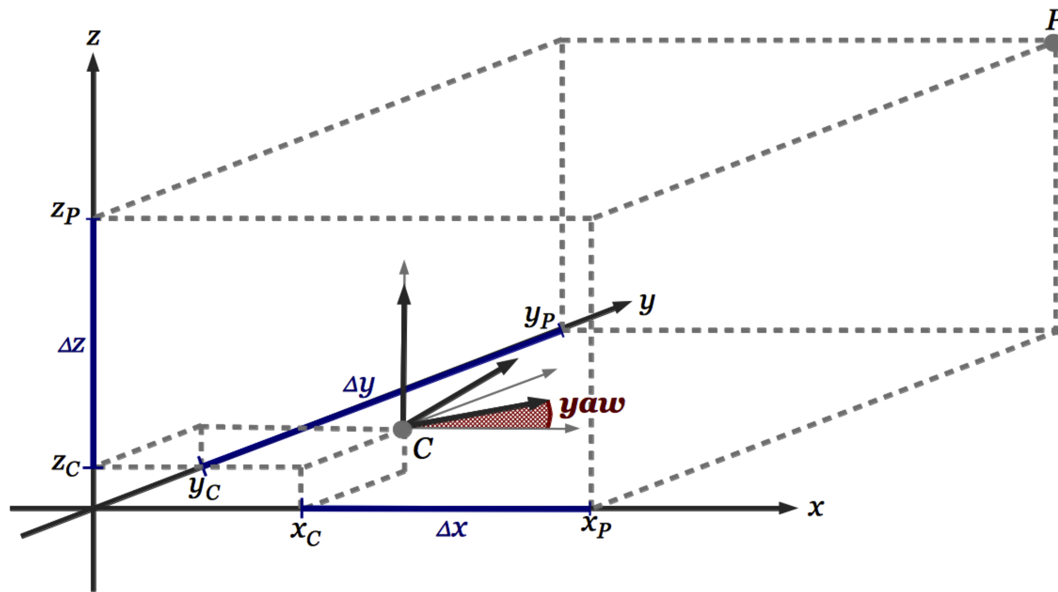


Bild 3.12: Anordnung Auto(C für car) und einem beliebigen Punkt P

die dreidimensionale affine Transformation verwenden:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \tag{3.1}$$

$$R_y(\alpha) = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix} \tag{3.2}$$

$$R_z(\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

3.2 Aktionsdarstellung

Die dem Agenten zur Verfügung stehenden Aktionen sind die Festsetzung des Gas- bzw. Bremspedals(acc) und die Einstellung der Lenkung(steer). Diese Aktionen stehen in jedem Zustand zur Verfügung, also gilt

$$\mathcal{A}(s) = \{\text{acc}, \text{steer}\}\ \forall\ s \in \mathcal{S}. \tag{3.4}$$

Der Wert für die Lenkung und die Stellung des Gas- bzw. Bremspedals können im Intervall $[-1, 1]$ gewählt werden. Dabei gilt für die Lenkung:

$$\text{steer} = -1 \rightarrow \text{Volleinschlage nach LINKS}$$
$$\text{steer} = +1 \rightarrow \text{Volleinschlage nach RECHTS}$$

Im Bereich von $[-1, 0]$ entspricht der acc-Wert der Einstellung des Bremspedals, wobei die Werte folgendes bedeuten:

$$\text{acc} = -1 \rightarrow \text{Bremspedal maximal gedrückt}$$
$$\text{acc} = 0 \rightarrow \text{Bremspedal in Ruhestellung}$$

Dementsprechend wird im Intervall $[0, 1]$ das Gaspedal bedient:

$$\text{acc} = +1 \rightarrow \text{Gaspedal maximal gedrückt}$$
$$\text{acc} = 0 \rightarrow \text{Gaspedal in Ruhestellung}$$

Die Gänge werden durch ein Automatiksystem von alleine eingestellt. Dies wird in Abschnitt 3.5 auf Seite 48 näher erläutert.

Diskretisierung

Die Werte für einzelne Aktionen können mit beliebig vielen Nachkommastellen eingestellt werden. Also handelt es sich hier um einen kontinuierlichen Aktionsraum $\mathcal{A}$. In jedem Zeitschritt t muss der Agent jedoch eine konkrete Aktion a, zusammen mit dem aktuellen Zustand s in die Aktion-Werte-Funktion $Q(s, a)$ einsetzen. Also muss der Aktionsraum hinreichend abgetastet werden, sodass

ein Q-Wert gefunden wird, der dem besten[8] Q-Wert zumindest nahe ist.
Eine Möglichkeit ist den Aktionsraum erst grob und dann in der nähe des besten gefunden Wertes nochmal feiner abzutasten. Bild 3.13 verdeutlicht dies. Die

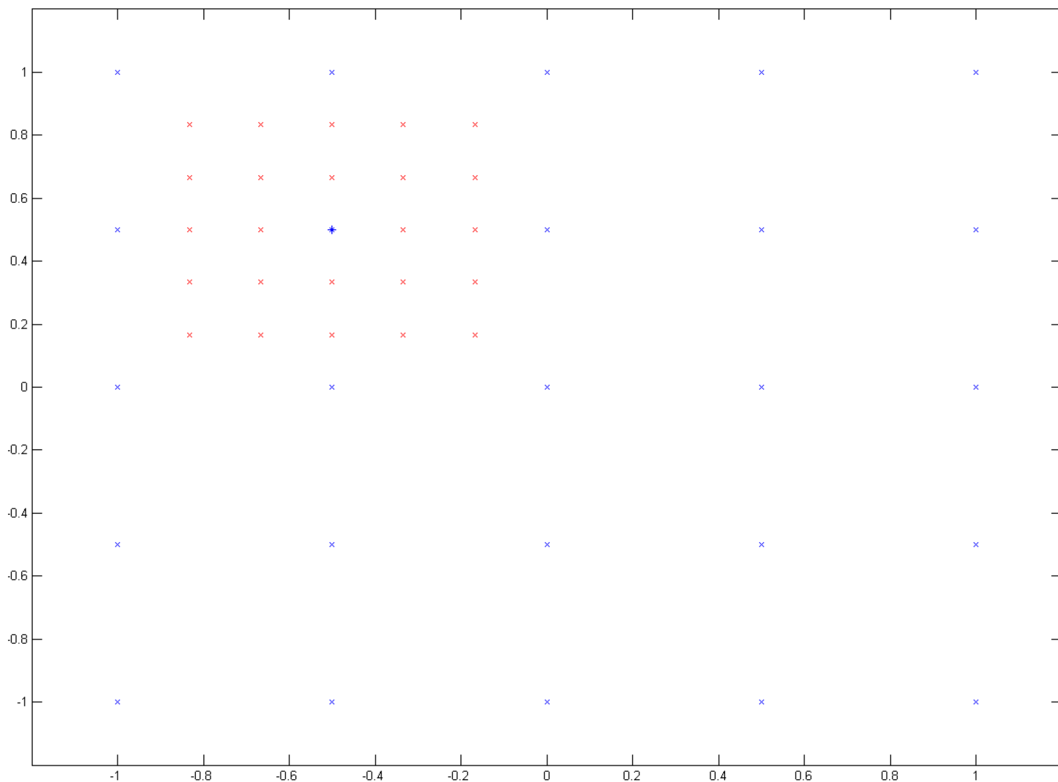


Bild 3.13: Abtastung des Zustandsraum(x→steer,y→acc): Erst grob(blau), dann fein(rot)

grobe Abtastung ist blau dargestellt, wohingegen die feinere rot ist. Der etwas dickere blaue Stern symbolisiert den besten gefundenen Q-Wert der groben Abtastung.
Diese Variante der Diskretisierung hat einen entscheidenden Nachteil. Wichtige Bereiche müssen feiner abgetastet werden als unwichtigere. Solch ein Bereich ist z.B. das Intervall $[-0.1, 0.1]$ der Lenkung. Das Auto sollte dort häufiger abgetastet werden, damit kleinere Lenkbewegungen möglich sind, denn der Agent führt nur eine Aktion aus der Menge der abgetasteten Aktionen aus. Dies ist zum einen für das „geradeaus"-Fahren wichtig, um Zick-Zack-Kurse zu vermeiden. Zum anderen hat eine kleine Veränderung der Lenkung bei hohen

[8] Ob der beste Wert möglichst hoch oder möglichst niedrig ist wird durch die Verwendung von Kosten $c(s, a, s')$ oder Belohnungen $r(s, a, s')$ entschieden. Der beste Wert ist für beide Implementierungen eindeutig.

Geschwindigkeiten eine höhere Wirkung. Die Abtastung dieses Bereichs kann jedoch nicht garantiert werden.
Man stelle sich nun vor, bei der groben Abtastung wird die beste Aktion nicht exakt abgetastet, sondern der Wert daneben. Dies passiert jedoch in einem wichtigen Bereich des Zustandsraums. Der Wert wird als zweitbester eingestuft und der beste abgetastete Wert liegt in einem unwichtigeren Bereich. Eine feinere Abtastung nutzt in diesem Fall nichts, da der eigentlich beste Wert nicht erreicht wird. So wird die schlechtere Aktion verwendet, was durch eine höhere Abtastung des, für das Problem, wichtigeren Bereiches verhindert werden könnte.
In [Engesser, 2011] wird eine Diskretisierung vorgestellt, die dieses Problem löst. Hier wird die Aktion steer in gröberen Schritten abgetastet, je näher der Wert an die Ränder des Intervalls tritt. Es wird die Funktion

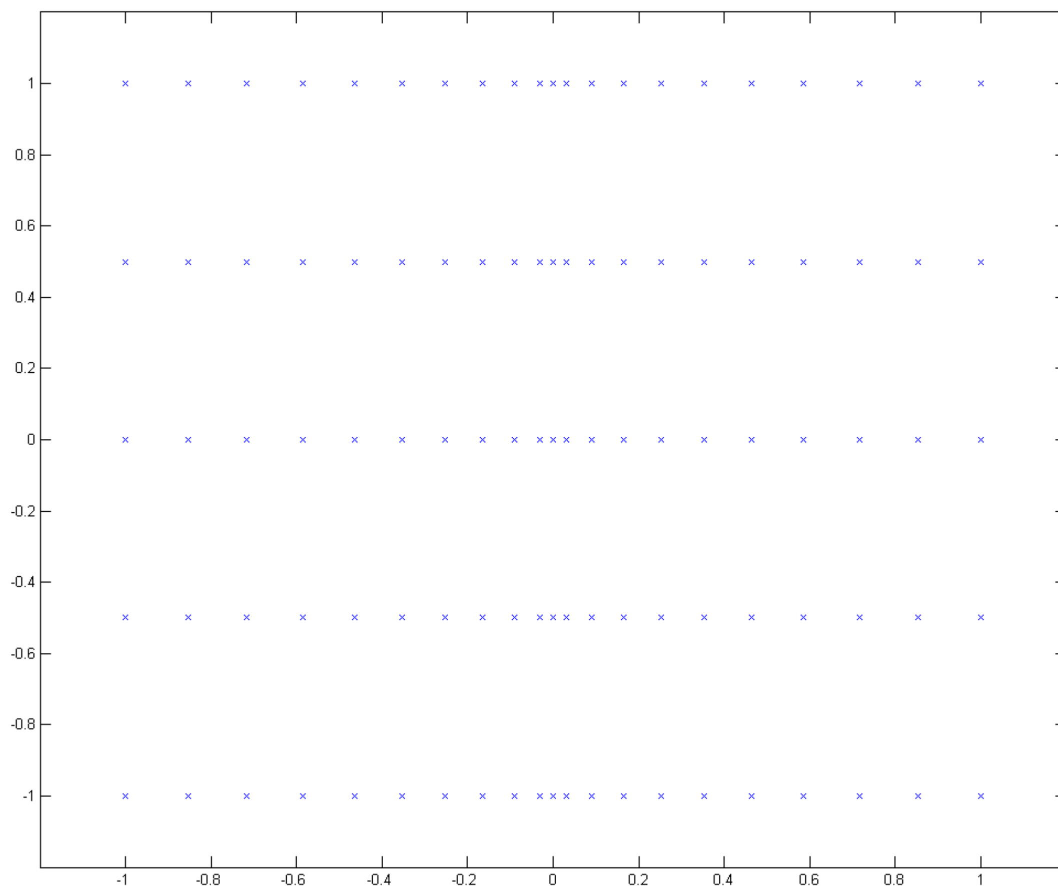


Bild 3.14: Abtastung des Zustandsraum(x→steer,y→acc): acc grob, steer feiner je näher an 0

$$f(x) = \frac{x}{10}\sqrt{\frac{|x|}{10}} \tag{3.5}$$

verwendet. Setzt man für x Werte im Bereich $[-10, 10]$ ein, bleibt $f(x)$ im vorgegebenen Intervall von $[-1, 1]$. Bild 3.14 zeigt die Anwendung von Gleichung 3.5 auf das Aktionselement steer. Mit der Abtastung wird die Schätzungsstrategie in Form der Funktion

$$\max_b Q(s, b) \tag{3.6}$$

umgesetzt.
Generell sollte man die Anzahl der Abtastwerte möglichst gering halten. Zum Einen muss der Agent die Suche nach dem besten Wert in jedem Zeitschritt durchführen, um eine Aktion zu wählen. Zum Anderen wird für jede Transition ein Q-Wert aktualisiert. Dafür muss der Aktionsraum ebenfalls abgetastet werden.

3.3 „Unstuck“-Modus

Der Agent kann sich auf der Strecke z.B. an der Seite festfahren, wenn das Auto sich nicht fortbewegt und der Zustand sich somit nicht ändert. Dies hat zur Folge, dass die Abtastung des Aktionsraums immer die selbe Aktion hervorbringt, die jedoch das Auto nicht bewegt. Es wird zwar mit der Wahrscheinlichkeit ϵ eine zufällige Aktion gewählt, trotzdem kann es lange dauern bis sich das Auto von selbst befreit. Torcs bricht nach einer gewissen Zeit ab, wenn sich das Auto nicht bewegt. Daher wurde der „Unstuck“-Modus eingeführt, der das Auto mit höchster Beschleunigung ein Stück zurückfahren lässt und dann dem Beispiel-Agenten aus [Wymann] für einen gewissen Zeitraum folgt. In diesem Zeitraum wird das Auto so ausgerichtet, dass es auf der Strecke in Richtung Ziel orientiert ist.
In Bild 3.15 wird dies verdeutlicht. Die blauen Punkte stellen die Streckenbegrenzungen dar, in rot fährt der Agent im „normalen“ Modus, also kontrolliert selbst Gas- bzw. Bremspedal und das Steuer, und in grün ist der Agent im „Ünstuck’-Modus dargestellt. Er kommt von der Strecke ab, fährt gegen die

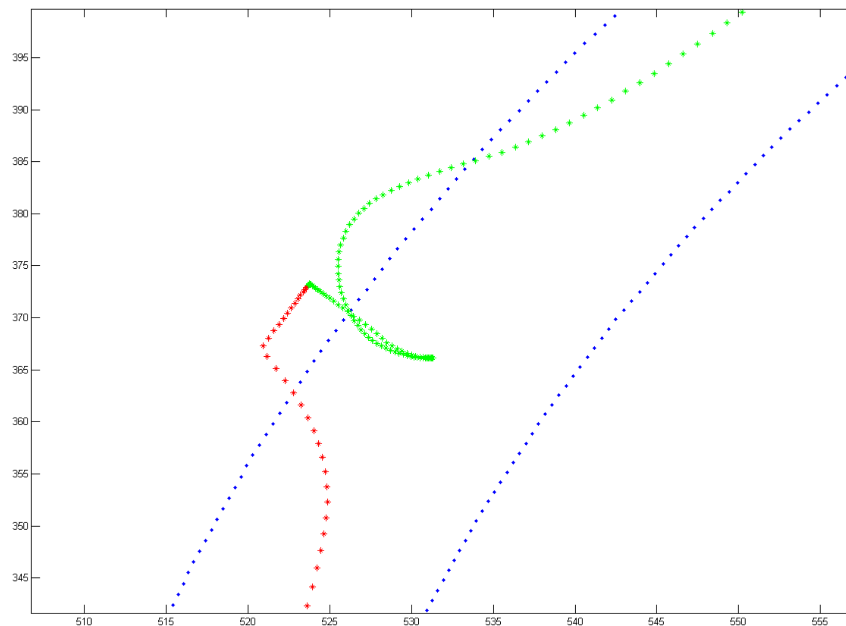


Bild 3.15: „Unstuck"-Vorgang

Bande und unterschreitet einen bestimmten „Distsum"-Wert. Diese Grenze wird in Abschnitt 4 auf 49 je nach Kosten- bzw. Belohnungsfunktion bestimmt. Daraufhin setzt der oben beschriebene Vorgang ein, sodass der Agent wieder auf die Straße fährt.
Der „Unstuck"-Modus ist noch aus einem zweiten Grund wichtig. Nehmen wir an, das Auto hätte keine zeitliche Beschränkung, wenn es sich festgefahren hat. Dann würden in jedem Zeitschritt Erfahrungen darüber gesammelt werden, wie sich das Auto am besten befreit, anstatt zu lernen schnell über die Strecke zu fahren. Um das Ziel „schnelles Fahren" zu lernen, muss der Agent über den Bereich, in dem er sich festgefahren hat, nur wissen, dass dieser nicht erlaubt ist.

3.4 Neuronales Netzwerk

In diesem Abschnitt sollen die Methoden erläutert werden, die die Lerngeschwindigkeit des Netzwerks beschleunigen. Damit ist nicht die Geschwindigkeit des Lernvorgangs des RL-Problems gemeint, sondern die Schnelligkeit, mit der das neuronale Netz es schafft, sich Trainingsdaten mit einem geringen Fehler anzueignen. Zunächst wird über den Aufbau des neuronalen Netzes

diskutiert und danach die zwei Methoden erläutert, die die Lerngeschwindigkeit am meisten zum positiven beeinflusst haben.

3.4.1 Topologie

Für diese Aufgabe wurde ein Netz mit 2 versteckten Schichten, einer Input- und einer Output Schicht verwendet. Bild 3.16 zeigt den Aufbau des Netzes.

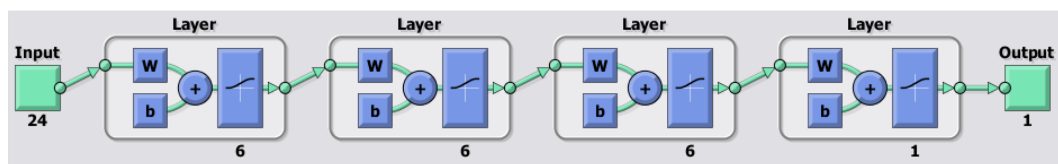


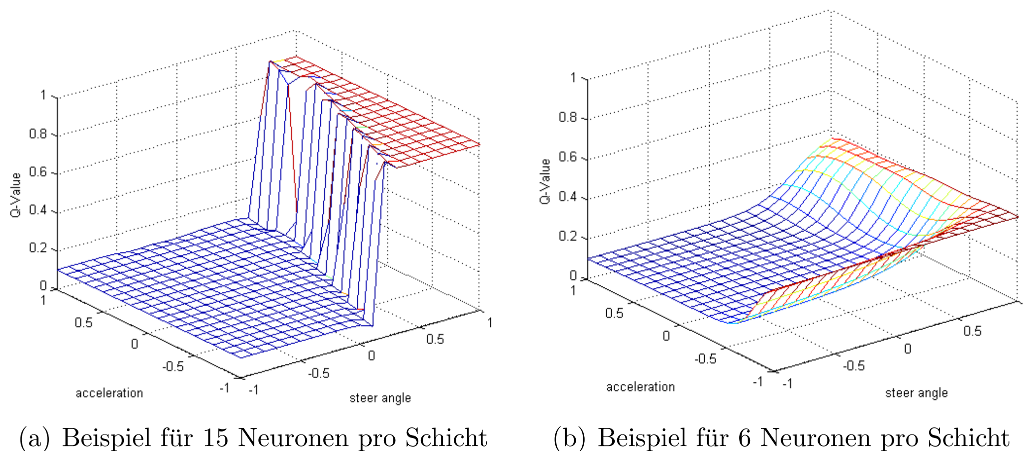
Bild 3.16: Topologie des neuronalen Netzwerks

Es besitzt 24 Eingänge für den Zustand und die Aktion:

- Jeweils drei Eingänge für die 6 Streckenpunkte(4 für den aktuellen Abschnitt, 2 für nächsten($k = 1$))$\rightarrow$ 18.
- Einen Eingang besetzt der „Distsum“-Wert aus Abschnitt 3.1.3 auf Seite 35
- Drei Eingänge sind für die Geschwindigkeitskomponenten reserviert.
- Die Aktion beansprucht 2 Eingänge.

Als Aktivierungsfunktion wurde für jedes Neuron die Log-Sigmoid-Funktion verwendet, die Eingangswerte $\in [-\infty, \infty]$ auf das Intervall $[0, 1]$ zusammenzieht und differenzierbar ist(wird für Trainingsverfahren benötigt).
Als Aktivierungsfunktion der Output-Schicht wurde ebenfalls die Log-Sigmoid-Funktion gewählt, obwohl die Q-Werte auch außerhalb des Ausgangsintervalls der Log-Sigmoid-Funktion liegen können. In Abschnitt 3.4.3 auf Seite 47 wird dieses Problem gelöst.
Die Anpassung der Anzahl der Neuronen und versteckten Schichten ist schwer vorzunehmen, da diese stark von den Trainingsdaten abhängen. Die Eingangsschicht hat dabei immer die selbe Anzahl an Neuronen, wie die versteckten Schichten. Ein Test der Lerngeschwindigkeit hat gezeigt, dass weniger Neuronen in zwei versteckten Schichten ,anstatt einer , verwendet werden sollen.

Bild 3.17 zeigt die Abtastung zweier Aktionsräume, die mit verschiedenen neuronalen Netzwerken approximiert werden. In Bild 3.17(a) wird ein Netzwerk mit 15 Neuronen in den 2 versteckten Schichten, sowie in der Eingangsschicht verwendet. Es bilden sich scharfe Kanten. Die Anzahl der Neuronen pro Schicht



(a) Beispiel für 15 Neuronen pro Schicht (b) Beispiel für 6 Neuronen pro Schicht

Bild 3.17: Abgetastete Aktionsraum für einen Zustand

wurde kontinuierlich gesenkt bis die Bilder der Abtastung für verschiedene Zustände weiche Übergänge aufzeigten. In Bild 3.17(b) sind es jeweils 6 Neuronen, die sich in der Eingangsschicht und den verdeckten Schichten befinden. Allgemein kann man sagen, dass ein neuronales Netz gut generalisiert, wenn geringe Änderungen am Eingang geringe Änderungen am Ausgang erzeugen.

3.4.2 Skalierung der Eingangswerte

Die einzelnen Werte der Zustandskomponenten s_{it} $\forall\ i = 1, 2 \ldots n$ treten in verschiedenen Intervallen auf. Die z-Komponente eines Streckenpunktes hat andere Minimal- und Maximalwerte als die x-Komponente. Wenn einen Eingang zwischen sehr hohen und sehr niedrig Werten schwankt kann eine Multiplikation mit dem jeweiligen Eingangsgewicht nicht verhindern, dass die Log-Sigmoid-Funktion gesättigt wird. Dies verlangsamt die Lerngeschwindigkeit[Matlabhilfe, 2011].

Vor dem Batch-Training des Netzes werden die gesammelten Eingangsdaten mithilfe der Matlab-Funktion „mapminmax“ in jeder Komponente skaliert. D.h.

es wird der Maximal- und der Minimalwert jeder Komponente des Zustandsignals bestimmt und auf das Intervall $[-1, 1]$ normiert. Wird das neuronale Netz abgefragt, müssen die Eingangswerte mit der selben Normierung umgerechnet werden.

3.4.3 Skalierung der Ausgangswerte

Das neuronale Netz soll Q-Werte als Output ausgeben. Diese über- bzw. unterschreiten den Wertebereich der Ausgangswerte der Log-Sigmoid-Funktion. Daher kann man entweder eine lineare Ausgangsfunktion wählen oder die Ausgangswerte skalieren. Ein Versuch hat gezeigt, dass die Lerngeschwindigkeit wesentlich höher und der Fehler geringer ist, wenn zweite Variante verwendet wird.
In [Gabel u. a., 2011] wird eine lineare Skalierung vorgeschlagen, die das Intervall der Q-Werte $[out_{\min}, out_{\max}]$ auf den Wertebereich der Log-Sigmoid-Funktion $[net_{\min}, net_{\max}]$ skaliert. In Bild 3.18 wird dies verdeutlicht. Dabei sind entspre-

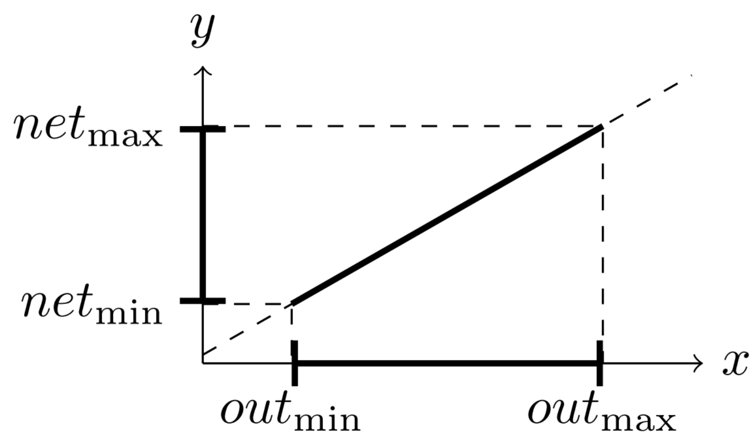


Bild 3.18: Skalierung der Q-Werte

chend die Q-Werte auf der x-Achse und die skalierten Q-Werte auf der y-Achse aufgetragen und werden mit der Funktion

$$f_{scale} : x \mapsto y = Ax + B \tag{3.7}$$

und der Steigung

$$A = \frac{net_{\max} - net_{\min}}{out_{\max} - out_{\min}} \tag{3.8}$$

sowie dem y-Achsenabschnitt

$$B = -\frac{out_{\min}(net_{\max} - net_{\min})}{out_{\max} - out_{\min}} + net_{\min} \tag{3.9}$$

berechnet. Greift man auf die Q-Werte wieder zurück, müssen diese mit den selben Parametern A und B zurückgerechnet werden:

$$x = \frac{y - B}{A} \tag{3.10}$$

Da der Improved NFQ-Algorithmus implementiert wurde, wurde auch auf dessen Skalierungsmethode zurückgegriffen anstatt die Matlab-Funktion „mapminmax" zu verwenden.

3.5 Gangschaltung

Der Agent soll möglichst wenige Aktionskomponenten a_{it} $\forall$ $i = 1, 2 \ldots m$ einstellen müssen, damit der Lernvorgang nicht mit vermeidbaren Aufgaben aufgehalten wird. Daher wurde eine Automatikschaltung implementiert, die abhängig von der Drehzahl den Gang(engl. gear) umschaltet. Der Wertebereich der Gänge wird um den Rückwärtsgang(-1) und den neutralen Gang(0) dezimiert, sodass im Bereich $[1, 6]$ geschaltet wird. Das Auto soll sich ständig im höheren Drehzahlbereich bewegen. Daher wurde der Parameter für die maximale Drehzahl *redLine* ausgelesen und folgendermaßen geschaltet:

- gear = gear + 1, wenn Drehzahl >= 0.95·redLine
- gear = gear − 1, wenn Drehzahl <= 0.6·redLine

4

Kostenfunktion

Neben der Zustandskomposition ist die Kostenfunktion[1] $c(s, a, s')$ das Hauptelemente eines RL-Problems. Sie definiert die Lernziele des Agenten. Der Agent soll

- die komplette Runde so schnell, wie möglich fahren.
- niemals von der Strecke abkommen.

Die benötigten Informationen muss das Zustandssignal liefern, was in Abschnitt 3.1 auf Seite 24 erläutert wurde.
In diesem Abschnitt werden vier Ansätze für die Kostenfunktion gegeben und diskutiert.

4.1 Grundidee/Leitfaden

Nun sollen Möglichkeiten gezeigt werden, wie Ziele mit einer Kostenfunktion formuliert werden können. Dafür muss die Menge aller möglichen Zustände $\mathcal{S}$ in drei Gruppen unterteilt werden.

$$\begin{aligned} \mathcal{S}^+ &\rightarrow \text{ erwünschte Zustände} \\ \mathcal{S}^- &\rightarrow \text{ unerwünschte Zustände} \\ \mathcal{S}^0 &\rightarrow \text{ Zustände, die besucht werden dürfen, um zu } S^+ \text{ zu gelangen.} \end{aligned}$$

[1] Die Theorie wurde mit Belohnungen hergeleitet. Da jedoch Kosten modelliert werden und in diesem Kapitel konkrete Zahlenwerte vorkommen wird von Kosten gesprochen.

Es gilt

$$\mathcal{S} = \mathcal{S}^{+} + \mathcal{S}^{-} + \mathcal{S}^{0}. \tag{4.1}$$

Zustände der Menge S^0 werden hier auch Wegzustände genannt. Erwünschte Zustände S^+ sind die Zustände, die der Agent erreicht, nachdem er die Start/Ziellinie überquert hat. Da auf dem Weg zum Ziel die Strecke nicht verlassen werden soll, sind die Menge der Zustände, die sich außerhalb der Strecke befinden, der Menge S^- zugeordnet. Die Positionen auf der Straße, die nicht zu dem aktuellen Zeitpunkt bereits S^+ zugeordnet sind, entsprechen der Menge S^0. Logischerweise erhalten Zustände der Menge

- S^+ die niedrigsten Kosten $c = c^+$ und
- S^- die höchsten Kosten $c = c^-$.

Bei episodischen Aufgaben[2] ist bei Erreichen eines Zustandes der Menge $\mathcal{S}^+$ die Episode beendet und es beginnt eine neue. Dies ist meistens durch die Aufgabe natürlich gegeben. Man könnte annehmen, dass die Aufgabe des „schnellen Fahrens“ pro Runde gesehen wird. Diese Schlussfolgerung wird jedoch in den weiteren Abschnitten in Frage gestellt.
Bei kontinuierlichen Aufgaben sollte der Agent die Position in dem gewünschten Zustandsraum halten.

Ziel: benötigte Zeit minimieren

Nun soll die Frage beantwortet werden, wie es möglich ist, dem Agenten zu sagen, dass er den schnellsten Weg zu den Zielzuständen wählen soll. Dies wird realisiert, indem der Agent geringe Kosten erhält, wenn er sich in einen Wegzustand befindet. Die Kosten sind geringfügig höher als die für einen erwünschten Zustand.

$$\mathcal{S}^0 \to c = c^0, c^+ < c^0 << c^-$$

[2] siehe 2.1.1

Erhält der Agent in jedem Zeitschritt geringe Kosten wird er die Häufigkeit des Auftretens der Kosten minimieren, damit der Verlust[3]

$$C_t = \sum_{k=0}^{T} \gamma^k c_{t+k+1} \tag{4.2}$$

gering gehalten wird. Im Falle einer episodischen Aufgabe($\gamma = 1$) würde die Anzahl der Zeitschritte T minimiert werden. Bei einer kontinuierlichen Aufgabe($T = \infty$) würde der Agent versuchen so lange, wie möglich, im gewünschten Zustandsraum zu bleiben. Es sei dazu gesagt, dass die Kosten nicht auf den Ausgangswert der Funktionsapproximationsmethode angepasst werden müssen, da die berechneten Q-Werte vor dem Training skaliert werden(siehe Abschnitt 3.4.3 auf Seite 47).
Aufbauend auf diesen erarbeiteten Kenntnissen werden 4 Vorschläge für die Kostenfunktion $c(s, a, s')$ gegeben. Der erste Ansatz beschreibt die grundlegende Vorgehensweise. Die drei weiteren Ansätze bauen darauf auf und verbessern diesen.

4.2 Variante 1

Die Variante der Kostenfunktion $c(s, a, s')$, die nachfolgend beschrieben wird, ist als erste Idee entstanden und wurde nicht implementiert. Hier tauchen jedoch viele Probleme auf, die an dieser Stelle erwähnt werden sollen.

Aufspannen des Zustand-Aktionsraums

In Abschnitt 3.1.1 auf Seite 25 wurde darauf hingewiesen, dass eine weiträumige Abtastung des Zustand-Aktionsraums gegeben sein muss, wenn eine Funktionsapproximationsmethode angewendet wird. Daher werden, bevor der Agent das Auto steuert, drei Runden mit dem Torcs-Agenten aus [Wymann] gefahren. Dabei wird in jeder Runde die Stellung des Gaspedals erhöht. In der dritten Runde wird mit Vollgas gefahren. Währenddessen werden Transitionen gesammelt. Die Aktualisierungen der Q-Funktion, die anhand der Transitionen

[3] Hier ist der Gewinn aus Gleichung 2.4 auf Seite 8 gemeint. Da Kosten modelliert wurden, wird hier von Verlust anstatt Gewinn gesprochen.

vorgenommen werden, spannen den Zustand-Aktionsraum auf. Der Agent hat außerdem durch die vortrainierte Q-Funktion ein paar unerwünschte Zustände kennen gelernt. Ein weiterer Vorteil ist, dass genug Trainingsdaten für das neuronale Netz gesammelt wurden. Wird ein neuronales Netz trainiert, das mehr Gewichte und Bias-Werte aufweist als Trainingsdaten vorhanden sind, können nicht alle Unbekannte angepasst werden. Ein künstliches neuronales Netz mit der in Abschnitt 3.4 auf Seite 44 erwähnten Topologie(6-6-6-1) hat 217 Gewichte und 19 Bias-Werte. Während der 3 Runden werden circa 2700 Transitionen gesammelt. Somit wird eine Unterbestimmung vermieden.

Festsetzung der Kosten und Bereiche

Zunächst wird eine Episode als eine Runde definiert. Also befindet sich der Bereich, in dem gewünschte Zustände $\mathcal{S}^+$ auftreten können, kurz hinter der Start/Ziel-Linie. Daher erhält der Agent bei einer neuen Runde die Kosten $c^+ = 0$ und in jedem sonstigen Zeitschritt $c^0 = 1$. Befindet sich das Auto auch nur einmal außerhalb der Strecke, wird der Agent mit so hohen Kosten bestraft, dass jede weitere Kombination von Zuständen bis zum Ziel diesen Austritt im Verlust C_t nicht mehr attraktiv machen kann. Es wurde ermittelt, dass der Agent circa 2000 Transitionen pro Runde sammelt. Daher wurde

$$c^- = 2000 \tag{4.3}$$

festgelegt.

Unstuck

Bevor das Auto autonom fahren kann muss bedacht werden, dass es sich festfahren kann und Torcs das Rennen abbricht. Daher wurde der „Unstuck"-Modus aus Abschnitt 3.3 auf Seite 43 verwendet. Ein weiterer Vorteil ist, dass weniger Trainingsdaten gesammelt werden, die im festgefahrenen Situationen entstehen. Der Agent soll sich in dem Zustandsbereich befinden, den er verbessern soll.
Zu beachten ist, dass nicht in jedem Zeitschritt Transitionen gesammelt werden dürfen. In der ersten Phase des „Unstuck"-Modus wird das Auto rückwärts gefahren. Der Agent ist jedoch nicht in der Lage den Rückwärtsgang einzulegen, sodass in dieser Phase Aktionen ausgewählt werden, die der Agent nicht im Stande ist auszuführen.

Nun muss definiert werden, wann der „Unstuck-“-Modus in Kraft tritt. Als gut erwiesen haben sich folgende Fälle, in denen der „Distsum“-Wert aus Abschnitt 3.1.3 auf Seite 35 auftaucht:

1. Auto auf der Fahrbahn und Distsum< 0.5
2. Auto neben der Fahrbahn und Distsum< 3

Da der „Distsum“-Wert eine Summe der Distanzen in Fahrtrichtung der Strecke der letzten 10 Zeitschritt ist, darf das Auto kurzzeitig auf der Strecke sogar stehen bleiben. Bleibt es jedoch dauerhaft stehen und bewegt sich nicht vorwärts, weil in diesem Zustand immer die Aktion $acc = -1$ gewählt wird, soll der Unstuck-Modus den Wagen wieder in eine neue Situation bringen. Also wird die Grenze 0.5 gewählt. Man hätte sie auch höher wählen können. Es soll jedoch so wenig, wie möglich eingegriffen und beschränkt werden.
Der Agent soll den Unstuck-Modus genauso vermeiden, wie von der Strecke abzukommen. Also erhält er für ein Zustand-Aktionspaar, das ihn in den Unstuck-Modus führt, ebenfalls $c = c^{-}$.
Der „Distsum“-Wert muss im Zustandssignal auftauchen, da sonst die Markov-Eigenschaft nicht gilt.

Beispiel Man stelle sich vor, der Wagen würde auf der Strecke langsam fahren und schließlich anhalten. In einem zweiten Fall fährt der Wagen etwas schneller, hält aber an der selben Position an. In beiden Fällen ist der „Distsum“-Wert nicht im Zustandssignal enthalten. Im jeweils nächsten Zeitschritt kommt der Agent im ersten Fall in den „Unstuck“-Modus und im zweiten Fall nicht, da im ersten Fall „Distsum“ die Schwelle von 0.5 unterschritten hat und im zweiten Fall nicht. Dies würde unterschiedliche Kosten zur Folge haben, obwohl der Agent beide Male den selben Zustand sieht. Der Grund liegt darin, dass die angewendete Aktion nur aufgrund des aktuellen Zustands gewählt wurde, obwohl diese auch von der Vergangenheit(vorher langsamer/schneller gefahren) abhängt.

Nun soll gezeigt werden, wo sich die Zielzustände auf der Strecke befinden. In Bild 4.1 ist die Teststrecke dargestellt. Der grüne Bereich in der Nähe von

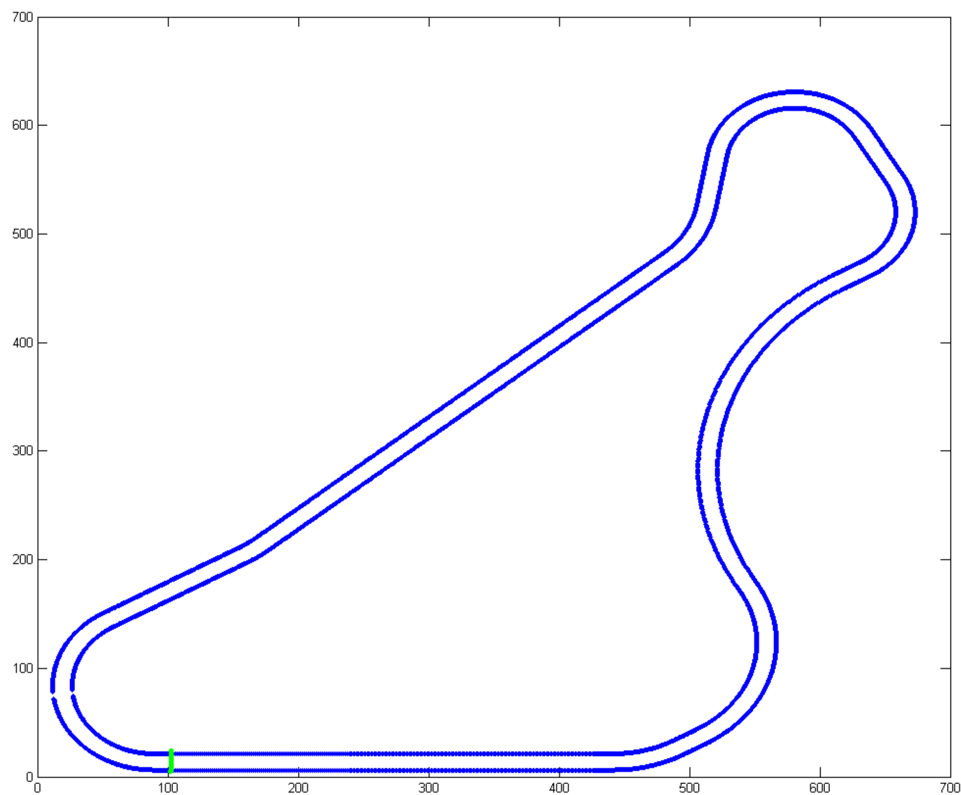


Bild 4.1: Zielbereich für Variante 1

$(100, 10)$ bildet die Start/Ziel-Linie.

Probleme dieses Ansatzes

Man kann hier schon erkennen, dass nur in einem kleinen Bereich, im Verhältnis zu der ganzen Strecke, Zielzustände auftreten können. Darin liegt das Hauptproblem dieses Ansatzes. Die Menge der Zielzustände S^+ sind dem Agent zu Beginn unbekannt. Zwar fährt das Auto in den ersten drei Runden dreimal durch das Ziel, dies steht jedoch nicht im Verhältnis der möglichen Zustände. In [Riedmiller, 2005] wird die „hint-to-goal“-Heuristik vorgeschlagen. Diese sieht vor, im Vorfeld künstlich Trainingsdaten zu entwickeln, die den $\mathcal{S}^+$ Bereich des Zustandsraums „herunterdrücken“. Befindet sich der Agent in der Nähe solch eines Zustandes wird er die Aktion wählen, die ihn näher zu dem S^+-Bereich führt. Jedoch wird die Überschreitung einer Linie mit den geringsten Kosten c^+ belohnt und nicht ein Bereich auf der Strecke. Ein Zustand, der in einem Zeitschritt ein Zielzustand ist, ist kein Zielzustand mehr, wenn der Agent rückwärts fährt und an die selbe Stelle auf der Strecke zurückkehrt. So kann man die Zustände der Menge S^+ nicht genau benennen. Man könnte diese

Trainingsdaten so generieren, dass trotzdem alle Zustände in der Nähe des Ziels im Zustands-Aktionsraum nahe Null sind, doch dann riskiert man auch, dass der Agent nach dem Ziel anhält und zurück fährt.
Dieser Ansatz beinhaltet ein weiteres Problem. Um dies erläutern zu können, muss die Zustandsdarstellung untersucht werden. Erreicht das Auto das Ende des aktuellen Abschnittes, wechseln die Streckenpunkte und der vorliegende Abschnitt wird der aktuelle, was in Bild 3.5 auf Seite 30 dargestellt wird. Hat das Auto mit dem Abschnittsende auch das Rundenende erreicht, erhält der Agent die Kosten c^+ eines Zielzustandes. Ist dies nicht der Fall erhält er „nur“ c^0 für einen weiteren Wegzustand. Damit wurde der Forderung nach einem Zustandsignal, das die Markov-Eigenschaft besitzt, nicht nachgekommen. Es müsste eine weitere Komponente zu der Zustandsdarstellung hinzugefügt werden, die dem Agenten mitteilt, dass er sich nun in einem Abschnitt befindet, dessen Endlinie die Ziellinie ist. Dies würde jedoch den Verlust eines großen Teils des generischen Charakters dieser Formulierung bedeuten(siehe Seite 31).

4.3 Variante 2

Die zweite Variante der Kostenfunktion $c(s, a, s')$ soll die Probleme der ersten Variante lösen. Damit der Agent häufiger einen Zielzustand antrifft, erhält er bei der Überschreitung der Endlinie des aktuellen Abschnitts c^+. In Bild 4.2 kann man erkennen, dass die grünen Bereiche, die grob die möglichen Bereiche der Zielzustände markieren, viel häufiger vorkommen.
Eine solche Formulierung hat zur Folge, dass nun nicht nach einer kompletten Runde die Episode zu Ende ist, sondern nach jedem Abschnitt. Daher benötigt der Agent keine Informationen, ob der aktuelle Abschnitt der letzte der Runde ist.

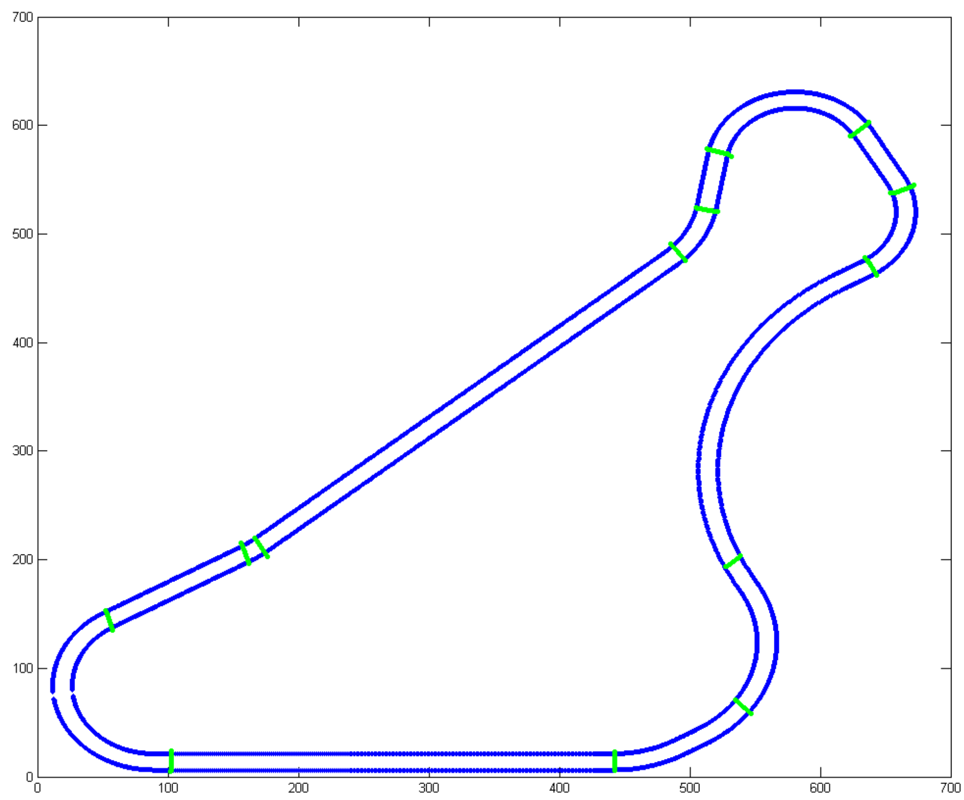


Bild 4.2: Zielbereich für Variante 2

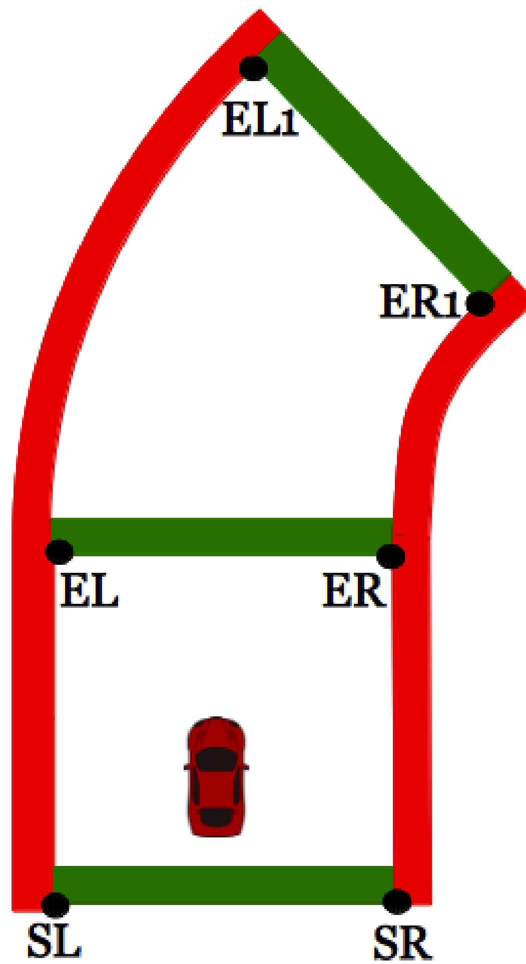


Bild 4.3 zeigt zwei aufeinander folgende Abschnitte. Der rote Bereich deutet die Zustände der Menge S^- und der grüne die Menge der S^+ an. Problematisch an diesem Ansatz, ist die Formulierung der Aufgabe als episodische Aufgabe. Der Agent erhält nach Überschreitung der Linie $\overline{EL, ER}$ keine Kosten. Also wird ihm so mitgeteilt, dass er sein Ziel erreicht hat und eine neue Episode beginnt. Das Problem ist, dass der Agent auch dann in einen Zielzustand kommt, wenn sich die zukünftigen Abschnitte beliebig ändern. So wird er keinen Zusammenhang zwischen der Belohnung und den Streckenpunkten der nächsten Abschnitte knüpfen können.
Des Weiteren bleibt die Problematik bestehen, dass Zielzustände nach Überschreitung einer Linie generiert werden anstatt an festen Positionen auf der Strecke. Eine Lösung würde evtl. eine Generierung von Trainingsdaten im Vorfeld ermöglichen(hint-to-goal Heuristik).

4.4 Variante 3

Die Quintessenz der letzten beiden Ansätze für die Kostenfunktion ist zum einen, dass die Positionierung der Ziellinie Probleme bereitet. Damit ist nicht die Start/Ziel-Linie gemeint, sondern die Linie, die das Auto überschreiten muss, um in einen Zielzustand zu kommen. Zum anderen stört die Notwendigkeit für den Agenten eine Linie überschreiten zu müssen, um in einen Zielzustand zu kommen. Wünschenswert wäre es den Zielzuständen feste Positionen zuweisen zu können.
In [Engesser, 2011] und [Abdullahi und Lucas, 2011] wird vorgeschlagen überhaupt keine Zielzustände zu formulieren. Stattdessen erhält der Agent in jedem Zeitschritt Kosten proportional zum Kehrwert der zurückgelegten Distanz in Fahrtrichtung der Strecke

$$c^0 = \frac{u}{\mathrm{dist}(s, s')}. \tag{4.4}$$

Dabei ist u eine Proportionalitätskonstante, die es erlaubt das Verhältnis zu anderen Kosten anzupassen. Die Funktion dist() misst die Distanz zwischen zwei Zuständen in Fahrtrichtung der Strecke, wie es in Bild 3.10 auf Seite 36

gezeigt wird. Die Kosten für unerwünschte Zustände c^- bleiben identisch.

Vorteile

Im Grunde wurde durch den Verzicht auf Zustände der Menge $\mathcal{S}^+$ aus der episodischen eine kontinuierliche Aufgabe. Dadurch fallen alle Probleme weg, die durch die Formulierung von Zielzuständen entstehen. Trotz allem erhält der Agent eine Resonanz über die Güte seiner Aktionswahl in einem bestimmten Zustand hinsichtlich der vorgegebenen Ziele. Zu beachten ist nun, dass $T = \infty$ und dadurch die Diskontierung $\gamma < 1$ sein muss.

Versuch

Mit diesen Annahmen wurde ein Testlauf gestartet. Als Explorationsstrategie wurde eine kontinuierliche Dezimierung von ϵ gewählt. Von $\epsilon = 0.9$ wurde in jeder Runde 0.02 abgezogen bis $\epsilon = 0.04$. Weiterhin wurde $\gamma = 0.9$ festgesetzt. Zwei Kennzahlen sollen den Lernfortschritt messen.

$$\text{dN/tN} = \frac{\text{dist(s,s') in „normal“}}{\text{Zeitschritte in „normal“}} \tag{4.5}$$

$$\text{tNontrack} = \frac{\text{Zeitschritte in „normal“ auf Fahrbahn}}{\text{Zeitschritte in „normal“}} \tag{4.6}$$

Sie werden nach jeder Runde zurückgesetzt. dN/tN ist ein Indikator für das Ziel „schnell Fahren“,tNontrack für das Ziel „auf der Fahrbahn bleiben“.
Bild 4.4 zeigt die Resultate. In den ersten Runden ist das ϵ sehr hoch, sodass diese Werte nicht sehr aussagekräftig sind. In Runde 30 wurden etwa 30% der Aktionen vom Agenten bestimmt.
Im oberen Graphen in Bild 4.4 ist dn/tN für jede gefahrene Runde dargestellt worden. Um den Wert deuten zu können, wird ein Relationswert zu dn/tN benötigt. Fährt das Auto mit 150 $\frac{\text{km}}{\text{h}}$ genau der Mittellinie entlang, so legt es eine Distanz von 2.5 in Richtung der Fahrbahn zurück. dn/tN schwankt in Bild 4.4 um den Wert 1, was einer Geschwindigkeit von durchschnittlich 60 $\frac{\text{km}}{\text{h}}$ in Fahrbahnrichtung bedeutet. Dies ist nicht viel,wenn man bedenkt, dass das Auto mit einer hohen Geschwindigkeit aus dem „Unstuck“-Modus heraus kommt(etwa 10mal pro Runde). Also kann man schlussfolgern, dass dn/tN sich mit zunehmender Rundenzahl nicht ändert. Dem Kurvenverlauf ist kein eindeutiger Trend zu entnehmen.

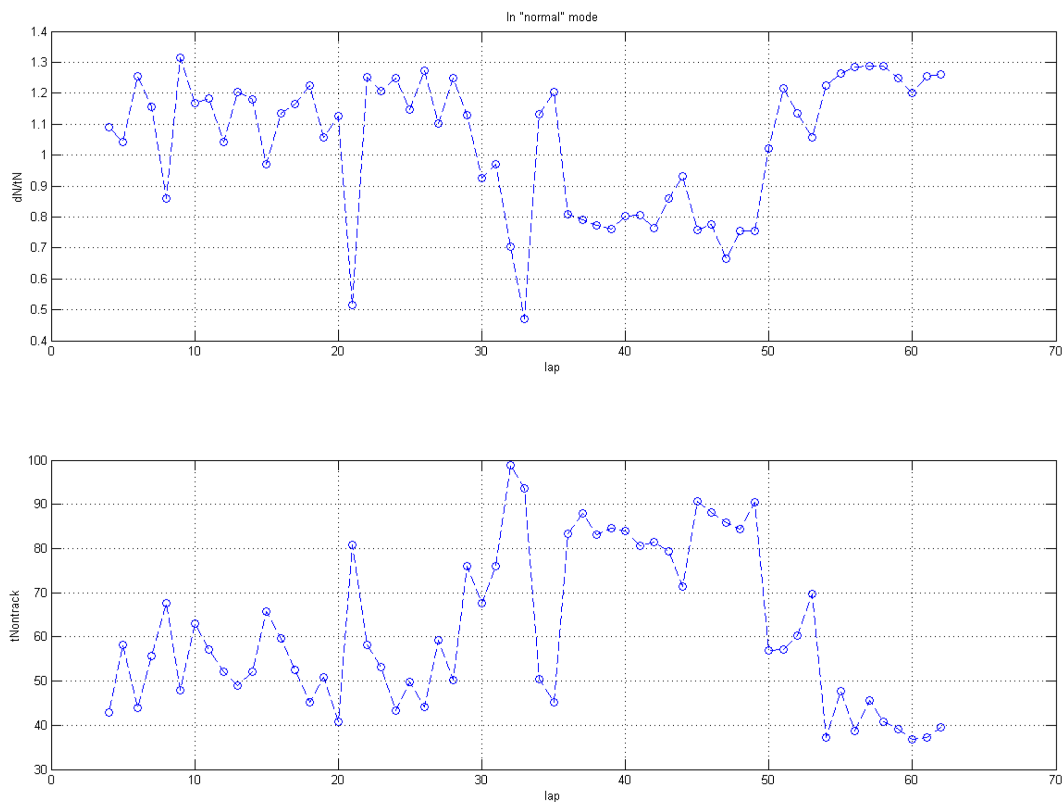


Bild 4.4: Versuchsergebnisse

Im unteren Graph ist tNontrack gegen die Rundenzahl aufgetragen. Auch hier schwanken die Werte stark und es kann Trend ausgemacht werden.
Auffallend ist, dass dieser Graph gegenläufig zu dem oberen ist. D.h. je mehr sich das Auto auf der Strecke befindet, umso weniger Distanz legt es in die richtige Richtung zurück. Im Bereich von Runde 36 bis 50 befindet sich der Wagen eher auf der Strecke und in den folgenden Runden legt er mehr Distanz in Fahrbahnrichtung zurück.
Bei Betrachtung der Trajektorien für den ersten Bereich kann man feststellen, dass das Verhalten aus Bild 4.5 vermehrt auftritt. Dabei beschreiben rote Punkte den „normalen“ und grüne den „Unstuck“-Modus. Im Rundenbereich ab Runde 50 tritt wiederholt das Muster aus Bild 4.6 auf. An den verschiedenen Fahrweisen kann man sehen, dass in Bild 4.6 weniger gebremst wird als in Bild 4.5. Dies spiegelt sich auch in den Kenndaten von Bild 4.4 wieder. In beiden Fällen wird nach links gelenkt.

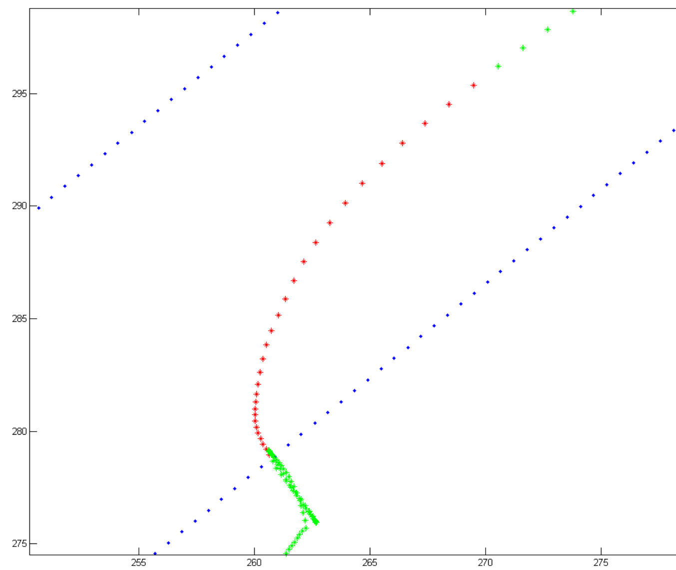


Bild 4.5: Häufiges Verhalten in Runden 36 bis 50

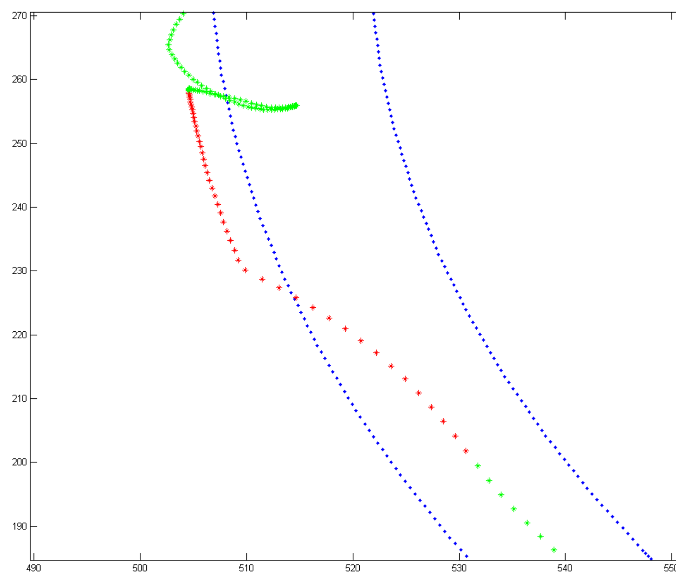


Bild 4.6: Oft vorkommendes Muster zu Beginn des Lernvorgangs

Auswertung

Es wurde festgestellt, dass das Auto keinen Fortschritt beim Ziel „schnelles Fahren“ sowie „auf der Strecke bleiben“ gemacht hat. Des Weiteren konnte beobachtet werden, dass zwei Fahrweisen aus Bild 4.5 und 4.6 in verschiedenen Rundenbereichen sich häuften. Bei beiden Fahrweisen wurde nach links gelenkt. Das kann bedeuten, dass der Agent zu wenig Zustände besucht hat und die Funktionsapproximation höhere Bereiche für den linken Bereich der Lenkung erzeugt. Dafür spricht die Tatsache, dass hier ein 24-dimensionaler Zustand-Aktionsraum vorhanden ist. Man kann also annehmen, dass trotz der ersten 3 Runden mit verschiedenen Geschwindigkeiten nicht genug Trainingsdaten vorhanden sind, um so einen großen Zustandsraum hinreichend aufzuspannen. Dies wird anscheinend auch nicht nach 62 Runden erreicht.

4.5 Variante 4

Die Dimension kann mit diesem Ansatz für das Zustandssignal nicht geändert werden, da der Agent sonst zu wenig Informationen erhält, um seine Ziele umzusetzen[4]. Daher muss der Agent häufiger in ähnliche Zustände kommen. In einem weiteren Versuch könnte das Rennen jedes mal neu gestartet werden, wenn der Agent in Zustände der Menge S^- gelangt. Mit den bis dahin gesammelten Transitionen werden die Q-Werte berechnet und das neuronale Netz trainiert. Dieses wird vom Agenten geladen, bevor er erneut von der Startposition losfährt. Dadurch werden öfter ähnliche Zustände besucht. In unbekanntere Bereiche kommt der Agent, je weiter er sich von der Start/Ziel-Linie entfernt. Dadurch würde ein Ungleichgewicht zwischen den Bereichen entstehen, da der Weg zu unbekannteren Bereichen über bekanntere führt. Daher lernt der Agent die vorderen Abschnitte besser zu fahren, als die hinteren. So ein Ansatz dient eher zu Testzwecken, als dem Ziel das „schnelle Fahren“ zu erlernen.

[4] siehe Abschnitt 3.1.2 auf Seite 27

5 Zusammenfassung

In dieser Arbeit wurde ein Ansatz für das „schnelle Fahren“ über eine Rennstrecke entwickelt. Andere Ansätze, die das selbe Ziel verfolgen, geben dem Agenten zu wenig Informationen über den Verlauf der Strecke. So ist es nicht möglich eine optimale Bahn über die *komplette* Strecke zu finden, sondern nur kurzfristige Reaktion auf Fahrbahnänderungen vorzunehmen.
Das Zustandssignal ist möglichst generisch zusammengesetzt worden, damit das gesammelte Wissen auf einer Strecke auf andere Strecken übertragen werden kann. Der Agent erhält Wissen über spätere Abschnitte der Strecke und hat so die Möglichkeit sich schon im aktuellen Abschnitt eine gute Ausgangslage für den nächsten Abschnitt zu schaffen.
Der Agent interagiert mit der Umwelt, von der kein Modell vorhanden ist, um Erfahrungen in Form von Transitionen zu sammeln. Mittels Q-Learning werden die Werte für die Q-Funktion anhand der gesammelten Transitionen berechnet. Nach dem hier angewendeten Neural Fitted Q Iteration Algorithmus wird ein neuronales Netz nach Rundenende im Batch-Verfahren trainiert. So wird der Nachteil des Online-Lernverfahrens vermieden, bei dem Bereiche im Zustand-Aktionsraum, die nicht trainiert werden, sich unkontrolliert verändern. Gleichzeitig kann die generalisierende Eigenschaft neuronaler Netze genutzt werden.
Damit der Agent das „schnelle“ Fahren erlernt, muss die Kostenfunktion ihm geringere Kosten für einen Zeitschritt geben, je weiter er in Richtung der Fahrbahn gefahren ist. Bei der Variation der Kostenfunktion ist darauf zu achten, dass weiterhin die Markov-Eigenschaft gilt.
Ein Versuch konnte den erwarteten Lernvorgang nicht bestätigen. Am wahrscheinlichsten wird die zu hohe Dimension des Zustand-Aktionsraums als Grund angesehen.

6 Ausblick

Zum Schluss sollen Ideen vorgestellt werden, die nicht mehr in der Bearbeitungszeit umgesetzt werden konnten.

6.1 ABS und Trajektionskontrolle

Ein Anti-Blockier-System verhindert, dass einzelne Reifen während des Bremsvorgangs blockieren. Dann wird nur noch über die Gleitreibung gebremst und die Lenkung hat keine Wirkung, da die Haftreibung wegfällt. Durch Absenken der Bremskraft können sich die Räder wieder drehen und ein Bremsvorgang mit Haftreibung kann erneut gestartet werden.
Analog dazu sollte die Beschleunigungskraft mittels einer Trajektionskontrolle angepasst werden, wenn die Reifen beim Beschleunigungsvorgang durchdrehen. In [Onieva u. a., 2009] und [Engesser, 2011] wird vorgeschlagen, wie beide Systeme für Torcs umgesetzt werden können.

6.2 Teststrecke

Das große Problem dieses Ansatzes besteht in seiner hohen Dimension. Könnte man auf die z-Koordinate verzichten, würde sich die Ordnung bei $k = 1$, von 24 auf 17 verkleinern. Eine Strecke die keine Unterschiede in der z-Koordinate hat und simpel aufgebaut ist, wäre ein besserer Startpunkt um diesen Ansatz zu testen.

6.3 Online- anstatt Batch-Lernen

Beim NFQ-Algorithmus werden alle Transitionen gesammelt und damit nach jeder Runde ein Q-Wert pro Transition berechnet. Folglich dauert die Aktualisierung der Q-Werte mit zunehmender Rundenzahl sehr lange. Pro Runde wurden hier circa 2000 Transitionen gesammelt. Die Aktualisierung eines Q-Werts dauert durchschnittlich 19 msec. In der 50. Runden muss also 32 Minuten lang gerechnet werden. Beim Online-Lernen werden die Transitionen nur einmal für die Aktualisierung eines Q-Werts verwendet. Die Zeitersparnis würde also für die Auswahl eines Online-Lernverfahren sprechen. In [Wilson und Martinez, 2003] wird sogar behauptet, dass Online-Verfahren dem Batch-Lernen in vielen Situation hinsichtlich der Lerngeschwindigkeit überlegen sind.

Abbildungsverzeichnis

2.1 Interaktion zwischen Agent und Umwelt bei RL 5
2.2 Skizze zum Beispiel „Roboter balanciert eine Stange“ 9
2.3 Quelle:[Sutton und Barto, 1998] 15
2.4 Quelle:[Meisel, 2011] 20
2.5 Neuron 21
2.6 Feedforward Netzwerk 21

3.1 Gute und schlechte Abtastung eines Zustandsraums 25
3.2 Beispiel: Flugzeug soll zu Punkt P fliegen lernen 26
3.3 Streckenpunkte,Abschnitte und Segmente 29
3.4 Veränderung des aktuellen Abschnitts während der Fahrt 29
3.5 Wechsel des aktuellen Abschnitts 30
3.6 Form aus Anordnung der Streckenpunkte 31
3.7 mögliche Trajektorie für $k = 0$ 33
3.8 mögliche Trajektorie für $k = 1$ 33
3.9 mögliche Trajektorie für $k = 3$ 34
3.10 Unterschied zwischen zurückgelegter Strecke und Distanz in Fahrtrichtung der Strecke 36
3.11 pitch, roll, yaw 38
3.12 Anordnung Auto(C für car) und einem beliebigen Punkt P . . . 39
3.13 Abtastung des Zustandsraum(x→steer,y→acc): Erst grob(blau), dann fein(rot) 41
3.14 Abtastung des Zustandsraum(x→steer,y→acc): acc grob, steer feiner je näher an 0 42
3.15 „Unstuck“-Vorgang 44
3.16 Topologie des neuronalen Netzwerks 45
3.17 Abgetastete Aktionsraum für einen Zustand 46

3.18 Quelle:[Gabel u. a., 2011] 47

4.1 Zielbereich für Variante 1 54
4.2 Zielbereich für Variante 2 56
4.3 Detaillierte Darstellung der zweiten Variante 56
4.4 Versuchsergebnisse 59
4.5 Häufiges Verhalten in Runden 36 bis 50 60
4.6 Oft vorkommendes Muster zu Beginn des Lernvorgangs 60

Literaturverzeichnis

[torcsrestart] : *The Official TORCS 1.3.3 FAQ*. – URL http://torcs.sourceforge.net/index.php?name=Sections&op=viewarticle&artid=30#c6_8

[Abdullahi und Lucas 2011] Abdullahi, A.A. ; Lucas, S.M.: Temporal difference learning with interpolated n-tuples: Initial results from a simulated car racing environment. In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on* IEEE (Veranst.), 2011, S. 321–328

[Alpaydin 2008] Alpaydin, Ethem: *Maschinelles Lernen*. Oldenbourg Wissenschaftsverlag, 4 2008. – ISBN 9783486581140

[Bellman 1957] Bellman, R.: A Markovian decision process / DTIC Document. 1957. – Forschungsbericht

[Butz und Lonneker 2009] Butz, M.V. ; Lonneker, TD: Optimized sensory-motor couplings plus strategy extensions for the torcs car racing challenge. In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* IEEE (Veranst.), 2009, S. 317–324

[Cardamone u. a. 2009] Cardamone, L. ; Loiacono, D. ; Lanzi, P.L.: On-line neuroevolution applied to the open racing car simulator. In: *Evolutionary Computation, 2009. CEC'09. IEEE Congress on* IEEE (Veranst.), 2009, S. 2622–2629

[Cardamone u. a. 2010] Cardamone, L. ; Loiacono, D. ; Lanzi, P.L.: Learning to Drive in the Open Racing Car Simulator Using Online Neuroevolution. In: *Computational Intelligence and AI in Games, IEEE Transactions on* 2 (2010), sept., Nr. 3, S. 176 –190. – ISSN 1943-068X

[Engesser 2011] ENGESSER, Thorsten: *Generalisierendes Neural Fitted Q Learning im TORCS-Competition-Framework*, Universität Freiburg, Diplomarbeit, 3 2011

[Ertel 2007] ERTEL, Wolfgang: *Grundkurs Künstliche Intelligenz . Eine praxisorientierte Einführung*. 1. Vieweg+Teubner, 11 2007. – ISBN 9783528059248

[Espié u. a. 2006] ESPIÉ, Eric ; GUIONNEAU, Christophe ; WYMANN, Bernhard ; DIMITRAKAKIS, Christos ; COULOM, Rémi ; SUMNER, Andrew: *TORCS, The Open Racing Car Simulator*. http://www.torcs.org. 2006. – URL http://www.torcs.org

[Gabel u. a. 2011] GABEL, T. ; LUTZ, C. ; RIEDMILLER, M.: Improved neural fitted Q iteration applied to a novel computer gaming and learning benchmark. In: *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on* IEEE (Veranst.), 2011, S. 279–286

[van Hoorn u. a. 2009] HOORN, N. van ; TOGELIUS, J. ; WIERSTRA, D. ; SCHMIDHUBER, J.: Robust player imitation using multiobjective evolution. In: *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, may 2009, S. 652 –659

[Knabe u. a. 2005] KNABE, J. ; RIEDMILLER, M. ; SPERSCHNEIDER, V.: Kooperatives Reinforcement learning in Multiagentsystemen. (2005). – URL http://www-lehre.inf.uos.de/~jknabe/papers/CooperativeRLinMAS.pdf

[Lange u. a.] LANGE, S. ; GABEL, T. ; RIEDMILLER, M.: Batch Reinforcement Learning.

[Loiacono u. a. 2009] LOIACONO, D. ; CARDAMONE, L. ; LANZI, P.L.: Simulated car racing championship 2009: Competition software manual. In: *Dipartimento di Elettronica e Informazione, Politecnico di Milano, Tech. Rep* (2009)

[Loiacono u. a. 2010] LOIACONO, D. ; PRETE, A. ; LANZI, P.L. ; CARDAMONE, L.: Learning to overtake in TORCS using simple reinforcement learning. In: *Evolutionary Computation (CEC), 2010 IEEE Congress on*, july 2010, S. 1 –8

[Matlabhilfe 2011] MATLABHILFE: Neural Network Toobox, Preprocessing and Postprocessing / Matlab. 2011. – Forschungsbericht

[Meisel 2011] MEISEL, Andreas: *Musterklassifikation und Neuronale Netze*. 1 2011

[Nikolov 2012] NIKOLOV, Ivo: Maschinelles Lernen zur Optimierung einer autonomen Fahrspurführung. (2012). – URL http://opus.haw-hamburg.de/volltexte/2012/1453/pdf/Ivo_Nikolov_Masterarbeit.pdf

[Onieva u. a. 2009] ONIEVA, E. ; PELTA, DA ; ALONSO, J. ; MILANÉS, V. ; PÉREZ, J.: A modular parametric architecture for the TORCS racing engine. In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* IEEE (Veranst.), 2009, S. 256–262

[Riedmiller 2005] RIEDMILLER, M.: Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In: *Machine Learning: ECML 2005* (2005), S. 317–328

[Riedmiller u. a. 2007] RIEDMILLER, M. ; MONTEMERLO, M. ; DAHLKAMP, H.: Learning to drive a real car in 20 minutes. In: *Frontiers in the Convergence of Bioscience and Information Technologies, 2007. FBIT 2007* IEEE (Veranst.), 2007, S. 645–650

[Röttger 2009] RÖTTGER, M.C.: *Reinforcement Learning für kontinuierliche Zustands-und Aktionsräume unter Berücksichtigung der wissenschaftlichen Informationsverarbeitung*, Universitätsbibliothek Freiburg, Dissertation, 2009

[Sivanandam 2003] SIVANANDAM, Manu: *Introduction to Neural Networks with matlab 6.0*. 1st. McGraw-Hill, 2003

[Sutton und Barto 1998] SUTTON, Richard S. ; BARTO, Andrew G.: *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The Mit Press, 5 1998. – ISBN 9780262193986

[Van Hoorn u. a. 2009] VAN HOORN, N. ; TOGELIUS, J. ; WIERSTRA, D. ; SCHMIDHUBER, J.: Robust player imitation using multiobjective evolution. In: *Evolutionary Computation, 2009. CEC'09. IEEE Congress on* IEEE (Veranst.), 2009, S. 652–659

[Watkins 1989] WATKINS, Christopher J. C. H.: *Learning from Delayed Rewards*. Cambridge, UK, King's College, Dissertation, 1989

[Wikipedia 2012] WIKIPEDIA: *Ralley — Wikipedia, The Free Encyclopedia*. 2012. – URL http://de.wikipedia.org/wiki/Rallye

[Wilson und Martinez 2003] WILSON, D.R. ; MARTINEZ, T.R.: The general inefficiency of batch training for gradient descent learning. In: *Neural Networks* 16 (2003), Nr. 10, S. 1429–1451

[Wolter 2008] WOLTER, Anne: *Reinforcement Learning in der Roboter-Navigation: Grundlagen, Methoden, Realisierung und Experimente*. Vdm Verlag Dr. Müller, 6 2008. – ISBN 9783639047028

[Wymann] WYMANN, Bernhard: *T.O.R.C.S. Manual installation and Robot tutorial*. – URL www.berniw.org/aboutme/publications/torcs.pdf

A

APPENDIX

A.1 Parallelisierung

Im NFQ Algorithmus wird Batch-Lernen angewendet. Daher muss nach jedem Durchgang aus allen bisher gesammelten Transitionen neue Trainingsdaten für das neuronale Netz berechnet werden. Dies wurde mit Matlab umgesetzt. Möchte man Matlab-Scripte parallelisieren, wird die Parallel Computing Toolbox benötigt. So kann anstatt einer `for`-Schleife kann dann eine `parfor`-Schleife verwendet werden, die für jede Iteration einen neuen Thread eröffnet. Vorher muss mit `matlabpool` eine Parallel-Umgebung geschaffen werden. So konnte die Berechnungszeit um das 3,5-fache gesenkt werden.
Des Weiteren können auch Matlab-Funktionen, die mit der Compiler Toolbox per C/C++ Shared Libraries in ein externes Programm ausgelagert wurden, parallelisiert werden. Dafür muss der Befehl `matlabpool` ebenfalls mit in die C/C++ Shared Librariy eingefügt und im externen Programm ausgeführt werden.

A.2 selbstgeschriebene Simfunktion für Matlab

Die sim-Funktion simuliert ein erstelltes neuronales Netzwerk. Es legt den angegebenen Vektor an die Eingänge des Netzwerks und gibt den Rückgabewert zurück. Es hatt sich herausgestellt, dass diese Funktion langsam arbeitet, da sie viele Funktionen beinhaltet, die hier nicht benötigt werden.
Also wurde sie selbst geschrieben. Zunächst wird das betreffende neuronale Netz durch die Funktion readNet() ausgelesen, sodass benötigte Informationen über das Netz im Matlab-Workspace vorhanden sind. Es wird vorausgesetzt,

dass die Eingabeschicht und die versteckten Schichten die selbe Anzahl an Neuronen enthalten. Außerdem muss das Netz den Namen „net“ tragen.

```
1  %% NN laden
2   clear biases lastbias inweights lweights lastlweights ...
       trfcn trfcnout;
3  %% biases auslesen
4  [br,bc]=size(net.biases);
5  for i=1:br−1 % last bias is of size 1
6      biases(:,i)=net.b{i};
7  end
8  lastbias=net.b{br};
9  clear bc br i
10 %% weights und transfer function auslesen
11 % input weights:
12 % nur der input−layer hat eine Verbindung zu den inputs
13 % −> keine Schleife
14 inweights=net.IW{1};
15 % layer weights
16 [lr,lc]=size(net.LW);
17 k=1;
18 lweights=[];
19 for i=1:lr
20     for j=1:lc
21         [lwr,lwc]=size(net.LW{i,j});
22         if(¬isempty(net.LW{i,j}) && (lwr==lwc))
23             lweights=[lweights;net.LW{i,j}];
24             k=k+1;
25         elseif(¬isempty(net.LW{i,j}) && (lwr≠lwc))
26             lastlweights=net.LW{i,j};
27             k=k+1;
28         end
29     end
30 end
31 clear k lr lc i j lwr lwc
32 % transfer function:
33 [tr,tc]=size(net.layers);
34 for i=1:tr
35 trfcn{i}=net.layers{i}.transferFcn;
36 end
37 trfcnout=trfcn{tr};
38 trfcn=trfcn{1};
39 clear i tr tc
```

Danach wird auf diese Werte mit der selbst geschriebenen Funktion mysim() zugegriffen:

```
1 function y=mysim(in)
2 biases=evalin('base','biases');
3 lastbias=evalin('base','lastbias');
4 inweights=evalin('base','inweights');
5 lweights=evalin('base','lweights');
```

```
6  lastlweights=evalin('base','lastlweights');
7  trfcn=evalin('base','trfcn');
8  trfcnout=evalin('base','trfcnout');
9  [layersize,numLayer]=size(biases);
10 result=feval(trfcn,(in'*inweights'+biases(:,1)'));
11 for i=1:numLayer-1
12     result=feval(trfcn,(result*lweights((i-1)*layersize...
13     +1:i*layersize,:)'+biases(:,i+1)'));
14 end
15 y=feval(trfcnout,result*lastlweights'+lastbias);
```

Dabei spart man sich die Übergabe der Parameter, indem evalin() verwendet wird. Ein Test der Bearbeitungszeit hat folgendes Ergebnis in Matlab geliefert:

```
1 tic;for i=1:1000;sim(net,in);end;toc
2 Elapsed time is 5.082917 seconds.
3 tic;for i=1:1000;mysim(in);end;toc
4 Elapsed time is 0.633699 seconds.
```

A.3 Restartbefehl in Torcs einfügen

Um einen Restart-Befehl in Torcs zu implementieren, muss in Datei `torcsInstallFolder/src/libs/raceengineclient/raceengine.cpp` an zwei Stellen etwas verändert werden. Diese sind mit „hawdb“ gekennzeichnet. Die Software dieses Projekts ist bei Prof.Dr.-Ing. Meisel hinterlegt. Siehe auch [torcsrestart].

A.4 Kommunikation

Die Berechnung der neuen Q-Werte und das Trainieren des neuronalen Netzes wurde auf einem externen Rechner durchgeführt, auf dem Torcs nicht installiert war. Die Transitionen wurden in eine Datei geschrieben, die durchschnittlich 1 MB groß war, und in einen Dropbox-Verzeichnis gespeichert wurde. Danach wird die Lockdatei übertragen. Auf dem externen Rechner wird auf die Datei gewartet, indem jede Sekunde auf die Existenz der Lockdatei abgefragt wird. Sind beide Dateien vorhanden wird die Datei mit den Transitionen ausgelesen und die Q-Daten berechnet. Sind diese bereit wird damit das neuronale Netz

trainiert und über das Dropbox-Verzeichnis auf den Torcs-Rechner übertragen. In der Zwischenzeit kann der Agent weiter Transitionen sammeln. Ist er fertig, wird das neue Netz geladen und der Vorgang beginnt von neuem.

A.5 Tank und Schaden

Damit das Auto beim Fahren keinen Schaden nimmt, muss in der `car` Struktur in der Funktion drive() der Schwierigkeitsgrad auf 0 gesetzt werden.
`car->info.skillLevel=0`
Der Spritverbrauch wird in der Datei `torcsInstallFolder/src/modules/simu/simuv2/car.cpp` in der Funktion `SimCarUpdate`() eingestellt. Am Anfang muss die Zeile `car->fuel=100;` hinzugefügt werden.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel „Optimale Trajektorien mit Reinforcement Learning“ im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, den 12. April 2012 Dieter Büchler