Institut für Logik, Komplexität und Deduktionssysteme
Fakultät für Informatik
Universität Karlsruhe

Studienarbeit

Erweiterungen des TD(λ)-Learning – eine alternative Explorationsstrategie und Offline-Lernen unter Ausnutzung von Trajektorienspeicherung

Lutz Frommberger*

Betreuer: Dr. Martin Riedmiller**

17. September 2001

*Lutz.Frommberger@ira.uka.de
**riedml@ira.uka.de

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit befasst sich mit Methoden zur Lösung Sequenzieller Entscheidungsprobleme mit *Temporal Difference Learning* (TD(λ)-Learning) und den Problemen, die bei der Anwendung dieser Lernmethode auftreten können.

Das Einlernen der Neuronalen Netze zur Problemlösung gestaltet sich oft schwierig: Es gilt dabei nicht nur, die richtige Parameterkombination auszuwählen und eine passende Netztopologie zu finden, sondern auch zwischen den Varianten des Verfahrens auszuwählen. Diese Auswahl geschieht in der Regel auf der Basis von Heuristiken und ist für jedes Problem von Neuem vorzunehmen.

In dieser Arbeit werden zwei Schwierigkeiten aufgegriffen, die in der praktischen Anwendung von Bedeutung sind: Die Frage, auf welche Weise Exploration im Zustandsraum vonstatten geht, und die Vermeidung des Phänomens, dass bereits zeitweise erreichte Teillösungen im Laufe des Lernvorgangs wieder vergessen werden. Als Grundlage dafür dient die Speicherung der Abfolgen von Zustand-Aktions-Paaren, die während des Lernens auftreten.

Kapitel 2 ist eine kurze Einführung in das Thema *Sequenzielle Entscheidungsprobleme* und deren Lösung mittels TD(λ)-Learning. In Kapitel 3 wird eine Methode vorgeschlagen, die eine zielgerichtete und effektive Exploration ermöglichen soll. Diese *Phasenweise Exploration* soll sowohl dem Wunsch nach Ausnutzung des bestehenden Wissens über das System als auch einer weitestgehenden Exploration des Zustandsraums nachkommen. Kapitel 4 schlägt eine Möglichkeit vor, aus dem durch die Speicherung der bereits bekannten Trajektorien erworbenen Wissen eine Möglichkeit des *Offline-Lernens* abzuleiten, indem die gespeicherte Datenbasis für den Lernvorgang herangezogen wird. Beide Abschnitte beginnen dabei mit einer Analyse der auftretenden Schwierigkeiten und leiten daraus einen Lösungsvorschlag ab. Kapitel 5 schließt mit einem Ausblick.

Die Arbeit ist so gestaltet, dass sie für Informatikstudierende im Hauptstudium auch ohne Hintergrundwissen über neuronale Lernmethoden verständlich sein sollte. Ein rudimentäres Grundwissen über künstliche Neuronale Netze und ihre Funktionsweise wird vorausgesetzt.

2 TD(λ)–Learning

2.1 Lösung Sequenzieller Entscheidungsprobleme

Sequenzieller Entscheidungsprobleme sind Probleme, bei denen ein System aus einem (beliebigen) Startzustand mittels Ausführen von Aktionen in einen wohldefinierten Zielzustand überführt werden soll. Zu jedem Zeitpunkt wird dabei der Zustand des Systems durch Anwendung einer Aktion in den Folgezustand gebracht, wobei diese Transformation von Systemzustand und ausgeführter Aktion abhängige Kosten verursacht.

Die Lösung des Sequenziellen Entscheidungsproblems ist eine Aktionsfolge, die, auf den Startzustand angewendet, das System schrittweise in den Zielzustand überführt. Gesucht ist also eine Folge von Einzelentscheidungen. Jede dieser Entscheidungen ist dabei nur von dem aktuellen Zustand des Systems abhängig, getroffen werden soll sie mit dem Ziel einer optimalen Lösung des Gesamtproblems, also der nicht unmittelbaren Folgen der Entscheidung. Da diese im Moment der Entscheidung im Allgemeinen nicht bekannt sind, bezeichnet man solche Systeme als *Dynamische Systeme*.

Ein Sequenzielles Entscheidungsproblem besteht aus

- einer Menge von Zuständen X
- einer Menge von Zielzuständen $X^+ \subset X$
- einer Menge von Randzuständen $X^- \subset X$, $X^- \cup X^+ = \emptyset$
- einer endlichen Menge von Aktionen A
- einer Zustandsübergangsfunktion $f(x, a)$ mit $x \in X$ und $a \in A$, die durch Anwendung von a auf x das System in einen Folgezustand überführt
- einer Kostenfunktion $r(x, a) \in \mathbb{R}$ mit $x \in X$ und $a \in A$, die die durch Anwendung von a im Zustand x entstehenden Kosten liefert (*Payoff*).

Zu einem Zeitpunkt t ist das System in einem Zustand $x_t \in X$, unter Anfallen von Kosten $r(x_t, a_t)$ wird es in den Folgezustand $x_{t+1} = f(x_t, a_t)$ überführt.

2.1.1 Der Zustandsraum

Der Zustandsraum X ist theoretisch unendlich, durch die Beschränkung der Laufzeit des Dynamischen Systems kann er allerdings als endlich betrachtet werden, da in endlicher Zeit nur endlich viele Zustände erreicht werden können. Ein Systemzustand x_t besteht aus allen Parametern, die zur eindeutigen Beschreibung des Systems zum Zeitpunkt t notwendig sind. Das sind bspw. die Sensordaten eines Roboters oder die aktuelle Position, Geschwindigkeit und Beschleunigung eines Autos. Auch Informationen aus der Vergangenheit, wie die Position zu einem früheren Zeitpunkt, können Teil des Zustands sein.

2.1.2 Die Zielzustände

Das Sequenzielle Entscheidungsproblem gilt als gelöst, wenn das System einen Zustand erreicht, der Teil der Zielzustandsmenge X^+ ist. In dieser Arbeit werden die Zielzustände stets *absorbierende* Zustände sein, d. h., sie können nach Erreichen nicht mehr verlassen werden, es gilt also für $x \in X^+$ und für alle $a \in A$: $f(x, a) = x$.

2.1.3 Die Randzustände

Die Menge X^- bezeichnet die Randzustände, die das System nicht erreichen sollte. Dies können Zustände sein, die das System schädigen (wie bspw. eine Gelenküberstreckung bei einem Roboterarm) oder einfach Zustände, an denen ein Weiterlaufen des Systems bekanntermaßen keine Hoffnung mehr auf Lösung zulässt. Auch die Randzustände werden in dieser Arbeit stets absorbierende Zustände sein.

2.1.4 Die Aktionsmenge

Theoretisch kann auch die Aktionsmenge A unendlich sein. Beim Autofahren bspw. kann beliebig viel Gas gegeben werden, die Abstufung ist kontinuierlich, es gibt also unendlich viele Aktionen. In dieser Arbeit soll sich aber nur mit endlichen Aktionsmengen beschäftigt werden, also bspw. mit dem dummen Autofahrer, der nur die Aktionen „Vollgas“ und „Kein Gas“ kennt.

Das Ausführen von Aktionen wird durch einen *Agent* herbeigeführt, der auf dem System operiert. Zu jeden Zeitpunkt t wird genau eine Aktion a_t ausgeführt, die das System in den Zustand x_{t+1} überführt. Die Auswahl, welche Aktion ausgeführt werden soll, trifft der Agent auf Grund eines Regelwerks, der *Strategie* oder *Policy* π. Dabei hat er als Entscheidungsgrundlage den gesamten Systemzustand x_t zur Verfügung, in welchen Zustand die Befolgung der Strategie das System aber bringen wird, ist dem Agenten im Allgemeinen nicht bekannt (bspw. weiß der Autofahrer nicht, wie schnell sein Auto nach dem Treten von Vollgas wirklich sein wird). Ziel wird es immer sein, eine möglichst gute Strategie des Agenten zu finden.

2.1.5 Die Übergangsfunktion

Mittels der Übergangsfunktion $f(x_t, a_t)$ wird das System vom Zustand x_t nach x_{t+1} überführt. Der Folgezustand hängt also nur vom aktuellen Zustand und der gewählten Aktion ab, insbesondere gilt die *Markov-Eigenschaft*, das heißt, dass der Weg, auf dem x_t erreicht wurde, irrelevant ist[1].

In der Praxis wird $f(x_t, a_t)$ allerdings nicht immer in identische Zustände x_{t+1} überführen. Ein Auto wird bei Gegenwind auf Vollgas anders reagieren als bei Rückenwind, der Faktor „Wind“ wird also als *Störgröße* Einfluss auf das Systemverhalten haben. Somit gilt eigentlich $x_{t+1} = f(x_t, a_t, p_t)$, wenn man p_t als Vektor aller Störeinflüsse definiert. Da die Gesamtheit all dieser Einflüsse in der Regel nicht beschreibbar ist (und somit auch nicht sinnvoll Teil des Systemzustands sein kann), wird sich f im Allgemeinen nicht deterministisch verhalten. Diese Eigenschaft von f wird sich unter Umständen als problematisch erweisen (siehe Kapitel 4.2.2).

2.1.6 Die Kostenfunktion

Die Ausführung einer Aktion a_t führt zu Kosten $r(x_t, a_t)$. Über die Kostenfunktion erfolgt eine qualitative Bewertung der einzelnen Zustände. Ziel des Agenten wird es sein, die Aktionen so auszuwählen, dass die Gesamtkosten, also die Summe aller Einzelkosten r_t von der Startstellung bis zum Erreichen eines Zielzustandes, minimiert werden. Das bedeutet, dass im Allgemeinen nicht die im jeweiligen Zustand kostengünstigste Aktion ausgewählt werden wird, sondern diejenige, die am Ende die niedrigsten Gesamtkosten zur Folge haben wird.

In einem absorbierenden Zielzustand $x \in X^+$ gilt, dass für alle Aktionen $a \in A$ Kosten von $r(x, a) = 0$ entstehen.

2.2 Das Problem als Optimierungsaufgabe

2.2.1 Startstellungen und Horizont

Bei der Lösung eines Sequenziellen Entscheidungsproblems spielt nicht zuletzt die Menge der Startzustände $X^S \subseteq X$ eine Rolle. X^S kann theoretisch identisch mit X sein, in der Praxis wird man aber häufig Probleme antreffen, die nur aus bestimmten Zuständen starten können. In dieser Arbeit wird die Menge der Startzustände stets als endlich angesehen. Von ihrer Auswahl stark der Erfolg bei der Lösung des Problems ab.

Der *Horizont* $T \in \mathbb{N}$ bezeichnet die maximale Anzahl von Zeitschritten, die für die Lösung des Problems zugelassen wird; ist Zeitschritt $t = T$ erreicht, wird das System abbrechen. Die sinnvolle Wahl von T ist ebenfalls wichtig für den Lösungserfolg:

[1] Vorherige Zustände oder Aktionen können allerdings selbst Bestandteil des Zustandsraumes sein.

Einerseits muss T mindestens so groß gewählt sein, dass das System von jedem Startzustand auch in maximal T Schritten in einen Zielzustand überführt werden kann (bevorzugt etwas größer, um dem System zu ermöglichen, auch suboptimale Lösungen zu finden), andererseits erhöht ein zu großer Wert von T die Komplexität der Lernaufgabe.

Da in einem absorbierenden Zielzustand $x \in X^+$ für alle $a \in A$ $f(x,a) = x$ und $r(x,a) = 0$ gilt, kann man o. B. d. A. annehmen, dass ein Zielzustand immer genau nach T Schritten erreicht wird, ein Lauf des Agenten also stets T Zeitschritte dauert. In der Praxis wird der Lösungsversuch hier natürlich (erfolgreich) abgebrochen werden.

Im einem Endzustand x_T können noch zusätzlich aktionsunabhängige Kosten $r(x_T)$ auftreten. In einem Zielzustand $x_T \in X^+$ gilt $r(x_T) = 0$. In einem Randzustand $x_r \in X^-$ werden ebenfalls Kosten auftreten, die eine Bestrafung (*Penalty*) darstellen. Der Penalty sollte so hoch gewählt werden, dass ein weiteres Besuchen dieser Zustände unattraktiv wird.

2.2.2 Bewertung der Strategie

Ein Sequenzielles Entscheidungsproblem hat im Allgemeinen mehrere Lösungen, von denen die „beste“ gesucht wird. Die beste Lösung wird die Strategie sein, die das System aus jeder beliebigen Startstellung mit minimalen Kosten in einen Zielzustand bringt.

Die „Güte“ einer Strategie π wird durch die *Bewertungsfunktion* V^π beschrieben, die die bei Überführen des Systems von einem Start- in einen Endzustand entstehenden Gesamtkosten beschreibt, wenn Strategie π angewandt wird. Nach der Strategie π werden die jeweiligen Aktionen in jeden Zeitschritt t ausgewählt, also $a_t = \pi(x_t)$. Für einen Startzustand $x_0 \in X^S$ gilt:

$$V^\pi(x_0) = r(x_T) + \sum_{t=0}^{T-1} r(x_t, a_t) \tag{2.1}$$

$$= r(x_0, a_t) + r(x_T) + \sum_{t=0}^{T-1} r(f(x_t, a_t), \pi(f(x_t, a_t))) \tag{2.2}$$

Das Finden einer optimalen Lösung wird also zu einem Optimierungsproblem: Es gilt, die Strategie π zu finden, die die Funktion V^π minimiert.

V ist auf dem gesamten Zustandsraum X definiert, Folgezustände von x_0 kann man als Startstellung desselben Problems mit einem entsprechend kleineren Horizont ansehen. $V^\pi(x)$ liefert also für jedes $x \in X$ eine Prognose der noch zu erwartenden Kosten. Wenn π für alle Zustände, die nach Ausführung einer Aktion entstehen können, bekannt ist, wird zur Lösung des Problems die Aktion a angewendet, für

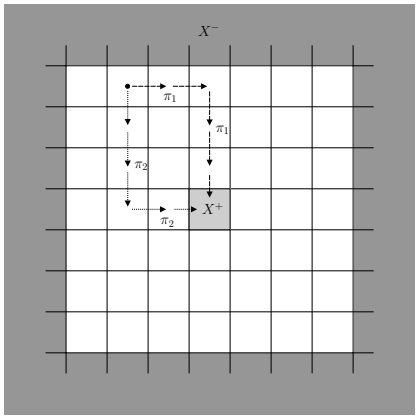

Abbildung 2.1: Zielfinden im zweidimensionalen Raum

die $r(x,a) + V^\pi(f(x,a))$ minimal ist. Üblicherweise ist π allerdings nicht vollständig bekannt.

Eine optimale Strategie π^* ist dann gefunden, wenn V^{π^*} ein absolutes Minimum von V ist ($V^* := V^{\pi^*} = \min_\pi V^\pi$). V^* ist die optimale Bewertung. Die Anwendung von π^* liefert also mit Hilfe von V^* diejenige Aktionenfolge, die das Sequenzielle Entscheidungsproblem löst.

2.2.2.1 Beispiel: Zweidimensionales Zielfinden

Ein simples Beispiel für ein Sequenzielles Entscheidungsproblem ist ein diskreter zweidimensionaler Raum, in dem das Ziel darin besteht, den Agenten in die Mitte zu steuern, ohne sich in einen Randbereich zu begeben (siehe Abbildung 2.1). Mögliche Aktionen sind Bewegungen in x- und y-Richtung, für jede Bewegung fallen Kosten von jeweils 0,1 an, wird ein Randbereich erreicht, gibt es einen Penalty von 0,5. Hierbei wäre also

- $X = \mathbb{Z}^2; \quad X^+ = (0,0); \quad X^- = \mathbb{Z}^2 \setminus \{[-3,3] \times [-3,3]\}$
- $A = \{(-1,0),(1,0),(0,-1),(0,1)\}$
- $f(x,a) = x + a \ \ \forall x \notin X^+ \cup X^-, \quad f(x,a) = x$ sonst
- $r(x,a) = 0,1 \ \ \forall x \notin X^- \cup X^+, \, a \in A$
- $r(x) = 0,5 \ \ \forall x \in X^-; \quad r(x) = 0 \ \ \forall x \in X^+$

Für dieses Beispiel ist es möglich, die optimale Bewertungsfunktion direkt aus der Betrachtung des Systems anzugeben: Für $x = (p,q) \in X \setminus X^-$ gilt $V(x) = 0,1 *$

($|p| + |q|$). Die in Abbildung 2.1 eingezeichneten Lösungswege folgen verschiedenen Strategien. Strategie π_1 findet sich in Algorithmus 1. Bei Strategie π_2 sind die beiden

Algorithmus 1 Lösungsstrategie für das Zielfinden im zweidimensionalen Raum

```
betrachte Position (p, q)
while p ≠ 0 do
  if p > 0 then
    führe Aktion (−1, 0) aus {gehe nach links}
  else
    führe Aktion (1,0) aus {gehe nach rechts}
  end if
end while
while q ≠ 0 do
  if q > 0 then
    führe Aktion (0, −1) aus {gehe nach oben}
  else
    führe Aktion (0,1) aus {gehe nach unten}
  end if
end while
```

While-Blöcke vertauscht. Beide Strategien lösen das Problem mit minimalen Kosten, sind also optimale Strategien.

2.3 Lösungsmethoden

2.3.1 Dynamisches Programmieren

Die Bewertungsfunktion V wird in der Praxis selten in einer Form vorliegen, die direkt die Berechnung der Bewertung aus dem Startzustand x_0 zulässt. Die Konsequenz einer zu einem Zeitpunkt getroffenen Entscheidung wird erst in der Folgezeit sichtbar werden. Der Ansatz des *Dynamischen Programmierens* besteht darin, die Lösung des Gesamtproblems auf Lösungen von Teilproblemen zurückzuführen.

Für einen Zustand $x_T \in X^+$ ist die optimale Bewertung bekannt, $V^*(x_T) = r(x_T, a) = 0$. Dies wäre sozusagen ein nullschrittiges Problem. Daraus lässt sich das einschrittige Problem ableiten: Für jeden Zustand x_{T-1}, der durch Ausführen einer Aktion a in den Zustand x_T überführt werden kann, ist eine optimale Lösung sofort durch Ausprobieren aller Aktionen ermittelbar, es ist die Aktion $a \in A$, für die $r(x_{T-1}, a) + V^*(x_T) = r(x_{T-1}, a)$ minimal ist. Ist also eine optimale Lösung für ein $(n-1)$-schrittiges Problem bekannt, so lässt sich daraus auch eine Lösung für ein n-schrittiges Problem gewinnen, indem man das verbleibende einschrittige Problem löst. Eine Formalisierung dieses Verfahrens als Theorem findet sich in [Ber87]:

Theorem 1 *Sei $V^*(x_0)$ die optimale Kostenfunktion für ein Grundproblem mit dem Anfangszustand x_0. Dann gilt: $V^*(x_0) = V_0(x_0)$, wobei V_0 sich aus dem letzten Schritt des folgenden Algorithmus' ergibt:*

$$V_T(x_T) = r(x_T)$$

$$V_t(x_t) = \min_{a_t \in A}(r(x_t, a_t) + V_{t+1}(f(x_t, a_t))), \qquad t = 0, \ldots, T-1 \tag{2.3}$$

Außerdem gilt: Wenn für alle t und alle x_t gilt, dass $a_t = \pi^*(x_t)$ die Aktion ist, die $V_t(x_t)$ minimiert, dann ist π^* optimal.

Gleichung 2.3 ist der *DP-Algorithmus*.

Dadurch, dass V im Allgemeinen nicht so vorliegt, dass man die Bewertungen direkt ausrechnen kann, wird, kann eine optimale Lösung in der Praxis nur garantiert werden, wenn für alle Zustände als Startstellung alle möglichen Strategien befolgt wurden. Stellt man sich dies als Entscheidungsbaum der Tiefe T und mit dem Zustand $x_0 \in X^S$ als Wurzelknoten vor, muss dieser vollständig besucht werden, um Optimalität gewährleisten zu können. So kann selbst das einschrittige Problem erst als gelöst betrachtet werden, wenn für jeden Zustand $x \in X$, für den es eine Aktion $a \in A$ gibt, für die $f(x, a) \in X^+$ gilt, $V(x) = r(x, a) + r(f(x, a))$ bekannt ist. Insbesondere müssen alle diese Zustände auch besucht worden sein, was in der Praxis auch für Teilprobleme schwierig zu erreichen ist, da im Allgemeinen gar nicht bekannt ist, wie die möglichen Vorgängerzustände eines Zielzustandes aussehen. Diese Suboptimalität ist aber hinnehmbar: Für jeden Zielzustand x_T sind die Kosten $r(x_T) = 0$ bekannt. Wenn der Agent einen Zielzustand findet, ist für den Vorgängerzustand x_{T-1} als Bewertung $V(x_{T-1}) = r(x_{T-1}, a_{T-1})$ anzunehmen. Natürlich kann es über einen „Umweg" über einen anderen Zustand noch eine günstigere Bewertung für x_{T-1} geben, in diesem Falle wäre die Strategie dann entsprechend anzupassen. Aber selbst wenn ein solcher „Umweg" nicht gefunden würde, wäre die Bewertung als Grundlage für die Bewertung weiter „entfernter" Zustände brauchbar. Aus der (unvollständigen und suboptimalen) Bewertung $V(x_{T-1})$ ließe sich gemäß Gleichung 2.3 auf eine (natürlich ebenfalls suboptimale) Bewertung von Vorgängerzuständen von x_{T-1} schließen, die im Falle einer besseren Lösung des Teilproblems entsprechend anzupassen sind.

2.3.1.1 Beispiel: Tabelle als Lösung des zweidimensionalen Zielfindens

Das Beispielproblem aus Kapitel 2.2.2.1 lässt sich auf einfache Weise durch Eintragen von Kosten in eine Tabelle lösen. Für jeden Zustand wird ein Tabelleneintrag angelegt[2], der die Kostenprognose für den bis dahin ermittelten besten Weg nach X^+ enthält. Als Strategie wird also in jedem Zustand diejenige Aktion ausgewählt, die das

[2]Was für nicht diskrete Zustandsräume natürlich so einfach nicht geht.

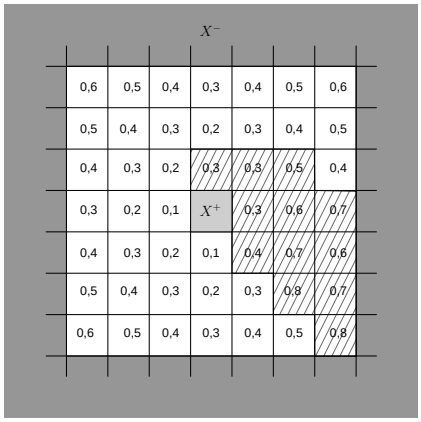

Abbildung 2.2: Lösung des Zielfindens im diskreten zweidimensionalen Raum mit Kostentabelle

System in einen Folgezustand mit laut Tabelle minimaler Kostenprognose überführt, Abweichungen davon muss man zulassen, die Einzelheiten über diese *Exploration* finden sich in Kapitel 2.4.1.

Nach Erreichen eines Finalzustands wird die durchlaufene Trajektorie rückwärts durchwandert, und die real angefallenen Kosten werden für jeden Zustand eingetragen, falls sie niedriger sind als der vorherige Tabelleneintrag bzw. falls noch kein Eintrag vorhanden ist. In Abbildung 2.2 sind die jeweils erwarteten Kosten für jeden Zustand eingetragen. Im schraffierten Bereich allerdings stimmen die Kostenprognosen (noch) nicht.

Im nächsten Trainingsdurchgang wird der Agent die in Abbildung 2.3 gezeigte Trajektorie durchlaufen. Im Zielzustand angekommen werden nun auf Grund der gewonnenen Information über die angefallenen Kosten die Prognosewerte angepasst. Die Tabelle nähert die optimale Lösung jetzt besser an, liefert aber immer noch nicht für alle Zustände eine korrekte Prognose, auch einige der korrigierten Werte entsprechen noch nicht der optimale Lösung.

2.3.2 Temporal Difference Learning

Das *Temporal Difference Learning* (TD-Learning) wurde von R. Sutton in [Sut88] vorgestellt, um Vorhersagen in einem zeitlichen Ablauf aufzustellen. Es ist eine Anwendung der Funktionsapproximation mittels Neuronaler Netze auf das Konzept des Dynamischen Programmierens. Das TD-Learning ist zur Lösung Sequenzieller Entscheidungsprobleme verwendbar [Jan94]. Die Neuronalen Netze, die für diese Arbeit verwendet werden, sind stets mehrschichtige *Feed-Forward-Netze*. Die Grundlagen bzgl. der Verwendung Neuronaler Netze werden vorausgesetzt.

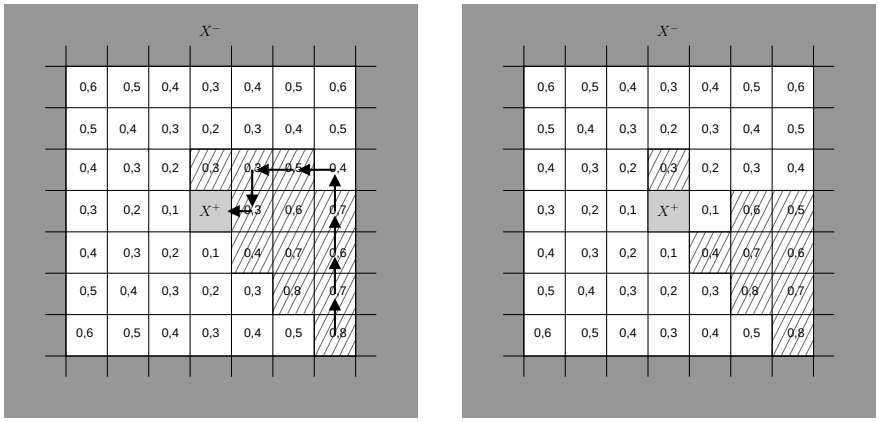

Abbildung 2.3: Update der Tabelle nach Durchlaufen einer weiteren Trajektorie

Sutton unterscheidet die zu treffenden Vorhersagen zwischen *Single Step Predictions* und *Multi Step Predictions*. Single Step Predictions sind Vorhersagen, deren Korrektheit sich nach einem Schritt bewerten lässt. Zu einem Zeitpunkt t kann also auf die Güte der Prognose zum Zeitpunkt $t-1$ geschlossen werden, eine Aufgabe, die sich mit Neuronalen Netzen durch *Überwachtes Lernen* bewerkstelligen lässt. Kommt das System bspw. in einen Endzustand x_T, sind die dort entstehenden Kosten $r(x_T)$ bekannt, die Kostenprognose eines Netzes zum Zeitpunkt $T-1$ könnte also sofort durch Veränderung der Netzgewichte angepasst werden. Multi Step Predictions, wie sie bei Sequenziellen Entscheidungsproblemen vorliegen, sind mit Überwachtem Lernen nicht in den Griff zu bekommen. Hier lässt sich die Qualität der Prognose erst nach mehreren Schritten, am Ende der Sequenz, feststellen. Dabei entsteht zu jedem Zeitpunkt eine Prognose, die wiederum eine Multi Step Prediction ist.

Ziel des TD-Learning ist es, ein Neuronales Netz so zu trainieren, dass es die erwarteten Gesamtkosten zu jedem Zeitpunkt $t < T$ voraussagen kann. Dabei wird nun die Qualität einer Prognose auf Grund der folgenden Prognosen bewertet, nur im letzten Schritt der Sequenz wird mit den – dort tatsächlich ermittelbaren – Kosten verglichen.

Im folgenden ist $P(x)$ die Prognose des Neuronalen Netzes für die zu erwartenden Kosten im Zustand x. Der Fehler, den das Netz bei dieser Voraussage macht, ist $P(x) - V(x)$, die Fehlerfunktion E für das Netz über einer Lernmenge $X' \subseteq X$ ist $E = -\frac{1}{2}\sum_{x \in X'}(P(x) - V(x))^2$. Die Lernmenge ist in unserem Fall eine Sequenz von zeitlich aufeinanderfolgenden Zuständen, so dass gilt:

$$E = -\frac{1}{2}\sum_{t=0}^{T-1}(P(x_t) - V(x_t))^2 \tag{2.4}$$

Die vorzunehmende Änderung Δw_t an den Gewichten w des Netzes zum Zeitpunkt t

ist

$$\Delta w_t = \eta(V(x_t) - P(x_t))\frac{\partial P(x_t)}{\partial w} \tag{2.5}$$

Hierbei ist der Faktor η die *Lernrate* (siehe Kapitel 2.4.2). Aus der Tatsache, dass man zur Bestimmung von V_t die Prognose im Schritt $t+1$ und die beim Übergang dahin entstehenden Kosten heranzieht (vgl. Gleichung 2.3), erhält man

$$\Delta w_t = \eta(r(x_t, a_t) + P(x_{t+1}) - P(x_t)) \sum_{i=1}^{t} \lambda^{t-i} \frac{\partial P(x_t)}{\partial w} \tag{2.6}$$

Der Parameter $\lambda \in \mathbb{R}$ $(0 \leq \lambda \leq 1)$ ist ein eine Gewichtung, die bewirken soll, dass der Einfluss älterer Vorhersagen auf die Gewichtsänderung geringer ist als der neuerer Prognosen, da erwartbar ist, dass die neueren Vorhersagen genauer ausfallen und so stärker gewichtet werden sollen. Er ist namensgebend für die *TD(λ)-Methode*. Insbesondere ist ersichtlich, dass für den Fall $\lambda = 0$ nur der unmittelbar vorherige Schritt in die Formel eingeht, was prinzipiell dem üblichen überwachten Lernen ähnelt[3]. In dieser Arbeit wird stets $\lambda = 0$ verwendet werden, ein Wert, der sich in der Praxis bewährt hat. Eine ausführliche Herleitung dieser Lernmethode findet sich in [Jan94].

2.3.3 Q-Learning

Q-Learning [WD92] ist eine Variante des TD-Lernverfahrens. Es vermeidet einen Nachteil der TD-Methode, nämlich dass zur geeigneten Auswahl einer Aktion im Zustand x_t eine Vorhersage über die jeweiligen Folgezustände von Nöten ist. Um die in Frage kommenden Folgezustände betrachten zu können, muss man sie aber erst einmal kennen, und dazu ist aber ein (zumindest unvollständiges) Systemmodell notwendig. Q-Learning kommt ganz ohne Systemmodell aus.

Beim Q-Learning ist die Aufgabe des Netzes, die Bewertung nicht allein auf Grund des Systemzustands, sondern zusätzlich auch auf Grund der auszuführenden Aktion vorzunehmen. Es lernt also statt der Zustands-Bewertung V eine Zustands-Aktions-Bewertung $Q(x, a)$ mit $x \in X$ und $a \in A$:

$$Q^{\pi}(x, a) = r(x, a) + P^{\pi}(f(x, a)) \tag{2.7}$$

Der einzige Unterschied ist also, dass die Netzeingabe zusätzlich zum Systemzustand noch aus einer möglichen Aktion besteht. Damit ist es möglich, Vorhersagen über die Kosten eines Folgezustandes zu treffen, ohne diesen selbst zu kennen. Für die Auswahl einer geeigneten Aktion zum Zeitpunkt t genügt die Betrachtung von $Q(x_t, a)$ für alle $a \in A$.

[3] Allerdings gilt es hier zu beachten, dass sich durch das zeitliche Fortschreiten die Gewichtsvektoren w ändern und der Fehler extern berechnet werden muss. Außerdem ändern sich die Lernmuster über die Zeit

Alle in dieser Arbeit aufgeführten Lösungsversuche werden mittels Q-Learning durchgeführt werden.

2.4 Der Lösungsalgorithmus

Algorithmus 2 ist der Grundalgorithmus zur Lösung des Problems mit Q-Learning.

Algorithmus 2 Grundalgorithmus Q-Learning

```
    wähle Startstellung x_0
    for t = 0 to T − 1 do
      for all Aktionen a_i do {suche Aktion mit minimaler Kostenprognose}
        berechne Netzausgabe Q(x_t, a_i)
 5:   end for
      wähle a_i mit minimalem Q
      führe Aktion a_i aus {x_{t+1} = f(x_t, a_i)}
      if t > 0 then
        if f(x_t, a_i) ∈ X^+ then
10:       Q := 0
        else if f(x_t, a_i) ∈ X^− then
          Q := Q+penalty
        end if
        berechne Fehler Q − (Q_alt + r(x_t, a_t))
15:     passe Netzgewichte an
      end if
      Q_alt := Q
    end for
```

2.4.1 Exploration

Um zu einer Lösung zu kommen, ist es unbedingt notwendig, dass der Agent einen Zielzustand findet – passiert dies nicht, ist Lernen nicht möglich. Nach Algorithmus 2 richtet sich der Agent bei der Aktionsauswahl aber nur nach der Netzausgabe, und um einen Zustand bewerten zu können, muss das System auch in diesen überführt werden. Je nachdem, wie die Netzprognosen am Anfang ausfallen, kann es sein, dass das System nie in einem Zielzustand landet. Auch ist es wahrscheinlich, dass, wenn bereits eine Strategie gefunden wurde, die zu einem Zielzustand führt, dies oft nur auf einem suboptimalen Weg passiert und der optimale Weg nicht gefunden wird. Es wäre wünschenswert, wenn der Agent also beim Auswählen der Aktion öfters mal bewusst von der Prognose des Netzes abweicht, um den Zustandsraum in der Umgebung der als günstig vorhergesagten Wege zu explorieren und zu prüfen, ob

dort vielleicht günstigere Prognosen zu erwarten sind. Diese *Exploration* wird bspw. dadurch erreicht, dass mit einer gewissen Wahrscheinlichkeit, der *Explorationsrate*, die Aktion nicht auf Grund der Kostenprognose, sondern per Zufall ausgewählt wird. Ohne Exploration ist im Allgemeinen keine Lösung zu erreichen.

Der Lösungsalgorithmus mit Exploration findet sich in Algorithmus 3.

Algorithmus 3 Q-Learning mit Exploration

```
wähle Startstellung x_0
for t = 0 to T do
  for all Aktionen a_i do {suche Aktion mit minimaler Kostenprognose}
    berechne Netzausgabe Q(x_t, a_i)
  end for
  if Explorationsdurchgang then {Zufallswert < Explorationsrate}
    wähle zufällige Aktion a_i
  else
    wähle a_i mit minimalem Q
  end if
  führe Aktion a_i aus {x_{t+1} = f(x_t, a_i)}
  if t > 0 then
    if f(x_t, a_i) ∈ X^- then
      Q := 0
    else if f(x_t, a_i) ∈ X^- then
      Q = Q+penalty
    end if
    if kein Explorationsdurchgang then {kein Lernen bei Exploration}
      berechne Fehler Q − (Q_alt + r(x_t, a_t))
      passe Netzgewichte an
    end if
  end if
  Q_alt := Q
end for
```

2.4.2 Wichtige Parameter

Ein wichtiger Parameter ist die in Gleichung 2.5 erwähnte Lernrate η. Sie spielt bei Erreichen des Lernziels eine große Rolle. Ist sie zu gering gewählt, finden die Gewichtsupdates zu langsam statt und das Lernen verläuft sehr langsam;zudem besteht die Gefahr, dass das Netz in einem lokalen Minimum konvergiert. Ein zu hoher Wert von η dagegen kann dazu führen, dass Minima wieder verlassen oder schlichtweg „übersprungen“ werden und Lernerfolge somit wieder verloren gehen, bzw. sich gar

nicht erst einstellen. Für die Wahl von η gibt es keine allgemeingültigen Weisheiten, ein guter Wert muss für jedes Problem neu gefunden werden.

3 Das Explorationsproblem

3.1 Probleme bei der Exploration

Die in Kapitel 2.4.1 bereits angesprochene Exploration spielt eine zentrale Rolle bei der Lösung Sequenzieller Entscheidungsprobleme mit TD–Methoden. Dieses Kapitel analysiert die Bedeutung der Exploration und die Probleme, die durch diese hervorgerufen werden. Als Ergebnis dieser Beobachtungen wird eine alternative Explorationsmethode vorgeschlagen werden, die diese Probleme vermeiden soll.

3.1.1 Das Finden des Zielzustandes

Ein zielgerichtetes Lernen mit TD-Methoden kann nicht stattfinden, bevor das System in einen Zielzustand $x \in X^+$ überführt worden ist. Bevor das eintritt, wird der Agent mehr oder weniger planlos seine Aktionen wählen und sich dabei auf Prognosen des Netzes verlassen, die noch keinerlei Aussagekraft haben[1]. Im schlimmsten Fall werden die Netzprognosen so ausfallen, dass X^+ gar nicht erreicht wird, dann ist der Lernversuch gescheitert. Hier sind die Explorationsdurchgänge, die ein Abweichen von der noch falschen Strategie erzwingen, essenziell für das Finden eines Zielzustands und somit für das gesamte Lernen.

Wie schnell ein Zielzustand gefunden wird, hängt also auch an der Wahl der Startstellungen X^S, insbesondere ihrer „Entfernung“ zu X^+. Je weiter die Startstellungen von X^+ entfernt liegen, desto später wird ein Zielzustand erreicht werden und desto später wird ein Lernen einsetzen können. Besonders problematisch wird es, wenn Trainingsläufe nur von wenigen oder gar nur von einer einzigen Startstellung aus begonnen werden können. Nehmen wir an, es gäbe nur eine Startposition x_0, X^+ könnte in 5 Schritten erreicht werden und es gäbe jeweils 4 verschiedene Aktionen zur Auswahl: Schon dann läge die Wahrscheinlichkeit, diesen optimalen Weg zu finden (angenommen, es sei der einzige), unter 0,1%.

[1] Bestenfalls wird so das Überführen des Systems in einen Randbereich X^- vermieden, aber auch X^- muss erstmal erreicht werden.

3.1.2 Trainingskosten vs. Lernqualität

Insbesondere fällt der Anteil der Explorationsdurchgänge ins Gewicht, wenn Kosten der Trainingsdurchgänge relevant sind. Bei real existierenden Systemen kann es durchaus sein (vor allem, wenn nicht an einem Simulator gelernt werden kann), dass der bloße Betrieb zu größeren finanziellen Aufwendungen führt. Wenn man es sich nicht leisten kann oder will, viel im laufenden Betrieb zu lernen, wird man nicht mehr explorieren wollen als unbedingt nötig.

Exploration ist nötig, um die relevanten Zustände im Bereich erfolgsversprechender Lösungen zu besuchen und so zu einem Lernerfolg zu kommen, aber wie in Kapitel 2.4.1 umgesetzt, geschieht sie vollkommen ungerichtet und wird deshalb bevorzugt ins Leere laufen. Werden die real anfallenden Trainingskosten zu einem Problem, wird man das vermeiden wollen. Ein Heruntersetzen der Explorationsrate wird allerdings dazu führen, dass dem Agenten weniger Zustände sichtbar werden. Oftmals spielen sich die für den Lösungserfolg wirklich wichtigen Entscheidungen in Zuständen ab, die etwas weiter von den Startstellungen entfernt liegen. Um sie zu finden, sollte optimalerweise der gesamte Zustandsraum auch besucht worden sein – eine Forderung, die praktisch unerreichbar ist und auch unter Kosten-Nutzen-Gesichtspunkten nicht sinnvoll ist. Hier kommt es notwendigerweise zu einem Konflikt zwischen den Bedürfnissen nach niedrigen Trainingskosten und größtmöglicher Exploration.

3.1.3 Verharren in suboptimalen Strategien

Exploration ist auch notwendig, um zu vermeiden, dass sich der Agent auf eine suboptimale Lösung einschießt. Oftmals kann ein Lösungweg durch geringe Abweichungen von der gelernten Linie noch verbessert werden. Dass aber schon eine suboptimale Lösung bekannt ist, kann dies behindern. Dazu ein Beispiel:

In einer Vorarbeit zu dieser Studienarbeit wurde versucht, mit der Hilfe von Q-Learning eine vollkommen selbstlernende Robotersteuerung zu realisieren. Ein rundum mit Sensoren ausgestatteter, sich auf Rollen vorwärts bewegender kleiner Roboter sollte lernen, in einem Labyrinth geradlinig zu fahren und Hindernisse mit möglichst geringer Aufwendung von Motorkraft zu umgehen. Vorkenntnisse waren keine vorhanden, als Zielzustand wurde nur „Keine Hindernisse voraus“ ausgegeben, einfache Bewegungen wie Kurvenfahren sollten selbstständig online erlernt werden, indem sich im Labyrinth vorwärts bewegt wurde. So lernte der Agent auch, dass das Ausführen einer Drehung gegen den Uhrzeigersinn eine gute Strategie war, um einem in Fahrtrichtung auftauchenden Hindernis auszuweichen, und zwar solange, bis die Sensoren kein Hindernis mehr wahrnahmen. Traf der Roboter auf ein Hindernis, führte er eine 90°-Drehung durch und setzte seinen Weg fort, er bog also links ab. Für diese Drehung waren ca. 5 Bewegungsschritte notwendig. Auch beim Erreichen einer Ecke mit Hindernissen links und vorne führte diese Strategie zum Erfolg; bis der Weg wie-

der frei war, wurde eine 180°-Drehung durchgeführt. Es ist leicht ersichtlich, dass dies nicht die optimale Lösung war, denn eine 90°-Drehung in die andere Richtung hätte das Problem mit nur der Hälfte der Bewegungsschritte gelöst. Allein: Die Drehung mit dem Uhrzeigersinn hatte der Agent noch nicht gelernt, und die Wahrscheinlichkeit, es noch zu lernen, war sehr gering. Denn in dem Moment, in dem der Roboter auf ein Hindernis trifft, hat das Drehen nach links, da es langfristig bekanntermaßen zu einem Erfolg führen wird, die bessere Prognose als das Drehen nach rechts, von dem unbekannt ist, wo es endet. Dass links in dem Fall jetzt auch eine Wand sichtbar ist, wird durch die (erwünschte) Generalisierungsfähigkeit des Netzes nicht ins Gewicht fallen. Um die Rechtsdrehung zu erlernen, müsste sie auch mindestens einmal durchgeführt worden sein, damit die langfristige Erfolgsaussicht auch Eingang in das Lernen findet. Dazu muss aber voraussichtlich mehr als ein Explorationsschritt durchgeführt werden: Auch wenn sich nach einem Schritt das System in einem relativ „unbekannten“ Zustand befinden wird, werden aller Voraussicht nach weitere Explorationsschritte von Nöten sein, um wirklich eine Rechtsdrehung zu erlernen, anstatt auf die „bewährte“ Linkskurve zurückzugreifen. Die Wahrscheinlichkeit dafür ist äußerst gering. Aber auch, wenn die bessere Aktionsfolge nur marginal von der laut Prognose besten Aktionsfolge abweicht, ist Exploration notwendig. Es ist erfolgsversprechend, in der Umgebung bereits als gut erkannter Trajektorien nach Optimierungsmöglichkeiten zu suchen.

Die einfache Methode, einen gewissen Anteil der Schritte als Explorationsdurchgang zu wählen, erweist sich also mitunter als problematisch. Erstrebenswert wird eine Explorationsstrategie sein, die einen Mittelweg findet zwischen vollständigem Absuchen und striktem Befolgen der Netzprognose, um sowohl die Prognose- und Generalisierungsfähigkeit Neuronaler Netze auszunutzen als auch den Zustandsraum ausreichend, vor allem in der Nähe guter Lösungen, zu erforschen.

3.2 Speicherung explorierter Trajektorien

Für eine zielgerichtete Explorationsmethode ist es günstig, Wissen über bereits besuchte Zustände zu konservieren. Die in dieser Arbeit behandelten Lernmethoden speichern die Information, welche Zustände als Eingabemuster bereits anlagen, nicht. Das Wissen muss also separat gespeichert werden.

3.2.1 Die Datenstruktur

Auf Grund der im Kapitel über Q-Learning (Kapitel 2.3.3) beschriebenen Tatsache, dass bei den meisten Problemen kein oder nur ein unvollständiges Systemmodell vorliegt, fällt eine einfache Tabelle als Datenstruktur zur Speicherung des Explorationwissens aus: Da zum dem Zeitpunkt, an dem die Entscheidung getroffen werden muss, nichts über den Folgezustand bekannt ist, nützt es es für die Aktionsauswahl bei der

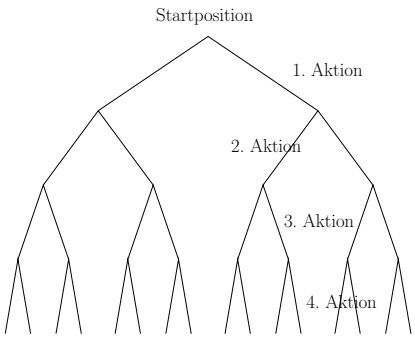

Abbildung 3.1: Repräsentation des Entscheidungsproblems als Baum

Exploration wenig, Informationen darüber zu haben, an welchen Folgezuständen man bereits war. Auch eine ans Q-Learning angelehnte Speicherung von Zustand und Aktion hätte Nachteile: Über den Weg, auf dem dieser Zustand erreicht wurde, würde genauso wenig ausgesagt wie über den Zeitschritt, in dem sich das System gerade befindet.

Betrachtet man den Algorithmus zur Lösung des Problems, so lässt er sich als Entscheidungsbaum darstellen (siehe Abbildung 3.1): Von jedem Startzustand aus gibt es eine Anzahl von möglichen Aktionen zur Auswahl und in jedem Folgezustand wird dieser Entscheidungsprozess wiederholt. Es böte sich also an, das Explorationswissen in Bäumen zu speichern, in denen von der Wurzel ausgehend jede mögliche Entscheidung nachvollzogen werden kann. Das explizite Speichern des Systemzustands ist dabei nicht nötig, da nur Aktionsfolgen, also Trajektorien im Zustandsraum, gespeichert werden.

Für diese Arbeit wurde folgende Datenstruktur verwendet: Für jeden Startzustand $x \in X^S$ gibt es einen Baum $B_x = (E, K)$. Dabei ist E die Menge der Knoten und K die Menge der Kanten. Jede Kante $k \in K$ steht dabei für einen Zustandsübergang, also eine Aktion. Ein Knoten $e \in E$ steht für den Zustand des Systems, den es nach Ausführen der durch die zu ihm führenden Kanten repräsentierten Aktionenfolge einnimmt; dieser Zustand ist natürlich nicht eindeutig bestimmt. Jeder Knoten e hat genau so viele Söhne, wie Aktionen in den Zuständen, die e symbolisieren kann, möglich sind. Jede Kante $k_i(e)$ (die i-te Kante, die von e ausgeht) beinhaltet

- die Aktion $a_i \in A$
- ein boolsches Visit-Flag, ob die Kante schon besucht wurde

Eine Repräsentation des jeweiligen Systemzustands kann dabei im Knoten e abgelegt werden[2]. Die maximale Baumtiefe entspricht dem Horizont T.

[2]Zum Speichern der Trajektorien wäre das nicht nötig, die Gründe hierfür werden erst in Kapitel 4.2 aufgeführt.

Während des Trainings legt man nun simultan diese Datenstruktur an; für jeden Startzustand $x \in X^S$ wird ein eigener Baum B_x erzeugt. Nach jeder Aktion wird ggf. eine neue Kante und ein neuer Knoten hinzugefügt. So ist in einem Zustand $x_t \in X$ stets überprüfbar, ob ausgehend von einem Startzustand x_0 zu einem Zeitpunkt t nach Durchführen der Aktionen $a_0 \dots a_{t-1}$ eine Aktion $a \in A$ bereits einmal ausgeführt wurde oder nicht. Es ist dagegen *nicht* feststellbar, ob der Zustand x_t bereits zuvor im Laufe des Trainings schon besucht erreicht worden ist, da dies von einem anderen Startzustand oder auf einem anderen Weg passiert sein konnte. Die Speicherung in Bäumen speichert Information über Aktionsfolgen, nicht über Systemzustände.

3.2.1.1 Schnittstelle der Datenstruktur

Für die spätere Verwendung in den Algorithmen gelten für die Baum-Datenstruktur folgende Bezeichnungen:

- e ist der jeweils aktuelle Knoten
- $k_i(e)$ ist die i-te Kante von Knoten e
- $relevant(k)$ bezeichnet, ob Kante k „relevant“ ist
- state(e) liefert den in Knoten e gespeicherten Zustand
- action(k) liefert die zu Kante k gehörende Aktion

3.3 Exploration durch Phasenweises Fortschreiten

3.3.1 Der Algorithmus

Mit Hilfe der in den Bäumen gespeicherten Informationen kann nun ein Verfahren durchgeführt werden, das die Kapitel 3.1 aufgeführten Probleme bei der Exploration umgehen soll. Dabei soll ein Trainingsdurchgang in drei Phasen unterteilt werden:

Greedy-Phase In dieser ersten Phase werden die Aktionen streng nach der Kostenprognose ausgewählt, zu jedem Zeitpunkt t wird die Aktion a_i mit der minimalen Kostenprognose $\min_i Q(x_t, a_i)$ ausgeführt. Exploration findet in dieser Phase überhaupt nicht statt.

Verteilphase Diese Phase folgt auf die Greedy-Phase. Sie dauert nur einen einzigen Zeitschritt t lang. Hier werden sukzessive alle möglichen Aktionen a_i ausgeführt. Nach jedem dieser Schritte wird in die Explorationsphase übergegangen und das System nach deren Ende auf den Ausgangszustand x_t zurückgesetzt. In dieser Phase wird also das gesamte unmittelbare Umfeld um den Zustand x_t abgesucht.

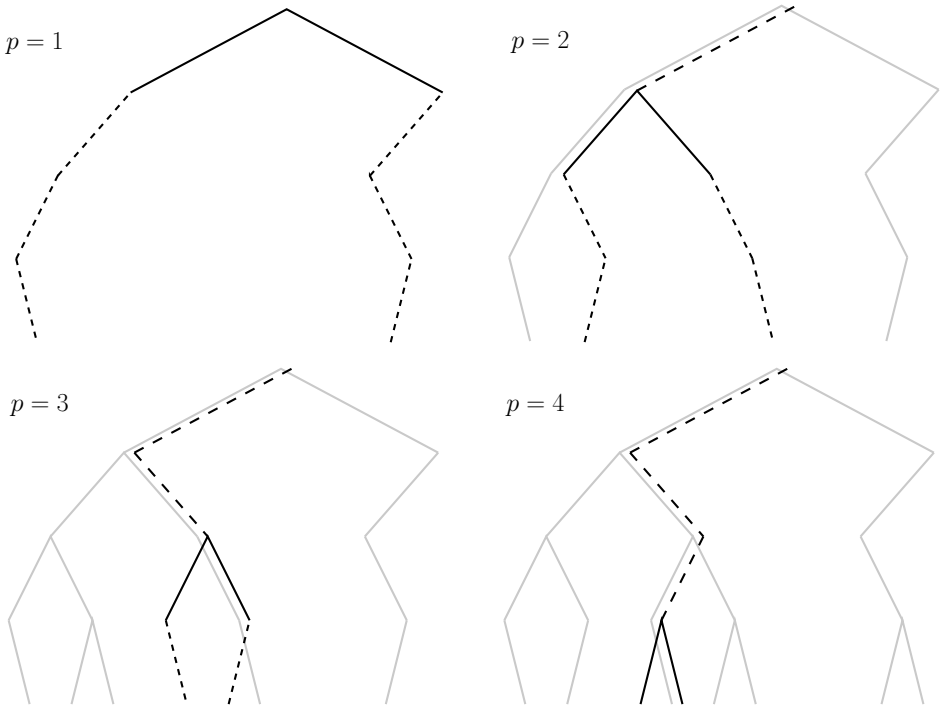

Abbildung 3.2: Phasenweise Exploration – Aufbau des Baums. Der bereits bekannte Baum ist jeweils hellgrau eingezeichnet, die Verteilphase ist mit durchgezogener, die Greedy-Phase mit lang gestrichelter und Explorationsphase mit kurz gestrichelter Linie markiert.

Explorationsphase Hier werden auf Basis der Trajektorien-Bäume bevorzugt Aktionen ausgewählt, die eine bislang noch nicht ausgesuchte Trajektorie ergeben, um eine weitestmögliche Exploration des Zustandsraums zu gewährleisten; im Baum B_x wird also erstmal an Kanten entlang geschritten, die noch nicht besucht sind. Sind mehrere Aktionen möglich, wird die Aktion mit der besten Kostenprognose gewählt, sind bereits alle Kanten besucht, wird die Auswahl nach einem beliebigen, am besten zufälligen, Verfahren getroffen. Die Explorationsphase endet, wenn ein Finalzustand ($x \in X^+ \vee x \in X^- \vee t = T$) erreicht ist.

Um das Gewicht von Explorationsphase und Greedy-Phase auszubalancieren, wird ein Phasenzähler p eingeführt. Für $t < p$ gilt die Greedy-Phase, für $t = p$ die Verteilphase und für $t > p$ die Explorationsphase. Für jeden Startzustand x_0 durchläuft p die Werte von 1 bis T, in jedem dieser Durchgänge wird die Verteilphase also stetig nach hinten verschoben. Da sie immer an bereits als gut prognostizierte Trajektorien

anschließt, ist sichergestellt, dass die Exploration zielgerichtet stattfindet, für kleinere Werte von p und somit einer längeren Explorationsphase wird ungerichteter und tiefer exploriert. Für $p = 1$ entfällt offensichtlich die Greedy-Phase, für $p = T$ die Explorationsphase. Abbildung 3.2 veranschaulicht den Aufbau eines Baumes für ein Problem mit 2 Aktionen und einem Horizont von $T = 4$. Algorithmus 4 beschreibt die Exploration durch phasenweises Fortschreiten. Ein Lernvorgang ist aus Übersichtsgründen in der Beschreibung dieses Algorithmus' nicht enthalten.

Algorithmus 4 Phasenweise Exploration

```
wähle Startposition x_0
wähle Baum B_{x_0} und gehe zur Wurzel
for p = 1 to T do
  for t = 1 to p do
    if t < p then {Greedy-Phase}
      wähle Aktion a_i mit minimalem Q(x_t, a_i)
      führe Aktion a_i aus
      markiere k_i als besucht und gehe zu Sohn i
    else if t = p then {Verteilphase}
      speichere Zustand x_p
      for all Aktionen a_i do
        setze System auf Zustand x_p zurück
        führe Aktion a_i aus
        markiere Kante k_i als besucht und gehe zu Sohn i
        for t = p + 1 to T do {Explorationsphase}
          if alle Kanten k bereits besucht then
            wähle zufällige Aktion a_i
          else
            wähle Aktion a_i mit minimalem Q(x_t, a_i)
          end if
          führe Aktion a_i aus
          markiere Kante k_i als besucht und gehe zu Sohn i
        end for
      end for
    end if
  end for
end for
```

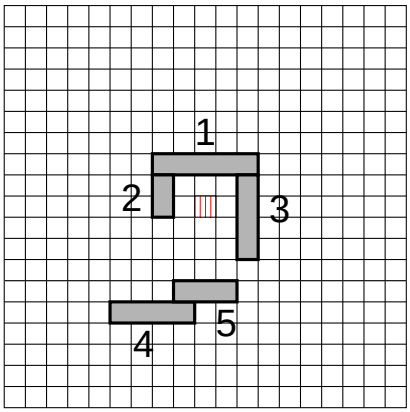

Abbildung 3.3: Testszene zum Vergleich der Explorationsstrategien

Bei Phasenweiser Exploration werden pro Startstellung statt T Schritten abhängig von der Anzahl der möglichen Aktionen a

$$\sum_{p=1}^{T} p - 1 + (a(T - p + 1)) = \frac{1}{2}T((1 + a)T + a - 1) \tag{3.1}$$

Schritte ausgeführt[3], das entspricht $O(T^2 \cdot a)$. Im Fall von 4 Aktionen bei einem Horizont von $T = 20$ würden also von jeder Startstellung statt 20 jeweils 1030 Aktionen ausgeführt.

3.4 Ergebnisse

3.4.1 Vergleich mit herkömmlichen Explorationsstrategien

3.4.1.1 Der Benchmark

Zum Vergleich der Explorationsstrategien wurde ein simples Beispiel gewählt. Das Problem ist eine Abwandlung des in Kapitel 2.2.2.1 vorgestellten Zielfindens im zweidimensionalen, diskreten Zustandsraum. Der erlaubte Bereich ist hier größer, zusätzlich sind noch einige Hindernisse hinzugefügt worden (siehe Abbildung 3.3). Es gilt also

$$X^- = \mathbb{Z}^2 \setminus (\{[-9, 9] \times [-9, 9]\} \cup R_i) \tag{3.2}$$

[3]Wenn, wie üblich, in absorbierenden Zuständen abgebrochen wird, ist das natürlich ein Maximalwert.

Hind.	Zufällige Exploration Kosten	X^-	Phasenw. Exploration Kosten	X^-
0	23544	5773	4661	4283
1	23160	14063	3900	3171
2	11579	10658	3546	5651
3	58500	40806	3331	5132
4	(95%)	–	5494	9952
5	(97,5%)	–	19459	12917

Tabelle 3.1: Gegenüberstellung der Ergebnisse der Explorationsmethoden. Aufgelistet sind jeweils Trainingsgesamtkosten und Anzahl der Läufe, die im Training X^- erreichten.

wobei R_i die Menge der durch Hindernisse „blockierten“ Zustände ist (für nur Hindernis 1 bspw. gilt $R_i = \{[-2, 2] \times \{2\}\}$. Als Horizont wurde $T = 20$ gewählt, die restlichen Größen entsprechen denen in Kapitel 2.2.2.1. Für die Gegenüberstellung wurde das Problem zunächst ohne, dann mit jeweils einem weiteren Hindernis gelöst. Zur Repräsentation der Bewertungsfunktion wurde, wie in Kapitel 2.3.1.1 beschrieben, eine Tabelle gewählt, um die Ergebnisse der Exploration nicht durch Seiteneffekte neuronaler Lösungsstrategien zu verwässern.

3.4.1.2 Gegenüberstellung

Tabelle 3.1 stellt die Ergebnisse von Lösungsstrategien mit zufälliger Exploration (wie in Kapitel 2.4.1 beschrieben) und Phasenweiser Exploration gegenüber. In der ersten Spalte ist die Anzahl der Hindernisse aufgetragen (in der Reihenfolge der Nummerierung von Abbildung 3.3). Die aufgeführten Kosten sind jeweils die während des Trainings angefallenen reinen Bewegungskosten, also ohne Penaltys, dahinter ist jeweils die Anzahl der Durchgänge angegeben, die in den Trainingsläufen im Randbereich X^- endeten. Die angegeben Werte sind diejenigen zu dem Zeitpunkt, ab dem ein stabiler Lernerfolg erreicht wurde, also alle Läufe mit Startpositionen aus einer Kreuzvalidierungsmenge im Test X^+ erreichten. Für die Fälle mit mehr als drei Hindernissen konnte dies bei zufälliger Exploration nicht hundertprozentig erreicht werden, der Anteil der erfolgreichen Läufe ist in der Tabelle in Klammern angegeben.

Bei zufälliger Exploration konnte kein stabiles Lernergebnis erzielt werden, schon ab vier Hindernissen war im Gegensatz zur Phasenweisen Exploration in absehbarer Zeit kein Erfolg von 100% zu erreichen. Und auch in den Durchgängen, in denen gelernt wurde, waren beim herkömmlichen Verfahren die Kosten stets mindestens dreimal so hoch. Entsprechend signifikant fallen die Vorteile des phasenweisen Explorierens auch bei Betrachtung der X^--Durchgänge aus: Sieht man von dem Fall ohne Hindernisse

ab, bei dem ja naturgemäß wenig Trajektorien in den Randbereichen enden, kamen hier zwischen zwei- und achtmal weniger X^--Zustände vor, was sich vor allem bei Systemen lohnen wird, bei denen ein Erreichen der Randbereiche hohe reale Kosten zur Folge haben würde.

Nicht zuletzt wird durch diese Art der Exploration ein Parameter, die Explorationsrate, eingespart. Bei der Ermittlung der Tabellenwerte mussten etliche Möglichkeiten durchprobiert werden; für den Fall ohne Hindernisse lieferte ein Wert von $0,3$, für die andern von $0,2$ das beste Ergebnis. Die Explorationsrate kann zumeist nur durch Ausprobieren ermittelt werden, ein Wegfall dieses Parameters bedeutet also eine nicht unerhebliche Ersparnis an Arbeit und Trainingskosten.

3.4.2 Nicht-rücksetzbare Systeme

Um die Phasenweise Exploration für Trainingsläufe anwenden zu können, ist es notwendig, dass das System nach Erreichen eines Finalzustands x_T in den Zustand x_p zurückgesetzt werden kann, in dem die Verteilphase ausgeführt wurde. Dies ist aber nicht immer möglich, zumindest nicht in vertretbarem Aufwand. Und auch, wenn komplementäre Aktionen ausführbar sind, ist in realen Systemen auf Grund von Störeinflüssen nicht zu garantieren, dass dieser Zustand dann mit dem Zustand x_p übereinstimmt; das Lernen würde so verfälscht. In diesem Fall muss man die Verteilphase weglassen und direkt in die Explorationsphase übergehen. Dann wird, wie üblich, einer ununterbrochene Trajektorie $x_0, x_1, \ldots, x_T$ gefolgt, ein Rücksetzen ist nicht mehr nötig. Es könnte durch Speicherung der ausgewählten Aktion sichergestellt werden, dass das Verhalten der Verteilphase bei nochmaligem Erreichen dieser Position im Baum simuliert wird, indem dann zunächst eine andere Aktion ausgewählt wird. Auch wenn man diese Simulation der Verteilphase nicht implementiert, wird die Phasenweise Exploration bei einem Hindernis die Lösung mit Kosten von 13400 (statt 3900 mit Verteilphase) finden, was immer noch deutlich unter den Kosten von 23160 der Zufälligen Exploration liegt.

3.5 Speicherverbrauch der Verfahrens

Größtes Problem dieses Verfahrens ist der exponentielle Speicherverbrauch durch die Speicherung der Trajektorien in Bäumen. Bereits für die Lösung des simplen Problems aus dem letzten Kapitel musste ein Speicheraufwand von 80 bis 120 Megabyte hingenommen werden. Damit scheidet die Phasenweise Exploration, wie sie in dieser Arbeit verwendet wurde, für die praktische Anwendung aus. Um das zu vermeiden, müssen zusätzlich folgende Maßnahmen implementiert werden:

Purging komplett besuchter Teilbäume Vollständig besuchte Teilbäume können abgeschnitten werden, wenn sie nicht in relevante Bereiche führen. Dies spart in

der Folgezeit auch Kosten, da in diese Bereiche nie mehr vorgedrungen werden braucht. Auch vollständig relevante Bereiche können so behandelt werden.

Purging offensichtlich erfolgloser Bereiche Auch nicht komplett besuchte Teilbäume können abgeschnitten werden, wenn der Füllgrad entsprechend hoch und die Wahrscheinlichkeit gering ist, darin noch relevante Trajektorien zu finden. Dabei muss natürlich eine gute Bewertungsmöglichkeit für die Bewertung der Erfolgsaussichten gefunden werden. Ein Indiz mag bspw. sein, dass auch in benachbarten Teilbäumen keine relevanten Zustände erreicht wurden.

Reichen diese Maßnahmen noch nicht aus, sind zusätzlich auch die folgenden Modifikationen denkbar:

Verkürzen der letzten Phasen Am Ende der Trajektorie könnte man darauf verzichten, die letzten Entscheidungen überhaupt zu speichern. Der Wissensverlust dürfte gering sein.

Vergessen des gespeicherten Wissens Nach guten Lernerfolgen zu Beginn des Lernens wäre es möglich, nacheinander die Bäume für einzelne Startstellungen zu löschen. Aufgrund des akquirierten Wissens in der Bewertungsfunktion wird der Agent zielgerichteter agieren und die Bäume werden nicht so voll werden.

Rückgriff auf Tabellen Die vorher genannten Maßnahmen werden den exponentiellen Speicherverbrauch nicht verhindern können. Deshalb könnte man trotz des unvollständigen Wissen über das System und des oft kontinuierlichen Zustandsraums von der Speicherung von Trajektorien abgehen und doch Zustands-Aktions-Paare in einer Tabelle speichern. Diese grobe Repräsentation des Wissens über bereits explorierte Gebiete dürfte gegenüber der zufälligen Exploration immer noch Vorteile mit sich bringen.

4 Das Vergesslichkeitsproblem

4.1 Vergessen guter Lösungen

Beim Lösen von Problemen mit Hilfe von TD(λ)-Learning kommt es oft vor, dass während des Trainingsvorgangs relativ schnell für gewissen Startpositionen eine gute Strategie gefunden wird, diese aber beim Fortschreiten der Lernens wieder verloren geht. Über dem weiteren Lernen werden die guten und erfolgreichen Trajektorien, die der Agent im Zustandsraum zieht, geradezu wieder „vergessen“, wenn diese Zustände länger nicht besucht werden und somit auch nicht in den Update der Netzgewichte eingehen.

Dieses Problem trat auch in der auf Seite 19 erwähnten Vorarbeit der Robotersteuerung auf. Wenn der Roboter in freier Umgebung gestartet wurde, konnte er in relativ kurzer Zeit das Geradeaus-Fahren erlernen. Beim ersten Auftreffen auf ein Hindernis konnte auch hier, allerdings erst nach geraumer Zeit und etlichen Versuchen, ein Vermeiden erlernt werden; Geradeausfahren konnte der Roboter aber nach Umfahren des Hindernisses nicht mehr – dieser Teil der Strategie war vergessen, da das System sehr lange nicht mehr in die zum Geradeausfahren gehörenden Zustände überführt worden war.

Auch wenn es nicht, wie in diesem Fall, zu einem kompletten Vergessen guter Teillösungen kommt, ist doch oft zu beobachten, dass das Netz für zumindest einige Startstellungen gegen eine suboptimale Strategie konvergiert, obwohl die optimale Lösung bereits in früheren Trainingsdurchgängen gefunden und auch gelernt worden war. Oder es tritt das in Kapitel 3.1.3 geschilderte Szenario auf, dass die optimale Aktionsfolge aufgrund der im Vorfeld gelernten Strategie einfach zu selten vorkommt, als dass sie nennenswert Niederschlag in den Gewichtsänderungen des Netzes finden würde. Es wäre also wünschenswert, gute Trajektorien zu konservieren, um im weiteren Verlauf des Trainings wieder auf sie zurückgreifen zu können.

4.2 Offline-Lernen

Mit den in Kapitel 3.2.1 vorgestellten Bäumen ist eine Datenstruktur zur Speicherung von Trajektorien bereits vorhanden, Algorithmus 4 nimmt eine solche Speicherung auch vor. Um aber nicht nur zwischen schon besuchten und noch unbekannten

Trajektorien unterscheiden zu können, muss man zusätzlich in dieser Datenstruktur an jeden Ast ein boolsches „Relevanz-Flag“ hinzufügen, um „gute“, in unserem Sinne also für das erfolgreiche Lernen relevante Trajektorien zu markieren. Dazu muss nach Abschluss eines Trainingslaufes entschieden werden, ob dieser Lauf relevant ist; üblicherweise ist er das, wenn er in einem Finalzustand (in X^+ oder X^-) endet. Ist eine Trajektorie in der Szene als relevant erkannt, wird im Baum wieder bis zur Wurzel nach oben gewandert und die einzelnen Kanten werden als relevant markiert.

Die Information über gute Trajektorien kann dazu genutzt werden, das Netz zu trainieren, ohne dass das System reale Läufe durchführen muss; das Lernen findet also „offline“ statt, indem man in den Bäumen B_x die relevanten Aktionsfolgen betrachtet. Nach Abschluss einer Anzahl von Trainingsläufen können die dort gespeicherten Zustands-Aktions-Paare dem Netz beliebig oft eingelernt werden, ohne dass dies reale Kosten durch den Lauf des Systems zur Folge hätte. So wird verhindert, dass im Laufe der Zeit wichtige Trajektorien „vergessen“ werden.

Wenn das Lernen wirklich offline, also ohne Anfallen von realen Kosten erfolgen soll, ist es, wie in Kapitel 3.2.1 schon kurz erwähnt, auch notwendig, den Systemzustand x_t in den entsprechenden Knoten des Baumes zu speichern, um die Auswirkungen einer Aktion auch betrachten zu können, ohne dass das System läuft.

4.2.1 Offline-Q-Learning

4.2.1.1 Der Algorithmus

Dieses Verfahren funktioniert im Grunde wie das normale Q-Learning (siehe Kapitel 2.3.3), nur mit dem Unterschied, dass die Lernschritte des Neuronalen Netzes nicht im Anschluss an jede Aktion, sondern „offline“ nach Beendigung der realen Aktionen des Agenten mit den Daten aus allen gespeicherten Trajektorien durchgeführt werden. Natürlich könnte aber auch wie beim normalen Q-Learning das Netz bereits während der realen Läufe trainiert werden; angesichts des im Vergleich zu einem realen Lauf des Systems geringen Aufwands des Offline-Lernens wird dies aber für das Lernergebnis nicht nennenswert ins Gewicht fallen.

Algorithmus 5 Markierung relevanter Trajektorien

Require: $t = T$, aktueller Knoten ist ein Blatt
  if $x_t \in X^+ \vee x_t \in X^-$ **then** {oder eine andere Relevanzbedingung}
    while aktueller Knoten ist nicht die Wurzel **do**
      markiere Kante zum Vaterknoten als relevant
      gehe zum Vaterknoten
    end while
  end if

Die Trajektorien (Zustandsfolgen und ausgeführte Aktionen) werden wie in Ka-

pitel 3.2.1 beschrieben gespeichert. Zunächst wird eine gewisse Anzahl von realen Trainingsläufen mit Phasenweiser Exploration nach Algorithmus 4 absolviert, wobei die Gesamtkosten unberücksichtigt bleiben; betrachtet wird nur, ob die Trajektorie in X^+ oder X^- endet, dann wird sie als relevant markiert (Algorithmus 5). Zur der Frage, was als relevant anzusehen ist und was nicht, und welche Folgen dies auf das Lernverhalten hat, nimmt Abschnitt 4.3.3 Stellung. Nach einer gewissen Anzahl von realen Trainingsläufen werden dann die relevanten Trajektorien offline eingelernt (Algorithmen 6 und 7). Dazu wird jeder Baum per Tiefensuche durchlaufen und die als relevant markierten Zustandsübergänge gemäß dem „normalen“ TD-Verfahren durch das Netz propagiert. Das Netz wird hier nur mit den bereits als zielführend erkannten Zustandsfolgen konfrontiert, ein Lernen von Trajektorien, die keinen Erfolg bringen und so den Lernvorgang verzögern können, wird komplett vermieden.

Algorithmus 6 Offline-Q-Learning

```
    wähle Startstellung x_0
    betrachte Baum B_{x_0}
    gehe zur Wurzel von B_{x_0}
    for all Kanten k_i do {Tiefensuche}
 5:   if relevant(k_i) then
        berechne Netzausgabe Q = Q(state(e),aktion(k_i))
        gehe zu Sohn i
        go_down_and_learn(Q,r(state(e), aktion(k_i))) {Algorithmus 7}
      end if
10: end for
```

4.2.2 Offline oder nicht?

Im vorgestellten Verfahren wird während der Offline-Lernphase für die Netzeingabe der im Baum gespeicherte Zustandswert state(e) verwendet. Dies kann unter Umständen unerwünschte Effekte hervorrufen. Der Zustandsvektor wird beim ersten Durchlaufen der Trajektorie im Baumknoten gespeichert. Wenn dieser Zustand unter Einfluss eines Störfaktors erreicht wurde, kann der gespeicherte Zustand „falsch“ sein, sprich, mit dem realen Systemverhalten nicht übereinstimmen. Die „falsche“ Zustandsfolge wird in den Offline-Läufen immer wieder eingelernt, was das Lernverhalten extrem beeinträchtigen kann. Im Extremfall können so auch inkonsistente Lernmuster entstehen, und das Netz konvergiert nicht.

Anstatt state(e) zu verwenden, könnte man auch das System mitlaufen lassen und den jeweils real ablesbaren Zustand durchpropagieren. So würde ein größerer Störeinfluss nicht mitgeschleppt werden, das eine unkorrekte Lernmuster wird nicht ins Gewicht fallen. Bei diesem Vorgehen verschwindet dann allerdings der Vorteil, dass

Algorithmus 7 Procedure go_down_and_learn(Q_{alt}, $costs_{alt}$) {Teil von Algorithmus 6}

```
    for all Aktionen a_i do {Beste Prognose suchen}
      berechne Netzausgabe Q = Q(state(e),a_i)
    end for
    wähle a_i und Q mit minimalem Q
 5: if state(e) ∈ X^+ then
      Q := 0
    else if state(e) ∈ X^- then
      Q := Q+penalty
    end if
10: berechne Fehler Q − (Q_alt + costs_alt)
    passe Netzgewichte an
    if aktueller Knoten e ist kein Blatt then
      for all Kanten k_i do {Tiefensuche}
        if relevant(k_i) then
15:       berechne Netzausgabe Q = Q(state(e),aktion(k_i))
          gehe zu Sohn i
          go_down_and_learn(Q, r(state(e),aktion(k_i)))
        end if
      end for
20: end if
```

beim Offline-Lernen keine realen Kosten anfallen. Ist mit größeren Störeinflüssen zu rechnen, wird das unter Umständen aber nötig sein. Eine weitere, relativ kostengünstige Alternative wäre, relevante Aktionsfolgen mehrere Male zu wiederholen, um Ausreißer aussortieren zu können.

4.3 Ergebnisse

4.3.1 Implementierung

Für die Implementierung der Neuronalen Netze wurde das Neuro-Simulator-Paket *n++* [Rie95] verwendet. Verwendet wurden stets dreischichtige Feed-Forward-Netze mit zwischen 10 und 30 Neuronen in der versteckten Schicht. Getestet wurde an der in Kapitel 3.4.1.1 beschriebenen Szene. Gemessen wurde der Lernerfolg an einer von der für das Training verwendeten Startzustandsmenge unabhängigen Testmenge.

4.3.2 Ausbleiben des Lernerfolgs - Ursachen und Lösungsversuche

4.3.2.1 Konvergenz der Netzgewichte

Mit Offline-Q-Learning konnte in der in dieser Arbeit beschriebenen Form kein befriedigendes Ergebnis erzielt werden. Schon bei leichtesten Problemen mit einem einzigen Hindernis kam kein stabiler Lernerfolg mehr zustande.

Im Allgemeinen kam überhaupt kein Lernen zustande, abhängig von der gewählten Lernrate η konvergierten die Netzausgabewerte mehr oder weniger schnell alle gegen 1. Der Grund dafür ist darin zu finden, dass jeweils nach jedem Lauf *alle* relevanten Trajektorien eingelernt wurden. Insbesondere die Vielzahl der Aktionsfolgen, die in X^- endeten und somit einen Zielwert von 1 hatten (als Penalty wurde die maximale Netzausgabe gewählt) beschleunigte die Konvergenz: Das Problem ohne Hindernisse (und somit wenig X^--Gebieten) konnte vom Netz gelernt werden, aber bereits ein Hindernis führte zu keinem Erfolg mehr. Das Konvergenzproblem konnte nicht durch Einstellungen der Parameter in den Griff bekommen werden. Auch einfache Lösungsansätze scheiterten: Sowohl ein selteneres Aufrufen der Lernroutine als auch zwischenzeitliche Neuinitialisierungen des Neuronalen Netzes führten ebenfalls zu der beobachteten Konvergenz der Netzausgabewerte.

4.3.3 Bewertung relevanter Trajektorien und Behandlung der Randbereiche

Der Misserfolg des Lernens führte also auf die Frage zurück, welche Trajektorien relevant sind, also in den Lernvorgang eingehen sollen. Läufe, die weder in X^+ noch in X^- enden, sind nie relevant, Läufe die in X^+ enden immer[1]. Interessant ist war nun, ob und wie die X^--Durchgänge eingelernt werden.

Um eine bessere Gewichtung von X^+- und X^--Läufen zu erreichen, wurden die Trajektorien, die in einem Randbereich endeten, nicht als relevant markiert und stattdessen nur das Zustands-Aktions-Paar in einer separaten Liste gespeichert, das das System nach X^- überführt hatte. Die Lernmuster aus dieser Liste wurden dann zusätzlich zu den relevanten Aktionen in den Offline-Lerndurchgängen durchs Netz propagiert. Ziel dieser Maßnahme war es, die hohen Zielwerte nur für genau die Eingabewerte zu erreichen, die das System auch in einen Randbereich gebracht hatten. Dies war aber auch gleichzeitig der Schwachpunkt dieser Variante: Trajektorien, die direkt nach X^- führten, wurden nicht mehr als solche eingelernt. Das Konvergenz-Problem konnte so nicht behoben werden, es veränderte sich nur dahingehend, dass jetzt die Ausgaben für einen Teil des Eingaberaums gegen 0, die andern gegen 1 konvergierten.

[1] Natürlich bräuchte man nicht alle erfolgreichen Läufe einlernen, allerdings würde eine Auswahl unter diesen Trajektorien eine Betrachtung unter weiteren qualitativen Merkmalen erfordern, die im Allgemeinen zum Lernzeitpunkt nicht bekannt sind.

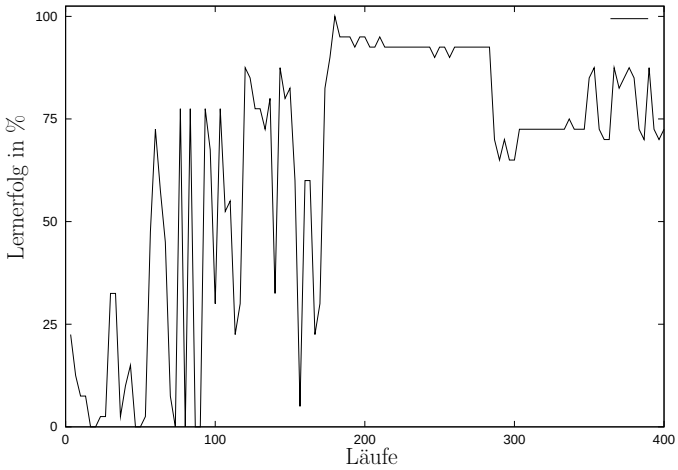

Abbildung 4.1: Typischer Lernverlauf beim Offline-Q-Learning mit begrenztem Einlernen von X^--Trajektorien

Vielversprechender erschien zunächst der Versuch, die Anzahl der zu lernenden X^--Trajektorien dadurch zu begrenzen, dass sie nach einer gewissen Anzahl von Lerndurchgängen nicht mehr eingelernt wurden, indem die Relevanz-Flags in den Bäumen gelöscht wurden. Ein typischer Lernverlauf ist in Abbildung 4.1 dargestellt: Durch das „Vergessen“ (hier wurde jeder X^+-Durchgang fünfmal eingelernt) entsteht der unstete Lernvorgang zu Beginn, das Ziel wird nur kurzzeitig erreicht und das Lernen pendelt sich zwischen 90 und 95% ein. Bei den 5–10% der Startpositionen, von denen aus das Ziel in den Testläufen nicht erreicht wurde, scheiterte der Agent jeweils an einem Hindernis – die Information, dass dort ein Übergang nach X^- stattfindet, wurde irgendwann nicht mehr relevant, insofern auch nicht mehr eingelernt und verschwand so aus dem Gedächtnis des Netzes. Die Generalisierungsfähigkeit überwog und die Hindernisse wurden nicht beachtet. Die „unsaubere“ Lösung des Vergessens von X^--Trajektorien bleibt auch im Ergebnis bestenfalls Flickwerk, das auch noch einen weiteren freien Parameter zur Folge hat: die Zeitspanne, zu dem das „Vergessen“ einsetzt.

Ein weniger unsteter Lernverlauf zu Beginn des Trainings wurde erreicht, wenn statt Backpropagation RPROP [RB93] verwendet wurde. Allerdings ist RPROP aufgrund der sich während des Einlernens ändernden Lernmuster eigentlich gar nicht für diese Aufgabenstellung geeignet, es konnte niemals auch nur temporär ein Lernerfolg von 100% erreicht werden. Abgesehen von dem ruhigeren Verlauf und marginal schnelleren Anfangserfolgen unterschieden sich die Lernverläufe kaum von denen mit Backpropagation. Abbildung 4.2 stellt die Lernverläufe eines eines Offline-Lernens mit RPROP und „Vergessen“ der X^--Durchgänge einem Lernen mit herkömmlichen

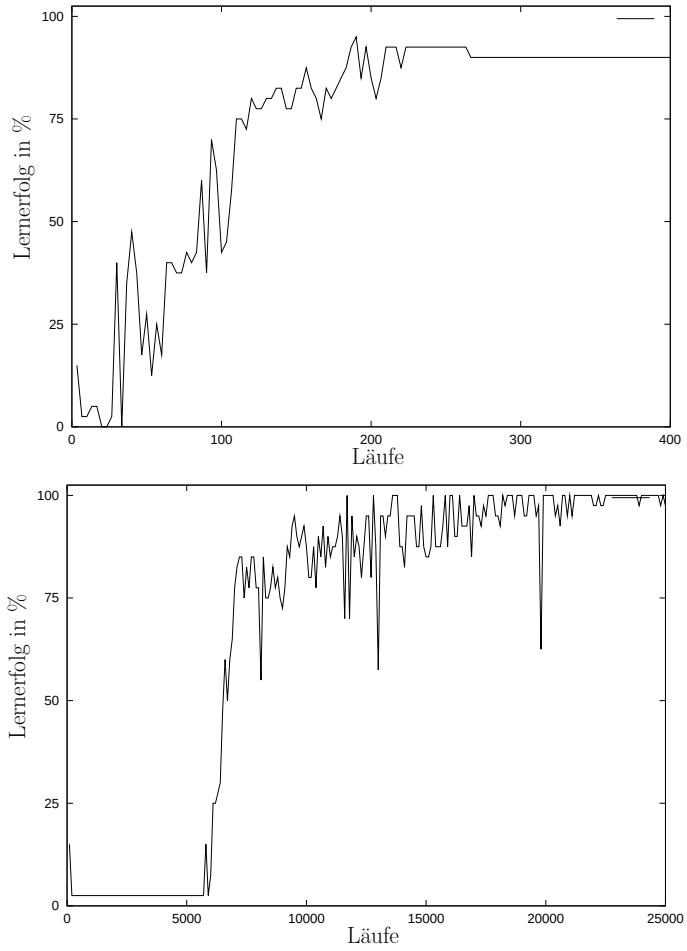

Abbildung 4.2: Vergleich der Lernverläufe von Offline-Q-Learning (oben) und TD(λ)-Lernen (unten)

TD(λ)-Lernen gegenüber[2]. Zu erkennen ist, dass das Offline-Lernen zwar schneller zu einem teilweisen Lernen kommt, aber nie zu einer vollständigen stabilen Problemlösung.

[2]Auf der X-Achse ist dabei jeweils die Anzahl der Trainingsläufe aufgetragen. Da für das Offline-Lernen die Phasenweise Exploration verwendet wurde, wurden zum besseren Vergleich die Diagramme so skaliert, dass an entsprechenden Punkten ungefähr gleiche Trainingskosten entstanden waren. Dazu wurde die Abschätzung der Laufanzahl gemäß Gleichung 3.1, Seite 25, vorgenommen.

4.4 Offline-Lernen – ein Ausblick

Offline-Lernen in der hier vorgestellten Form führt nicht annähernd zu einem Erfolg. Dennoch könnte es sich lohnen, die Idee nicht vollkommen fallen zu lassen. Was könnte man probieren?

Bessere Auswahl „wichtiger" Trajektorien Um die riesige Masse von gespeicherten Trajektorien zu vermeiden, müsste sich auf die wirklich wichtigen von ihnen beschränkt werden. Dazu müsste man ein Verfahren finden, dass ständig auf der gespeicherten Information operiert, ähnliche Trajektorien identifiziert, unnötige Schleifen vermeidet und suboptimale Aktionsfolgen löscht. Um dies zu bewerten, wird allerdings wohl Wissen um das Systemverhalten von Nöten sein, um wirklich effektiv zu funktionieren.

Andere Behandlung für X^--Trajektorien Aktionsfolgen, die das System in Randbereiche führen, sind zwingend notwendig für den Lernerfolg, vermeiden ihn aber bei den hier vorgestellten Methoden. Es müsste eine Möglichkeit gefunden werden, diese Trajektorien einzulernen, ohne dass es zur Konvergenz kommt. Dazu müssten insbesondere die Netzausgaben während des Lernens überwacht werden.

Offline–Lernen als Ergänzung Eventuell könnte man Offline-Lernen nur als Ergänzung zum herkömmlichen TD(λ)-Lernen verwenden und das gespeicherte Wissen um erfolgreiche Trajektorien nur in definierten Zeiträumen (bspw. nachdem gute Anfangslernerfolge erreicht wurden oder wenn der Lernerfolg wieder zurückgeht) zusätzlich einlernen.

Am Ende steht natürlich wie immer die Frage, in welchem Verhältnis der Aufwand zum Nutzen steht. Nicht zuletzt aufgrund des ohnehin schon extrem hohen Speicheraufwands des Offline-Verfahrens bleibt es fraglich, ob evtl. gefundene Lösungen im praktischen Einsatz bestehen können.

5 Fazit

Zur Zeit wird viel im Bereich des Temporal Difference Learning geforscht, um die in dieser Arbeit vorgestellten Probleme in den Griff zu bekommen. Die Arbeit mit neuronalen Methoden und selbst lernenden Systemen gestaltet sich vielfach aufgrund der komplizierten Parameter-Auswahl sehr schwierig, die Auswirkungen von neuen Methoden und Varianten sind im Vorfeld schwer vorhersehbar, auch verheißungsvolle Ansätze führen oft nicht zum Erfolg, und oft ist noch nicht einmal der Grund dafür nachzuvollziehen. Auch in dieser Arbeit erwiesen sich verheißungsvolle Ansätze im Endeffekt als nutzlos.

Abschließend betrachtet können die Ergebnisse dieser Arbeit nicht befriedigen. Die vorgestellte Methode zum Offline-Lernen erwies sich als erfolglos und ist in dieser Form nicht verwendbar. Phasenweise Exploration dagegen konnte mit beachtenswerten Ergebnissen aufwarten, allerdings unter dem Preis eines immensen Speicherverbrauchs. Die dort sichtbaren Steigerungen in der Lernqualität und -geschwindigkeit und die damit verbundene Senkung der Trainingskosten zeigen, dass auf dem Gebiet der Exploration längst noch nicht alle Möglichkeiten ausgeschöpft sind. Die Speicherung bereits bekannten Wissens ist dabei nachweislich hilfreich. Offen bleibt die Frage, welche Alternativen es zu den in dieser Arbeit vorgestellten Möglichkeiten gibt und wie erfolgreich sich die hier nur angedachten Erweiterungen und Varianten erweisen werden.

Literaturverzeichnis

[Ber87] D. P. Bertsekas. *Dynamic Programming. Deterministic and Stochastic Models.* Prentice-Hall, 1987.

[BSW90] A. Barto, R. Sutton, and C. Watkins. Learning and Sequential Decision Making. In M. Gabriel and J. Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 539–602. The MIT Press, Cambridge, MA, 1990.

[Jan94] Barbara Janusz. Reinforcement Learning und Dynamisches Programmieren. Diplomarbeit, Universität Karlsruhe, 1994.

[RB93] Martin Riedmiller and Heinrich Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, 1993.

[Rie95] Martin Riedmiller. Dokumentation zu n++. Technical report, 1995.

[Sut88] R. Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[WD92] C. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8:279–292, 1992.