Aktionenlernen mit Selbstorganisierenden Karten und Reinforcement Learning

Dissertation
zur Erlangung des Grades
Doktor der Naturwissenschaften

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz

vorgelegt von

Felix Flentge

geboren in Wiesbaden

Mainz, im Juni 2005

Mündliche Prüfung: 28. Oktober 2005

D77 - Mainzer Dissertation

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung eines Funktionsapproximators und dessen Verwendung in Verfahren zum Lernen von diskreten und kontinuierlichen Aktionen:

1. **Ein allgemeiner Funktionsapproximator** – *Locally Weighted Interpolating Growing Neural Gas* (LWIGNG) – wird auf Basis eines Wachsenden Neuralen Gases (GNG) entwickelt. Die topologische Nachbarschaft in der Neuronenstruktur wird verwendet, um zwischen benachbarten Neuronen zu interpolieren und durch lokale Gewichtung die Approximation zu berechnen. Die Leistungsfähigkeit des Ansatzes, insbesondere in Hinsicht auf sich verändernde Zielfunktionen und sich verändernde Eingabeverteilungen, wird in verschiedenen Experimenten unter Beweis gestellt.

2. **Zum Lernen diskreter Aktionen** wird das LWIGNG-Verfahren mit Q-Learning zur *Q-LWIGNG*-Methode verbunden. Dafür muss der zugrunde liegende GNG-Algorithmus abgeändert werden, da die Eingabedaten beim Aktionenlernen eine bestimmte Reihenfolge haben. Q-LWIGNG erzielt sehr gute Ergebnisse beim Stabbalance- und beim Mountain-Car-Problem und gute Ergebnisse beim Acrobot-Problem.

3. **Zum Lernen kontinuierlicher Aktionen** wird ein REINFORCE-Algorithmus mit LWIGNG zur *ReinforceGNG*-Methode verbunden. Dabei wird eine Actor-Critic-Architektur eingesetzt, um aus zeitverzögerten Belohnungen zu lernen. LWIGNG approximiert sowohl die Zustands-Wertefunktion als auch die Politik, die in Form von situationsabhängigen Parametern einer Normalverteilung repräsentiert wird. ReinforceGNG wird erfolgreich zum Lernen von Bewegungen für einen simulierten 2-rädrigen Roboter eingesetzt, der einen rollenden Ball unter bestimmten Bedingungen abfangen soll.

Abstract

This doctoral thesis deals with the development of a function approximator and its application to methods for learning discrete and continuous actions:

1. **A general function approximator** – *Locally Weighted Interpolating Growing Neural Gas* (LWIGNG) – is developed from Growing Neural Gas (GNG). The topological neighbourhood structure is used for calculating interpolations between neighbouring neurons and for applying a local weighting scheme. The capabilities of this method are shown in several experiments, with special considerations given to changing target functions and changing input distributions.

2. **To learn discrete actions** LWIGNG is combined with Q-Learning forming the *Q-LWIGNG* method. The underlying GNG-algorithm has to be changed to take care of the special order of the input data in action learning. Q-LWIGNG achieves very good results in experiments with the pole balancing and the mountain car problems, and good results with the acrobot problem.

3. **To learn continuous actions** a REINFORCE algorithm is combined with LWIGNG forming the *ReinforceGNG* method. An actor-critic architecture is used for learning from delayed rewards. LWIGNG approximates both the state-value function and the policy. The policy is given by the situation dependent parameters of a normal distribution. ReinforceGNG is applied successfully to learn continuous actions of a simulated 2-wheeled robot which has to intercept a rolling ball under certain conditions.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Entwicklung von „intelligenten“ Systemen, die komplexe Aufgaben lösen oder bei deren Bewältigung helfen können, ist ein zentrales Anliegen der Informatik. Bei der Entwicklung solcher Systeme ist es oft zweckmäßig, von einem in einer Umwelt agierenden autonomen Agenten auszugehen, der Teile dieser Umwelt wahrnimmt und Aktionen ausführt, um bestimmte Ziele zu erreichen. Bei der Konstruktion autonomer Roboter oder ganz allgemein bei Steuerungsproblemen mag dies selbstverständlich erscheinen, aber auch in anderen Kontexten, angefangen von Problemen bei der Zuweisung von Ressourcen bis hin zu Computerspielen, kann eine solche Sicht äußerst nützlich sein. Mit Übernahme der agentenorientierten Perspektive stellt sich die Frage, welche Aktionen der Agent abhängig vom Zustand der Umwelt ausführen muss, um seine Ziele zu erreichen. Grundsätzlich gibt es zwei Möglichkeiten, diese Aktionen zu bestimmen: Entweder müssen dem Agenten die Aktionen in irgendeiner Form vorgeschrieben werden, oder aber der Agent bestimmt die Aktionen, die zur Verwirklichung der Ziele führen, selbsttätig durch die Verwendung geeigneter Algorithmen. Die erste Möglichkeit setzt voraus, dass die richtigen Aktionen bereits bekannt sind und dem Agenten „beigebracht“ werden können. Dabei können die Aktionen direkt einprogrammiert werden oder durch Anwendung eines überwachten Lernverfahrens gelehrt werden, wobei dem Agenten die auszuführenden Aktionen für bestimmte Zustände vorgegeben werden. Das bedeutet aber auch, dass spezifisches Wissen über das System vorhanden sein muss. Zudem ist es bei komplexen Problemen oft auch gar nicht möglich, die benötigten Aktionen direkt anzugeben. Die zweite Möglichkeit hat den Vorteil, dass dabei allgemeine Methoden zum Einsatz kommen, um die nötigen Aktionen zu bestimmen. Ist die Dynamik der Umwelt bekannt, können Such- und Planungsalgorithmen helfen, die richtigen Aktionen zu finden, indem Aktionen und ihre Wirkungen auf die Umwelt simuliert werden. Bei unbekannter Umweltdynamik kann entweder zunächst ein Modell der Umwelt bestimmt werden, oder es können Optimierungsverfahren wie genetische Algorithmen benutzt werden, die direkt den Raum der möglichen Verhaltensweisen durchsuchen. Bei diesen Verfahren wird jedoch in der Regel nur das Gesamtverhalten bewertet; die Reaktion der Umwelt auf bestimmte Aktionen geht bei vielen Ansätzen nur über den Umweg dieser Gesamtbewertung in den Optimierungsprozess ein.

Eine Möglichkeit, geeignete Aktionen direkt während der Interaktion mit der Umwelt zu lernen, bietet *Reinforcement Learning*. Dabei erhält der Agent eine Belohnung oder eine Bestrafung für das Erreichen bestimmter Zustände oder das Ausführen bestimmter Aktionen. Im Gegensatz zum überwachten Lernen wird also nicht die konkrete Aktion vorgegeben, sondern lediglich das Verhalten des Agenten bewertet. Dabei kann die Belohnung auch zeitverzögert sein. Sie bezieht sich dann nicht ausschließlich auf die gerade ausgeführte Aktion, sondern beispielsweise auf das Erreichen eines bestimmten Zustands, wobei dafür gleich eine ganze Reihe von vergangenen Aktionen verantwortlich sein kann. In der Interaktion mit der Umwelt lernt der Agent, seine Aktionen so zu bestimmen, dass die Summe der erhaltenen Belohnungen maximiert wird. Viele Reinforcement-Learning-Methoden sind auch dann einsetzbar, wenn die Dynamik der Umwelt unbekannt ist. Gegenüber Methoden wie genetischen Algorithmen bieten sie den Vorteil, dass diese Dynamik durch die beobachteten Zustandsübergänge dennoch berücksichtigt wird, was dabei hilft, bessere Aktionen zielgerichtet zu finden.
Bei Problemen, die einen diskreten, nicht allzu großen Zustandsraum besitzen, lässt sich Reinforcement Learning gut einsetzen und es gibt ein solides theoretisches Fundament, welches die Anwendbarkeit sicherstellt. Bei kontinuierlichen Zustandsräumen müssen Reinforcement-Learning-Methoden mit Funktionsapproximatoren kombiniert werden. Diese Kombinationen sind theoretisch meist weniger gut begründet und es gibt Beispiele dafür, dass die Kombination von Reinforcement Learning mit bestimmten Funktionsapproximatoren nicht stabil ist. Dennoch funktionieren diese Kombinationen in der Praxis meist gut, wobei jedoch nicht alle Funktionsapproximatoren gleich gut geeignet sind. Ein Problem besteht auch, wenn der Zustandsraum viele Dimensionen aufweist, da viele Approximatoren dann deutlich mehr Ressourcen benötigen und die Zeit für das Training stark ansteigt. Ein Ziel der vorliegenden Arbeit besteht darin, einen Funktionsapproximator zu entwickeln, der auf die speziellen Anforderungen beim Reinforcement Learning eingeht und auch bei höher-dimensionalen Zustandsräumen gut anwendbar ist.
Betrachtet man höher-dimensionale Zustandsräume, wie sie beim Lernen von Aktionen auftreten, etwas genauer, so kann man oft feststellen, dass die lokale Dimension der tatsächlich auftretenden Eingabedaten aufgrund von wechselseitigen Abhängigkeiten erheblich niedriger als die globale Dimension ist. Zudem ändert sich während des Lernprozesses auch die Verteilung der auftretenden Eingabedaten durch bereits realisierte Lernerfolge. Um diese beiden Beobachtungen bei der Funktionsapproximation auszunutzen, werden in der vorliegenden Arbeit entsprechend erweiterte und angepasste *Wachsende Neurale Gase* (Growing Neural Gas, GNG) benutzt. Wachsende Neurale Gase sind Selbstorganisierende Karten, die sich der lokalen Häufigkeit einer unbekannten, möglicherweise hoch-dimensionalen Eingabeverteilung anpassen und dieser folgen können, falls sie sich im Laufe der Zeit ändert. Außerdem können sie lokale Strukturen in den Eingabedaten entdecken und wiedergeben. Ein Anliegen dieser Arbeit ist es, die Vorteile dieser adaptiven Anpassung an die Eingabedaten bei der Funktionsapproximation, insbesondere in Verbindung mit dem Lernen von Aktionen, aufzuzeigen. Als „unüberwachte“ Lernverfahren bieten Wachsende Neurale Gase allerdings zunächst keine Möglichkeit, Funktionen zu approximieren und müssen daher entsprechend erweitert werden. Dabei werden bestehende Ansätze vereinheitlicht, miteinander kombiniert und zu einem neuen Verfahren – *Locally Weighted Interpolating Growing Neural Gas* (LWIGNG) – erweitert.
Das LWIGNG-Verfahren wird dann mit einem Reinforcement-Learning-Algorithmus (Q-Learning) zur *Q-LWIGNG*-Methode verbunden und zum Lernen von diskreten Aktionen eingesetzt. Eine bislang weitgehend offene Frage im Reinforcement Learning betrifft das Lernen von kontinuierlichen Aktionen. Dazu gibt es zwar verschiedene Ansätze,

aber bislang haben sich keine Standardverfahren etablieren können. In der vorliegenden Arbeit wird mit *ReinforceGNG* ein Verfahren zum Lernen kontinuierlicher Aktionen entwickelt, welches ebenfalls auf dem zuvor vorgestellten Funktionsapproximator LWIGNG beruht.
In den folgenden beiden Kapiteln wird kurz der Hintergrund beschrieben, vor dem dann die eigenen Arbeiten in den Kapiteln 4 - 6 präsentiert werden. Kapitel 2 beschäftigt sich zunächst mit den Grundlagen von Reinforcement Learning. Dabei wird insbesondere auf die Verwendung von Funktionsapproximatoren eingegangen. Zudem werden die speziellen Anforderungen an einen Funktionsapproximator beim Reinforcement Learning hergeleitet. Es wird auch kurz das Lernen von kontinuierlichen Aktionen betrachtet. Die Beschäftigung mit Policy-Gradient- und Actor-Critic-Ansätzen schließt dieses Kapitel ab.
Selbstorganisierende Karten und ihre Anwendung als Funktionsapproximator sind das Thema von Kapitel 3. Nachdem grundlegende Begriffe eingeführt worden sind, wird das Wachsende Neurale Gas genauer beschrieben. Die verschiedenen Möglichkeiten mit Selbstorganisierenden Karten Funktionen zu approximieren werden danach dargestellt. Schließlich wird noch genauer betrachtet, wie Selbstorganisierende Karten bisher im Reinforcement Learning eingesetzt wurden.
In Kapitel 4 wird dann ein eigener Funktionsapproximator (LWIGNG) auf Basis eines Wachsenden Neuralen Gases unter Beachtung der in Kapitel 2 beschriebenen Anforderungen hergeleitet. Dieser wird in verschiedenen Experimenten getestet und mit einem aktuellen Verfahren zur inkrementellen Funktionsapproximation verglichen.
Kapitel 5 verbindet das LWIGNG-Verfahren mit einem Q-Learning-Algorithmus zur Q-LWIGNG-Methode, um diskrete Aktionen zu lernen. Dazu sind noch einige kleinere Änderungen an LWIGNG nötig, die zunächst beschrieben werden. Zur Evaluation von Q-LWIGNG wurden Experimente mit drei Standardproblemen durchgeführt: dem Stabbalance-Problem, dem Mountain-Car-Problem und mit dem Acrobot. Die Ergebnisse dieser Experimente werden ausführlich beschrieben und mit Ergebnissen aus der Literatur verglichen. Am Ende des Kapitels werden die Ergebnisse der einzelnen Experimente überblickartig zusammengefasst, um verschiedene Aspekte des entwickelten Verfahrens zu beurteilen.
In Kapitel 6 wird dann ein Algorithmus zum Lernen kontinuierlicher Aktionen auf Grundlage des LWIGNG-Verfahrens hergeleitet. Dazu wird ein REINFORCE-Algorithmus in einer Actor-Critic-Architektur eingesetzt und LWIGNG sowohl zur Approximation der Politik als auch der Wertefunktion eingesetzt. Die Leistungsfähigkeit dieses Verfahrens soll dann in einem Szenario demonstriert werden, für das eine direkte algorithmische Lösung nicht trivial erscheint. Daher werden Experimente mit einem simulierten 2-rädrigem Roboter durchgeführt, der lernen soll, einen sich bewegenden Ball so anzufahren, dass dieser in Richtung eines vorgegebenen Ziels geschossen werden kann. Dabei wird auch der Fall betrachtet, dass die Wahrnehmung von Zuständen oder die Ausführung von Aktionen durch Rauschen gestört ist.
Kapitel 7 rundet die Arbeit mit einer Zusammenfassung und einer kurzen Bewertung der entwickelten Verfahren ab. Es werden offene Fragen benannt und mögliche Richtungen für weitergehende Forschungen vorgeschlagen.

Kapitel 2

Reinforcement Learning

In diesem Kapitel wird eine knappe Einführung in Reinforcement Learning, also das Lernen von Aktionen aufgrund von Belohnungen, gegeben. Nach einem ersten Abschnitt, in dem die allgemeinen Grundlagen vorgestellt werden, wird dann genauer auf *Temporal-Difference-Methoden* eingegangen, die im weiteren Verlauf der Arbeit zur Anwendung kommen. Die Darstellung orientiert sich dabei vor allem an [Sutton und Barto, 1998].
Während Reinforcement Learning bei einer kleinen Anzahl von Zuständen mit tabellierten Werten auskommt, müssen bei größeren oder kontinuierlichen Zustandsräumen Funktionsapproximatoren eingesetzt werden. In Abschnitt 2.3 geht es daher um die Kombination von Reinforcement Learning mit Funktionsapproximatoren. Im darauf folgenden Abschnitt wird dann kurz auf das Lernen von kontinuierlichen Aktionen eingegangen, da die bis dahin betrachteten Verfahren von einer kleinen Menge diskreter Aktionen ausgehen. Zum Schluss werden dann noch *Actor-Critic*- und *Policy-Gradient*-Ansätze und insbesondere die so genannten *REINFORCE*-Algorithmen betrachtet, die im weiteren Verlauf der Arbeit ebenfalls zur Anwendung kommen.

2.1 Grundlagen

Beim Lernen von Aktionen soll ein in einer Umwelt agierender Agent lernen, bestimmte Ziele zu erreichen. Er nimmt dabei verschiedene Zustände der Umwelt wahr und führt aufgrund der wahrgenommenen Zustände seine Aktionen aus. Als Ziel strebt er dabei entweder bestimmte Zustände an, oder aber er versucht bestimmte Zustände zu vermeiden. Sein Verhalten, also die Verknüpfung von Zuständen mit Aktionen, ist dabei nicht von Anfang an fest vorgegeben, sondern wird in einem Lernprozess angepasst, während der Agent mit der Umwelt interagiert. Ein solches Vorgehen ist vor allem dann von Vorteil, wenn eine feste Verknüpfung von Zuständen mit Aktionen, die zum Erreichen des Ziels führen, nur schwer oder überhaupt nicht angegeben werden kann. Das kann beispielsweise daran liegen, dass es zu viele Zustände gibt, oder dass erst im Nachhinein geklärt werden kann, ob eine ausgewählte Aktion hilfreich oder weniger hilfreich für die Realisierung des Ziels ist.

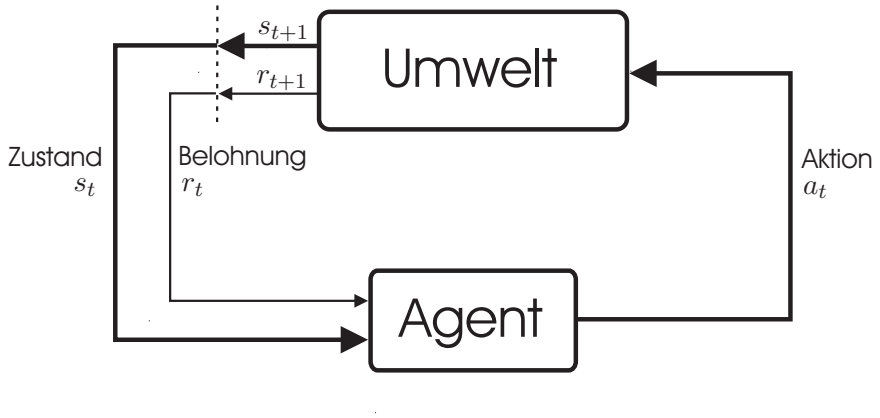


Abb. 2.1: Agent und Umwelt beim Reinforcement Learning (nach [Sutton und Barto, 1998]).

Anders als bei so genannten „überwachten Lernverfahren“, bei denen im Lernprozess bei einem Zustand auch gleich die gewünschte Aktion für diesen Zustand vorgegeben wird, wird beim Reinforcement Learning lediglich das Erreichen bestimmter Zustände oder das Ausführen bestimmter Aktionen belohnt oder bestraft. Dabei muss die Belohnung bzw. die Strafe nicht unmittelbar die jeweils ausgeführte Aktion bewerten, sondern kann zeitlich verzögert sein. Das Ziel eines Agenten beim Reinforcement Learning ist es, die Summe seiner Belohnungen über die Zeit durch „Versuch-und-Irrtum“ zu maximieren.

2.1.1 Markov-Entscheidungsprozesse

Die Problematik soll nun mathematisch formalisiert werden. Der Agent nimmt seine Umwelt als Folge von Zustandssignalen $s_0, s_1, s_2, ... \in S$ wahr und führt Aktionen $a_0, a_1, a_2, ... \in A(s_t)$ aus (siehe Abbildung 2.1). S ist dabei die Menge aller Zustände, $A(s_t)$ die Menge der im Zustand s_t möglichen Aktionen a_t. Wenn in allen Zuständen alle Aktionen möglich sind, schreiben wir einfach $a_t \in A$. Wir gehen im Folgenden davon aus, dass die Zustandsignale, die der Agent wahrnimmt, den tatsächlichen Zuständen der Umwelt weitgehend entsprechen und ihm erlauben, zwischen den für ihn wichtigen Aspekten der Umwelt zu unterscheiden, d.h. dass verschiedene Zustände in der Umwelt auch zu unterschiedlichen Zustandssignalen führen[1]. Im Folgenden wird daher nicht zwischen Zustandsignalen und den Zuständen der Umwelt unterschieden.
Die Auswahl der Aktionen wird durch die aktuelle *Politik* π_t des Agenten bestimmt. Die Politik gibt die Wahrscheinlichkeit für eine bestimmte Aktion a in einem bestimmten Zustand s an:

$$\pi_t : S \times A \to [0,1] \quad \pi_t(s,a) \mapsto \Pr(a_t = a | s_t = s) \tag{2.1}$$

Anstelle einer solchen *stochastischen* Politik kann der Agent auch eine *deterministische* Politik verfolgen, d.h. dass in jedem Zustand jeweils eine Aktion eine Wahrscheinlichkeit von Eins hat und daher immer dieselbe Aktion ausgeführt wird.
Bei Reinforcement-Learning-Problemen erhält der Agent nach seiner Aktion a_t eine Belohnung oder Bestrafung $r_{t+1} \in \mathbb{R}$ und nimmt den nächsten Zustand s_{t+1} wahr. Wir werden im Folgenden nur von Belohnungen sprechen und Bestrafungen einfach als negative

[1] In Kapitel 6 wird allerdings auch der Fall betrachtet, dass die Wahrnehmung des Agenten durch Rauschen gestört ist.

Belohnungen ansehen. Der gesamte Prozess besteht also aus einer Folge von Zuständen, Aktionen und Belohnungen: $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots)$. Sind diese Folgen endlich, so werden sie als *Episoden* bezeichnet. Sie enden mit einem zu erreichenden oder zu vermeidenden *Terminalzustand* zum Zeitpunkt $t = T$. Die Belohnung r_t kann von den Zuständen s_{t-1} und s_t sowie der Aktion a_{t-1} abhängen und zusätzlich zufällig verteilt sein. Da sich Belohnungen oft auf das Erreichen bestimmter Zustände beziehen und dafür meist eine ganze Reihe von Aktionen „verantwortlich“ ist, ist nicht unbedingt klar, welche Aktionen in der Vergangenheit für die aktuelle Belohnung ausschlaggebend sind. Soll der Agent beispielsweise Schachspielen lernen, so kann es sinnvoll sein, eine Niederlage mit -1 zu „belohnen“ und ansonsten die Belohnung auf Null zu setzen. Es ist klar, dass die allerletzte Aktion normalerweise nicht alleine für eine Niederlage verantwortlich ist. Dieses Problem wird als *temporal credit-assignment*-Problem bezeichnet. Reinforcement Learning beschreibt einen Ansatz zur Lösung dieses Problems.
Eine entscheidende Voraussetzung für Reinforcement-Learning-Algorithmen ist, dass die Wahrscheinlichkeit für einen bestimmten Nachfolgezustand s_{t+1} und eine bestimmte Belohnung r_{t+1} nur vom Vorgängerzustand s_t und der gewählten Aktion a_t abhängt und nicht von weiter zurück liegenden Zuständen oder Aktionen:

$$\begin{aligned} & \Pr \{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \\ = \; & \Pr \{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} \end{aligned} \tag{2.2}$$

Diese Eigenschaft wird als *Markov-Eigenschaft* bezeichnet. Die Markov-Eigenschaft garantiert, dass es ausreichend ist, den gegenwärtigen Zustand und die Aktion zu kennen, um Aussagen über den zukünftigen Zustand und die zu erwartende Belohnung zu treffen. Ein Reinforcement-Learning-Problem, welches die Markov-Eigenschaft erfüllt, heißt *Markov-Entscheidungsprozess (Markov decision process = MDP)*. Sind die Mengen der Zustände und der Aktionen endlich, heißt der MDP endlich. Ein MDP ist vollständig gegeben durch die Menge der Zustände S, die Mengen der Aktionen $A(s)$ und durch die von den Aktionen a abhängigen Übergangswahrscheinlichkeiten zwischen den Zuständen

$$P^a_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{2.3}$$

zusammen mit den Erwartungswerten für die Belohnungen bei diesen Zustandsübergängen

$$R^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}. \tag{2.4}$$

Gerade bei Realwelt-Problemen ist die Markov-Eigenschaft (2.2) nicht immer vollständig erfüllt, da nicht alle Aspekte der Realität, die für die Zustandsübergänge und Belohnungen relevant sind, in die Zustandsbeschreibungen aufgenommen werden können. Die meisten Reinforcement-Learning-Algorithmen funktionieren jedoch auch, wenn die Markov-Eigenschaft zumindest annähernd erfüllt ist, also die Zustandsübergänge und Belohnungen im Wesentlichen vom aktuell beobachteten Zustand und der gewählten Aktion abhängen. Probleme ergeben sich auch, wenn bestimmte Auswirkungen erst verzögert sichtbar werden. Für eine bestimmte Teilklasse solcher nicht-markovscher Entscheidungsprozesse, die so genannten *Partial Observable Markov Decision Processes* (POMDP), gibt es daher auch spezielle Algorithmen (siehe z.B. [Lovejoy, 1991; Murphy, 2000]).

2.1.2 Wertefunktionen

Das Ziel eines Agenten beim Reinforcement Learning besteht darin, die Gesamtheit der Belohnungen, die er über die Zeit erhält, zu maximieren. Diese wird als *Return* bezeichnet. Es reicht also nicht aus, immer die Aktionen zu wählen, die kurzfristig zu einer hohen Belohnung führen, da der Agent dabei vielleicht in Zuständen landet, in denen keine weiteren hohen Belohnungen mehr möglich sind.
Man kann zwischen kontinuierlichen Aufgaben, die kein bestimmtes Ende besitzen, und episodischen Aufgaben, bei denen es bestimmte Terminalzustände gibt, unterscheiden. Bei episodischen Aufgaben, lässt sich der Return R_t zum Zeitpunkt t als die Summe aller dem Zustand s_t folgenden Belohnungen definieren:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T = \sum_{k=0}^{T-t-1} r_{t+k+1} \tag{2.5}$$

Bei kontinuierlichen Aufgaben, die ja keine Terminalzustände kennen, muss zusätzlich ein so genannter *Diskontierungsfaktor* $0 \leq \gamma \leq 1$ eingeführt werden. Dieser sorgt dafür, dass der Return endlich bleibt[2], indem weiter in der Zukunft liegende Belohnungen weniger stark berücksichtigt werden:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.6}$$

Zur Vereinfachung werden beide Definitionen zusammengefasst:

$$R_t = \sum_{k=0}^{K} \gamma^k r_{t+k+1} \tag{2.7}$$

Dabei ist $K = T - t - 1$ für episodische Aufgaben und $K = \infty$ mit $\gamma < 1$ für kontinuierliche Aufgaben. Natürlich kann auch bei episodischen Aufgaben Diskontierung eingesetzt werden.
Reinforcement-Learning-Algorithmen versuchen meist, möglichst genau die zu erwartenden Returns zu schätzen. Dabei werden entweder *Zustands-Wertefunktionen (state-value functions)* oder *Aktions-Wertefunktionen (action-value functions)* gelernt, die den zu erwartenden Return für einen bestimmten Zustand bzw. ein Zustands-Aktions-Paar schätzen. Dabei hängt der Return natürlich von der aktuellen Politik π des Agenten ab. Bei Zustands-Wertefunktionen ist der Wert $V^\pi(s)$ eines Zustandes s der zu erwartende Return, wenn man in s beginnt und der Politik π folgt:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi \left\{ \sum_{k=0}^{K} \gamma^k r_{t+k+1} \mid s_t = s \right\} \tag{2.8}$$

E_π steht dabei für den von der Politik π abhängigen Erwartungswert. Ähnlich gibt die Aktions-Wertefunktion $Q^\pi(s,a)$ den zu erwartenden Return wieder, wenn in Zustand s Aktion a ausgeführt und dann der Politik π gefolgt wird:

$$Q^\pi(s,a) = E_\pi\{R_t \mid s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{K} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} \tag{2.9}$$

[2] Dies ist der Fall, falls $\gamma < 1$ und die Folge der Belohnungen $\{r_k\}$ beschränkt ist.

Fundamental für das Reinforcement Learning im Allgemeinen ist die *Bellman-Gleichung*. Diese beschreibt einen Zusammenhang zwischen dem Wert eines Zustandes und den Werten möglicher Nachfolgezustände[3]:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right] \tag{2.10}$$

Der Wert eines Zustandes ist also gleich der nach der Übergangswahrscheinlichkeit $P^{a}_{ss'}$ gewichteten, diskontierten Summe der Werte $V^{\pi}(s')$ der Nachfolgezustände s' plus den möglichen Belohnungen $R^{a}_{ss'}$ bei den Übergängen. Für Aktions-Wertefunktionen hat die Bellman-Gleichung die Form:

$$Q^{\pi}(s,a) = \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a') \right] \tag{2.11}$$

Mit Hilfe von Wertefunktionen lassen sich unterschiedliche Politiken vergleichen. Eine Politik π' ist besser bzw. genauso gut wie eine anderen Politik π, falls $V^{\pi'}(s) \geq V^{\pi}(s)$ für alle Zustände $s \in S$ gilt. Es gibt immer eine oder mehrere Politiken, die besser als alle anderen sind. Eine Politik, die in einem Zustand schlechter als eine andere ist, lässt sich nämlich immer verbessern, indem für diesen Zustand einfach die Aktionswahrscheinlichkeiten der besseren Politik übernommen werden. Die optimalen Politiken werden mit π^* bezeichnet. Sie besitzen alle dieselbe optimale Zustands-Wertefunktion V^* mit:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \;\; \forall s \in S \tag{2.12}$$

Analog gibt es eine eindeutige optimale Aktions-Wertefunktion $Q^*(s,a)$ mit:

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \;\; \forall s \in S \;\wedge\; \forall a \in A(s) \tag{2.13}$$

Diese gibt den zu erwartenden Return an, wenn in Zustand s Aktion a ausgeführt wird und danach einer optimalen Politik π^* gefolgt wird:

$$Q^{*}(s,a) = E\{ r_{t+1} + \gamma V^{*}(s_{t+1}) \mid s_t = s, a_t = a \} \tag{2.14}$$

Für die optimalen Wertefunktionen lassen sich die Bellman-Gleichungen in einer speziellen Form als *Bellmann-Optimalitätsgleichungen* schreiben:

$$\begin{aligned} V^{*}(s) &= \max_{a} E\{ r_{r+1} + \gamma V^{*}(s_{t+1}) \mid s_t = s, a_t = a \} \\ &= \max_{a} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{*}(s') \right] \end{aligned} \tag{2.15}$$

Für die Aktions-Wertefunktion gilt:

$$\begin{aligned} Q^{*}(s,a) &= \max_{a} E\{ r_{r+1} + \gamma \max_{a'} Q^{*}(s_{t+1},a') \mid s_t = s, a_t = a \} \\ &= \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma \max_{a'} Q^{*}(s',a') \right] \end{aligned} \tag{2.16}$$

Statt dem expliziten Bezug zur Politik π taucht hier einfach das Maximum über alle möglichen Aktionen auf, da die optimale Politik ja gerade dadurch charakterisiert ist, dass sie die Aktionen ausführt, die einen maximalen Wert liefern.

[3] Für die Herleitung siehe z.B. [Sutton und Barto, 1998].

Ist die optimale Wertefunktion bekannt, so lässt sich eine optimale Politik bestimmen. Bei Zustands-Wertefunktionen ist die optimale Aktion a für einen Zustand s diejenige, welche den Erwartungswert $\sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V^*(s'))$ maximiert. Dafür müssen aber die möglichen Nachfolgezustände s' und auch deren Wahrscheinlichkeiten $P^a_{ss'}$ sowie die Erwartungswerte der Belohnungen $R^a_{ss'}$ bekannt sein. Bei einer optimalen Aktions-Wertefunktion ist dies nicht nötig: Hier kann einfach die Aktion a gewählt werden, die $Q^*(s,a)$ maximiert.
Theoretisch ist es möglich, die Bellmann-Optimalitätsgleichungen als Gleichungssystem in $|S|$ Zuständen zu betrachten und zu lösen. In der Praxis scheitert das jedoch an der Anzahl der Zustände und daran, dass die Dynamik der Umwelt, also die Übergangswahrscheinlichkeiten $P^a_{ss'}$ und die Erwartungswerte der Belohnungen $R^a_{ss'}$, meist nicht hinreichend bekannt ist. Die meisten Reinforcement-Learning-Methoden versuchen daher, die Bellman-Gleichungen approximativ zu lösen und schätzen die meist unbekannte Umweltdynamik mehr oder weniger implizit durch die beobachteten Zustandsübergänge und empfangenen Belohnungen während der Interaktion mit der Umwelt.

2.1.3 Politik-Evaluation und Politik-Verbesserung

Beim Reinforcement Learning lassen sich in der Regel zwei grundsätzliche Operationen unterscheiden. Bei der *Politik-Evaluation* wird die Wertefunktion der aktuellen Politik möglichst gut approximiert. Anschließend wird bei der *Politik-Verbesserung* auf Grundlage der geschätzten Wertefunktion die Politik verbessert und dann wiederum die Wertefunktion der neuen Politik geschätzt. Dieses Vorgehen wird als *Politik-Iteration* bezeichnet (vgl. Abbildung 2.2).
Die Politik-Verbesserung beruht dabei auf dem *Politik-Verbesserungs-Satz*:

Satz (Politik-Verbesserung). *Seien π und π' Politiken, so dass*

$$\sum_a \pi'(s,a) Q^\pi(s,a) \geq \sum_a \pi(s,a) Q^\pi(s,a) .$$

Dann ist π' besser oder genauso gut wie π, d.h.

$$V^{\pi'}(s) \geq V^\pi(s) \; \forall s \in S.$$

Beweis. Zum Beweis siehe [Sutton und Barto, 1998]. □

Erscheint es also unter Verwendung der Aktions-Wertefunktion der aktuellen Politik Q^π besser (oder gleich gut), den Aktionen in einem Zustand s andere Wahrscheinlichkeiten als in der aktuellen Politik zuzuweisen, dann ist die dadurch entstehende neue Politik tatsächlich mindestens so gut wie die aktuelle Politik.
Um eine bessere Politik π' zu erhalten, wird daher die aktuelle Politik π meist so verändert, dass

$$\pi'(s,a) = \begin{cases} 0 & a \notin \arg\max\limits_{a'} Q^\pi(s,a') \\ p_i & a \in \arg\max\limits_{a'} Q^\pi(s,a') \text{ mit } p_i > 0 \text{ und } \sum_i p_i = 1 \end{cases} \tag{2.17}$$

gilt. Die neue Politik π' muss also allen Aktionen, die $Q^\pi(s,a)$ nicht maximieren, eine Wahrscheinlichkeit von Null zuweisen. An einen solchen Politik-Verbesserungs-Schritt

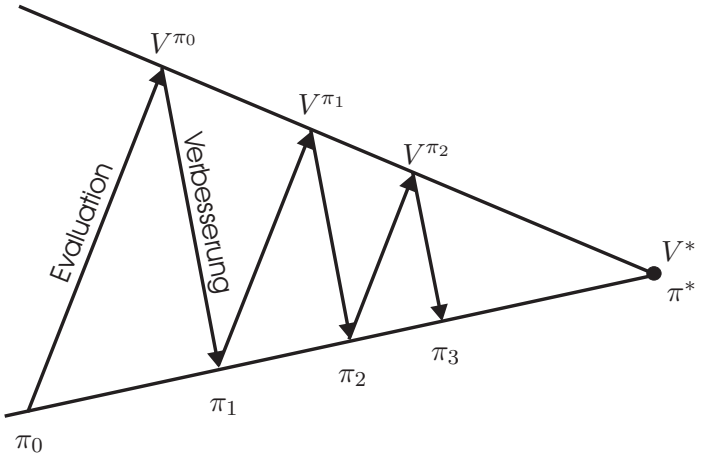


Abb. 2.2: Schematische Darstellung der Politik-Iteration nach [Sutton und Barto, 1998].

schließt sich dann wieder ein Politik-Evaluations-Schritt an. Der Prozess der Politik-Iteration endet, wenn sich die Schätzung für die aktuelle Zustands-Wertefunktion stabilisiert hat und keine bessere Politik mehr gefunden werden kann. Bei endlichen Markov-Entscheidungsprozessen bedeutet das, dass die optimale Politik gefunden ist [Sutton und Barto, 1998].

Die verschiedenen Reinforcement-Learning-Verfahren unterscheiden sich zum einen darin, wie die aktuelle Wertefunktionen geschätzt wird, also in der Politik-Evaluation, zum anderen darin, wie die Schritte Politik-Evaluation und Politik-Verbesserung miteinander verwoben werden. Außerdem kann man zwischen *on-policy*- und *off-policy*-Verfahren unterscheiden. On-policy-Methoden bewerten die Politik, die sie gegenwärtig verfolgen. Im Gegensatz dazu verfolgen off-policy-Methoden eine andere Politik als die, die sie bewerten und verbessern. Off-policy-Methoden versuchen meist, die optimale Wertefunktion und die optimale Politik direkt zu approximieren.

Ein weiterer wichtiger Unterschied zwischen den einzelnen Methoden besteht darin, wie viel über die Umwelt bekannt sein muss. Sind alle Übergangswahrscheinlichkeiten $P^a_{ss'}$ und die Erwartungswerte der Belohnungen $R^a_{ss'}$ bekannt, so können Methoden des *dynamischen Programmierens* verwendet werden. Dabei können die Wertefunktionen iterativ aus diesen Wahrscheinlichkeiten und den Erwartungswerten bestimmt werden, ohne dass der Agent mit der Umwelt interagiert. Allerdings handelt es sich hierbei um keine Lernverfahren im engeren Sinne, da es nur noch darum geht, aus den bekannten Informationen die Wertefunktion zu berechnen. „Echte“ Lernverfahren kommen dann zum Einsatz, wenn diese Wahrscheinlichkeiten nicht bekannt sind oder dynamisches Programmieren zu aufwändig wäre. Wenn ein Modell der Umwelt bekannt ist oder gelernt wird, d.h. wenn die Auswirkungen von Aktionen abgeschätzt werden können, reicht es aus, Zustands-Wertefunktionen zu schätzen. Mit Hilfe des Modells können dann Aktionen simuliert werden und diejenigen ausgesucht werden, die zum größten Wert führen. Modellfreie Verfahren hingegen arbeiten mit Aktions-Wertefunktionen. Die aktuell als am besten eingeschätzte Politik ergibt sich, wenn in einem Zustand s die Aktion a ausgeführt wird, die $Q(s, a)$ maximiert.

Dynamisches Programmieren

Dynamisches Programmieren (vgl. z.B. [Barto u. a., 1993; Bertsekas und Tsitsiklis, 1996; Sutton und Barto, 1998]) bestimmt die Wertefunktionen aus den Übergangswahrscheinlichkeiten $P^a_{ss'}$ und den Erwartungswerten für die Belohnungen $R^a_{ss'}$. Zur Politik-Evaluation wird die Bellmann-Gleichung (2.10) iterativ näherungsweise gelöst. Dazu wird die Gleichung in eine Anpassungsformel überführt:

$$\begin{aligned} V_{k+1}(s) &= E_\pi\{r_{r+1} + \gamma V_k(s_{t+1}) \mid s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right] \end{aligned} \tag{2.18}$$

Die neue Schätzung $V_{k+1}(s)$ für einen Zustand s basiert also auf der aktuellen Schätzung $V_k(s')$ für den Nachfolgezustand s'. Ein solches Vorgehen, bei dem Schätzungen aufgrund von anderen Schätzungen berechnet werden, wird als *bootstrapping* bezeichnet. Die Anpassungsschritte können auf verschiedene Weise durchgeführt werden. Unterschiede bestehen z.B. darin, in welcher Reihenfolge die Zustände behandelt werden und ab wann die neuen Schätzungen verwendet werden. Es kann gezeigt werden, dass die Folge $\{V_k\}$ für $k \to \infty$ gegen V^π konvergiert[4]. Bei der Anwendung beobachtet man die maximalen Veränderungen in den Anpassungsschritten und bricht den Prozess ab, wenn diese klein genug werden. Ist die gegenwärtige Wertefunktion V^π hinreichend genau approximiert, schließt sich der Politik-Verbesserungs-Schritt an.
Es ist auch möglich, den Prozess der Politik-Evaluation mit der Politik-Verbesserung zu verbinden, indem ähnlich wie in (2.15) der Bezug zur aktuellen Politik durch die Aktion ersetzt wird, die gegenwärtig den größten zu erwartenden Return liefert:

$$\begin{aligned} V_{k+1}(s) &= \max_a E\{r_{r+1} + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right] \end{aligned} \tag{2.19}$$

Die Politik-Verbesserung wird sozusagen direkt bei der Evaluation vollzogen, indem die aktuell als am besten angesehene Aktion berücksichtigt wird. Das Verfahren ist unter dem Namen *Werte-Iteration* bekannt. Unter sehr schwachen Bedingungen[5] kann man zeigen, dass Anpassungsregel (2.19) gegen die optimale Wertefunktion V^* konvergiert [Bertsekas und Tsitsiklis, 1996].
Die Methoden des dynamischen Programmierens haben zwei entscheidende Nachteile: Zum einen sind sie bei einer großen Anzahl von Zuständen kaum praktikabel, zum anderen müssen alle Übergangswahrscheinlichkeiten und die Erwartungswerte der Belohnungen bekannt sein, um die Politik-Evaluation durchzuführen.

Monte-Carlo-Methoden

Sind die Übergangswahrscheinlichkeiten und die Erwartungswerte der Belohnungen bei Markov-Entscheidungsprozessen unbekannt, müssen sie aus der Erfahrung geschätzt werden. Der Agent muss also mit der Umwelt interagieren, damit die möglichen Zustandsübergänge und Belohnungen beobachtet werden und in die Schätzungen für die

[4] Unter der Voraussetzung, dass die Wertefunktion V^π überhaupt existiert, der politikabhängige Erwartungswert E_π also für jeden Zustand endlich ist.

[5] Entweder muss $\gamma < 1$ gelten, oder es muss mindestens eine Politik geben, die von jedem Zustand mit Wahrscheinlichkeit 1 einen Endzustand erreicht und der Return für jede Politik, die das nicht tut, muss unendlich sein.

Wertefunktion mit eingehen können. Aber auch, wenn ein Problem zu viele Zustände und Aktionen kennt, um durch dynamisches Programmieren gelöst zu werden, kann ein solches Vorgehen vorteilhaft sein. Während sich traditionelles dynamisches Programmieren nämlich um alle Zustände gleich kümmert (auch um solche, die vielleicht gar nicht auftreten), findet hierbei eine Konzentration auf häufig auftretende Zustände statt [Bertsekas und Tsitsiklis, 1996].
Eine Möglichkeit, Wertefunktionen aus Erfahrung zu lernen, besteht darin, einfach die Durchschnittswerte für die Returns, die bestimmten Zuständen bzw. Zustands-Aktions-Paaren folgen, zu bestimmen. Diese Methoden heißen *Monte-Carlo-Methoden* (kurz MC-Methoden), da über zufällige Stichproben von Returns gemittelt wird. Da Monte-Carlo-Methoden diese Durchschnittswerte erst nach dem Ende einer Episode berechnen, können sie bei kontinuierlichen Aufgaben nicht angewendet werden. Ein weiterer wichtiger Unterschied zu den Methoden des dynamischen Programmierens besteht darin, dass Monte-Carlo-Methoden keine bootstrapping-Methoden sind, also nicht auf der Bellmann-Gleichung beruhen. Es wird immer der komplette Return betrachtet, während beim dynamischen Programmieren nur die unmittelbare Belohnung verwendet wird und der restliche Return durch die Schätzung des Wertes des Nachfolgezustandes angenähert wird.
Um eine optimale Politik zu finden, werden bei Monte-Carlo-Methoden dieselben Prinzipien angewandt wie beim dynamischen Programmieren, also Politik-Evaluation und Politik-Verbesserung. Bei der Politik-Evaluation wird zunächst eine vollständige Episode generiert. Dann wird für jeden auftretenden Zustand s der auf diesen folgende Return bestimmt[6] und der Wert des Zustandes auf den Durchschnitt aller bisher für diesen Zustand beobachteten Returns gesetzt. Die Politik-Verbesserung kann genauso wie im Falle des dynamischen Programmierens stattfinden. Auch eine Verbindung der Politik-Evaluation mit der Politik-Verbesserung wie bei der Werte-Iteration ist möglich. Hierbei sollten jedoch geeignet gewichtete Durchschnitte verwendet werden, da sich die Politik in jedem Schritt ändert und daher Returns, die unter länger vergangenen Politiken erreicht wurden, nicht so stark berücksichtigt werden sollten.

2.2 Temporal-Difference-Methoden

Temporal-Difference-Methoden (kurz TD-Methoden) verbinden den bootstrapping-Ansatz des dynamischen Programmierens mit dem modellfreien Vorgehen der Monte-Carlo-Methoden. Temporal-Difference-Methoden kommen also ohne Kenntnis der Übergangswahrscheinlichkeiten und der Erwartungswerte der Belohnungen aus, müssen aber auch nicht das Ende einer Episode abwarten, um einen Lernschritt durchzuführen. Das hat den Vorteil, dass Erfahrungen innerhalb einer Episode sofort im weiteren Vorgehen berücksichtigt werden können. Bei der Politik-Evaluation wird ganz ähnlich wie beim dynamischen Programmieren vorgegangen und eine auf der Bellmann-Gleichung beruhende Anpassungsregel verwendet. Anstelle der unbekannten Wahrscheinlichkeiten und Erwartungswerte werden die während der Interaktion mit der Umwelt beobachteten Zustandsübergänge und Belohnungen benutzt. Die Schätzung für einen Zustand $V(s_t)$ beruht auf der unmittelbaren Belohnung r_{t+1} und der gegenwärtigen Schätzung

[6] Tritt derselbe Zustand mehrmals auf, wird entweder nur das erste Auftreten berücksichtigt (*first-visit MC*), oder aber jedes Auftreten (*every-visit MC*).

für den aktuell auftretenden Nachfolgezustand $V(s_{t+1})$[7]:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right) \tag{2.20}$$

Der Ausdruck $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ beschreibt den Unterschied zwischen zwei aufeinander folgenden Zeitschritten und wird auch als „TD-Fehler“ bezeichnet. Die Anpassung kann erst zum Zeitpunkt $t+1$ stattfinden, da r_{t+1} und s_{t+1} zum Zeitpunkt t noch nicht bekannt sind. Der Wert eines Zustandes wird der Summe aus direkter Belohnung und dem Wert des diskontierten Nachfolgezustandes angenähert. Der Parameter $0 < \alpha \leq 1$ ist dabei die so genannte Lernrate. Sie bestimmt wie stark die aktuelle Beobachtung in die neue Schätzung mit einfließt. Wird $\alpha = 1$ gewählt, so wird der Wert des Zustandes ausschließlich nach der neuen Beobachtung gesetzt. Dies ist normalerweise aber ungeschickt, da Belohnungen und Zustandsübergänge stochastisch sein können. Ist $\alpha < 1$, so wird im Prinzip ein gewichtetes Mittel über alle beobachteten Belohnungen und Zustandsübergänge gebildet, wobei neuere Beobachtungen stärker gewichtet werden. Es kann gezeigt werden, dass die Anpassungsformel (2.20) für feste, kleine Lernraten α im Durchschnitt gegen V^π konvergiert [Sutton, 1988]. Wird α geeignet im Lernprozess verringert, kann auch gezeigt werden, dass (2.20) mit Wahrscheinlichkeit 1 gegen V^π konvergiert [Dayan, 1992].
Natürlich lassen sich auch Aktions-Wertefunktionen mit Temporal-Difference-Methoden lernen. So benutzt der *SARSA-Algorithmus*[8] [Rummery und Niranjan, 1994; Sutton, 1996] zur Politik-Evaluation das Äquivalent von (2.20) für Aktions-Wertefunktionen:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \tag{2.21}$$

Temporal-Difference-Methoden können auch so erweitert werden, dass nicht nur die unmittelbar folgende Belohnung in die neue Schätzung eingeht, sondern auch eine Reihe weiterer. Dies kann sehr elegant mit so genannten Aktivierungsspuren (eligibility traces) geschehen. Dabei wird die Aktivierung des gerade besuchten Zustandes erhöht und sinkt mit der Zeit wieder ab. Bei Auftreten einer Belohnung werden alle Zustände berücksichtigt, deren Aktivierung nicht Null ist. Damit werden auch nicht unmittelbar auftretende Belohnungen bei der Anpassung der Zustandswerte berücksichtigt. Dabei regelt ein Parameter λ, wie stark weiter in der Zukunft liegende Belohnungen gewichtet werden. Daher werden diese Methoden als TD(λ)-Methoden bezeichnet. Die oben vorgestellte Methode entspricht TD(0), da nur die unmittelbar folgende Belohnung berücksichtigt wird. Monte-Carlo-Methoden entsprechen TD(1), da alle Belohnungen bis zum Ende einer Episode benutzt werden[9].

2.2.1 Exploration vs. Ausnutzung von bereits erworbenem Wissen

Da bei Monte-Carlo- und Temporal-Difference-Methoden die Wertefunktionen aufgrund von Beispieltrajektorien durch den Zustandsraum geschätzt werden, stellt sich die Frage, wie diese Trajektorien erzeugt werden. Wird nur der aktuell als am besten angesehenen Politik gefolgt, besteht das Problem, dass eventuell bessere Alternativen gar

[7] In der Anpassungsformel verzichten wir auf den Index k bei der Wertefunktion und verwenden „$\leftarrow$“ in der Notation, um deutlich zu machen, dass hier ein einzelner Wert $V(s_t)$ sofort aktualisiert wird, wenn r_{t+1} und s_{t+1} bekannt sind.

[8] Der Name erklärt sich durch das Werte-Tupel, das für die Anpassung benutzt wird: $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{r}_{t+1}, \mathbf{s}_{t+1}, \mathbf{a}_{t+1})$.

[9] Mit dem Unterschied, dass TD(1) inkrementell und online verwendet werden kann und auch für nicht episodische Aufgaben geeignet ist.

nicht erst ausprobiert werden. Daher sollte der Agent gelegentlich von der bisher als am besten angesehenen Politik abweichen, um neue Aktionen zu testen. Dabei stellt sich das so genannte *exploration-exploitation*-Dilemma, d.h. der Agent muss sich entscheiden, ob er sein vorhandenes Wissen ausnutzen soll, um eine große Belohnung zu erzielen, oder ob er neue Aktionen ausprobieren soll, die eventuell zu noch größeren Belohnungen führen. Oft wird die ε-greedy-Methode zur Aktionsauswahl verwendet, d.h. dass mit einer Wahrscheinlichkeit von ε eine zufällige Aktion und ansonsten die so genannte „greedy-Aktion“ gewählt wird, also die Aktion, die im Augenblick als am besten angesehen wird:

$$\pi(s,a) = \begin{cases} \frac{1-\varepsilon}{|\arg\max_a Q(s,a)|} + \frac{\varepsilon}{|A|} & \text{für } a \in \arg\max_a Q(s,a) \\ \frac{\varepsilon}{|A|} & \text{für } a \notin \arg\max_a Q(s,a) \end{cases} \tag{2.22}$$

Der Parameter ε wird oft zu Beginn auf einen hohen Wert gesetzt und dann mit der Zeit reduziert. Eine weitere Methode, um am Anfang eine ausreichende Exploration sicherzustellen, besteht darin, die Anfangswerte für die Wertefunktion optimistisch zu wählen, also höhere Werte zu verwenden, als durch Belohnungen erreicht werden können. Damit werden Zustände bzw. Zustands-Aktions-Paare, die noch nicht besucht wurden, bevorzugt, da diese immer besser als bereits besuchte Zustände erscheinen. Diese Methode ist als *exploring starts* bekannt.

2.2.2 Q-Learning

Im Gegensatz zum SARSA-Algorithmus stellt der *Q-Learning*-Algorithmus [Watkins, 1989] eine off-policy-Methode dar, die versucht, die optimale Aktions-Wertefunktion $Q^*(s,a)$ direkt zu approximieren, während einer anderen Politik (meist ε-greedy) gefolgt wird. Dabei wird die SARSA-Anpassungsregel (2.21) so geändert, dass anstelle der tatsächlich ausgeführten Aktion das Maximum der Q-Werte der Nachfolgesituation benutzt wird:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right) \tag{2.23}$$

Falls s_{t+1} ein Terminalzustand ist, wird $\max_a Q(s_{t+1},a)$ als Null angenommen. Der vollständige Q-Learning-Algorithmus wird in Abbildung 2.3 wiedergegeben. Wenn alle Zustandspaare (s_t,a_t) unendlich oft besucht werden und α geeignet reduziert wird, konvergiert die Anpassungsformel (2.23) mit Wahrscheinlichkeit 1 gegen Q^* [Watkins, 1989; Watkins und Dayan, 1992; Jaakkola u. a., 1994; Tsitsiklis, 1994].

Initialisiere $Q(s,a)$ beliebig
Wiederhole(für jede Episode):
    Beobachte Startzustand s_0
    Wiederhole (für jeden Schritt der Episode, $t = 0,\ldots,T-1$):
        Wähle Aktion a_t nach einer von Q abgeleiteten Politik (z.B. ε-greedy)
        Führe Aktion a_t aus, beobachte r_{t+1}, s_{t+1}
        $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right)$
    bis s_{t+1} ein Terminalzustand ist

Abb. 2.3: Der Q-Learning-Algorithmus nach [Sutton und Barto, 1998].

2.3 Reinforcement Learning und Funktionsapproximation

Bisher sind wir davon ausgegangen, dass bei der Schätzung der Wertefunktion für jeden Zustand bzw. jedes Zustands-Aktions-Paar ein Wert gespeichert wird, etwa in Form einer Tabelle. Mit wachsender Größe des Zustandsraums $|S|$ und steigender Anzahl der möglichen Aktionen $|A|$ brauchen die Verfahren jedoch immer länger, um zu guten Ergebnissen zu gelangen, da immer mehr Zustände besucht werden müssen. Zudem können die Wertefunktionen für sehr große oder kontinuierliche Zustandsräume nicht mehr tabelliert werden. Eine Möglichkeit dennoch Wertefunktionen zu lernen besteht darin, zunächst den Raum geeignet zu diskretisieren und danach wie gewohnt eine tabellierte Funktion auf den diskreten Zuständen zu lernen. Dies ist aber nicht immer möglich, da eventuell zu viele Zustände unterschieden werden müssten. Außerdem ist i.A. auch nicht vorher bekannt, wie der Zustandsraum am besten diskretisiert werden kann.
Daher werden meist Funktionsapproximatoren auf der Basis von Gradientenabstiegsverfahren verwendet, um die Wertefunktion zu approximieren (etwa durch Neuronale Netze oder lineare Approximation, vgl. z.B. [Sutton und Barto, 1998]). Ein Vorteil bei der Verwendung von Funktionsapproximatoren liegt in deren Fähigkeit zur Generalisierung. Man kann meist annehmen, dass ähnliche Zustände ähnliche Werte besitzen und ähnliche Aktionen erfordern. Deswegen sollte nicht nur der Wert eines einzelnen Zustands bzw. Zustands-Aktions-Paars, sondern in geringerem Maße auch die Werte von ähnlichen Zustände bzw. Zustands-Aktions-Paaren aktualisiert werden. Hierbei stellt sich das *structural credit-assignment*-Problem: Welche Eigenschaften eines Zustands sind entscheidend für seinen Wert? Wie kann eine geeignete Generalisierung auf ähnliche Zustände stattfinden? Neben der Möglichkeit zur Generalisierung besteht ein Vorteil bei der Verwendung von Funktionsapproximatoren darin, dass die Funktionen viel kompakter repräsentiert werden können, als das bei Tabellen der Fall ist [Kaelbling u. a., 1996].
Allgemein geht es bei Funktionsapproximation darum, eine unbekannte Funktion

$$f(x) : \mathbb{R}^d \to \mathbb{R}^m \quad x \mapsto f(x) \tag{2.24}$$

aus möglicherweise fehlerbehafteten Beispielen $(x_t, y_t = f(x_t) + \varepsilon)$ mit einer Funktion

$$\widetilde{f}(x) : \mathbb{R}^d \to \mathbb{R}^m \quad x \mapsto \widetilde{f}(x) \tag{2.25}$$

möglichst gut zu wiederzugeben. Möglichst gut heißt dabei, dass der Fehler

$$\int_{\mathbb{R}^d} \left\| f(x) - \widetilde{f}(x) \right\|^2 dx \tag{2.26}$$

möglichst klein werden soll. Da $f(x)$ unbekannt ist, kann dieser Fehler nicht exakt bestimmt werden. Man kann lediglich den Fehler auf der Menge der bekannten Punkte (x_t, y_t) berechnen:

$$\sum_{t=1}^{n} \left\| y_t - \widetilde{f}(x_t) \right\|^2 \tag{2.27}$$

Ein Funktionsapproximator kann abstrakt als eine durch einen Parametervektor θ parametrisierte Funktion $\widetilde{f}_\theta(x)$ betrachtet werden. Bei einem Trainingsbeispiel (x_t, y_t) wird der aktuelle Parametervektor θ_t so geändert, dass die Ausgabe des Approximators $\widetilde{f}_{\theta_t}(x_t)$ der gewünschten Ausgabe y_t ähnlicher gemacht wird. Sehr oft werden dabei Gradientenabstiegsverfahren verwendet. Dazu wird der Fehler aus (2.27) als Funktion von θ betrachtet[10]:

$$E(\theta) = \frac{1}{2} \sum_{t=1}^{n} \left\| y_t - \widetilde{f}_\theta(x_t) \right\|^2 \tag{2.28}$$

[10] Der Faktor $\frac{1}{2}$ ist nur dazu da, die folgenden Berechnungen zu vereinfachen.

Um einen optimalen Parametervektor θ^* zu finden, muss das Minimum dieser Funktion gefunden werden. Dabei kann es zwei Probleme geben: Zum einen kann die Anzahl der Trainingsbeispiele recht hoch sein, so dass die Berechnungen sehr aufwändig sind. Zum anderen ist es oft so, dass die Trainingsbeispiele nicht alle auf einmal vorliegen, sondern nach und nach eintreffen und sofort in die Approximation mit eingehen sollen. Das ist beispielsweise bei Wertefunktionen im Reinforcement Learning der Fall, wenn neue Schätzungen auf aktuellen Werten beruhen sollen. Noch problematischer wird es, wenn sich die zu approximierende Zielfunktion während des Trainings ändert. Dann stellt sich die Frage, welche Trainingsbeispiele überhaupt noch und wie stark berücksichtigt werden sollen.
Eine Lösung für diese Probleme bieten die Gradientenabstiegsverfahren. Um (2.28) zu minimieren, wird dabei immer nur das aktuelle Trainingsbeispiel (x_t, y_t) betrachtet und inkrementell vorgegangen. Da wir an der Approximation von Wertefunktionen interessiert sind, gehen wir im Folgenden davon aus, dass y_t reellwertig ist, also f und $\widetilde{f}$ reellwertige Funktionen aus $\mathbb{R}^d$ nach $\mathbb{R}$ sind. Dadurch werden die folgenden Betrachtungen etwas vereinfacht. Prinzipiell ist ein Gradientenabstieg aber auch für vektorwertige Funktionen möglich. Der aktuelle Fehler in Abhängigkeit von θ ist durch $E_t(\theta) = \frac{1}{2}(y_t - \widetilde{f}_\theta(x_t))^2$ gegeben. Um diesen zu verringern, wird θ_t um einen Faktor α in die entgegengesetzte Richtung des Gradienten $\nabla_{\theta_t} E_t(\theta)$ verschoben[11], da der Gradient immer in Richtung des steilsten Anstieges zeigt, also:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \alpha \nabla_{\theta_t} E_t(\theta) \\ &= \theta_t - \alpha \nabla_{\theta_t} \left[\frac{1}{2} \left(y_t - \widetilde{f}_{\theta_t}(x_t) \right)^2 \right] \\ &= \theta_t + \alpha \left(y_t - \widetilde{f}_{\theta_t}(x_t) \right) \nabla_{\theta_t} \widetilde{f}_{\theta_t}(x_t) \end{aligned} \tag{2.29}$$

Der Parametervektor θ wird dabei nicht vollständig, sondern nur skaliert mit einer Lernrate α in die Richtung verschoben, die den aktuellen Fehler verringert, da ansonsten die Fehler für die anderen Trainingsbeispiele stark ansteigen könnten. Außerdem können die zu approximierenden Werte für einen festen Zustand x auch stochastisch verteilt sein, so dass es geschickter ist, eine Art gewichtetes Mittel zu bilden. Nimmt die Lernrate α nach bestimmten Bedingungen im Laufe des Trainings ab, so konvergiert θ gegen ein lokales Minimum von (2.28). Benutzt man hingegen eine feste Lernrate α, so werden aktuellere Beispiele stärker gewichtet als weiter zurückliegende. Dies ist immer dann wichtig, wenn sich die zu approximierende Zielfunktion während des Trainings noch ändern kann, wie dies auch bei Wertefunktionen der Fall ist.
Sollen Wertefunktionen approximiert und durch Gradientenabstieg angepasst werden, so wird anstelle von y_t eine Schätzung v_t für den Wert des Zustandes benutzt, etwa $v_t = r_{t+1} + \gamma V_t(s_{t+1})$ bei Temporal-Difference-Methoden:

$$\theta_{t+1} = \theta_t + \alpha \left(v_t - V_t(s_t) \right) \nabla_{\theta_t} V_t(s_t) \tag{2.30}$$

Die Wertefunktion V_t wird nur vom Parametervektor θ_t bestimmt, der nun anstelle der einzelnen Werte angepasst werden muss. Wenn der Parametervektor angepasst wird, ändern sich gleichzeitig die Werte verschiedener Zustände und führen so zu der angesprochenen Generalisierung. Falls v_t ein erwartungstreuer Schätzer für $V^\pi(s_t)$ ist, also $Ev_t = V^\pi(s_t)$, und α geeignet reduziert wird, konvergiert θ_t, so dass eine lokal optimale Approximation von $V^\pi(s_t)$ erreicht wird. Allerdings wird dabei nur ein lokales Mi-

[11] Der Gradient $\nabla_{\theta_t} E_t(\theta)$ existiert, falls $\widetilde{f}_\theta(x_t)$ als Funktion von θ an der Stelle θ_t differenzierbar ist.

nimum der Fehlerfunktion gefunden und es gibt normalerweise keine Möglichkeit zu garantieren, dass dieses lokale Minimum auch ein globales Minimum ist[12].
Aus den speziellen Problemen beim Reinforcement Learning ergeben sich bestimmte Anforderungen an den Funktionsapproximator:

- Inkrementelles Training
  Bei Reinforcement-Learning-Problemen liegt keine feste Menge von Trainingsbeispielen vor. Die Trainingsbeispiele entstehen entweder nacheinander während der Interaktion mit der Umwelt, etwa bei Temporal-Difference-Methoden, oder episodenweise, wie bei Monte-Carlo-Methoden. Der Funktionsapproximator muss also in der Lage sein, ein Trainingsbeispiel dann zu verarbeiten, wenn es eintrifft. Dies ist vor allem bei *bootstrapping*-Methoden wichtig, da dabei die Schätzungen ja auf der bereits gelernten Funktion beruhen.

- Umgang mit sich dynamisch ändernden Zielfunktionen
  Bei on-policy-Verfahren ändert sich die zu approximierende Zielfunktion ständig, da sich ja auch die Politik ständig ändert. Der verwendete Funktionsapproximator muss daher in der Lage sein, diesen Änderungen zu folgen. Aber auch bei off-policy-Methoden stellt sich das Problem: Die ersten Schätzungen für die Werte der Wertefunktion sind meist so schlecht, dass der Einfluss der ersten Beispiele mit der Zeit zurückgehen muss. Aktuelle Beispiele sollten ein größeres Gewicht erhalten.

- Hohe Güte der Approximation
  Natürlich ist es notwendig eine möglichst hohe Approximationsgüte zu erreichen, da eventuell zwischen einzelnen Zuständen nur geringe Unterschiede in deren Werten bestehen[13].

- Gute Generalisierungseigenschaften
  Ein Funktionsapproximator soll bei einem Trainingsbeispiel nicht nur die Werte für das entsprechende Beispiel ändern, sondern die gemachte Erfahrung sollte sich auch in der gesamten Nachbarschaft auswirken. Auf der anderen Seite darf aber auch nicht zu stark generalisiert werden, weil dies auf Kosten der Genauigkeit geht.

- Effizienz bei Training und Auswertung
  Für ein befriedigendes Lernergebnis ist oft eine große Anzahl von Lernschritten notwendig. Deswegen dürfen die Berechnungen nicht zu lange dauern. Das ist natürlich besonders wichtig, wenn in Situationen gelernt werden soll, in denen Echtzeitanforderungen herrschen und Aktionen innerhalb eines vorgegebenen Zeitintervalls gefunden werden müssen.

- Anpassung an die Verteilungen der Eingabevektoren
  Gerade bei hoch-dimensionalen Räumen treten meist nicht alle überhaupt möglichen Eingabevektoren auf. Aufgrund von wechselseitigen Abhängigkeiten unter den Eingabegrößen, stammen die Eingabevektoren oft nur aus bestimmten Unterräumen des eigentlichen Eingaberaums. In solchen Fällen muss der Funktionsapproximator nicht den gesamten Eingaberaum abdecken und kann sich auf diese Unterräume konzentrieren. Bei hoch-dimensionalen Eingaberäumen könnte

[12] Es sei denn, es kann gezeigt werden, dass jedes lokale Minimum auch ein globales Minimum ist, etwa wenn die Fehlerfunktion überhaupt nur ein Minimum besitzt.

[13] Im Prinzip ist eine hohe Approximationsgüte nicht notwendig, solange das „Verhältnis“ der einzelnen Werte zueinander gewahrt wird. Würde zu einer Wertefunktion für eine optimale Politik einfach ein fester Wert hinzuaddiert, würde sich die optimale Politik ja nicht ändern. Allerdings ist ein solcher Approximator nur schwer vorstellbar.

dies sogar der einzige Weg sein, um überhaupt mit vertretbarem Aufwand lernen zu können. Beim Lernen von Aktionen können sich diese Unterräume zusätzlich während des Trainingsprozesses stark ändern. Wenn der Agent lernt, werden einige Eingabevektoren mit der Zeit überhaupt nicht mehr auftreten und dafür andere Eingabevektoren häufiger werden oder neu hinzukommen. Dann ist es wünschenswert, wenn der Approximator sich dieser veränderten Verteilung anpassen kann, etwa indem die Zentren von Basisfunktionen verschoben werden.

Ein Problem bei der Verbindung von Reinforcement Learning mit Funktionsapproximation ist, dass es in der Regel keine Beweise für die Konvergenz der kombinierten Verfahren gibt. Für einige Kombinationen von Verfahren mit einfachsten Approximatoren lassen sich sogar Gegenbeispiele konstruieren, die Divergenz zeigen (vgl. z.B. [Baird, 1995; Tsitsiklis und Van Roy, 1997])[14]. Dennoch scheinen Kombinationen von Reinforcement Learning und Funktionsapproximatoren in der Praxis meist gut zu funktionieren. Dabei kommen verschiedenste Funktionsapproximatoren zum Einsatz. Insbesondere *Tiling*-Methoden wie CMAC (Cerebellar Model Articulator Controller) [Albus, 1971, 1981] werden oft benutzt. Dabei wird der Zustandsraum durch so genannte „*tilings*“ mehrfach vollständig partitioniert. Die „tilings“ können beliebig gewählt werden. Besonders einfach wird das Verfahren, wenn Gitter benutzt werden, die um einen kleinen Betrag gegeneinander verschoben werden. Jedem Zustandsvektor s_t kann nun je ein Element, ein „*tile*“, eines jeden „tilings“ zugeordnet werden. Zu jedem „tile“ gehört ein Wert, der im Laufe des Trainings angepasst wird. Der Wert $V(s_t)$ wird dann durch Addition der Werte der einzelnen „tiles“, die zum entsprechenden Zustandsvektor gehören, approximiert. Der Parametervektor θ besteht bei diesem Verfahren aus den Werten für alle „tiles“. Die Anpassung geschieht über ein Gradientenabstiegsverfahren, wobei der Gradient durch die Linearität eine extrem einfache Form annimmt. Aber auch Verfahren, die z.B. auf Entscheidungsbäumen [Pyeatt und Howe, 1998; Reynolds, 2000] oder verschiedensten Interpolationsalgorithmen beruhen [Boyan und Moore, 1995; Davies, 1997; Munos, 1997; Munos und Moore, 1999, 2002] werden eingesetzt. Auch neuronale Methoden wie Backpropagation-Netze [Anderson, 1987, 1993; Coulom, 2002, 2004] oder RBF-Netze [Kretchmar und Anderson, 1997; Buck u. a., 2002b] werden verwendet. Ein besonders erfolgreiches Beispiel für die Kombination eines neuronalen Backpropagation-Netzes mit einer TD(λ)-Methode, stellt das Programm TD-Gammon [Tesauro, 1992, 1994, 1995] dar. Hierbei besteht der Parametervektor θ aus allen Gewichtsvektoren des Neuronalen Netzes. Dieses Programm spielt gegen sich selbst und lernt so, auf Weltklasse-Niveau Backgammon zu spielen. Allerdings ist es bisher nicht gelungen, dieses Verfahren auch bei anderen Spielen erfolgreich anzuwenden.

2.4 Lernen von kontinuierlichen Aktionen

Funktionsapproximation stellt eine Möglichkeit dar, Reinforcement-Learning-Methoden auf Problemen mit kontinuierlichen Zustandsräumen anzuwenden. Manche Probleme erfordern aber auch kontinuierliche Aktionswerte oder lassen sich damit zumindest besser lösen, da dadurch eine feinere Steuerung möglich wird. Die Politik ist dann nicht

[14] Probleme scheinen dabei insbesondere dann aufzutreten, wenn bootstrapping-Methoden bei der Werte-Anpassung nicht die on-policy-Verteilung benutzen, also die Werte der Zustände nicht genau nach der Verteilung ihres Auftretens unter der aktuellen Politik geändert werden. Bootstrapping-Methoden werden aber trotzdem sehr häufig eingesetzt, da sie meist deutlich schneller als nicht-bootstrapping-Methoden sind.

mehr durch die Wahrscheinlichkeit für eine bestimmte Aktion in einem bestimmten Zustand gegeben, sondern über eine Wahrscheinlichkeitsdichtefunktion.
Da jedoch die üblichen Algorithmen wie Q-Learning oder SARSA bei kontinuierlichen Aktionen nicht direkt anwendbar sind, werden bei solchen Problemen die eigentlich kontinuierlichen Aktionen oft diskretisiert und dann die Algorithmen genauso angewendet wie bei diskreten Aktionen. Dies setzt aber voraus, dass das Problem mit der gewählten Diskretisierung tatsächlich gelöst werden kann, was im Voraus nicht unbedingt klar ist. Bei der Diskretisierung muss auch geklärt werden, in wie viele Einzelaktionen die kontinuierlichen Aktionen diskretisiert werden sollen. Sind es zu wenige, kann das Problem eventuell nicht befriedigend gelöst werden. Sind es zu viele, kann das Training sehr lange dauern und es müssen viele Werte gespeichert werden.
Bei kontinuierlichen Aktionen kann auf leicht veränderte Zustände mit leicht veränderten Aktionen reagiert werden. Auch eine Generalisierung über Aktionen wird möglich. Ähnliche Aktionen haben für ähnliche Situationen wahrscheinlich ähnliche Auswirkungen und daher ähnliche Zustands-Aktions-Werte. Bei der Diskretisierung gehen aber die Informationen über Ähnlichkeiten zwischen Aktionen verloren. Es kann also selbst dann von Vorteil sein, kontinuierliche Aktionen zu benutzen, wenn eine Diskretisierung im Prinzip möglich ist. Daher gibt es großes Interesse daran, Reinforcement-Learning-Methoden so zu erweitern, dass kontinuierliche Aktionen gelernt werden können.
Gerade für Q-Learning sind dabei in der Vergangenheit verschiedene Erweiterungen vorgeschlagen worden. Eine Möglichkeit besteht darin, den Aktionswert einfach als zusätzliche Eingabedimension zu betrachten und mit Hilfe von Funktionsapproximatoren eine in a kontinuierliche Q-Funktion $Q(s, a)$ zu approximieren. Dafür werden beispielsweise Neuronale Netze [ten Hagen und Kröse, 2000] oder CMAC-Funktionsapproximatoren verwendet [Santamaría u. a., 1997]. Ein Nachteil dieser Methode ist, dass die Approximation insgesamt schwieriger wird, da die Anzahl der Eingabedimensionen steigt. Ein Problem besteht auch darin, die Aktion zu finden, die $Q(s, a)$ maximiert. Bei diskreten Aktionen kann einfach die Q-Funktion für jede mögliche Aktion berechnet werden, bei kontinuierlichen Aktionen ist dies natürlich nicht mehr möglich und es müssen spezielle Verfahren angewendet werden, um dieses Maximum zu bestimmen. Andere Methoden wiederum benutzen mehrere Zustands-Aktionspaare (s, a), um dann für konkrete Zustände zwischen den Aktionswerten zu mitteln (vgl. z.B. [Millán u. a., 2002]). Beim *Wire-Fitting*-Ansatz [Baird und Klopf, 1993] wird für jeden Zustand s eine Menge von Kontrollpunkten $(a_i(s), q_i(s))$ für die Aktionen und die Q-Werte mit einem beliebigen Funktionsapproximator gelernt. Diese Kontrollpunkte beschreiben Kurven im Zustandsraum, so genannte *Wires*, und werden dazu benutzt, den Q-Wert für einen Zustand s durch eine Art gewichtete *Nearest-Neighbour*-Interpolation zu bestimmen. Diese ist so aufgebaut, dass das Maximum $\max_a Q(s, a)$ immer an einem der Kontrollpunkte liegt und somit schnell gefunden werden kann. Die Kontrollpunkte können dabei durch ein Gradientenabstiegsverfahren angepasst werden.
Einige Verfahren versuchen auch, den Raum der Aktionen durch die Verwendung geeigneter neuronaler Methoden abzubilden, etwa durch Neurale Felder [Gross u. a., 1997] oder Selbstorganisierende Karten [Smith, 2001, 2002]. Q-Werte beziehen sich dann auf bestimmte Regionen des abgebildeten Aktionsraums. Ähnlich gehen auch die *motorische Karte* [Ritter und Schulten, 1986; Ritter u. a., 1990], *Covariance Learning* [Wedel und Polani, 1996] und *Q-KOHON* [Touzet, 1997] vor. Dabei werden jedoch nicht tatsächlich kontinuierliche Aktionen gelernt, die auf leicht veränderte Zustände mit leicht veränderten Aktionen reagieren. Vielmehr werden aus dem kontinuierlichen Spektrum der Aktionen einige adaptiv ausgewählt und dann immer dieselbe Aktion angewandt, wenn die Zustände hinreichend ähnlich sind. Es handelt sich also eher um eine adaptive Diskreti-

sierung. Die genannten Ansätze werden in Abschnitt 3.4 genauer beschrieben, wo es um die Verbindung von Selbstorganisierenden Karten mit Reinforcement Learning geht.
Ein anderer Ansatz besteht darin, kontinuierliche Aktionen durch eine Normalverteilung zu realisieren. Der zustandsabhängige Mittelwert der Verteilung gibt dabei die aktuell favorisierte Aktion an und über die Standardabweichung kann das Explorationsverhalten geregelt werden. Ist die aktuell favorisierte Aktion mit ziemlicher Sicherheit die beste Aktion, so sollte die Standardabweichung klein sein, um möglichst oft diese Aktion auszuführen. Ist man sich hinsichtlich der Aktion noch unsicher, so sollte die Standardabweichung groß sein, um mögliche Alternativen zu testen. Die Parameter der Normalverteilung werden zustandsabhängig mit einem Funktionsapproximator geschätzt. Dies kann beispielsweise mit einem CMAC-Funktionsapproximator [Prescott, 1994] oder mit Neuronalen Netzen [Rummery, 1995] geschehen. In [Gullapalli, 1990, 1992a,b] werden dazu so genannte *Stochastic Real-Valued Units (SRV)* entwickelt. Der Erwartungswert der Normalverteilung wird dabei ebenfalls mittels einer neuronalen Architektur bestimmt. Zusätzlich wird der geschätzte Erwartungswert der Belohnung benutzt, um die Standardabweichung der Normalverteilung einzustellen und so das explorative Verhalten zu steuern. Belohnungen sind dabei jedoch auf $[0, 1]$ beschränkt. In Kapitel 6 soll ebenfalls von der Möglichkeit, kontinuierliche Aktionen mit Hilfe einer Normalverteilung zu lernen, Gebrauch gemacht werden. Der verwendete Lernalgorithmus basiert dabei auf den im nächsten Abschnitt vorgestellten Methoden.

2.5 Policy-Gradient- und Actor-Critic-Ansätze

Bei Policy-Gradient- und Actor-Critic-Ansätzen wird die Politik nicht aus der Wertefunktion abgeleitet, sondern explizit repräsentiert. Dabei parametrisiert der Parametervektor θ nicht mehr die Wertefunktion sondern die Politik direkt. Ist ρ ein Maß für die Performanz der aktuellen Politik (z.B. die durchschnittliche Belohnung pro Schritt), so wird der Parametervektor ungefähr proportional zum Gradienten der Performanz geändert [Sutton u. a., 2000]:

$$\theta_{t+1} \approx \theta_t + \alpha \nabla_\theta \rho \tag{2.31}$$

Dieser Gradient kann entweder aus Beispielen geschätzt oder auch berechnet werden, wenn der verwendete Funktionsapproximator ganz bestimmten Anforderungen genügt [Sutton u. a., 2000]. Die im Folgenden beschriebenen REINFORCE-Algorithmen stellen eine Möglichkeit dar, die Performanz einer Politik bei direkter Belohnung ohne explizite Berechnung des Gradienten zu maximieren. Bei Actor-Critic-Algorithmen wird die Politik durch den so genannten Actor repräsentiert. Der Critic lernt eine Wertefunktion, um dessen Politik zu bewerten. In Kapitel 6 wird ein REINFORCE-Algorithmus in einer Actor-Critic-Architektur eingesetzt, um kontinuierliche Aktionen aus zeitverzögerten, also nicht-direkten, Belohnungen zu lernen.

2.5.1 REINFORCE-Algorithmen

Williams stellte 1992 gleich eine ganze Klasse so genannter REINFORCE-Algorithmen vor [Williams, 1992]. REINFORCE-Algorithmen lernen dabei nur aus direkten Belohnungen[15]. Das Besondere an REINFORCE-Algorithmen ist, dass sie zwar ein stochastisches

[15] Es gibt eine Ausnahme für episodische Aufgaben, bei denen nur eine einzige Belohnung am Ende der Episode geliefert wird.

Gradientenverfahren darstellen, aber ohne eine explizite Berechnung dieses Gradienten auskommen.
REINFORCE-Algorithmen gehen von einem Netzwerk von (stochastischen) Einheiten u_i aus, bei denen eine Eingabe x durch das Netz geleitet wird, um eine Ausgabe a zu erzeugen. Eine stochastische Einheit u_i bestimmt dabei ihre Ausgabe a_i mit Hilfe einer Wahrscheinlichkeitsverteilung, die nur von ihrer Eingabe x_i (entweder direkt der Eingabevektor x oder die Ausgabe einer anderen Einheit) und ihrem Gewichtsvektor θ_i abhängt. Die Wahrscheinlichkeitsverteilung ist dabei entweder durch eine Wahrscheinlichkeitsfunktion $g_i(\xi, \theta_i, x) = \Pr\{y = \xi | \theta_i, x\}$ im diskreten bzw. durch eine Wahrscheinlichkeitsdichte $g_i(\xi, \theta_i, x)$ im kontinuierlichen Fall gegeben. Die Ausgabe a führt zu einer direkten Belohnung $r \in \mathbb{R}$. Diese wird dann von den Einheiten u_i des Netzes benutzt, um ihre Gewichtsvektoren θ_i in den Komponenten θ_{ij} anzupassen.
REINFORCE-Algorithmen versuchen, den Erwartungswert der direkten Belohnung r über die Zeit zu maximieren. Sind die Verteilungen, nach denen die Umwelt die Eingaben für das Netz und die Belohnungen bestimmt, stationär und unabhängig von der Zeit, so hängt der Erwartungswert der direkten Belohnung nur von den Gewichtsvektoren θ_i des Netzwerkes ab. Daher werden alle Gewichtsvektoren in einer Matrix Θ zusammengefasst und dieser Erwartungswert mit $E\{r|\Theta\}$ bezeichnet.
Die Anpassung der Gewichte θ_{ij} geschieht bei REINFORCE-Algorithmen gemäß der folgenden Anpassungsformel:

$$\Delta\theta_{ij} = \alpha_{ij}(r - b_{ij})e_{ij} \tag{2.32}$$

Dabei bezeichnet $\alpha_{ij} > 0$ eine eventuell vom jeweiligen Gewicht θ_{ij} und der Zeit abhängige Lernrate, b_{ij} die so genannte *reinforcement baseline* und

$$e_{ij} = \frac{\partial \ln g_i(a_i, x_i, \theta_i)}{\partial \theta_{ij}} \tag{2.33}$$

die *characteristic eligibility* von θ_{ij}[16]. Die „characteristic eligibility“ gibt sozusagen an, wie stark das jeweilige Gewicht für die ausgeführte Aktion und somit für die empfangene direkte Belohnung verantwortlich ist. Die „reinforcement baseline“ b_{ij} ist ein beliebiger Wert, der aber bedingt unabhängig von der Ausgabe a_i bei gegebenem Θ und x_i sein muss. In der Praxis hat es sich als vorteilhaft erwiesen, diese so zu wählen, dass die aktuelle Belohnung mit der zu erwartenden Belohnung verglichen wird (*Reinforcement Comparison*). Dabei wird $b_{ij} = \overline{r}$ gesetzt, wobei $\overline{r}$ eine Schätzung der zu erwartenden Belohnung ist.
Für alle Algorithmen der Form (2.32) gilt folgender Satz:

Satz (REINFORCE). *Für jeden REINFORCE-Algorithmus ist das Skalarprodukt* $E\{\Delta\Theta|\Theta\} \cdot \nabla_\Theta E\{r|\Theta\} \geq 0$.
Falls $\alpha_{ij} > 0$ *für alle* i, j *gilt:* $E\{\Delta\Theta|\Theta\} \cdot \nabla_\Theta E\{r|\Theta\} = 0 \Rightarrow \nabla_\Theta E\{r|\Theta\} = 0$.
Ist $\alpha_{ij} = \alpha$ *konstant, so gilt:* $E\{\Delta\Theta|\Theta\} = \alpha \nabla_\Theta E\{r|\Theta\}$

Beweis. Zum Beweis siehe [Williams, 1992]. □

Vereinfacht ausgedrückt bedeutet das, dass der durchschnittliche Anpassungs-Vektor $\Delta\Theta$ aus (2.32) im Raum der Gewichte in die Richtung des Gradienten der Performanz zeigt, also im Durchschnitt die Anpassungen zu einer besseren Performanz führen. Somit führen REINFORCE-Algorithmen im Prinzip einen Gradientenanstieg aus, wobei der Gradient nicht explizit berechnet werden muss.

[16] Von der Form dieser Anpassungen stammt auch die Bezeichnung für diese Klasse von Algorithmen: **RE**ward **I**ncrement = **N**onnegative **F**actor × **O**ffset **R**einforcement × **C**haracteristic **E**ligibility.

Im Folgenden soll noch ein konkreter REINFORCE-Algorithmus abgeleitet werden, der dann in Kapitel 6 zur Anwendung kommen wird. Dabei wird von einer einzigen Einheit u ausgegangen. Auf die Indizes i kann daher verzichtet werden. Als Verteilung zur Bestimmung der Ausgabe a wird eine Normalverteilung $N(\mu(x), \sigma(x))$ benutzt, wobei der Erwartungswert $\mu(x)$ und die Standardabweichung $\sigma(x)$ vom Eingabevektor x abhängen. Dabei steuert die Standardabweichung $\sigma(x)$ das Explorationsverhalten des Algorithmus: Große $\sigma(x)$ führen zu einer starken Exploration, während bei kleinen $\sigma(x)$ eher das bereits gelernte Wissen ausgenutzt wird.
Mit der Dichtefunktion der Normalverteilung

$$g(\xi, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\xi-\mu)^2}{2\sigma^2}} \tag{2.34}$$

können die „characteristic eligibilities“ berechnet werden:

$$e_{\mu(x)} = \frac{\partial \ln g\left(a, \mu(x), \sigma(x)\right)}{\partial \mu(x)} = \frac{a - \mu(x)}{\sigma(x)^2} \tag{2.35}$$

$$e_{\sigma(x)} = \frac{\partial \ln g\left(a, \mu(x), \sigma(x)\right)}{\partial \sigma(x)} = \frac{\left(a - \mu(x)\right)^2 - \sigma(x)^2}{\sigma(x)^3} \tag{2.36}$$

Damit ergeben sich die folgende Anpassungsregel analog zu (2.32):

$$\Delta\mu(x) = \alpha_\mu (r - b_\mu) \frac{a - \mu(x)}{\sigma(x)^2} \tag{2.37}$$

$$\Delta\sigma(x) = \alpha_\sigma (r - b_\sigma) \frac{\left(a - \mu(x)\right)^2 - \sigma(x)^2}{\sigma(x)^3} \tag{2.38}$$

Dem Vorschlag in [Williams, 1992] folgend, werden die Lernraten $\alpha_\mu = \alpha_\sigma = \alpha\sigma(x)^2$ gewählt. Die „reinforcement baselines“ b_μ und b_σ sollen so gewählt werden, dass sie Schätzungen für die zu erwartenden Belohnungen aufgrund der bisher im jeweiligen Zustand x gemachten Erfahrung sind, und werden daher mit $\overline{r}(x)$ bezeichnet. Das führt zu folgenden Anpassungsformeln:

$$\Delta\mu(x) = \alpha(r - \overline{r}(x))(a - \mu(x)) \tag{2.39}$$

$$\Delta\sigma(x) = \alpha(r - \overline{r}(x)) \frac{\left(a - \mu(x)\right)^2 - \sigma(x)^2}{\sigma(x)} \tag{2.40}$$

Die Analyse dieser Gleichungen gibt einen Einblick in die Arbeitsweise des Algorithmus: Falls die direkte Belohnung r besser als die erwartete Belohnung $\overline{r}(x)$ ist, wird $\mu(x)$ in Richtung der Ausgabe a verschoben. Ist r schlechter als erwartet, so wird $\mu(x)$ in die entgegengesetzte Richtung verschoben. In Gleichung (2.40) wird die Standardabweichung $\sigma(x)$ so geändert, dass die Wahrscheinlichkeit für das Auftreten von a erhöht wird, wenn r besser als erwartet ist. Ist z.B. $r > \overline{r}(x)$ und die Distanz $|a - \mu(x)|$ größer als eine Standardabweichung $\sigma(x)$, so wird $\sigma(x)$ erhöht und somit die Wahrscheinlichkeit für a erhöht. Ist r schlechter als erwartet, so wird $\sigma(x)$ verringert und somit das Auftreten von a unwahrscheinlicher.
Das Problem des hergeleiteten Algorithmus ist, dass dieser auf direkte Belohnungen r angewiesen ist und auch nicht geklärt ist, wie die Schätzung für die zu erwartende Belohnung $\overline{r}(x)$ bestimmt werden soll. Eine Möglichkeit dieses Problem zu lösen, besteht in der Verwendung einer Actor-Critic-Architektur, wie sie im Folgenden kurz vorgestellt wird.

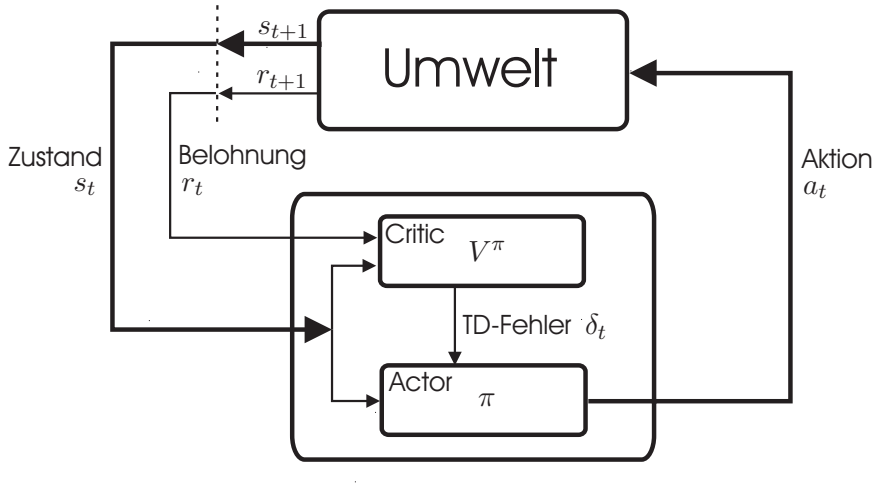


Abb. 2.4: Schematische Darstellung der Actor-Critic-Architektur.

2.5.2 Actor-Critic-Ansätze

Bei Actor-Critic-Ansätzen wird die Politik unabhängig von der Wertefunktion im Actor gespeichert. Der Critic lernt die Wertefunktion der gegenwärtig vom Actor verfolgten Politik und bewertet damit die vom Actor ausgeführten Aktionen. Dieser benutzt die Bewertungen des Critic, um seine Politik zu verbessern (siehe Abbildung 2.4).
Die Politik kann dabei auf unterschiedliche Art vom Actor gespeichert werden, etwa in Form eines Neuronalen Netzes oder durch Präferenzen für bestimmte Aktionen in bestimmten Zuständen (siehe [Sutton und Barto, 1998]). Der Critic benutzt in der Regel eine Zustands-Wertefunktion $V(s_t)$ und bewertet die vom Actor ausgewählte Aktion in Form des TD-Fehlers:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.41}$$

Falls δ_t positiv ist, ist die Summe aus empfangener Belohnung und dem Wert des Nachfolgezustands größer als der geschätzte Wert des gegenwärtigen Zustands. D.h. dass in der Vergangenheit wohl eher Aktionen gewählt wurden, die schlechter als die zuletzt gewählte Aktion waren. Daher sollte die Politik so verändert werden, dass die zuletzt gewählte Aktion häufiger in dem jeweiligen Zustand auftritt. Entsprechend sollte die zuletzt gewählte Aktion weniger oft ausgeführt werden, wenn δ_t negativ ist. Actor-Critic-Methoden haben insbesondere zwei Vorteile gegenüber Verfahren, die die Politik aus der Wertefunktion ableiten: Zum einen können stochastische Politiken durch Anpassung der Parameter der Politik gelernt werden. Zum anderen können Aktionen oft wesentlich schneller gefunden werden. Da Aktionen direkt aus der approximierten Politik bestimmt werden, müssen nicht erst Zustands-Wertefunktionen für mögliche Nachfolgezustände oder Aktions-Wertefunktionen ausgewertet werden, um dann das Maximum und damit die beste Aktion zu bestimmen. Dies ist insbesondere dann ein großer Vorteil, wenn die Anzahl der möglichen Aktionen sehr groß ist oder gar mit kontinuierlichen Aktionen gearbeitet werden soll.
In Kapitel 6 wird eine Actor-Critic-Architektur zum Lernen von kontinuierlichen Aktionen eingesetzt werden. Die Aktionen werden dabei mit Hilfe einer zustandsabhängigen Normalverteilung bestimmt und durch den zuvor beschriebenen REINFORCE-Algorithmus angepasst. Die Parameter dieser Verteilung werden dabei mit einer erweiterten Selbstorganisierenden Karte approximiert werden, die in den folgenden Kapiteln beschrieben wird.

Kapitel 3

Selbstorganisierende Karten

Selbstorganisierende Karten (Self-Organizing Maps, SOM) bilden mehrdimensionale Eingabesignale auf Strukturen künstlicher *Neuronen* ab. Als *unüberwachte* Lernverfahren finden sie dabei diese Abbildung selbsttätig während des Trainings mit den Eingabesignalen. Im ersten Teil dieses Kapitels sollen verschiedene Typen Selbstorganisierender Karten vorgestellt und deren Unterschiede beleuchtet werden. Von besonderem Interesse ist dabei das so genannte *Wachsende Neurale Gas (Growing Neural Gas, GNG)* [Fritzke, 1995a], welches im weiteren Verlauf dieser Arbeit breite Anwendung finden und deshalb im zweiten Teil des Kapitels ausführlich beschrieben wird.
Da Selbstorganisierende Karten zunächst nur eine Abbildung von Eingabesignalen auf Neuronen finden, müssen sie erweitert werden, wenn sie bei der Funktionsapproximation eingesetzt werden sollen. Im dritten Teil dieses Kapitels werden mögliche Erweiterungen, wie sie in der Literatur zu finden sind, beschrieben und miteinander verglichen. Zum Schluss des Kapitels wird noch auf die bisherigen Versuche, Selbstorganisierende Karten mit Reinforcement Learning zu kombinieren, eingegangen. Vor diesem Hintergrund werden dann im folgenden Kapitel die eigenen Erweiterungen vorgestellt.

3.1 Grundlagen

Zunächst soll der allgemeine Aufbau Selbstorganisierender Karten beschrieben und die wichtigsten Begriffe eingeführt werden. Wir wollen den Begriff der Selbstorganisierenden Karte dabei so verstehen, dass darunter alle unüberwacht gelernten Abbildungen einer Menge von Eingabesignalen auf eine Struktur von Neuronen fallen[1].
Selbstorganisierende Karten lernen entweder aus einer (prinzipiell unendlichen) Folge von Eingabesignalen x_t, deren Auftreten sich nach einer (in der Regel unbekannten) Häufigkeitsverteilung richtet, oder aber aus einer festen Menge von Eingabesignalen $\{x_1, x_2, \ldots, x_T\}$. Da in unserem Kontext die Eingabesignale durch Lernprozesse „online“

[1] In der Literatur wird der Begriff z.T. enger gefasst, so dass z.B. nur die *Kohonenkarte* (siehe Beispiel in diesem Kapitel) als Selbstorganisierende Karte bezeichnet wird.

generiert werden, beachten wir im Folgenden nur die erste Möglichkeit. In unserem Fall wird es sich bei den Eingabesignalen stets um eine zeitliche Abfolge reellwertiger Vektoren $x_t \in \mathbb{R}^d$ handeln. Wir werden daher $\mathbb{R}^d$ auch als Eingaberaum bezeichnen und von den Eingabevektoren x_t sprechen. Oft werden wir auch den expliziten Zeitbezug t weglassen, wenn dieser zum Verständnis nicht notwendig ist und nur die Notation unnötig verkomplizieren würde.
Selbstorganisierende Karten bestehen aus einer Menge von Neuronen:

$$A = \{c_1, c_2, \ldots, c_N\}$$

Jedes Neuron c_i besitzt eine Position im Eingaberaum (auch *Referenz-* oder *Gewichtsvektor* genannt):

$$w_i \in \mathbb{R}^d$$

Wir werden im Folgenden nicht immer scharf zwischen dem Neuron und seiner Position unterscheiden und teilweise auch von Neuronen sprechen, wenn eigentlich Positionen gemeint sind. Neuronen können über ungerichtete Kanten miteinander verbunden sein. Falls eine Kante zwischen zwei Neuronen c_i und c_j besteht, wird diese mit $e_{i,j} = e_{j,i}$ bezeichnet. E ist die Menge der Kanten. Neuronen, die eine gemeinsame Kante besitzen, heißen benachbart. Die Nachbarn eines Neurons c_i werden mit $c_{i,j}$ bezeichnet, also mit $j = 1, \ldots, N_i$ indiziert, wobei N_i die Anzahl der Nachbarn von c_i ist. Dabei soll per Konvention $c_{i,0} = c_i$ sein. Diese Notation werden wir für andere Variablen entsprechend anwenden, so soll z.B. $w_{i,j}$ die Position des j-ten Nachbars von c_i bezeichnen. Die von den Neuronen und den Kanten gebildete Struktur wird als Netz oder als Karte bezeichnet.
Wird ein Eingabevektor $x \in \mathbb{R}^d$ auf das Netz gegeben, wird zunächst das *Gewinnerneuron* $c_{b(x)}$ mit

$$d\left(w_{b(x)}, x\right) \leq d\left(w_i, x\right) \quad \forall\, i = 1, \ldots, N \tag{3.1}$$

bestimmt. Dabei ist $d(x, y)$ eine beliebige Metrik im Eingaberaum. In unserem Fall wird das stets der euklidische Abstand $\|x - y\|$ sein[2]. Statt $b(x)$ wird im Folgenden einfach b verwendet, wenn klar ist, dass es sich um den Index des Gewinnerneurons zum aktuellen Eingabevektor handelt. Das Gewinnerneuron wird dann einfach mit c_b bezeichnet. Die Selbstorganisierende Karte realisiert eine Abbildung vom Eingaberaum auf die Menge der Neuronen:

$$\mathbb{R}^d \to A,\ x \mapsto c_{b(x)} \tag{3.2}$$

Im einfachsten Fall besteht das Training solcher Selbstorganisierender Karten darin, dass die Position des Gewinnerneurons und die einer Reihe weiterer Neuronen, die nicht weiter als eine bestimmte Distanz l vom Eingabevektor x entfernt liegen, um $0 \leq \alpha \leq 1$ in Richtung des Eingabevektors verschoben werden (vgl. z.B. [Hastie u. a., 2001]):

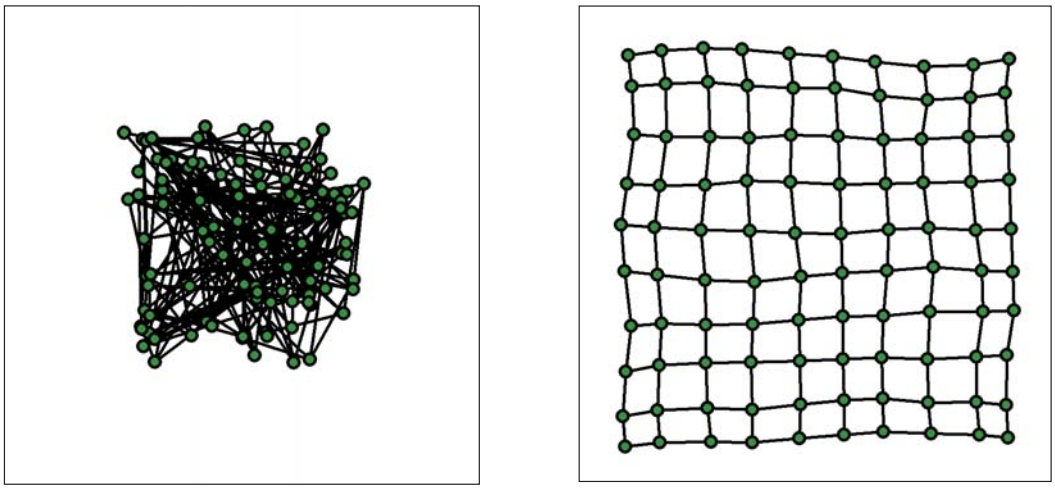
$$\begin{aligned} \Delta w_k &= \alpha\,(x - w_k) \\ &\quad k \in \{i \mid d(w_i, x) < l; i = 1, \ldots, N\} \end{aligned} \tag{3.3}$$

Oft fließen auch die Abstände zum Gewinnerneuron in der Neuronenstruktur (z.B. in der Manhattan-Metrik[3]) und die Anzahl der bereits vergangenen Lernschritte mit in die Lernregel ein:

$$\begin{aligned} \Delta w_k &= \alpha(t)\ h\left(d^N(c_b, c_k), t\right)(x - w_k) \\ &\quad k = 1, \ldots, N \end{aligned} \tag{3.4}$$

[2]Sollte $d(x, w_i) = d(x, w_j)$ für verschiedene i und j sein, so wird zufällig bestimmt, ob c_i oder c_j das Gewinnerneuron sein soll.

[3]Die Manhattan-Metrik ist gegeben durch die minimale Anzahl von Kanten, die zwischen einem Neuron und einem anderen liegen. So haben direkte Nachbarn den Manhattan-Abstand 1.



a) b)

Abb. 3.1: Ausbreitung einer Kohonenkarte im Einheitsquadrat. a) Vor Beginn des Trainings. b) Nach einem Training von 10000 Schritten.

Die Metrik $d^N(\cdot,\cdot)$ ist eine beliebige Metrik auf der Neuronenstruktur und keine Metrik im Eingaberaum wie $d(\cdot,\cdot)$ in (3.1). Die Funktion $h(d,t)$ ist stets größer oder gleich Null, erreicht ihr Maximum an der Stelle $d = 0$ und geht für große d gegen Null. Sie wird als *Nachbarschafts-* oder *Aktivierungsfunktion* bezeichnet und bestimmt, wie stark „weiter entfernte" Neuronen an den Eingabevektor angepasst werden. Zusätzlich kann $\alpha(t)$ mit wachsendem t monoton abnehmen und $h(d,t)$ mit der Zeit schmaler werden. Das führt dazu, dass sich das Netz zu Beginn des Trainings rasch ausbreitet und danach immer mehr erstarrt und sich eher lokalen Strukturen anpasst.

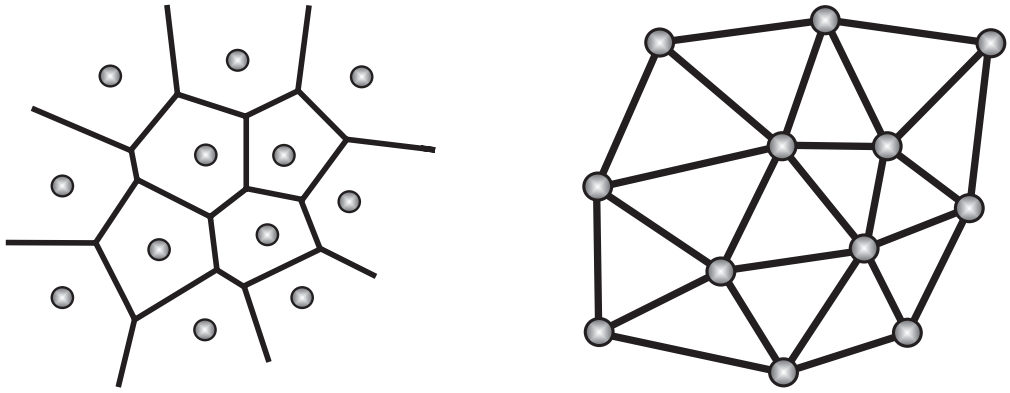
Beispiel: Kohonenkarte
Bei der Kohonenkarte [Kohonen, 1982] sind die Neuronen in der Regel in einer 2- oder 3-dimensionalen Gitterstruktur angeordnet[4]. Im hier gezeigten Beispiel besteht das Gitter aus 10 x 10 Neuronen. Der Eingaberaum ist das Einheitsquadrat $[0,1]^2$. Die Positionen der Neuronen werden zu Beginn mit zufälligen Werten aus $[0.25, 0.75]^2$ initialisiert. Als Lernregel wird (3.4) benutzt, wobei $\alpha(t)$ mit der Zeit linear abnimmt und $h(d,t)$ schmaler wird. Auf der Neuronstruktur wird die Manhattan-Metrik verwendet. In Abbildung 3.1 ist zu sehen, wie sich die zu Beginn zufällig verteilten Neuronen der Kohonenkarte nach dem Training regelmäßig im Eingaberaum angeordnet haben.

Wichtig im Zusammenhang mit Selbstorganisierenden Karten ist der Begriff der *Voronoi-Zerlegung* und der dazu duale Begriff der *Delaunay-Triangulierung*. Alle Punkte $x \in \mathbb{R}^d$, zu denen w_k der Positionsvektor mit dem geringsten Abstand ist, bilden die Voronoi-Zelle VZ_k zu w_k:

$$\mathrm{VZ}_k = \left\{ x \in \mathbb{R}^d \mid \|x - w_k\| \leq \|x - w_i\| \;\; i = 1, \ldots, N \right\} \tag{3.5}$$

Verbindet man die Positionsvektoren der Neuronen, deren Voronoi-Zellen benachbart sind, erhält man die Delaunay-Triangulierung der Positionsvektoren (siehe Abbil-

[4] Die Kohonenkarte wird hier formal als eine Struktur eingeführt, die eine Abbildung einer Menge von Eingabevektoren auf eine Menge von Neuronen realisiert. Das Kohonenmodell lässt sich jedoch auch biologisch motivieren und kann als, wenn auch sehr abstraktes, Modell neurophysiologischer Vorgänge im Gehirn angesehen werden [Ritter u. a., 1990].



a) b)

Abb. 3.2: a) Die Voronoi-Zerlegung einer 2-dimensionalen Punktmenge.
b) Die Delaunay-Triangulierung derselben Menge.

dung 3.2). Bei einigen Selbstorganisierenden Karten bildet die Menge der Kanten einen Subgraphen der Delaunay-Triangulierung.
Ein weiterer wichtiger Begriff im Zusammenhang mit Selbstorganisierenden Karten ist die *Topologieerhaltung*. Topologieerhaltung bedeutet,

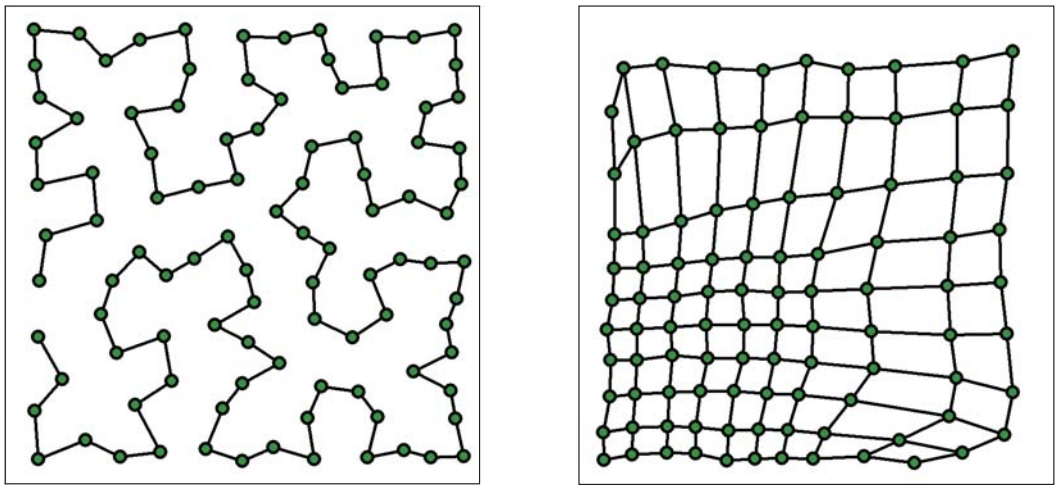
(i) dass im Eingaberaum benachbarte Positionsvektoren auf benachbarte Neuronen abgebildet werden

(ii) **und** dass die Positionsvektoren von in der Neuronenstruktur benachbarten Neuronen im Eingaberaum benachbart sind[5].

Die Nachbarschaft im Eingaberaum lässt sich dabei über die Voronoi-Zerlegung definieren: Zwei Punkte im Eingaberaum sind genau dann benachbart, wenn sie aneinander angrenzende Voronoi-Zellen besitzen. Topologieerhaltung ist nur möglich, wenn die Topologie der Mannigfaltigkeit, aus der die Eingabevektoren stammen, mit der Topologie der Neuronenstruktur übereinstimmt. Das ist zum Beispiel bei der Kohonenkarte in Abbildung 3.1 der Fall, da sowohl die Kohonenkarte als auch der Eingaberaum 2-dimensional ist. In der Regel ist für die Kohonenkarte jedoch nur Bedingung (ii) erfüllt. Das sieht man deutlich in Abbildung 3.3 a) wo 2-dimensionale Eingabevektoren aus $[0,1]^2$ auf eine 1-dimensionale Kohonenkarte trainiert wurden. Die Kohonenkarte nähert sich in etwa der Form einer Peano-Kurve[6] an, was dazu führt, dass benachbarte Neuronen benachbarte Positionsvektoren haben (Bedingung (ii)). Benachbarte Eingabevektoren werden jedoch keinesfalls immer auf benachbarte Neuronen abgebildet (Bedingung (i)). Topologieerhaltung ist besonders dann wünschenswert, wenn Ähnlichkeiten zwischen Eingabevektoren ausgenutzt werden sollen.
Eine weitere wichtige Eigenschaft der Lernregeln (3.3) und (3.4) ist, dass sich in der Verteilung der Neuronen die Häufigkeitsverteilung der Eingabevektoren widerspiegelt. Abbildung 3.3 b) verdeutlicht dies an Hand der Kohonenkarte. Aus dem Bereich $[0,0.5]^2$ stammen ungefähr doppelt so viele Eingabevektoren wie aus dem restlichen Einheitsquadrat. Deutlich ist zu sehen, dass die Neuronendichte in diesem Bereich wesentlich höher ist. Da es jedoch nicht immer von Vorteil ist, wenn sich die Verteilung der Neuronen nach der lokalen Häufigkeit richtet, werden im weiteren Verlauf dieser Arbeit

[5] Eine formale Definition findet sich in [Martinetz und Schulten, 1994].

[6] Eine Peano-Kurve ist eine raumfüllende Kurve. Sie wird durch eine rekursive Abbildungsvorschrift beschrieben und führt im Grenzwert zu einer surjektiven stetigen Abbildung von einem Intervall I in das Kreuzprodukt $I \times I$.



a) b)

Abb. 3.3: a) Ausbreitung einer 1-dimensionalen Kohonenkarte im Einheitsquadrat. b) Ausbreitung einer 2-dimensionalen Kohonenkarte bei nicht-gleichverteilten Eingabevektoren aus dem Einheitsquadrat.

Möglichkeiten vorgestellt, die Verteilung der Neuronen nach anderen Kriterien zu regeln.
Beim Lernen mit Selbstorganisierenden Karten wird meist eines der folgenden (oft einander ausschließenden) Ziele angestrebt [Van Hulle, 2000; Fritzke, 1995a, 1996]:

1. Minimierung des erwarteten Quantisierungsfehlers
   Es sollen Werte für die Positionsvektoren gefunden werden, die den erwarteten Quantisierungsfehler bei einer gegebenen Verteilung $P(x)$

   $$E\left(P(x), A\right) = \sum_{c \in A} \int_{\mathrm{VZ}_c} \|x - w_x\|^2 P(x) dx$$

   minimieren. Dieses Fehlerkriterium findet beispielsweise bei der *Vector-Quantization* Anwendung, bei der es darum geht, eine Menge von Vektoren durch eine kleinere Menge so genannter *Codebook-Vektoren* auszudrücken.

2. Entropiemaximierung[7]
   Jedes Neuron soll die gleiche Wahrscheinlichkeit haben, Gewinnerneuron zu werden:

   $$P\left(c_i = c_b\right) = \frac{1}{|A|} \; \forall c_i \in A$$

   Dieses Kriterium führt dazu, dass in Gebieten, aus denen viele Eingabevektoren stammen, viele Neuronen liegen, die jeweiligen Voronoi-Zellen dort also besonders klein sind.

3. Visualisierung
   Einige Selbstorganisierende Karten bilden hoch-dimensionale Eingabedaten auf niedrig-dimensionale Strukturen ab. So können z.B. höher-dimensionale Daten durch eine 2-dimensionale Kohonenkarte visualisiert werden (vorausgesetzt die

[7] Neben dem Quantisierungsfehler und der Neuronenentropie, gibt es noch eine Reihe weiterer Maße für Selbstorganisierende Karten (vgl. z.B. [Polani, 1995]). Diese spielen in der Anwendung allerdings meist keine Rolle.

Daten liegen zumindest lokal ungefähr auf einer 2-dimensionalen Mannigfaltigkeit).

4. Dimensions-Reduktion
   Bei der Dimensions-Reduktion geht es darum einen niedrig-dimensionalen Unterraum des Eingaberaums zu finden, der (fast) alle Eingabevektoren enthält.

5. Topologie-Lernen
   Das Ziel beim Topologie-Lernen ist, eine Neuronentopologie zu finden, die die Topologie der Verteilung der Eingabevektoren möglichst gut abbildet.

Ausgehend von diesen Zielen, wurde eine Reihe von verschiedenen Algorithmen vorgeschlagen. Diese unterscheiden sich meist in folgenden Punkten:

- hartes vs. weiches kompetetives Lernen
  Je nachdem ob nur das Gewinnerneuron oder auch andere Neuronen dem Eingabevektor angepasst werden, unterscheidet man zwischen *hartem und weichem kompetetivem Lernen*. Die Kohonenkarte ist ein Beispiel für weiches kompetetives Lernen, da neben dem Gewinnerneuron auch Neuronen in einer gewissen Nachbarschaft um einen geringeren Betrag in Richtung des Eingabevektors verschoben werden. Ein Beispiel für hartes kompetetives Lernen ist der *k-means*-Algorithmus [MacQueen, 1967], bei dem die Neuronenpositionen so angepasst werden, dass sie genau dem arithmetischem Mittel aller auf das jeweilige Neuron abgebildeten Eingabevektoren entsprechen.

- feste vs. adaptive Neuronenstruktur
  Manche Selbstorganisierende Karten besitzen eine feste Struktur, die sich während des Trainings nicht mehr ändert. So bestehen Kohonenkarten meist aus einer 2- oder 3-dimensionalen Gitter-Struktur. Ein Problem dabei ist, dass nur dann wirklich topologieerhaltende Abbildungen gefunden werden können, wenn die Neuronentopologie zur Topologie der Eingabeverteilung passt. Bei festen Strukturen erfordert das zumindest eine ungefähre Vorstellung von der Eingabeverteilung. Diese ist aber oft nicht im Voraus bekannt. Zudem kann sie auch sehr heterogen sein, wenn etwa die lokale Dimension variiert. Daher gibt es eine Reihe von Netzen, die ihre Topologie während des Trainings der Topologie der Eingabeverteilung anpassen. Werden diese Netze beispielsweise mit 3-dimensionalen Vektoren trainiert, die aber alle aus einem 2-dimensionalen Unterraum stammen, so ist die resultierende Struktur der Neuronen auch (nahezu) 2-dimensional. Ein extremes Beispiel ist das Neurale Gas, das überhaupt keine Struktur mehr besitzt (d.h. die Menge der Kanten ist leer) [Martinetz und Schulten, 1991].

- konstante vs. zeitlich dynamische Parameter
  Alle Parameter können entweder auf einen konstanten Wert festgelegt werden oder sich dynamisch mit der Zeit ändern. Die einfachste Möglichkeit ist, Parameter, wie z.B. die Lernraten, mit der Zeit zu reduzieren (*simulated annealing*). Solche Methoden führen dazu, dass das Netz nach einiger Zeit „einfriert“. Je nach Anwendung kann das gewünscht sein, oder nicht. Wenn sich z.B. die den Eingabevektoren zugrunde liegende Verteilung ändern kann, ist es von Vorteil, wenn die Lernraten noch nicht so weit reduziert sind, dass auf die veränderte Verteilung nicht mehr reagiert werden kann. Es gibt auch Netztypen, die die Lernraten zunächst reduzieren, dann aber versuchen, solche Änderungen in den Verteilungen zu registrieren und die Lernraten entsprechend anzupassen [Hawlitzky, 2001].

- globale vs. lokale Parameter
  Parameter wie die Lernrate oder der Aktivierungsradius können entweder global für das ganze Netz gelten wie bei der Kohonenkarte, oder aber jedes Neuron besitzt eigene lokale Parameter. Letzteres ist z.B. bei den DyCoN-Netzen der Fall, bei denen die lokalen Parameter über ein dynamisches System geregelt werden [Perl, 2001].

- feste Neuronenanzahl vs. wachsende Strukturen
  Die Anzahl der Neuronen kann entweder fest vorgegeben werden (z.B. bei der Kohonenkarte) oder sich im Laufe des Trainings ändern. So werden beispielsweise beim Wachsenden Neuralen Gas (siehe nächster Abschnitt) während des Trainings Neuronen neu eingefügt oder auch wieder gelöscht.

- online- vs. offline-Algorithmen
  Gibt es eine endliche Menge von Eingabevektoren, können diese gemeinsam betrachtet werden. So werden beim LBG-Algorithmus [Linde u. a., 1980] in jedem Durchgang zunächst für jedes Neuron alle Eingabevektoren in dessen Voronoi-Zelle bestimmt und anschließend die Positionsvektoren auf die Durchschnitte der Eingabevektoren in der jeweiligen Voronoi-Zelle gesetzt. Es gibt jedoch zahlreiche Situationen in denen die Menge der Eingabevektoren beliebig groß sein kann. In diesen Fällen müssen online-Verfahren angewendet werden, um jeweils nach der Präsentation eines einzelnen Eingabevektors das Netz entsprechend anzupassen.

Aus unserem Ziel, der adaptiven Anpassung an den Zustandsraum beim Lernen von Aktionen, ergeben sich in Hinblick auf die oben genannten Punkte folgende Anforderungen:

- beliebige, adaptive Neuronenstruktur:
  Da im Allgemeinen weder die Struktur noch die implizite Dimensionalität der Menge der Eingabevektoren im Vorhinein bekannt ist, bietet es sich an, adaptive Neuronenstrukturen zu verwenden, die sich der lokalen Dimension der Eingabeverteilung anpassen.

- wachsende Struktur:
  Da meist weder klar ist, wie viele Neuronen benötigt werden, noch wo diese benötigt werden, sollen wachsende Strukturen verwendet werden, die neue Neuronen an Stellen einfügen, an denen entsprechender „Bedarf“ besteht.

- online-Algorithmus:
  Da die Trainingsdaten aus der Interaktion mit der Umwelt stammen und diese Interaktion von dem bereits Gelernten abhängig ist, muss es sich um einen online-Algorithmus handeln.

- Parameter ohne explizite Zeitabhängigkeit:
  Da sich die Verteilung der Eingabevektoren während des Trainings durch bereits realisierte Lernerfolge verändert, dürfen Parameter nicht so weit reduziert werden, dass das Netz auf diese Veränderungen nicht mehr reagieren kann. Es bietet sich also an, entweder konstante Parameter zu wählen, oder aber die Anpassungsfähigkeit des Netzes durch eine geschickte Steuerung der Parameter zu gewährleisten.

Aus diesen Anforderungen heraus wurde das Wachsende Neurale Gas gewählt, welches im folgenden Abschnitt detailliert vorgestellt wird.

3.2 Wachsendes Neurales Gas

Das *Wachsende Neurale Gas (Growing Neural Gas, GNG)* [Fritzke, 1995a] ist eine Selbstorganisierende Karte mit adaptiver Neuronenstruktur, die sich im Laufe des Trainings der lokalen Verteilung der Eingabevektoren anpasst (sowohl der Häufigkeit als auch der lokalen Dimension). Der dabei entstehende Graph hat lokal im Wesentlichen dieselbe Dimensionalität wie die Menge der Eingabevektoren, die darauf abgebildet werden. Daher spricht man auch von *Topologie-Lernen*. Das Wachsende Neurale Gas stellt eine Erweiterung der so genannten *Topologieerhaltenden Netzwerke (Topology Representing Networks, TRN)* dar.
Topologieerhaltende Netzwerke basieren auf dem Neuralen Gas [Martinetz und Schulten, 1991] und auf der Anwendung einer kompetetiven Hebb'schen Lernregel[8]. Das Neurale Gas berücksichtigt keine Topologie auf der Neuronenstruktur; es kommt völlig ohne Kanten aus. Beim Neuralen Gas hängt die Anpassung der Positionsvektoren nicht von der Entfernung in einer gegebenen Neuronentopologie, sondern von den Abständen im Eingaberaum ab. Bei Präsentation eines Eingabevektors werden zunächst alle Neuronen nach dem Abstand zu diesem sortiert und dann nach ihrem Rang dem Eingabevektor angepasst (also das nächste am stärksten, das zweit nächste etwas weniger, usw.). Die Stärke der Anpassung wird über eine Nachbarschaftsfunktion geregelt, die auf diesem Rang der Neuronen basiert und mit der Zeit so abnimmt, dass weiter entfernte Neuronen weniger stark angepasst werden.
Um nun zu einer Topologie auf der Neuronenstruktur zu gelangen, wird das Prinzip des *Competetive Hebbian Learning* [Martinetz, 1993; Martinetz und Schulten, 1994] eingesetzt. Dabei werden für jeden Eingabevektor die beiden nächsten Neuronen bestimmt und eine Kante zwischen diesen eingefügt, wenn noch keine besteht. Der so entstehende Graph ist ein Subgraph der Delaunay-Triangulierung, die so genannte *induzierte Delaunay-Triangulierung*. So kann eine perfekt topologieerhaltende Karte erzeugt werden[9] (für formale Definitionen und Beweise siehe [Martinetz und Schulten, 1994]). Da beim GNG die Topologie auch auf diese Weise erzeugt wird, ergibt sich auch dabei ein Subgraph der Delaunay-Triangulierung. Somit wird auch durch das GNG eine perfekt topologieerhaltende Abbildung realisiert. Es ist außerdem zu vermuten, dass die induzierte Delaunay-Triangulierung auch eine gute Basis für die Anwendung in der Funktionsapproximation (siehe folgende Abschnitte) bildet. Zumindest bei der Verwendung einer stückweisen linearen Approximation auf einer Triangulierung, ist die Delaunay-Triangulierung diejenige Triangulierung, welche den kleinsten worst-case-Fehler liefert [Omohundro, 1990].
Das Erzeugen der Kanten kann entweder simultan zum Training der Positionsvektoren erfolgen, oder aber nachdem das Netz vollständig trainiert wurde. Soll das Erzeugen von Kanten simultan zum Training des Netzes erfolgen, müssen die Kanten aber noch mit einem „Alter" versehen werden und zu „alte" Kanten entfernt werden, weil sich die Nachbarschaften ja durch die Bewegung der Neuronen noch ändern und Kanten, die nicht mehr zur induzierten Delaunay-Triangulierung gehören, beseitigt werden müssen. Das Alter einer Kante $e_{i,j}$ wird im Folgenden mit $age_{i,j}$ (bzw. $age_{j,i}$, da die Kanten symmetrisch sind) bezeichnet. Das maximale Alter wird mit age_{max} bezeichnet.
Das GNG stellt eine Art inkrementelle Variante eines TRN dar. Der Algorithmus star-

[8] Nach dem Psychologen D. Hebb, der in 1940er Jahren die Hypothese aufstellte, dass sich die Verbindungsstärke zwischen zwei Neuronen proportional zur gemeinsamen Aktivität ändert [Hebb, 1949].

[9] Die einzige Voraussetzung ist, dass die Verteilung der Positionsvektoren „dicht" auf der gegebenen Mannigfaltigkeit sein muss. Es müssen sozusagen „genügend" Neuronen vorhanden sein, um die topologische Struktur der Eingabevektoren abzudecken.

Initialisierung:

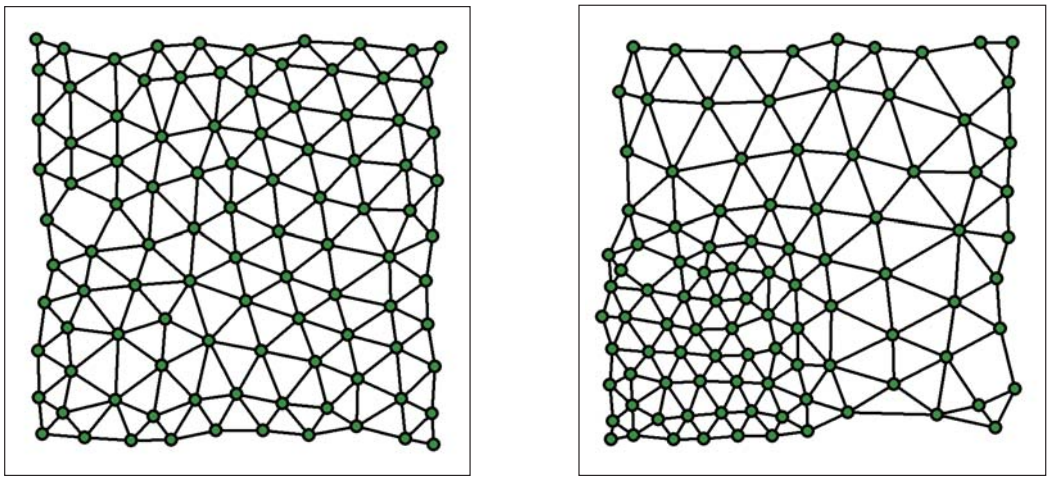
1. $A = \{c_1, c_2\}$ mit zufälligen Positionen w_1 und w_2
2. Erzeuge eine Kante $e_{1,2}$ mit Alter $age_{1,2} = 0$ zwischen diesen Neuronen

Wiederhole (bis ein beliebiges Stopp-Kriterium erfüllt ist):

1. Erhalte Eingabevektor $x \in \mathbb{R}^d$
2. Bestimme nächstes und zweit nächstes Neuron c_b und $c_{b'}$:
   $\|w_b - x\| \leq \|w_i - x\| \;\; i = 1, \ldots, N$
   $\|w_{b'} - x\| \leq \|w_i - x\| \;\; i = 1, \ldots, N; i \neq b$
3. Falls keine Kante zwischen c_b und $c_{b'}$ besteht, erzeuge Kante $e_{b,b'}$
4. Setze das Alter $age_{b,b'} = 0$
5. Akkumuliere den lokalen Quantisierungsfehler: $\Delta err_b = \|w_b - x\|^2$
6. Bewege c_b und dessen Nachbarn $c_{b,i}$ in Richtung des Eingabevektors:
   $\Delta w_b = \alpha_b (x - w_b)$
   $\Delta w_{b,i} = \alpha_n (x - w_{b,i}) \;\; i = 1, \ldots, N_i$ für alle Nachbarn $c_{b,i}$ von c_b
7. Erhöhe das Alter aller Kanten, die von c_b ausgehen um 1
8. Beseitige alle Kanten $e_{i,j}$ mit $age_{i,j} \geq age_{max}$
9. Beseitige alle Neuronen ohne Kanten
10. Alle N_{steps} Schritte: neues Neuron einfügen, falls $N \leq N_{max}$:
    Bestimme Neuron c_q mit maximalem Fehler err_q
    Bestimme den Nachbarn c_f von c_q mit maximalem Fehler err_f
    Füge Neuron c_r mit $w_r = 0.5 \cdot (w_q + w_f)$ ein
    Erzeuge Kanten $e_{r,f}$ zwischen c_r und c_f sowie $e_{r,q}$ zwischen c_r und c_q
    Lösche Kante $e_{f,q}$ zwischen c_f und c_q
    Verteile den lokalen Quantisierungsfehler neu:
    $\Delta err_q = -\delta_{new} \cdot err_q$
    $\Delta err_f = -\delta_{new} \cdot err_f$
    $err_r = 0.5 \cdot (err_q + err_f)$
11. Verringere den Quantisierungsfehler für alle Neuronen:
    $\Delta err_c = -\delta_{err} \cdot err_c$ für alle $c \in A$

Abb. 3.4: Der GNG-Algorithmus.

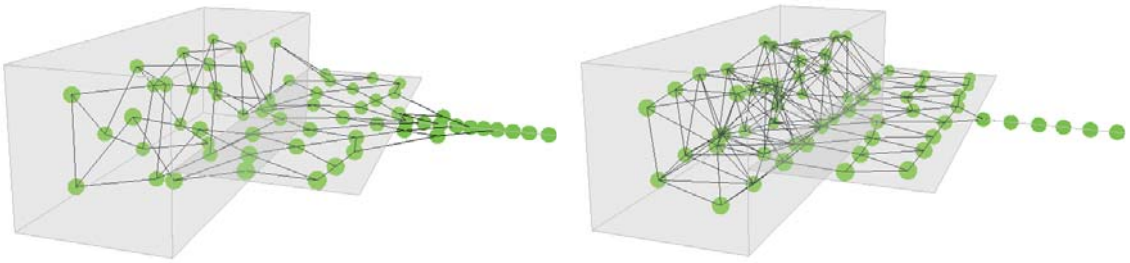
tet mit lediglich zwei Neuronen und fügt sukzessive neue Neuronen an Stellen ein, an denen der Quantisierungsfehler groß ist. Zu diesem Zweck erhält jedes Neuron c_i eine Fehlervariable err_i, mit der der lokale Quantisierungsfehler geschätzt wird. Der Hauptunterschied zum TRN besteht aber darin, dass die Anpassung der Neuronen auf der Topologie der Neuronenstruktur beruht. Diese Anpassung geschieht nämlich nicht mehr über eine Nachbarschaftsfunktion, die auf den Abständen im Eingaberaum basiert, sondern es werden das Gewinnerneuron und in schwächerem Maße alle Nachbarn an den Eingabevektor angepasst. Das hat auch den Vorteil, dass alle Parameter während des Trainings konstant bleiben können, was dazu führt, dass auch noch nicht vollständig trainierte Karten gute Ergebnisse hinsichtlich der Verteilung der Neuronen und der Topologie liefern. Beim TRN werden zunächst viele Neuronen sehr stark bewegt, was dazu führt, dass sie sich sehr stark in einer Region des Eingaberaums konzentrieren und dass sehr viele Verbindungen entstehen. Diese Situation löst sich erst gegen Ende des Trainings auf, wenn die Parameter weit genug reduziert sind. Beim GNG tritt dieses Problem nicht auf, da jeweils nur die Nachbarn mit angepasst werden und Neuronen erst nach und nach eingefügt werden (für diesbezügliche Simulationen siehe [Fritzke, 1995a,



a) b)

Abb. 3.5: a) Ausbreitung eines GNG im Einheitsquadrat bei gleichverteilten Eingabevektoren. b) Ausbreitung eines GNG bei lokal unterschiedlicher Verteilung der Eingabevektoren.

1996]). Gerade in Verbindung mit Reinforcement Learning ist das ein Vorteil, da sich ja die Verteilung und die Funktion, die approximiert werden soll, während des Trainings ständig ändern können und es daher äußerst schwierig wäre, die zeitliche Abnahme der Parameter richtig einzustellen. Zudem basieren ja die Schätzungen einzelner Werte auf der bereits gelernten Wertefunktion und erfordern somit schon während des Trainings ein möglichst großes Maß an Genauigkeit.
Der vollständige GNG-Algorithmus wird in Abbildung 3.4 skizziert [Fritzke, 1995a, 1996]. Die Parameter $0 \leq \alpha_b \leq 1$ und $0 \leq \alpha_n \leq 1$ sind die Lernraten für die Anpassung der Positionen des Gewinnerneurons und der benachbarten Neuronen. Neuronen werden alle N_{steps} Schritte eingefügt, bis eine maximale Anzahl von Neuronen N_{max} erreicht ist. Mit $0 \leq \delta_{err} \leq 1$ nimmt der akkumulierte lokale Quantisierungsfehler pro Schritt ab, was dafür sorgt, dass Fehler aus länger vergangenen Schritten weniger stark berücksichtigt werden. Der Parameter $0 \leq \delta_{new} \leq 1$ vermindert den lokalen Fehler, wenn ein neues Neuron eingefügt wird, um zu verhindern, dass das nächste Neuron wieder an derselben Stelle eingefügt wird.
In Abbildung 3.5 a) wird die Ausbreitung eines GNG bei gleichverteilten Eingabevektoren aus dem Einheitsquadrat $[0, 1]^2$ gezeigt. Man beachte, dass sich hierbei eine fast ausschließlich aus Dreiecken bestehende Struktur im Gegensatz zur festen Gitterstruktur der Kohonenkarte (Abbildung 3.1) bildet. Die entstehende Struktur entspricht nahezu der Delaunay-Triangulierung der Menge der Positionsvektoren. Abbildung 3.5 b) zeigt die Reaktion des GNG auf nicht-gleichverteilte Eingabevektoren. Wie in Abbildung 3.3 b) sind Eingabevektoren aus $[0, 0.5]^2$ etwa doppelt so häufig wie Eingabevektoren aus dem Rest des Einheitsquadrates. Genauso wie bei der Kohonenkarte konzentrieren sich auch hier die Neuronen in diesem Gebiet.
Einen Vergleich zwischen Kohonenkarte und GNG bei unterschiedlichen lokalen Dimensionen im Eingaberaum zeigt Abbildung 3.6. Der Eingaberaum besteht aus einem 3-dimensionalen Quader, einer 2-dimensionalen Ebene und einer 1-dimensionalen Geraden. Die Kohonenkarte konzentriert viele Neuronen auf der 1-dimensionalen Geraden, weil die 2-dimensionale Karte dort sozusagen „platt gedrückt“ werden muss. Die 2-dimensionale Ebene wird einigermaßen gut wiedergegeben, während der 3-dimensionale Quader äußerst dünn besetzt ist und sich die Karte falten muss, um die



a) b)

Abb. 3.6: a) Ausbreitung einer Kohonenkarte bei lokal unterschiedlich dimensionaler Eingabeverteilung. b) Ausbreitung eines GNG bei lokal unterschiedlich dimensionaler Eingabeverteilung.

Struktur wenigstens ungefähr wiederzugeben. So ist auch zu sehen, dass benachbarte Eingabevektoren keinesfalls immer auf benachbarte Neuronen abgebildet werden, da Verbindungen zwischen den entsprechenden Neuronen fehlen. Ganz anders beim GNG: Hier werden alle Strukturen gut wiedergegeben. Auf der 1-dimensionalen Geraden sammeln sich nicht zu viele Neuronen und die Verteilung der Neuronen auf der Ebene wirkt auch sehr regelmäßig. Die Kanten im Quader wirken in der 2-dimensionalen Abbildung vielleicht etwas verwirrend, was aber daher kommt, dass hier sehr viel mehr Kanten notwendig sind, um die Struktur wiederzugeben.
In [Bruske und Sommer, 1995] wird ein dem GNG sehr ähnlicher, ebenfalls auf den TRN basierender, Ansatz entwickelt. Bei den *Dynamic Cell Structures* geschieht die Anpassung der Neuronen ebenfalls über die Nachbarschaftsstruktur. Lediglich das Altern der Kanten und die Verteilung des Fehlers beim Einfügen neuer Neuronen wird etwas anders gehandhabt. In der Praxis dürfte das kaum einen Unterschied machen, da in beiden Fällen Kanten, die nicht mehr zur induzierten Delaunay-Triangulierung gehören, gelöscht werden.

3.3 Selbstorganisierende Karten und Funktionsapproximation

Als unüberwachte Lernverfahren bieten Selbstorganisierende Karten zunächst keine Möglichkeiten Eingabe-Ausgabe-Paare (x_t, y_t) zu lernen[10]. Sie müssen also geeignet erweitert werden, um für Funktionsapproximation eingesetzt werden zu können (vgl. Abschnitt 2.3). Grob lassen sich zwei unterschiedliche Ansätze unterscheiden: solche, die die Nachbarschaftsbeziehungen zwischen den Neuronen benutzen und solche Ansätze, die dies nicht tun.

[10] Im Folgenden werden wir der Einfachheit halber meist davon ausgehen, dass y_t reellwertig ist, dass also reellwertigen Funktionen approximiert werden müssen. Alle dargestellten Methoden können aber auf einfache Weise auf vektorwertige Funktionen erweitert werden.

3.3.1 Approximation ohne Berücksichtigung der Nachbarschaft

Der einfachste Weg, um Funktionen mit Hilfe Selbstorganisierender Karten zu approximieren ist, die Neuronen mit Werten v_i zu versehen und die Funktion konstant innerhalb der Voronoi-Zelle zu approximieren:

$$\widetilde{f}(x) = v_b \tag{3.6}$$

Trainiert werden können solche Netze bei einem Eingabepaar (x_t, y_t), indem der Wert v_b des Gewinnerneurons mit Lernrate α_v dem gewünschten Wert y_t angepasst wird:

$$\Delta v_b = \alpha_v (y_t - v_b) \tag{3.7}$$

Eine weitere Möglichkeit ist, die Funktion innerhalb der Voronoi-Zelle linear zu interpolieren (siehe z.B. [Martinetz u. a., 1993; Fritzke, 1995b]). Dabei wird in jedem Neuron zusätzlich zum Wert v_i eine lineare Abbildung L_i gespeichert und die Funktion durch

$$\widetilde{f}(x) = v_b + L_b(x - w_b) \tag{3.8}$$

approximiert[11]. Das Training eines solchen Netzes kann durch folgende Anpassungsregeln realisiert werden[12]:

$$\begin{aligned} \Delta v_b &= \alpha_v \left(y_t - \widetilde{f}(x) \right) \\ \Delta L_b &= \alpha_L \left(y_t - \widetilde{f}(x) \right) (x_t - w_b)^T \end{aligned} \tag{3.9}$$

Ein generelles Problem bei der Verwendung Selbstorganisierender Karten für Funktionsapproximation ist, dass sich die Verteilung der Neuronen nach der Häufigkeit der Eingabevektoren richtet und daher das eigentliche Ziel einer möglichst guten Approximation bei der Verteilung der Neuronen gar nicht berücksichtigt wird. So wäre es bei der stückweisen linearen Approximation wünschenswert in Gebieten, in denen sich die Zielfunktion annähernd linear verhält, wenige und in nicht-linearen Gebieten viele Neuronen zu haben. Daher schlägt [Fritzke, 1995b] vor, bei der Verwendung eines GNG zur Funktionsapproximation, dieses so zu verändern, dass der lokale Approximationsfehler anstelle des Quantisierungsfehlers das Einfügen neuer Neuronen regelt. Dazu werden die Fehlervariablen der Neuronen so geändert, dass sie sich nach dem Approximationsfehler richten:

$$\Delta err_b = \left\| y_t - \widetilde{f}(x_t) \right\|^2 \tag{3.10}$$

Neue Neuronen werden also in Gebieten eingefügt, in denen der Approximationsfehler groß ist. Auch in [Malcoci u. a., 1998] wird auf diese Weise vorgegangen. Dabei wird jedoch ein Neurales Gas mit lokal linearer Interpolation verwendet. Beim Training der v_i und der L_i, sowie bei der Akkumulation der Approximationsfehler werden die Neuronen nach ihrem Abstand zum Eingabevektor sortiert und eine auf dieser Reihenfolge basierende Nachbarschaftsfunktion benutzt. Im Gegensatz zu [Fritzke, 1995b] werden also stets die Fehlervariablen mehrerer Neuronen angepasst [13].

[11] Man beschreibt die Funktion sozusagen durch das Taylor-Polynom erster Ordnung, wobei L_b die Rolle der Jacoby-Matrix übernimmt, also möglichst gut die erste Ableitung im Punkt $x = w_b$ approximieren sollte.

[12] Durch diese Anpassungsregeln wird, wie übrigens auch bei Regel (3.7), ein Gradientenabstieg auf der Fehlerfunktion (vgl. Abschnitt 2.3) durchgeführt, wobei der Gradient eine sehr einfache Form annimmt.

[13] Ein weiterer Unterschied ist, dass sich an die Wachstumsphase eine Optimierungsphase anschließt, während der gegebenenfalls Neuronen auch wieder gelöscht werden können.

Eine weitere Methode zur Funktionsapproximation besteht darin, die Neuronen als Zentren für radiale Basisfunktionen zu verwenden [Fritzke, 1993, 1994b; Bruske und Sommer, 1995]. Die radialen Basisfunktionen haben ihr Maximum an der Stelle $x = w_i$ und gehen für wachsenden Abstand zu w_i gegen 0. Oft werden Gauß-Funktionen verwendet, um die Aktivierung eine Neurons zu bestimmen:

$$D_i(x) = e^{-\frac{\|x-w_i\|^2}{\sigma_i^2}} \tag{3.11}$$

Dabei regelt der Reichweitenparameter σ_i, wie stark das jeweilige Neuron auf weiter entfernte Eingabevektoren anspricht. Meist werden die Aktivierungen auch auf 1 normiert:

$$D_i(x) = \frac{e^{-\frac{\|x-w_i\|^2}{\sigma_i^2}}}{\sum_{j=1}^{N} e^{-\frac{\|x-w_j\|^2}{\sigma_j^2}}} \tag{3.12}$$

Bei einem Eingabevektor x werden die Aktivierungen der Neuronen berechnet, mit den Werten an den Neuronen multipliziert und aufaddiert:

$$\widetilde{f}(x) = \sum_{i=1}^{N} v_i D_i(x) \tag{3.13}$$

Das Training der Positionsvektoren findet entweder vor dem Training mit den Ausgabewerten oder gleichzeitig statt. Die Ausgabewerte der einzelnen Neuronen können wie bei der konstanten Approximation (siehe (3.7)) oder aber unter Berücksichtigung der Aktivierungen mit folgender Lernregel angepasst werden:

$$\Delta v_i = \alpha_v D_i(x_t) \left(y_t - \widetilde{f}(x_t) \right) \tag{3.14}$$

Die Reichweitenparameter σ_i können ebenfalls gelernt oder müssen vorab geeignet gesetzt werden. So wird in [Fritzke, 1994a] bei einem GNG die Reichweite einfach auf die durchschnittliche Länge aller von dem jeweiligen Neuron ausgehenden Kanten gesetzt. Auch bei diesen Verfahren bietet es sich an, beim Einfügen neuer Neuronen den Approximationsfehler und nicht den Quantisierungsfehler zu berücksichtigen. Es wird auch vorgeschlagen, neue Neuronen nicht in einem festen Rhythmus einzufügen, sondern den durchschnittlichen Fehler oder den Fehler auf einer Testmenge zu beobachten, um dann neue Neuronen einzufügen, wenn keine Verbesserungen mehr erreicht werden.
Ein Nachteil all dieser Methoden ist, dass der lokale Approximationsfehler nur beim Einfügen neuer Neuronen berücksichtigt wird und somit die Bewegung der Neuronen wieder dazu führt, dass sich die Positionen nach der Häufigkeitsverteilung der Eingabevektoren richten. Daher wird in [Bohn, 1997] vorgeschlagen, dieses Problem dadurch zu umgehen, dass dem Netz für schwierige Stellen häufiger Beispiele präsentiert werden. Da diese Methode etwas umständlich ist, schlägt [Aupetit u. a., 2000b] vor, einen zusätzlichen Parameter $\varepsilon(x)$ einzuführen, der die Schwierigkeit der Approximation an der Stelle x wiedergibt. Diese Schwierigkeit fließt mit in die Bewegung der Neuronen ein und „simuliert“ sozusagen eine größere Häufigkeit an den schwierigen Stellen. Als Basis dieses so genannten *Recruiting Neural Gas* wird ein Neurales Gas gewählt. Als Lernregel ergibt sich:

$$\Delta w_i = \alpha_w \varepsilon(x) h(k_i)(x - w_i) \tag{3.15}$$

Dabei ist $h(k)$ eine Nachbarschaftsfunktion, die sich nach dem auf den Entfernungen zum Eingabevektor basierenden Rang k des Neurons richtet. Da sich $\varepsilon(x)$ nicht direkt bestimmen lässt, erhält jedes Neuron einen zusätzlichen Parameter ε_i, der die geschätzte Fehlerdichte (durchschnittlicher quadratischer Fehler / approximierte Größe der Voronoi-Zelle) wiedergibt. Anstelle von $\varepsilon(x)$ wird dann einfach das ε_i des nächsten Neurons verwendet.
Ein anderer Weg wird in [Winter u. a., 2000] vorgeschlagen. Dort wird eine neue Fehlervariable für die Neuronen eingeführt, die den gewichteten Approximationsfehler des Gewinnerneurons wiedergibt und in jedem Schritt aktualisiert wird[14]:

$$\Delta err_b = \delta_{err} \cdot \left(\left| y - \widetilde{f}(x) \right| - err_b \right) \tag{3.16}$$

Die Lernrate δ_{err} sorgt dafür, dass länger vergangene Fehler weniger stark ins Gewicht fallen, $\widetilde{f}(x)$ ist der Wert der gegenwärtigen Approximation. Die Bewegung des Gewinnerneurons wird so abgeändert, dass es zusätzlich ein wenig in Richtung des Nachbarneurons mit dem größten Fehler bewegt wird, wenn dessen Fehler größer als der eigene ist. Die Anpassung des Gewinnerneurons sieht dann wie folgt aus:

$$\Delta w_b = \alpha_w \left(x - w_b \right) + \alpha_r \left(w_{b,i^*} - w_b \right) \frac{err_{b,i^*} - err_b}{err_{b,i^*}} \tag{3.17}$$

Dabei ist α_r die Lernrate für den fehlerabhängigen Teil der Bewegung und w_{b,i^*} der Positionsvektor des Nachbarneurons mit dem größten Fehler err_{b,i^*}. Ist der größte Fehler an den Nachbarneuronen kleiner als der Fehler am Gewinnerneuron, so wird die gewöhnliche Lernregel für die Anpassung des Gewinnerneurons (GNG-Algorithmus, Abb. 3.4, Schritt 6) verwendet. In [Winter u. a., 2000] wird für ein sehr einfaches Beispiel gezeigt, dass die so geänderte Lernregel tatsächlich zu einer Konzentration der Neuronen an der Stelle führt, an der die Funktion schwieriger zu approximieren ist, und somit die Approximationsgüte verbessert wird.

3.3.2 Approximation mit Berücksichtigung der Nachbarschaft

Ein Ansatz, Funktionen mit Selbstorganisierenden Karten unter Ausnutzung der Nachbarschaft zu approximieren, besteht darin, die Neuronen zunächst wie bei der konstanten Approximation zu trainieren (Lernregel (3.7)) und erst bei der Berechnung von Ausgabewerten zwischen den Werten einzelner Neuronen geeignet zu interpolieren. Insbesondere Göpperts *Interpolierende Selbstorganisierende Karte (Interpolating Self-Organizing Map, I-SOM)* [Göppert und Rosenstiel, 1997] und deren Erweiterungen sind hier zu nennen. Im Folgenden soll die Interpolierende Selbstorganisierende Karte etwas detaillierter vorgestellt werden, da einige der zugrunde liegenden Ideen die Basis für den von uns vorgeschlagenen Algorithmus in Kapitel 4 sind.
Beim I-SOM-Ansatz sind die Neuronen mit Ausgabewerten v_i versehen und können wie bei der konstanten Approximation trainiert werden (Lernregel (3.7))[15]. Erst bei der Berechnung von Ausgabewerten kommt die Interpolation ins Spiel. Generell geht es immer

[14] Es ist nicht klar, ob diese Fehlervariable den herkömmlichen Quantisierungsfehler ersetzt, oder ob dieser noch für das Einfügen von Neuronen verwendet wird. In der Gleichung wird dieselbe Fehlervariable err_i für den gewichteten Approximationsfehler verwendet, weil das später in unserem Verfahren genauso gehandhabt wird.

[15] In einigen der frühen Arbeiten scheint es allerdings so zu sein, dass die SOM nicht nur mit den Werten x_t trainiert wird, sondern ein zusammengesetzter Vektor $(x_t, y_t)^T$ gebildet und das Netz damit trainiert wird [Göppert und Rosenstiel, 1993a,b, 1995c]. Leider geht Göppert auf diesen Punkt nicht genauer ein.

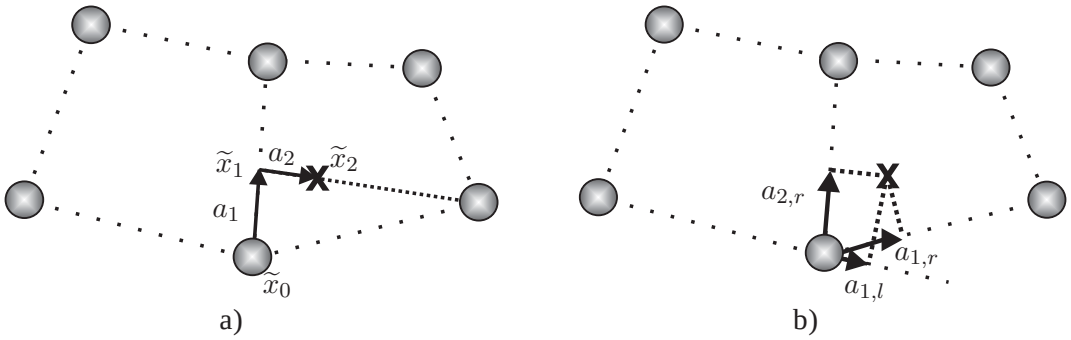


Abb. 3.7: a) Geometrische Interpolation bei Göppert. b) Topologische Interpolation bei Göppert.

darum, den Eingabevektor x_t durch die Summe aus dem Positionsvektor des nächsten Neurons w_b und einiger Differenzvektoren $(w_i - w_b)_{i=1,\ldots,k}$ (bzw. bei der geometrischen Interpolation $(w_i - \widetilde{x}_{i-1})_{i=1,\ldots,k}$, s.u.) zu approximieren:

$$x_t \approx \widetilde{x}_t = w_b + \sum_{i=1}^{k} a_i (w_i - w_b) \tag{3.18}$$

Hat man die entsprechenden Koeffizienten a_i bestimmt, so ergibt sich der interpolierte Wert ganz analog als:

$$\widetilde{f}(x_t) = v_b + \sum_{i=1}^{k} a_i (v_i - v_b) \tag{3.19}$$

Vorläufer der eigentlichen I-SOM finden sich in [Göppert und Rosenstiel, 1993a,b]. Dort werden zwei Interpolationmethoden vorgestellt, die auf geometrischer Information im Eingaberaum bzw. auf der topologischen Information der Neuronenstruktur beruhen. Bei der Interpolationsmethode, die auf geometrischer Information beruht, werden einfach die k nächsten Nachbarn im Eingaberaum ausgewählt (die topologische Struktur der Neuronen wird also nicht beachtet). Die a_i werden durch eine Folge von Approximationen $\widetilde{x}_i, i = 0, \ldots k$ dadurch bestimmt, dass $\widetilde{x}_0 = w_b$ gesetzt wird und dann jeweils das Residuum $(x - \widetilde{x}_{i-1})$ auf den Differenzvektor $(w_i - \widetilde{x}_{i-1})_{i=1,\ldots,k}$ zur gegenwärtigen Approximation projiziert wird (vgl. Abbildung 3.7 a)):

$$\begin{aligned} a_i &= \frac{(x - \widetilde{x}_{i-1})^T (w_i - \widetilde{x}_{i-1})}{\|w_i - \widetilde{x}_{i-1}\|^2} \\ \widetilde{x}_i &= \widetilde{x}_{i-1} + a_i (w_i - \widetilde{x}_{i-1}) \end{aligned} \tag{3.20}$$

Der Ausgabewert bei der geometrischen Interpolation ergibt sich dann aus der Folge:

$$\begin{aligned} \widetilde{y}_0 &= v_b \\ \widetilde{y}_i &= \widetilde{y}_{i-1} + a_i (v_i - \widetilde{y}_{i-1}) \end{aligned} \tag{3.21}$$

Ein Merkmal der geometrischen Interpolation ist, dass Abstände im Eingaberaum benutzt werden, d.h. es kann vorkommen, dass zwischen Neuronen interpoliert wird, die in der Neuronenstruktur weit auseinander liegen.
Bei der topologischen Interpolation werden nur direkte Nachbarn des Gewinnerneurons einbezogen. Dabei geht Göppert implizit davon aus, dass jedes Neuron in jeder Dimension ein oder zwei Nachbarn besitzt. In diesem Fall bezeichnet $a_{d,l}$ bzw. $a_{d,r}$ das zum linken bzw. rechten Nachbarn in der Dimension d gehörende a_i. Diese werden dann berechnet,

indem das Residuum $x - w_b$ auf die entsprechenden Differenzvektoren $(w_i - w_b)_{i=1,\ldots,k}$ projiziert wird (vgl. Abbildung 3.7 b)):

$$\begin{aligned} a_{d,l} &= \frac{(x - w_b)^T (w_{d,l} - w_b)}{\|w_{d,l} - w_b\|^2} \\ a_{d,r} &= \frac{(x - w_b)^T (w_{d,r} - w_b)}{\|w_{d,r} - w_b\|^2} \end{aligned} \tag{3.22}$$

Bei der Interpolation werden die Beiträge der rechten und der linken Nachbarn gemittelt:

$$\widetilde{f}(x_t) = v_b + \sum_{d=1}^{D} \frac{a_{d,l}(v_{d,l} - v_b) + a_{d,r}(v_{d,r} - v_b)}{k_d} \tag{3.23}$$

Hierbei bezeichnet k_d die Anzahl der Nachbarn in der jeweiligen Dimension, also eins, wenn das Neuron in der entsprechenden Dimension am Rand liegt, ansonsten zwei. In [Göppert und Rosenstiel, 1993a] werden beide Methoden mit der konstanten Approximation, distanz-basierter Interpolation[16] und Backpropagation-Netzen für einen Anwendungsfall verglichen. Dabei schneiden geometrische und topologische Interpolation am besten, die distanz-basierte und die konstante Approximation am schlechtesten ab.
Beim eigentlichen I-SOM-Algorithmus besteht die grundsätzliche Idee darin, ein lokales Koordinatensystem mit der Position des Gewinnerneurons als Ursprung und den Differenzvektoren zu einigen der benachbarten Neuronen als Achsen zu bilden[17]. Wie bei der topologischen Interpolation geht Göppert davon aus, dass jedes Neuron maximal zwei Nachbarn in jeder Dimension besitzt. Diesmal wird jedoch nur das zum Eingabevektor näher liegende Nachbarneuron für die Interpolation benutzt. Insgesamt werden also $d + 1$ Neuronen (das Gewinnerneuron und je ein Nachbarneuronen für jede Dimension) ausgewählt. Die Differenzvektoren $l_i = w_i - w_b, i = 1, \ldots, d$ werden als Koordinatenachsen gewählt (vgl. Abbildung 3.8). Dabei wird davon ausgegangen, dass die l_i linear unabhängig (aber nicht notwendigerweise orthogonal) sind. Die l_i werden in einer Matrix L zusammengefasst. Die affinen Koordinaten können dann mit Hilfe der Pseudo-Inversen bestimmt werden[18]:

$$a = (L^T L + \mu I)^{-1} L^T (x - w_b) \tag{3.24}$$

Hierbei bezeichnet I die Einheitsmatrix und $0 \leq \mu \ll 1$ einen Regularisierungsparameter, um Probleme mit fast singulären Matrizen zu vermeiden. Der interpolierte Wert wird dann gemäß Formel (3.19) berechnet. In [Göppert und Rosenstiel, 1995a] wird eine iterative Methode vorgeschlagen, um die Matrixinversion zu umgehen. Die Bedingung, dass maximal 2 Nachbarn pro Dimension vorhanden sein dürfen, wird in [Göppert und Rosenstiel, 1995b] etwas gelockert. Ist die Anzahl der ausgewählten Nachbarn niedriger als die Dimension des Eingaberaums, so wird über die quasi-inverse Matrix ähnlich wie in Gleichung (3.24) die Lösung mit dem kleinsten Abstand zum Eingabevektor gewählt. Ist die Anzahl der ausgewählten Nachbarn größer als die Dimension, so schlägt Göppert vor, die iterative Methode zu verwenden. Beim Vergleich mit Radial-Basisfunktions-Netzen und SOM mit lokal linearer Approximation zeigt sich diese Methode unempfindlicher gegenüber verrauschten Eingabedaten und der Auswahl der Parameter [Göppert und Rosenstiel, 1995b].

[16] Bei der distanz-basierten Interpolation werden nur die Abstände zum Eingabevektor im Eingaberaum berücksichtigt und die Ausgabewerte der einzelnen Neuronen entsprechend gewichtet, d.h. Richtungen spielen keine Rolle

[17] Die Darstellung des I-SOM-Algorithmus folgt vor allem [Göppert und Rosenstiel, 1997]. Eingeführt wurde der I-SOM-Algorithmus in [Göppert und Rosenstiel, 1995a,b,c].

[18] Zur Herleitung siehe [Göppert und Rosenstiel, 1995c].

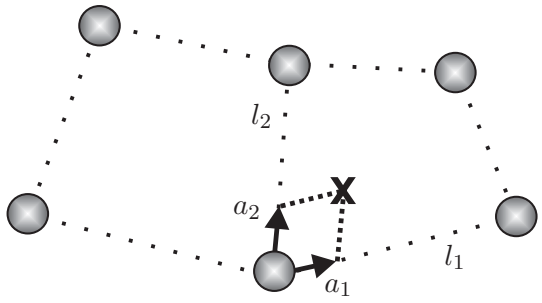


Abb. 3.8: Bildung eines lokalen Koordinatensystems bei I-SOM.

Bei der I-SOM führen wechselnde Gewinnerneuronen oder wechselnde Nachbarneuronen zu Unstetigkeitsstellen. Diese Unstetigkeiten versucht die CI-SOM (*Continuous Interpolating Self Organizing Map*) durch Gewichtung mehrerer linearer Interpolationen zu vermeiden [Göppert und Rosenstiel, 1997]. Um Unstetigkeiten bei wechselnden Nachbarneuronen zu vermeiden, werden die Differenzvektoren l_i durch eine gewichtete Summe der Differenzvektoren beider Nachbarn der jeweiligen Dimension ersetzt. Dabei werden die Beiträge der einzelnen Vektoren invers zu der Entfernung des jeweiligen Neurons zum Eingabevektor gewichtet. Um Unstetigkeiten bei wechselnden Gewinnerneuronen zu vermeiden, werden lokale Koordinatensysteme mit verschiedenen Neuronen als Ursprung erzeugt und jeweils eigene Ausgabewerte berechnet. Die verschiedenen Ausgaben werden ebenfalls nach der Distanz zum Eingabevektor gewichtet.
Mit dem Problem kontinuierlicher Approximation in Selbstorganisierenden Karten hat sich auch [Aupetit u. a., 1999a,b] beschäftigt. Die *Continuous Self-Organizing Map* (C-SOM) benutzt die Gradienteninformationen der lokal linearen Abbildungen (siehe Gleichung (3.8)) für kubische Spline-Interpolation zwischen benachbarten Neuronen. Dieser Ansatz liefert für 1-dimensionale Karten gute Ergebnisse, lässt sich aber nicht auf mehrdimensionale Karten übertragen. In späteren Arbeiten werden daher Techniken vorgeschlagen, die ebenfalls auf invers gewichteten Distanzen zu einzelnen Neuronen beruhen [Aupetit u. a., 2000a]. Im Gegensatz zu Göppert verwendet Aupetit lokal lineare Abbildungen schon beim Training (vgl. Gleichung (3.9)). Als Alternative zu den invers gewichteten Distanzen werden auch so genannte *Neighbouring Influence Kernels* zur Berechnung der Aktivierung einzelner Neuronen verwendet. Dabei ist die Aktivierung in einer bestimmten Region um ein Neuron herum 1 und nimmt dann mit wachsender Entfernung ab. Insgesamt jedoch sind Aupetits Methoden entweder ungeeignet für höhere Eingabedimensionen oder erscheinen als zu kompliziert.
Auch Göpperts Methoden scheinen zunächst nur für gewisse SOM, die bestimmten Anforderungen an die Neuronentopologie genügen, geeignet. Diese Anforderungen sind jedoch gerade beim GNG in der Regel nicht erfüllt. Zudem unterscheidet er zwischen einer Anzahl verschiedener Fälle und schlägt jeweils unterschiedliche Lösungen vor. Wie jedoch im nächsten Kapitel gezeigt werden wird, lässt sich auch beim GNG problemlos zwischen Neuronen interpolieren. Die vorgeschlagenen Methoden werden dabei so weit vereinheitlicht, dass eine einzige Methode in allen Fällen ausreicht. Außerdem wird die Interpolation zwischen den einzelnen Neuronen schon beim Training berücksichtigt werden. Auch die Idee durch Verwendung mehrerer Interpolationen an benachbarten Neuronen und einer Gewichtung, ähnlich wie bei Aupetits Neighbouring Influence Kernels, eine kontinuierlichere Approximation zu erreichen, wird dort, wenn auch in anderer Form, wieder aufgegriffen werden.

3.4 Selbstorganisierende Karten und Reinforcement Learning

Es gibt bereits einige Arbeiten, die sich mit dem Einsatz von Selbstorganisierenden Karten im Reinforcement Learning beschäftigen. Meist werden Selbstorganisierende Karten einfach dazu benutzt, den Zustandsraum adaptiv zu diskretisieren, um dann die üblichen Reinforcement-Learning-Algorithmen, meist Q-Learning, für diskrete Zustandsräume anzuwenden, wobei eventuell noch kleinere Modifikationen durchgeführt werden. Bei einigen Ansätzen werden Selbstorganisierende Karten auch dazu benutzt, kontinuierliche Aktionen zu lernen. Im Folgenden soll ein Überblick über die verschiedenen bestehenden Ansätze gegeben werden, um dann deutlich zu machen, worin sich der in der vorliegenden Arbeit entwickelte Ansatz von bereits bestehenden Ansätzen unterscheidet.

Einen ersten Ansatz, Aktionen mit Selbstorganisierenden Karten zu lernen, stellte die *motorische Karte* [Ritter und Schulten, 1986; Ritter u. a., 1990], eine Erweiterung der Kohonenkarte (siehe Abschnitt 3.1), dar. Dabei wird an jedem Neuron c_i zusätzlich ein Aktionswert a_i gespeichert. Die Bewegung der Neuronen erfolgt gemäß der Kohonen-Lernregel (3.4), die Aktionswerte werden analog trainiert, wobei die gewünschte Aktion explizit vorgegeben werden muss. Dabei wird nicht nur der Aktionswert des Gewinnerneurons trainiert, sondern auch Aktionswerte benachbarter Neuronen. Die Idee dahinter ist, dass ähnliche Eingabewerte wahrscheinlich ähnliche Aktionen erfordern. So findet eine Art Generalisierung statt. Im Laufe des Trainings nimmt die Größe der Nachbarschaft ab, so dass sich die Neuronen dann stärker spezialisieren. Bei der Auswertung wird nur das Gewinnerneuron c_b berücksichtigt und dessen Aktion a_b zurückgegeben. Da die gewünschten Aktionen vorgegeben werden müssen, handelt es sich bei solch einer motorischen Karte um ein überwachtes Lernverfahren. Es gibt aber auch eine Version der motorischen Karte, die eine Bewertungsfunktion anstelle von vorgegebenen Aktionen benutzt. Dabei wird die Aktion des Gewinnerneurons jeweils zufällig leicht abgewandelt. Am Ende eines jeden Zeitschrittes wird dann geprüft, ob die aktuelle Bewertung besser ist als der Durchschnitt aller Bewertungen, die das Neuron bisher erhalten hat. Ist dies der Fall, so werden dieselben Lernregeln wie im überwachten Fall ausgeführt. So lernt die Karte nur aus Aktionen, die besser als die aktuellen sind.

Die motorische Karte ist jedoch nicht in der Lage, mit zeitverzögerten Belohnungen umzugehen, wie sie für viele Reinforcement-Learning-Probleme typisch sind. Ein weiteres Problem, das sich bei mehrdimensionalen Aktionen stark verschärft, besteht darin, dass die Suche nach besseren Aktionen durch ungezieltes Ausprobieren geschieht. Dieses Problem versucht *Covariance Learning* [Wedel und Polani, 1996] zu lösen, bei dem der Aktionsraum systematisch durch Testaktionen untersucht wird. Die dabei erhaltenen Belohnungen werden dazu benutzt, die Parameter einer multi-dimensionalen Normalverteilung zu schätzen, die die Belohnungsfunktion in der Umgebung eines Neurons aktionsabhängig approximiert. Damit kann dann die Richtung bestimmt werden, in die die gegenwärtige Aktion verschoben werden soll, um eine höhere Belohnung zu erhalten. Auch Covariance Learning ist auf direkte Belohnungen angewiesen. Zudem ist es recht aufwändig, da jeweils eine ganze Reihe von Testaktionen ausgeführt werden muss, bevor die Aktion verbessert werden kann.

Die *Q-KOHON*-Methode [Touzet, 1997] zum Lernen kontinuierlicher Aktionen verbindet die Kohonenkarte mit Q-Learning. Die Eingabevektoren für die Kohonenkarte bestehen dabei aus Tupeln (Zustand, Aktion, Q-Wert der Aktion). Q-Werte können höchstens einen maximalen Wert von 1 annehmen. Um eine Aktion für einen Zustand zu bestimmen, wird das Gewinnerneuron für einen Eingabevektor (Zustand, * , 1) bestimmt, d.h. die Aktionswerte werden bei der Abstandsberechnung in der Kohonenkarte nicht berück-

sichtigt. Es wird also ein Neuron mit einer ähnlichen Situation und einem möglichst hohen Q-Wert gesucht. Die Aktion dieses Neurons wird dann noch zufällig leicht modifiziert und ausgeführt. Beim Training geht bei der Bestimmung des Gewinnerneurons nur die Situation und die ausgeführte Aktion ein. Der Q-Wert wird gemäß dem Q-Learning-Algorithmus bestimmt und das Gewinnerneuron und seine Nachbarn werden mit dem kompletten Eingabevektor trainiert. Q-KOHON wird erfolgreich zum Lernen eines Hindernisvermeidungs-Verhaltens für einen Khepera-Miniatur-Roboter eingesetzt. Ein Problem dieses Ansatzes liegt sicherlich in der Verwendung einer Kohonenkarte, die durch ihre vorgegebene Struktur bei höher-dimensionalen Eingaberäumen nicht unbedingt in der Lage ist, diese wirklich ähnlichkeitserhaltend abzubilden. Ein anderes Problem besteht darin, dass pro Neuron jeweils nur eine Aktion und ein Q-Wert gespeichert wird, was bei komplizierteren Politiken eine hohe Anzahl von Neuronen nötig macht.
In [Smith, 2001, 2002] werden Kohonenkarten nicht nur zur Diskretisierung des Eingaberaums, sondern auch zur Diskretisierung des Aktionsraums benutzt. Dabei wird eine zusätzliche Kohonenkarte dazu verwendet, mögliche Aktionswerte abzubilden. Zur Exploration werden die Aktionswerte zufällig ein wenig variiert. Die Q-Werte beziehen sich dabei auf Paare von Neuronen, je eines aus jeder Karte, und werden in Form einer Tabelle gespeichert. Bei der Anpassung der Q-Werte werden auch „benachbarte“ Zustände und Aktionen berücksichtigt. Probleme können wieder durch die Verwendung von Kohonenkarten bei höher-dimensionalen Eingaberäumen entstehen. Zudem besteht die Gefahr, dass sich während einer Episode die Karten so stark verändern, dass bereits Gelerntes wieder verlernt wird.
In [Großmann und Poli, 2000; Großmann, 2001] werden *Constructive Cell Structures* dazu benutzt, den Zustandsraum adaptiv zu diskretisieren. Constructive Cell Structures sind abgewandelte *Growing Cell Structures* [Fritzke, 1992, 1993] und bestehen wie diese aus k-dimensionalen Simplizes. Die Dimension dieser Simplizes muss vorab vorgegeben werden. Demnach ist auch keine Anpassung an lokal unterschiedlich-dimensionale Eingabeverteilungen möglich. Zum Lernen werden in [Großmann und Poli, 2000; Großmann, 2001] zunächst die Zustandsübergänge und erhaltenen Belohnungen beobachtet und gespeichert. Die kontinuierlichen Zustände werden dabei durch die Neuronen kodiert, die durch sie aktiviert werden. Die Q-Werte werden dann durch Werte-Iteration (siehe 2.1.3) neu berechnet, wobei die gespeicherten Zustandsübergänge und Belohnungen dazu benutzt werden, die Übergangswahrscheinlichkeiten und die Erwartungswerte für die Belohnungen zu schätzen. Dabei spielt die Topologie der Selbstorganisierenden Karte keine Rolle. Von Zeit zu Zeit wird getestet, ob neue Neuronen eingefügt werden sollen, um das Netz weiter zu verfeinern. Das Verfahren wird benutzt, um diskrete Aktionen für einen Roboter zu lernen, der Ziele in einer durch Wände unregelmäßige begrenzten Arena anfahren muss.
Neurale Gase und so genannte *Neural Fields* werden bei [Gross u. a., 1997] zur Kodierung von kontinuierlichen Zuständen und zum Lernen von kontinuierlichen Aktionen verwendet. Das Ergebnis scheint ein ähnlicher Ansatz zu sein, wie bei RBF-Netzen[19], wobei sich die Neuronen während des Trainings noch bewegen können. Das Verfahren wird dazu benutzt, ein Andockmanöver für einen mit einer Kamera ausgerüsteten Khepera-Miniatur-Roboter zu lernen. Bei der Berechnung von Q-Werten und der Auswahl der auszuführenden Aktion gehen dabei die distanzabhängigen Aktivierungen mehrerer Neuronen ein. Nachbarschaftsbeziehungen zwischen Neuronen spielen jedoch keine Rolle, da Neurale Gase keine Kanten besitzen und daher auch keine Nachbarschaftsbeziehungen kennen.

[19] Die Darstellung des Algorithmus in [Gross u. a., 1997] ist leider etwas undurchsichtig.

Es gibt eine ganze Reihe von Arbeiten, in denen Selbstorganisierende Karten im Wesentlichen dazu benutzt werden, den Zustandsraum zu diskretisieren, um dann auf diesen diskreten Zuständen die üblichen Reinforcement-Learning-Algorithmen einzusetzen. Dabei kommen, teilweise leicht modifiziert, Kohonenkarten [Barreca und Buttazzo, 1993; Gross u. a., 1996], Neurale Gase [Herrmann und Der, 1995; Gross u. a., 1996; Stephan u. a., 2003] oder auch Wachsende Neurale Gase [Millán u. a., 2002; Stephan u. a., 2003; Provost u. a., 2004] zum Einsatz. Zum Teil werden dabei auch die Kriterien, nach denen Neuronen verteilt werden, angepasst [Herrmann und Der, 1995; Finton und Yu, 1994]. Das Training der zugrunde liegenden Selbstorganisierenden Karte findet immer parallel zum Lernen der Aktionen durch Reinforcement Learning statt. Bei längeren Episoden kann das dazu führen, dass durch die Bewegung der Neuronen bereits Gelerntes wieder vergessen wird. Sollen kontinuierliche Aktionen benutzt werden, können einfach verschiedene Aktionswerte miteinander kombiniert werden, indem sie gemäß ihren Q-Werten gewichtet werden [Millán u. a., 2002]. Bei all diesen Ansätzen ist bei der Aktionsbestimmung allein das Gewinnerneuron ausschlaggebend. Benachbarte Neuronen gehen nicht in die Approximation ein.
Alle im weiteren Verlauf der vorliegenden Arbeit entwickelten Verfahren werden sich jeweils zumindest in einigen der folgenden Punkte von den bisherigen Ansätzen unterscheiden:

- Berücksichtigung der topologischen Nachbarschaft in der Neuronenstruktur beim Training und bei der Bestimmung von Aktionen
  Bei der Approximation der Q-Werte und der Bestimmung der auszuführenden Aktion, spielt bei allen betrachteten Verfahren die topologische Nachbarschaft in der jeweiligen Neuronenstruktur keine Rolle[20]. Die Approximation entspricht dabei vom Prinzip her der einfachen konstanten Approximation aus Abschnitt 3.3.1. Einzige Ausnahme ist das Verfahren in [Gross u. a., 1997], wo zumindest die Distanz zum Eingabevektor im Eingaberaum mit eingeht[21]. Bei den im weiteren Verlauf der vorliegenden Arbeit entwickelten Verfahren, wird die Topologie der Neuronenstruktur explizit in die Approximation mit einfließen. Dies wird durch lokal lineare Interpolation zwischen benachbarten Neuronen, durch lokale Gewichtung benachbarter Neuronen, oder durch eine Kombination aus beidem verwirklicht.

- Entwicklung eines allgemein einsetzbaren Funktionsapproximators
  Bei keinem der betrachteten Verfahren wird zunächst das allgemeine Problem der Funktionsapproximation betrachtet. Dementsprechend sind die Verfahren auch immer an einen bestimmten Reinforcement-Learning-Algorithmus gebunden. In der vorliegenden Arbeit wird zunächst das Problem der Funktionsapproximation gesondert betrachtet, wobei jedoch die besonderen Anforderungen, die allgemein beim Lernen von Aktionen zu berücksichtigen sind, in die Entwicklung des Funktionsapproximators mit eingehen. Dieser allgemeine Funktionsapproximator kann dann flexibel in verschiedenen Reinforcement-Learning-Architekturen zum Lernen diskreter und kontinuierlicher Aktionen eingesetzt werden.

- Anpassung an die lokale Dimension der Eingabeverteilung
  Bei vielen der bisherigen Ansätzen wurden Selbstorganisierende Karten mit einer explizit vorgegebenen Dimension benutzt (Kohonenkarte bzw. Constructive Cell

[20] Teilweise werden topologische Nachbarschaften in der Neuronenstruktur aber beim Training berücksichtigt.
[21] Da ein Neurales Gas benutzt wird, gibt es dort keine topologische Nachbarschaft zwischen Neuronen.

Structures). Dabei kann sich die Karte nicht an eventuell unterschiedlich dimensionale lokale Strukturen anpassen. Im Gegensatz dazu wird in den von uns entwickelten Verfahren ein Wachsendes Neurales Gas verwendet, dass zu solch einer Anpassung fähig ist.

- Berücksichtigung der besonderen Reihenfolge der Eingabedaten beim Lernen von Aktionen
  Beim Lernen von Aktionen haben die Eingabedaten eine ganz bestimmte Reihenfolge, da sie aus Trajektorien durch den Zustandsraum stammen. Wird dies bei der Anpassung der Neuronenpositionen nicht berücksichtigt, besteht die Gefahr, bereits Gelerntes durch die Bewegung der Neuronen wieder zu verlernen. Dies wird bisher nur bei den Constructive Cell Structures in [Großmann und Poli, 2000; Großmann, 2001] beachtet. In der vorliegenden Arbeit wird eine Version des GNG-Algorithmus vorgestellt, der ebenfalls die besondere Reihenfolge der Eingabedaten beim Lernen von Aktionen berücksichtigt.

- Lernen tatsächlich kontinuierlicher Aktionen
  Ein Vorteil von kontinuierlichen Aktionen ist, dass auf ähnliche Zustände mit ähnlichen Aktionen reagiert werden kann. Mit Ausnahme von [Gross u. a., 1997] lernen alle bisherigen Ansätze auf der Basis von Selbstorganisierenden Karten keine kontinuierlichen Aktionen in diesem Sinne. Zwar können sich die benutzten Aktionswerte während des Trainings ändern, aber die zurückgegebenen Aktionen sind stets für eine ganze Region im Eingaberaum dieselben, da jeweils nur das Gewinnerneuron den Ausschlag gibt. Im Gegensatz dazu wird das in der vorliegenden Arbeit entwickelte Verfahren ReinforceLWIGNG tatsächlich kontinuierliche Aktionen lernen und auf leicht veränderte Zustände mit leicht abgewandelten Aktionen reagieren.

Im nächsten Kapitel wird zunächst ein allgemeiner Funktionsapproximator auf Grundlage eines Wachsenden Neuralen Gases entwickelt, der die topologische Nachbarschaft in der Neuronenstruktur berücksichtigt. Dieser allgemeine Funktionsapproximator wird dann im weiteren Verlauf der Arbeit zunächst mit Q-Learning kombiniert, um diskrete Aktionen zu lernen (Kapitel 5). Anschließend wird er aber auch in einer Actor-Critic-Architektur zum Lernen tatsächlich kontinuierlicher Aktionen eingesetzt und dabei sowohl eine Zustands-Wertefunktion als auch eine Politik approximieren (Kapitel 6). Das ist unseres Wissens nach das erste Mal, dass eine auf einer Selbstorganisierenden Karte basierende Methode auf diese Weise eingesetzt wird.

Kapitel 4

Lokal Gewichtetes Interpolierendes Wachsendes Neurales Gas

In diesem Kapitel soll ein neu entwickelter Ansatz zur Funktionsapproximation auf der Basis eines Wachsenden Neuralen Gases schrittweise hergeleitet werden. Werden Funktionsapproximatoren beim Lernen von Aktionen eingesetzt, besteht das Problem, dass sich die Verteilung der Eingabevektoren und die Funktion, die gelernt werden soll, im Laufe des Trainings stark verändern können. Der entwickelte Funktionsapproximator soll auf diese Probleme eingehen und auch in hoch-dimensionalen Räumen mit möglicherweise irrelevanten oder redundanten Eingabedimensionen anwendbar sein.
Betrachtet man hoch-dimensionale Lernprobleme genauer, kann man oft feststellen, dass die lokale Dimension der Eingabedaten erheblich niedriger ist, als die globale Dimension (vgl. etwa [Vijayakumar u. a., 2005]). Dies kommt daher, dass die Eingabegrößen miteinander korreliert sind und daher viele Kombinationen von Eingabewerten gar nicht auftreten. Um diese Eigenschaft auszunutzen, soll ein Wachsendes Neurales Gas (siehe Abschnitt 3.2) als Basis für den Funktionsapproximator dienen. Ein entscheidender Vorteil ist dabei, dass das Neurale Gas in der Lage ist, sich der lokalen Dimension der Eingabeverteilung anzupassen. Die Neuronen werden dann als Zentren für eine lokal lineare Interpolation genutzt. Dabei wird die von [Göppert und Rosenstiel, 1997] vorgeschlagene Methode der Interpolierenden Selbstorganisierenden Karte (siehe Abschnitt 3.3.2) erweitert und vereinheitlicht. Darüber hinaus werden Techniken wie *Locally Weighted Learning*, d.h. die gewichtete Kombination mehrerer einfacher lokaler Modelle [Atkeson u. a., 1997], eingesetzt, um die Approximationsgüte weiter zu verbessern. Da dabei lokale Modelle durch Interpolation zwischen benachbarten Neuronen gebildet werden und aus diesen Modellen durch Gewichtung der einzelnen Ausgabewerte die endgültige Ausgabe bestimmt wird, soll der Ansatz als *Lokal Gewichtetes Interpolierendes Wachsendes Neurales Gas (Locally Weighted Interpolating Growing Neural Gas, LWIGNG)* bezeichnet werden. Dieser Ansatz wird mit *Locally Weighted Projection Regression* (LWPR) [Vijayakumar u. a., 2005], einem state-of-the-art Algorithmus für inkrementelle, nicht-lineare Funktionsapproximation, verglichen. LWIGNG wird die Grundlage des Funktionsapproximators für die Reinforcement-Learning-Algorithmen der nächsten Kapitel bilden.

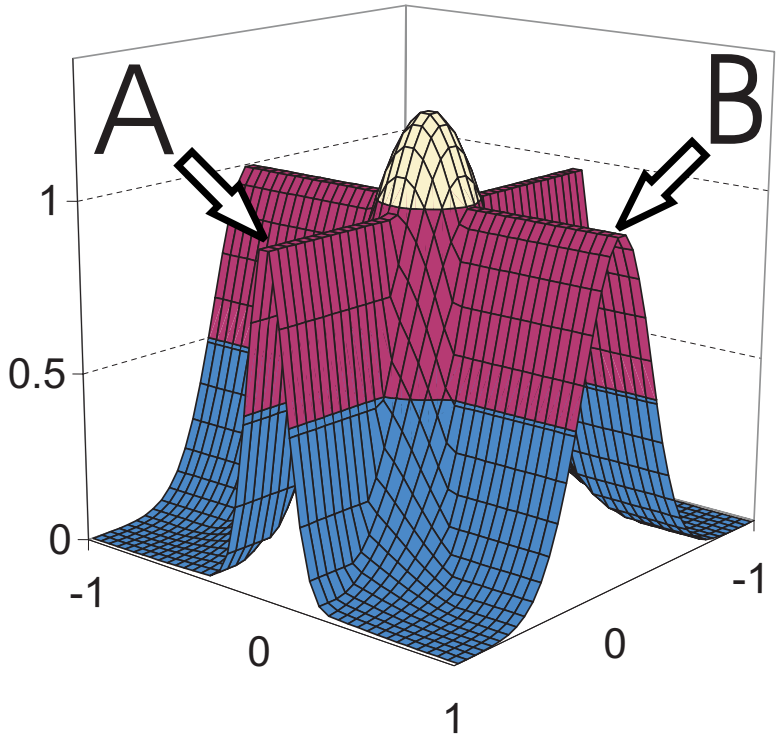


Abb. 4.1: Die zu approximierende Testfunktion.

4.1 Wachsendes Neurales Gas mit konstanter Approximation

Das Verfahren soll im Folgenden schrittweise entwickelt und anhand einer Testfunktion demonstriert werden. Als Testfunktion wird die 2-dimensionale Funktion

$$f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \max\left\{e^{-10x_1^2}, e^{-50x_2^2}, 1.25e^{-5(x_1^2+x_2^2)}\right\} \tag{4.1}$$

verwendet. Beim Training wird zusätzlich ein normalverteiltes Rauschen von $N(0, 0.01)$ zum Funktionswert addiert. Diese Funktion wurde auch in [Vijayakumar u. a., 2005] verwendet, um LWPR zu evaluieren. Am Ende dieses Kapitels wird sie dazu benutzt, um LWIGNG mit LWPR zu vergleichen. Wie man in Abbildung 4.1 sieht, gibt es Gebiete, in denen die Funktion annähernd konstant ist und andere Gebiete, in denen sie sich rasch ändert. Man beachte auch, dass die Funktion an der Stelle A „spitzer“ verläuft als an Stelle B. Es gibt also Stellen, an denen die Funktion einfacher und Stellen, an denen die Funktion schwerer zu approximieren sein sollte.
Um die Güte der Approximation zu quantifizieren, wird der *normalisierte mittlere quadratische Fehler (normalized Mean Square Error, nMSE)* benutzt, der auch üblicherweise bei der Beurteilung der Güte von Neuronalen Netzen verwendet wird. Da dabei das Quadrat der Abweichung vom Sollwert eingeht, werden größere Abweichungen stärker gewichtet als kleinere. Der nMSE wird berechnet, indem der mittlere quadratische Fehler durch die Varianz in der Stichprobenmenge geteilt wird:

$$\text{nMSE} = \frac{\text{MSE}}{\text{Var}\,(y)} = \frac{\sum_{i=1}^{N}\left(\widetilde{f}\,(x_i) - y_i\right)^2}{\sum_{i=1}^{N}\left(\bar{y} - y_i\right)^2} \tag{4.2}$$

Im Gegensatz zum gewöhnlichen mittleren quadratischen Fehler (MSE) wird damit die „Schwierigkeit“ der Approximation mit in Betracht gezogen. Wird die Funktion einfach

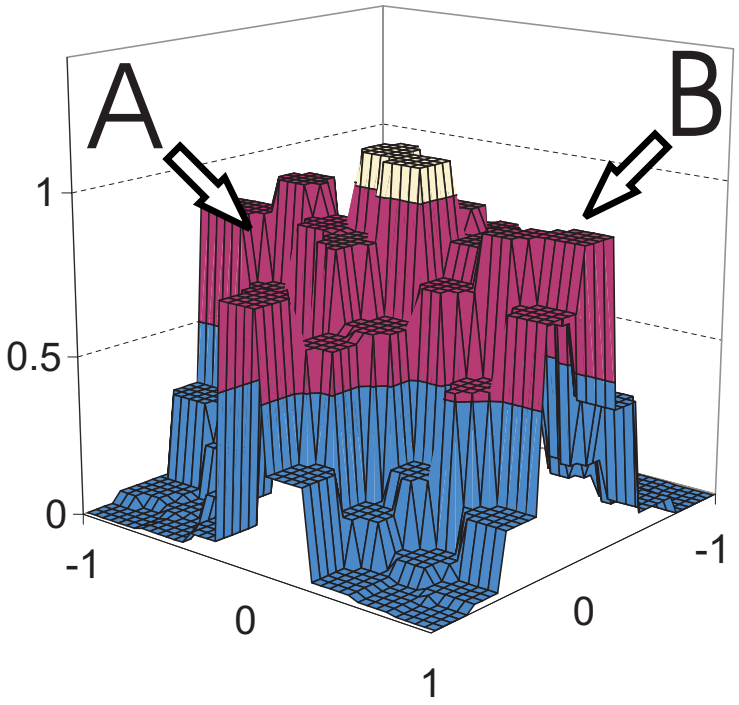


Abb. 4.2: Approximation durch konstante Werte.

konstant durch den Mittelwert approximiert, so ergibt sich ein nMSE von 1. Gute Verfahren sollten also einen nMSE erreichen, der deutlich kleiner als 1 ist. In unserem Fall wird die Funktion auf den Gitterpunkten eines regelmäßigen 21x21 Gitter in $[-1,1]^2$ getestet und der nMSE für diese Testpunkte berechnet.
Ausgangspunkt für LWIGNG ist das Wachsende Neurale Gas, wie es in Kapitel 3.2 beschrieben wurde. Alle Neuronen werden zusätzlich mit einem Wert $v_i \in \mathbb{R}$ versehen. Am Anfang werden die v_i auf Null gesetzt. Beim Einfügen neuer Neuronen wird zunächst der Wert der Approximation an der Stelle des neuen Positionsvektors bestimmt und dann der Wert v_i des neuen Neurons auf diesen Wert gesetzt. Wendet man dann das Verfahren der konstanten Approximation mit der Lernregel (3.7) zum Training der Testfunktion an, erhält man ein Ergebnis, wie es in Abbildung 4.2 zu sehen ist. Die Testfunktion wird stückweise konstant approximiert. Wie man sieht, wird die Funktion durch die terrassenartige Struktur nicht besonders gut wiedergegeben. Tabelle 4.1 zeigt die verwendeten Parameter. Diese wurden nach einigen Testläufen festgelegt, aber nicht gezielt optimiert.

Parameter	**Wert**	**Beschreibung**
α_b	0.025	Lernrate für die Bewegung des Gewinnerneurons
α_n	0.005	Lernrate für die Bewegungen der Nachbarneuronen
N_{max}	63	maximale Anzahl von Neuronen
N_{steps}	100	Anzahl der Schritte zwischen dem Einfügen von Neuronen
age_{max}	100	maximales Alter für Kanten
δ_{err}	0.001	Fehlerreduktionsrate pro Schritt
δ_{new}	0.01	Fehlerreduktionsrate beim Einfügen neuer Neuronen
α_v	0.1	Lernrate für die Werte v_i an den Neuronen

Tabelle 4.1: Parameter für die konstante Approximation.

4.2 Interpolierendes Wachsendes Neurales Gas

Als nächster Schritt soll nun zwischen den Werten benachbarter Neuronen interpoliert werden. Dadurch entsteht eine stückweise lineare Interpolation ähnlich wie bei den in Abschnitt 3.3.2 beschriebenen Verfahren von Göppert. Allerdings wird es durch die Verwendung von Ridge-Regression[1] möglich, von der Anzahl der Nachbarn abzusehen und immer das gleiche Verfahren zu verwenden. Außerdem wird sowohl beim Training als auch bei der Berechnung der Werte interpoliert, wo hingegen Göppert dasselbe Training wie bei der konstanten Approximation verwendet und nur bei der Berechnung der Werte interpoliert.
Die grundsätzliche Idee besteht darin, zunächst den Eingabevektor x als Linearkombination aus dem Positionsvektor des nächsten Neurons w_b und einiger Differenzvektoren $l_{b,i}$ auszudrücken. In unserem Fall sind dies immer die normierten Differenzvektoren zu den Nachbarn $c_{b,i}$[2].

$$x = w_b + \sum_{i=1}^{N_b} a_i l_{b,i} \tag{4.3}$$

mit

$$l_{b,i} = \frac{(w_{b,i} - w_b)}{\|w_{b,i} - w_b\|}$$

Abhängig von der Anzahl der Nachbarn N_b, der Dimension d und den Positionen der benachbarten Neuronen, gibt es für die a_i keine, eine eindeutige oder viele Lösungen (die Differenzvektoren könnten z.B. linear abhängig sein). Das Problem wird als lineares Gleichungssystem formuliert:

$$La = x_{rel} \tag{4.4}$$

wobei

$$L = (l_{b,1} \dots l_{b,N_b}),\ a = (a_1 \dots a_{N_b})^T \text{ und } x_{rel} = x - w_b.$$

Zur Lösung wird Ridge-Regression [Hastie u. a., 2001] verwendet. Ridge-Regression bestimmt a, so dass die folgende Summe minimiert wird:

$$\|x_{rel} - La\|^2 + \mu \|a\|^2 \tag{4.5}$$

Der erste Term ist der gewöhnliche quadratische Fehler, der zweite fügt eine zusätzliche Strafe für große $|a_i|$ ein. Da die Lösung von der Skalierung der Vektoren abhängt, ist es wichtig, dass in Gleichung (4.3) die normierten Differenzvektoren verwendet werden. Ansonsten würden große Differenzvektoren zu stark gewichtet werden, da sie zu kleinen a_i führen.
Die Ridge-Lösung ergibt sich als [3]:

$$a = \left(L^T L + \mu I\right)^{-1} L^T x_{rel} \quad \text{mit } 0 < \mu \ll 1 \text{ und Einheitsmatrix } I \tag{4.6}$$

[1] Auch als regularisierte minimale Quadrate oder Tychonov-Regularisierung bekannt.

[2] Natürlich könnten auch gewisse Differenzvektoren von vorne herein ausgeschlossen werden. So könnten beispielsweise nur Differenzvektoren verwendet werden, die in Richtung des Eingabevektors x zeigen (also solche mit positivem Skalarprodukt $l_{b,i} \cdot (x - w_b)$).

[3] In [Göppert und Rosenstiel, 1997] schlägt Göppert im Prinzip die gleiche Lösung vor. Dabei geht er aber von einer SOM-Topologie mit in der Regel zwei Nachbarn pro Dimension aus und davon, dass genau d Nachbarn gewählt wurden. In anderen Arbeiten schlägt er vor, die Kleinste-Quadrate-Lösung für überbestimmt Systeme zu verwenden [Göppert und Rosenstiel, 1995c] oder aber eine iterative Methode für unterbestimmte Systeme [Göppert und Rosenstiel, 1995a]. Bei Göppert werden die Differenzvektoren nicht normalisiert. Wir schlagen vor, ausschließlich die Ridge-Lösung mit normalisierten Differenzvektoren zu verwenden, da die Minimierung von (4.5) gute Lösungen für alle Fälle liefert.

Für überbestimmte Systeme (weniger Differenzvektoren als Dimensionen, also i.A. keine „echte“ Lösung) ergibt sich somit ungefähr die Kleinste-Quadrate-Lösung, d.h. der quadratische Abstand zum Eingabevektor x wird minimiert. Bei unterbestimmten Systemen (mehr Differenzvektoren als Dimensionen, also i.A. keine eindeutige Lösung) ergibt sich ungefähr die Minimum-Norm-Lösung[4], also die Lösung mit minimalen $\|a\|$.
Wenn einzelne Differenzvektoren linear oder nahezu linear abhängig sind[5], kann es vorkommen, dass einzelne $|a_i|$ dennoch sehr groß werden. Um zu gewährleisten, dass bei der Berechnung der Ausgabewerte nicht zu stark extrapoliert wird, wird daher folgende Bedingung überprüft:

$$\max_i \{|a_i|\} \stackrel{!}{\leq} \|w_{b,i} - w_b\| \tag{4.7}$$

Anschaulich bedeutet diese Bedingung, dass für die Linearkombination zur Approximation des Eingabevektors x keine Vektoren verwendet werden, die vom Betrag her größer als der jeweilige Differenzvektor zum Nachbarneuron sind. Wenn diese Bedingung nicht erfüllt ist, wird eine einfache Heuristik verfolgt, um einen Differenzvektor (also eine Spalte von L) zu entfernen und die a_i danach erneut nach Gleichung (4.6) zu bestimmen. Dabei werden Differenzvektoren gesucht, die in ähnliche oder nahezu entgegengesetzte Richtungen zeigen, damit möglichst viele „verschiedene Richtungen“ erhalten bleiben:

1. Bestimme die zwei normierten Differenzvektoren $l_{b,i}$ und $l_{b,j}$ mit maximalem absoluten Skalarprodukt $|l_{b,i} \cdot l_{b,j}|$
2. Bestimme davon den mit dem minimalen Skalarprodukt $l_{b,k} \cdot x_{rel}$ und entferne die entsprechende Spalte von L

Im ersten Schritt werden die beiden „linear abhängigsten“ Differenzvektoren bestimmt, um dann im zweiten Schritt denjenigen beizubehalten, der eher in Richtung von x_{rel} zeigt. Dieses Verfahren wird solange wiederholt bis Bedingung (4.7) erfüllt ist.
Sind die Koeffizienten a_i endgültig bestimmt, kann ein lokal lineares Modell aufgestellt werden, welches im Folgenden mit $M_b(x)$ bezeichnet wird. Dabei wird zwischen dem Wert des nächsten Neurons v_b und den Werten $v_{b,\delta(i)}$ der Nachbarneuronen, deren Differenzvektoren $l_{b,\delta(i)}$ nicht aus L entfernt wurden, interpoliert. Mit $\delta(i)$ wird der Index des zu Spalte i und damit zu a_i gehörenden Neurons und mit s die Anzahl der Spalten von L bezeichnet.

$$\widetilde{M}_b(x) \quad = \quad v_b + \sum_{i=1}^{s} a_i \frac{(v_{b,\delta(i)} - v_b)}{\left\|w_{b,\delta(i)} - w_b\right\|} \tag{4.8}$$

Abbildung 4.3 visualisiert diese Berechnung. Zunächst werden im Eingaberaum die a_i bestimmt, um den Eingabewert x möglichst gut zu approximieren. Diese werden dann dazu benutzt, mit Hilfe der Werte $v_{b,\delta(i)}$ an den Neuronen den Ausgabewert $\widetilde{M}_b(x)$ zu berechnen.
Um die v_b und die $v_{b,\delta(i)}$ zu trainieren, wird ein Gradientenabstiegsverfahren verwendet, welches zu folgenden Lernregeln führt:

$$\begin{aligned} \Delta v_b &= \alpha_v \left(1 - \sum_{i=1}^{s} \frac{a_i}{\left\|w_{b,\delta(i)} - w_b\right\|}\right) \left(y - \widetilde{M}_b(x)\right) \\ \Delta v_{b,\delta(i)} &= \alpha_v \frac{a_i}{\left\|w_{b,\delta(i)} - w_b\right\|} \left(y - \widetilde{M}_b(x)\right) \end{aligned} \tag{4.9}$$

[4] Bei überbestimmten Systemen konvergiert $\left(L^T L + \mu I\right)^{-1} L^T$ gegen die Pseudo-Inverse von L für $\mu \to 0$. Für unterbestimmte Systeme wird zusätzlich $\|a\|^2$ in (4.5) minimiert.

[5] Das kommt insbesondere bei gerade neu eingefügten Neuronen vor, da diese ja immer auf einer Geraden zwischen zwei benachbarten Neuronen eingefügt werden.

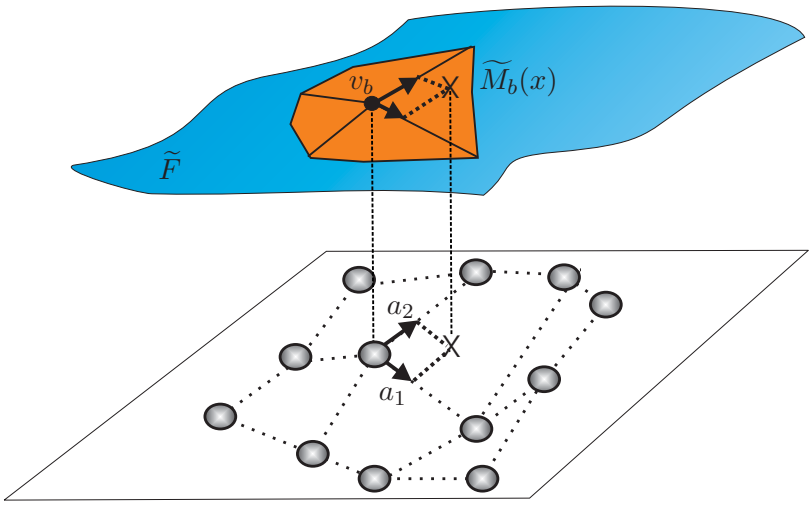


Abb. 4.3: Skizze des verwendeten Interpolationsverfahrens.

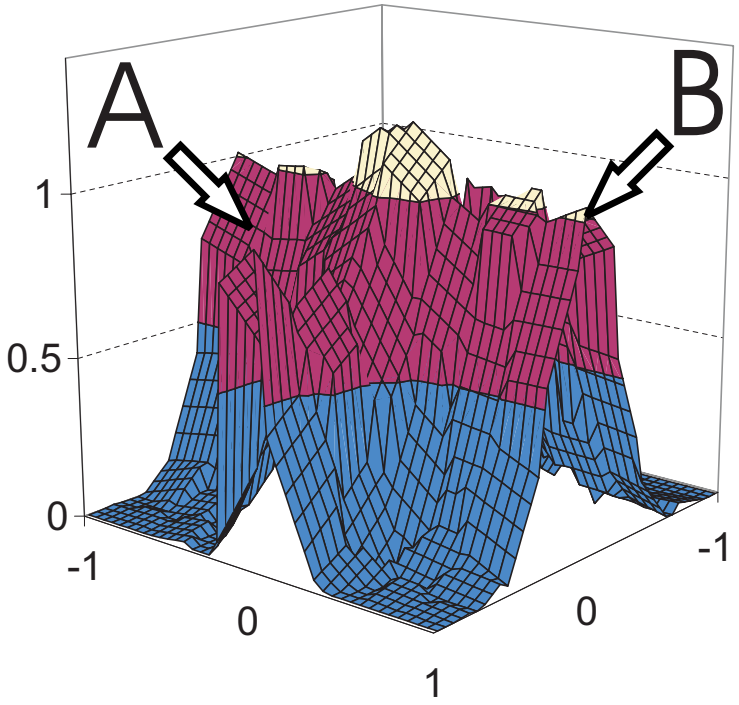


Abb. 4.4: Approximation durch stückweise lineare Approximation.

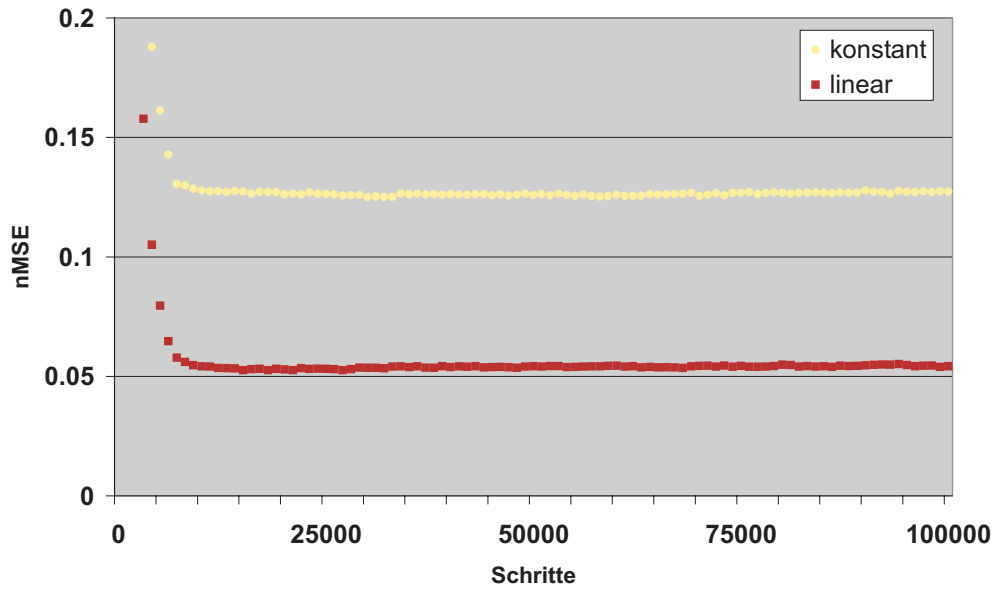


Abb. 4.5: Entwicklung des nMSE für konstante und stückweise lineare Approximation.

Auch durch diese Lernregeln unterscheidet sich das Verfahren deutlich von Göpperts I-SOM, wo lediglich das Gewinnerneuron mit der Lernregel für die konstante Approximation (Gleichung (3.7)) trainiert wird.
Wird dieses Verfahren mit den Parametern aus Tabelle 4.2 auf die Testfunktion (4.1) angewendet, ergibt sich eine Approximation wie in Abbildung 4.4. Es ist deutlich zu sehen, wie die Funktion stückweise linear approximiert wird. Deutlich sind auch die Schwierigkeiten zu sehen, die das Verfahren an der Stelle A hat. Da die Funktion dort relativ steil verläuft und das lokale Maximum ziemlich „eng“ ist, ist es schwierig, dieses gut wiederzugeben, wenn keine Neuronen dort liegen. Die Entwicklung des nMSE während des Trainings wird in Abbildung 4.5 im Vergleich mit der konstanten Approximation gezeigt. Die angegebenen und auch alle folgenden Werte sind jeweils Durchschnittswerte aus je 100 Läufen mit jeweils 100000 Schritten. Die Verwendung der stückweisen linearen Interpolation bringt also eine entscheidende Verbesserung bei der Güte der Approximation. Außerdem ist festzustellen, dass beide Verfahren sehr schnell lernen. Bereits nach etwa 10000 Schritten findet keine weitere Verbesserung der Approximationsgüte mehr statt.

Parameter	**Wert**	**Beschreibung**
μ	0.1	Regularisierungsparameter
alle übrigen Parameter sind die gleichen wie in Tabelle 4.1		

Tabelle 4.2: Parameter für die lineare Approximation.

4.3 Lokal Gewichtetes Interpolierendes Wachsendes Neurales Gas

Um eine kontinuierlichere und weiter verbesserte Approximation zu erreichen, wird die Interpolation $\widetilde{M}_b(x)$ nicht nur mit dem Gewinnerneuron als Zentrum berechnet. Auch für alle Nachbarn werden lokale Interpolationen $\widetilde{M}_{b,k}(x),\ k = 1, \ldots, N_b$ bestimmt. Das geschieht auf dieselbe Weise wie im vorhergehenden Abschnitt mit dem jeweiligen Nachbarneuron $c_{b,k}$ als Zentrum. Die Ausgaben dieser einzelnen lokalen Modelle sollen dann miteinander kombiniert werden. Dabei soll die Entfernung des Zentrums des jeweiligen Modells zum Eingabevektor berücksichtigt werden. Die endgültige Approximation $\widetilde{F}(x)$ wird dann als gewichtete Summe mit Gewichten $m_{b,k}(x)$ berechnet.

$$\widetilde{F}(x) = \sum_{k=0}^{N_b} m_{b,k}(x)\widetilde{M}_{b,k}(x) \tag{4.10}$$

Eine Möglichkeit, die Gewichte zu wählen, ist dem Vorschlag in [Fritzke, 1994a] zu folgen und eine normierte Gauß-Funktion (siehe Gleichung (3.12)) zu verwenden und die Reichweiten auf die durchschnittliche Kantenlänge aller vom jeweiligen Neuron ausgehenden Kanten zu setzen[6]. Damit ergibt sich für die Gewichte $m_{b,k}(x)$:

$$\begin{aligned} m_{b,k}(x) &= \frac{e^{-\frac{\left\|x-w_{b,k}\right\|^2}{\sigma_{b,k}^2}}}{\sum_{j=0}^{N_b} e^{-\frac{\left\|x-w_{b,j}\right\|^2}{\sigma_{b,j}^2}}} \\ \sigma_{b,j} &= \text{durchschnittliche Kantenlänge aller} \\ &\quad \text{vom Neuron } c_{b,j} \text{ ausgehenden Kanten} \end{aligned} \tag{4.11}$$

Eine weitere Möglichkeit ist, den Grad des Einflusses der lokalen Modelle auf die globale Approximation über die Kanten zu steuern. Jede Kante $e_{i,j}$ erhält dafür einen zusätzlichen Parameter $\lambda_{i,j}$, der die „Reichweite“ des lokalen Modells in Richtung der jeweiligen Kante beschreibt. Zusätzlich bekommt jedes Neuron c_i einen Parameter $\lambda_{i,0}$, der die „Reichweite“ steuert wenn das betreffende Neuron das Gewinnerneuron ist. Die Kante zwischen dem Gewinnerneuron c_b und dem Nachbarneuron $c_{b,k}$ wird mit $e_{b,(b,k)}$ bezeichnet, das zugehörige λ mit $\lambda_{b,(b,k)}$. Dabei ist $\lambda_{b,(b,0)} = \lambda_{b,0}$. Diese Parameter werden dann dazu verwendet, die Gewichte $m_{b,k}(x)$für die lokalen Modelle zu bestimmen.

$$m_{b,k}(x) = \frac{e^{-\lambda_{b,(b,k)}\left\|x-w_{b,k}\right\|}}{\sum_{j=0}^{N_b} e^{-\lambda_{b,(b,j)}\left\|x-w_{b,j}\right\|}} \tag{4.12}$$

In beiden Fällen werden die lokalen Modelle ähnlich wie im vorhergehenden Abschnitt trainiert. Statt der Ausgabe des lokalen Modells gehen nun die gesamte Approximation $\widetilde{F}(x)$ und die Gewichte mit in die Lernregeln ein. Das bedeutet für das Training des

[6] Im Unterschied zu [Fritzke, 1994a] gehen in die Berechnung der endgültigen Approximation in (4.10) allerdings nur die direkten Nachbarn des Gewinnerneurons und nicht alle Neuronen ein, wie es bei RBF-Netzen der Fall ist. Das hat zum einen den Vorteil, dass es schneller geht. Zum anderen gehen nur Neuronen ein, die dem Gewinnerneuron so ähnlich sind, dass sie eine gemeinsame Kante besitzen.

Modells $\widetilde{M}_{b,0}(x)$ mit dem Gewinnerneuron als Zentrum :

$$\begin{aligned} \Delta v_b &= \alpha_v m_{b,0}(x) \left(1 - \sum_{i=1}^{s} \frac{a_i}{\left\|w_{b,\delta(i)} - w_b\right\|}\right) \left(y - \widetilde{F}(x)\right) \\ \Delta v_{b,\delta(i)} &= \alpha_v m_{b,0}(x) \frac{a_i}{\left\|w_{b,\delta(i)} - w_b\right\|} \left(y - \widetilde{F}(x)\right) \end{aligned} \tag{4.13}$$

Alle übrigen Modelle $\widetilde{M}_{b,k}(x), k = 1, \ldots N_b$ werden mit den entsprechenden Gewichten $m_{b,k}(x)$, den jeweiligen Werten für die Koeffizienten a_i und den entsprechenden Differenzvektoren völlig analog trainiert.

4.3.1 Anpassung der Reichweite der lokalen Modelle

Es stellt sich nun die Frage, wie die $\lambda_{i,j}$ an den Kanten und die $\lambda_{i,0}$ der Neuronen gesetzt werden sollen. Es stellt sich heraus, dass es vorteilhaft ist, auch diese Parameter zu lernen. Die Gewichte lokaler Modelle $\widetilde{M}_{b,k}(x)$, die in eine bestimmte Richtung eine gute Approximation liefern, sollten erhöht werden, die Gewichte solcher, die eine schlechte Approximation liefern, verringert werden. Das lässt sich durch Gradientenabstieg mit folgender Formel erreichen:

$$\Delta\lambda_{b,(b,k)} = -\alpha_l \left\|x - w_{b,k}\right\| m_{b,k}(x) \left(\widetilde{F}(x) - \widetilde{M}_{b,k}(x)\right) \left(\widetilde{F}(x) - y\right) \tag{4.14}$$

Gleichung (4.14) führt dazu, dass bei zu kleinem Approximationswert ($\widetilde{F}(x) < y$) die $\lambda_{b,(b,k)}$ derjenigen Modelle, die einen größeren Wert liefern ($\widetilde{M}_{b,k}(x) > \widetilde{F}(x)$), verringert werden. Nach Gleichung (4.12) erhalten diese damit in Zukunft ein größeres Gewicht. Umgekehrt werden die $\lambda_{b,(b,k)}$ derjenigen Modelle, die einen kleineren Wert liefern, erhöht.
Abbildung 4.6 zeigt die Lernkurven für die verschiedenen Möglichkeiten der Bestimmung der Gewichte im Vergleich zur stückweisen linearen Interpolation (wiederum Durchschnittswerte aus 100 Läufen mit 100000 Schritten). Tabelle 4.3 enthält die verwendeten Parameter. In allen Fällen führt die Verwendung lokal gewichteter Modelle zu einer Verbesserung der Approximationsgüte. Die Verwendung von zusätzlichen Parametern an den Kanten und den Neuronen erreicht gerade wenn diese während des Trainings angepasst werden, eine deutlich bessere Approximationsgüte als die Verwendung der durchschnittlichen Kantenlänge. Dass auch die Verwendung von festen $\lambda_{i,j}$ besser abschneidet als die Verwendung der durchschnittlichen Kantenlänge, dürfte daran liegen, dass der feste Wert für die $\lambda_{i,j}$ glücklich gewählt wurde. Alle Methoden erreichen wieder relativ schnell eine hohe Approximationsgüte, wobei man sieht, dass die Approximationsgüte bei der Methode mit den Kantenparametern während des gesamten Trainings durch die Anpassung der Reichweite der Modelle noch weiter verbessert wird.

Parameter	**Wert**	**Beschreibung**
λ_{start}	30	Initialwert für die λ
α_l	1000	Lernrate für die Reichweitenanpassung
alle übrigen Parameter sind wie in den Tabellen 4.1 und 4.2		

Tabelle 4.3: Parameter für die Approximation mit Anpassung der Reichweite der lokalen Modelle.

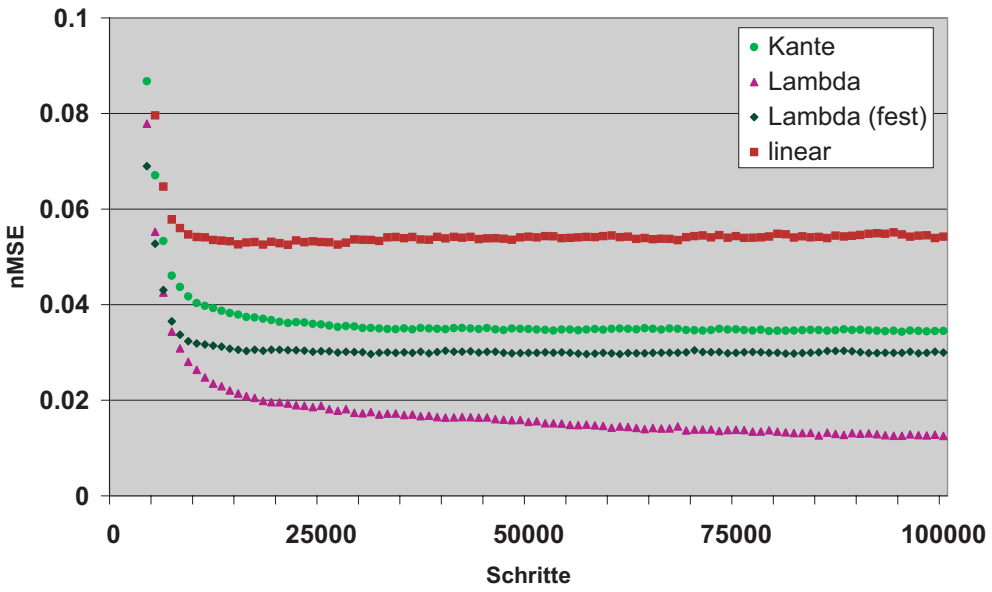


Abb. 4.6: Entwicklung des nMSE für die lokal gewichtete Approximation im Vergleich zur stückweisen linearen Interpolation. Bei „Kante“ wurden die Gewichte gemäß (4.11) über die durchschnittliche Kantenlänge bestimmt, bei „Lambda“ und „Lambda (fest)“ über zusätzliche Parameter an den Kanten und den Neuronen nach (4.12). Bei „Lambda“ wurden die $\lambda_{i,j}$ durch Lernregel (4.14) während des Trainings angepasst, bei „Lambda (fest)“ wurde keine Anpassung durchgeführt. Beachte, dass in dieser Abbildung das Maximum für den nMSE 0.1 und nicht 0.2 wie in Abbildung 4.5 ist.

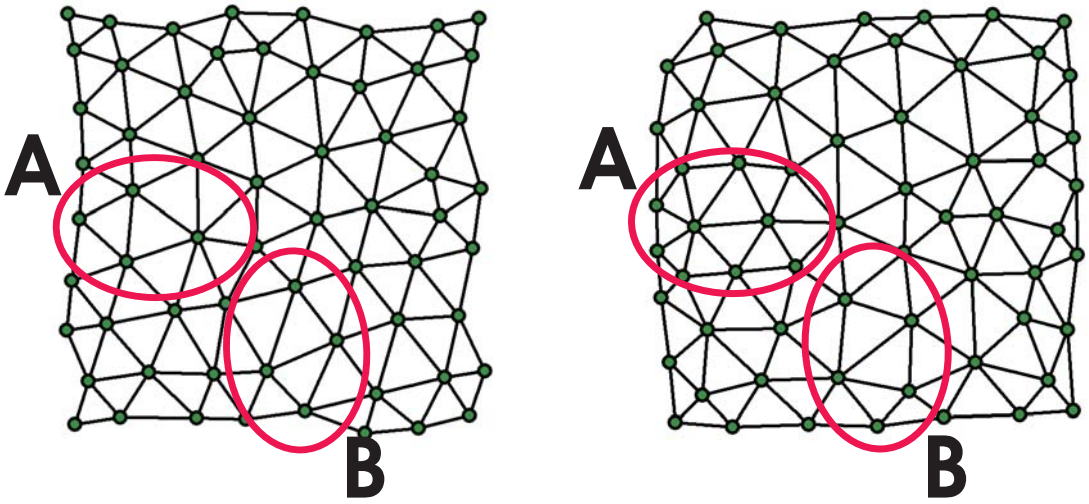
4.3.2 Fehlerabhängiges Anpassen der Positionen

Beim GNG-Algorithmus werden die Neuronen gemäß der relativen Häufigkeit der Eingabedaten verteilt. Möchte man Funktionen approximieren, ist das normalerweise nicht die geschickteste Möglichkeit. Besser wäre es die Güte der lokalen Approximation zu berücksichtigen. Zu diesem Zweck wird eine ähnliche Heuristik wie in [Winter u. a., 2000] für Neurale Gase mit lokal-linearer Regression verwendet (vgl. 3.3.1). Die Anpassungsformeln für die lokalen Fehlervariablen err_i werden so geändert, dass anstelle des Quantisierungsfehlers der Approximationsfehler geschätzt wird.

$$\Delta err_{b,k} = m_{b,k}\beta_{err}\left(\left|y_t - \widetilde{F}(x_t)\right| - err_{b,k}\right) \qquad k = 0, \ldots, N_b \tag{4.15}$$

Dabei bestimmt β_{err}, wie stark der aktuelle Fehler $|y_t - \widetilde{F}(x)|$ in die Schätzung eingeht. Kleinere Werte von β_{err} führen dazu, dass ältere Fehler stärker gewichtet werden. Durch die neue Fehlerdefinition werden Neuronen an Stellen eingefügt, an denen der Approximationsfehler hoch ist. Darüber hinaus werden auch die Anpassungsformeln für die Positionen der Neuronen geändert. Die grundsätzliche Idee ist dabei, das Gewinnerneuron c_b ein wenig in die Richtung des Nachbarneurons mit dem größten Fehler zu schieben, falls dessen Fehler größer ist als der des Gewinnerneurons. Andererseits werden Nachbarneuronen ein wenig in Richtung des Gewinnerneurons geschoben, wenn der Fehler des Gewinnerneurons größer als der eigene ist. Zu diesem Zweck wird eine Hilfsfunktion definiert:

$$\eta(err_1, err_2) = \max\left\{\frac{err_1 - err_2}{err_1}, 0\right\}, \quad err_1, err_2 \in \mathbb{R} \tag{4.16}$$



a) b)

Abb. 4.7: Vergleich der Verteilung der Neuronen bei unterschiedlichen Arten der Bewegung. Mit A ist der steile, mit B der etwas flachere Grat der Testfunktion markiert (vgl. Abbildung 4.1). a) Verteilung der Neuronen mit häufigkeitsabhängiger Bewegung der Neuronen. b) Verteilung mit fehlerabhängiger Bewegung der Neuronen. Diese Netz bildet die Grundlage der in Abbildung 4.9 dargestellten Approximation.

Die Anpassungsformeln für das Gewinnerneuron und dessen Nachbarn (GNG-Algorithmus, Abb. 3.4, Schritt 6) werden wie folgt geändert:

$$\begin{aligned} \Delta w_b &= \alpha_b(x - w_b) + \alpha_r \eta(err_{b,k_{max}}, err_b)(w_{b,k_{max}} - w_b) \qquad (4.17) \\ &\quad \text{wobei } k_{max} \text{ der Index des Nachbarn } c_{b,k} \text{ mit dem größten Fehler ist} \\ \Delta w_{b,k} &= \alpha_n(x - w_{b,k}) + \alpha_q \eta(err_b, err_{b,k})(w_b - w_{b,k}) \end{aligned}$$

Dabei bestimmen die Lernraten α_r und α_q, wie stark in Richtung des Neurons mit dem größten Fehler verschoben wird. Anders als in [Winter u. a., 2000] geht auch bei der Bewegung der Nachbarneuronen der Approximationsfehler mit ein. Abbildung 4.7 zeigt den Unterschied in der Verteilung der Neuronen mit und ohne fehlerabhängige Bewegung bei der Testfunktion (4.1). Man kann dabei eine etwas höhere Konzentration von Neuronen an der Stelle A feststellen, an der die Funktion am „spitzesten“ verläuft (vgl. Abbildung 4.1) und somit etwas schwerer zu approximieren ist. Außerdem liegen zwei Neuronen direkt auf dem Grat des lokalen Maximums und ermöglichen so eine bessere Approximation.

Parameter	**Wert**	**Beschreibung**
β_{err}	0.05	Gewicht des aktuellen Fehlers
α_r	0.025	Lernrate für die fehlerabhängige Anpassung des Gewinnerneurons
α_q	0.005	Lernrate für die fehlerabhängige Anpassung der Nachbarneuronen
alle übrigen Parameter sind wie in den Tabellen 4.1 bis 4.3		

Tabelle 4.4: Parameter für das fehlerabhängige Anpassen der Neuronenpositionen.

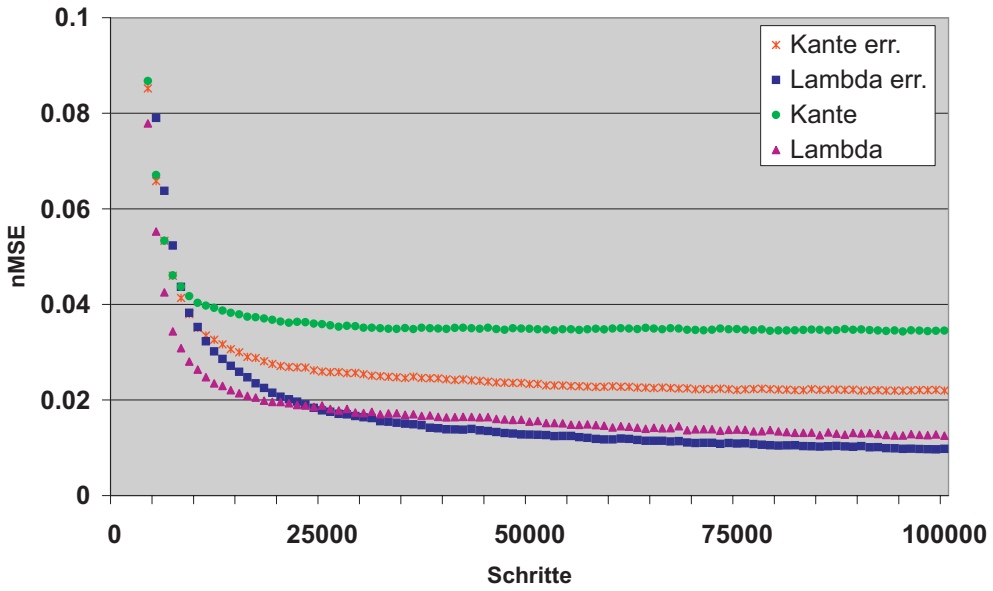


Abb. 4.8: Der nMSE mit (Kante err. , Lambda err.) und ohne (Kante, Lambda) fehlerabhängiger Bewegung der Neuronen.

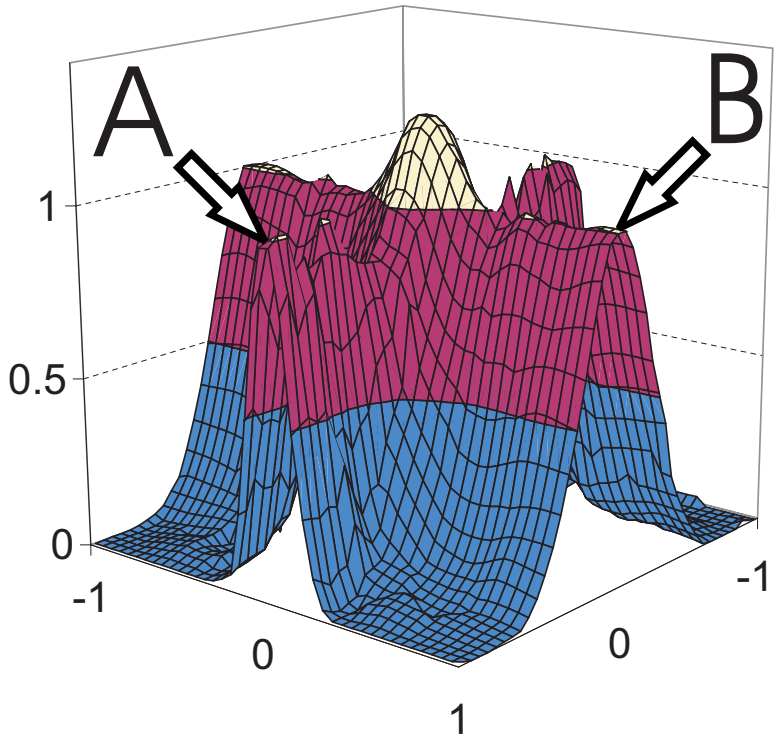


Abb. 4.9: Approximation mit lokal gewichteter linearer Interpolation und fehlerabhängiger Bewegung der Neuronen.

Ausgedrückt durch den nMSE zeigt sich eine weitere Verbesserung der Approximationsgüte durch die fehlerabhängige Anpassung der Positionen (Abbildung 4.8). Bei der Verwendung der durchschnittlichen Kantenlänge nach (4.11) wird dies besonders deutlich. Aber auch bei der Verwendung der $\lambda_{i,j}$ an den Kanten und den Neuronen nach (4.12) wird eine verbesserte Approximationsgüte erreicht, auch wenn dabei etwas langsamer gelernt wird. Erst nach etwa 25000 Schritten führt die fehlerabhängige Bewegung zu einem niedrigeren nMSE. Dass die Verbesserung bei der Verwendung der Parameter $\lambda_{i,j}$ nicht so stark ist, dürfte daran liegen, dass die Approximationsgüte ohnehin sehr hoch ist. Die verwendeten Parameter sind in Tabelle 4.4 aufgeführt. Abbildung 4.9 zeigt die dadurch realisierte Approximation. Die Approximation an der Stelle A ist zwar immer noch nicht optimal, aber doch wesentlich besser als zuvor. Der Rest der Funktion wird sehr gut wiedergegeben.

4.3.3 Der vollständige LWIGNG-Algorithmus

Abbildung 4.10 zeigt den kompletten Algorithmus für das Training. Soll nur der Approximationswert berechnet werden, werden lediglich die Schritte 2. bis 4. ausgeführt.

1. Trainiere das zugrunde liegende GNG gemäß dem gewöhnlichen GNG-Algorithmus (Abbildung 3.4) mit folgenden Änderungen:
   - Die Akkumulation des Quantisierungsfehlers wird durch die Akkumulation des Approximationsfehlers ersetzt (4.15).
   - Die Bewegung der Neuronen findet fehlerabhängig gemäß (4.17) statt.
   - Die $\lambda_{i,j}$ werden mit λ_{start} initialisiert.
   - Beim Einfügen eines neuen Neurons c_i wird $v_i = \widetilde{F}(w_i)$ gesetzt.
2. Berechne die lokalen Modelle $\widetilde{M}_{b,k}(x)$ gemäß (4.8).
3. Berechne die Gewichte $m_{b,k}(x)$ für die lokalen Modelle gemäß (4.12).
4. Bestimme die endgültige Approximation $\widetilde{F}(x)$ gemäß (4.10).
5. Trainiere die lokalen Modelle gemäß (4.13).
6. Trainiere gegebenenfalls die $\lambda_{b,k}$ gemäß (4.14).

Abb. 4.10: Der vollständige LWIGNG-Algorithmus.

4.4 Experimente

Die Leistungsfähigkeit von LWIGNG soll nun mit *Locally Weighted Projection Regression* (LWPR) [Schaal und Atkeson, 1998; Vijayakumar und Schaal, 1998, 2000; Vijayakumar u. a., 2002; Schaal u. a., 2002; Vijayakumar u. a., 2005] verglichen werden, einem Verfahren, das den derzeitigen Stand der Forschung repräsentiert. Beide Verfahren verfolgen zumindest teilweise dieselben Ziele, nämlich inkrementelle Funktionsapproximation

für hoch-dimensionale Eingaberäume mit möglicherweise irrelevanten oder redundanten Eingabedimensionen. Bei Data-Mining-Anwendungen ist LWPR konkurrenzfähig mit anderen modernen Verfahren wie *Gauss Process Regression* [Williams und Rasmussen, 1995] und *Support Vector Regression* [Smola und Schölkopf, 1998]. Was inkrementelle Funktionsapproximation anbelangt, ist LWPR nach Angaben der Entwickler „the first truly incremental spatially localized learning method that can successfully and efficiently operate in very high-dimensional spaces“ [Vijayakumar u. a., 2005]. LWPR soll im Folgenden zunächst kurz vorgestellt werden.

4.4.1 Locally Weighted Projection Regression

Bei LWPR werden eine Reihe von so genannten *rezeptiven Feldern (receptive fields)* mit Zentren $c_k \in \mathbb{R}^d$ im Eingaberaum verteilt. Diese Zentren sind jedoch im Gegensatz zu LWIGNG fest und nicht beweglich. Zudem gibt es auch keine Kanten zwischen einzelnen Zentren und somit auch keine explizite Nachbarschaftsstruktur.
Die Approximation bei LWPR beruht ebenfalls auf einer gewichteten Mittelung mehrerer lokal linearer Modelle. Diese werden durch eine inkrementelle Variante von *Partial Least Squares* (PLS) [Wold, 1975; Frank und Friedman, 1993] für jedes Zentrum gebildet. PLS projiziert den Eingabevektor x bzw. dessen Residuen sukzessive auf eine Reihe von orthogonalen Richtungsvektoren und führt jeweils eine univariate lineare Regression mit dem Ergebnis dieser Projektion durch. Die Ergebnisse der Regression werden dann addiert und liefern den Ausgabewert des Modells. Der Vorteil ist, dass keine komplette lineare Regression berechnet werden muss, sondern lediglich mehrere univariate Regressionen bestimmt werden. Die Richtungsvektoren werden dabei unter Beachtung der Korrelation zwischen Ein- und Ausgabedaten gewählt. Die Anzahl der verwendeten Projektionen richtet sich danach, wie stark der mittlere quadratische Fehler durch Einführung einer weiteren Projektion abnimmt. Jedes rezeptive Feld bestimmt also die lokal lineare Regression für sich ohne Beachtung der Position oder der Werte anderer Zentren. Die Parameter und die Richtungsvektoren werden gespeichert und während des Trainings entsprechend den präsentierten Ein-Ausgabepaaren angepasst.
Die endgültige Approximation wird als gewichtete Summe der Ausgaben der lokalen Modelle berechnet. Die Gewichte der lokalen Modelle werden durch multi-dimensionale Gauß-Funktionen mit Hilfe von Distanzmatrizen D_k bestimmt:

$$w_k = \frac{e^{-0.5(x-c_k)^T D_k (x-c_k)}}{\sum_{i=1}^{N} e^{-0.5(x-c_i)^T D_i (x-c_i)}} \tag{4.18}$$

Dabei bezeichnet c_k die Position des jeweiligen Zentrums des lokalen Modells und N die Anzahl der lokalen Modelle. Bei der Distanzmatrix D_k handelt es sich um eine positiv definite $(d \times d)$-Matrix, wobei d die Dimension des Eingabevektors ist. Neue Modelle werden eingefügt, wenn kein Modell stärker als ein bestimmter Schwellenwert aktiviert wird. Die Distanzmatrizen werden während des Trainings durch ein Gradientenverfahren angepasst, so dass sich die „Reichweiten“ der lokalen Modelle nach der Qualität der von ihnen gelieferten Approximation richten. Anders als bei LWIGNG, wo für die einzelnen Kanten die Parameter der ein-dimensionalen Gauß-Funktionen gelernt werden, muss hier also für jedes Zentrum eine komplette Matrix angepasst werden.

4.4.2 Wahl der Parameter und Durchführung

LWPR wurde nach der Beschreibung in [Vijayakumar u. a., 2005] implementiert und die dort beschriebenen Experimente mit Funktion (4.1) wurden durchgeführt. Die dort angegebenen Resultate konnten reproduziert werden.
Bei diesen Experimenten wurde jeweils eine feste Trainingsmenge von 500 Punkten (x_t, y_t) vorab bestimmt und diese dann zum Training verwendet, um LWPR mit state-of-the-art nicht-inkrementellen Data-Mining-Algorithmen (Gauss Process Regression [Gibbs und MacKay, 1997], Support Vector Regression [Saunders u. a., 1998]) zu vergleichen. LWPR erreichte bei Data-Mining-Experimenten ähnliche Ergebnisse wie diese Algorithmen. Bei den Experimenten mit der Zielfunktion (4.1) waren Gauss Process Regression und LWPR ungefähr gleich gut (nMSE ≈ 0.02 nach 100000 Schritten bei einer Testmenge von 500 Punkten), während Support Vector Regression deutlich schlechter war[7]. Da wir an einem Einsatz als Funktionsapproximator für Probleme interessiert sind, bei denen sich die Zielfunktion und die Verteilung der Trainingsdaten während des Trainings ändern können, wurden die Trainingsdaten $(x_t, f(x_t))$ online während des Trainings erzeugt und keine feste Menge von Trainingsbeispielen verwendet.
Alle Parameter für LWPR sind dieselben wie in [Vijayakumar u. a., 2005]. Für LWIGNG wurden die in den Tabellen 4.1 bis 4.4 angegebenen Parameter verwendet. Lediglich die maximale Neuronenanzahl wurde variiert, um die beiden Verfahren besser miteinander vergleichen zu können. Zunächst wurden die Experimente mit LWPR ausgeführt und die durchschnittliche Anzahl der benutzten lokalen Modelle bestimmt. Diese Anzahl wurde dann als Obergrenze für die Anzahl der Neuronen verwendet. Bei allen Experimenten wurden jeweils 100 Läufe mit je 100000 Schritten durchgeführt. Die angegebenen Werte sind die Mittelwerte aus diesen Läufen.

4.4.3 Approximation statischer Zielfunktionen

Zunächst wurden die Experimente mit der 2-dimensionalen verrauschten Testfunktion (4.1) aus [Vijayakumar u. a., 2005] mit den oben angegebenen Änderungen wiederholt, um herauszufinden, wie gut LWIGNG mit hoch-dimensionalen Eingaberäumen zurechtkommt.
Im ersten Experiment wurden LWPR und LWIGNG mit zufälligen 2-dimensionalen Eingabewerten aus $[-1, 1]^2$ und den reellen Ausgabewerten der Funktion trainiert. Da LWPR im Durchschnitt 63 lokale Modelle benutzte, wurde die maximale Neuronenanzahl auf 63 gesetzt. Wie in Abbildung 4.11 zu sehen, lernt LWPR während der ersten 10000 Schritte etwas schneller, was vor allem daran liegen dürfte, dass bei LWIGNG das zugrunde liegende Wachsende Neurale Gas erst noch Neuronen einfügen und sich ausbreiten muss. Ab 10000 Schritten ist LWIGNG besser und erreicht auch ein etwas besseres Endergebnis nach 100000 Schritten. Der nMSE von LWPR ist mit 0.01 besser als der nMSE bei der Verwendung einer festen Trainingsmenge von 500 Punkten in [Vijayakumar u. a., 2005]. Offensichtlich ist es leichter, die Funktion aus einer größeren Menge von Trainingsbeispielen zu lernen.
In einem zweiten Experiment wurden 8 Dimensionen mit jeweils konstanten Werten dem Eingabevektor hinzugefügt und der ganze Vektor mit einer Rotationsmatrix multipli-

[7] In [Schaal und Atkeson, 1998] wird außerdem berichtet, dass ein 3-schichtiges Backpropagation-Netz mit sigmoidaler Aktivierungsfunktion und verschiedenen Konfigurationen mit bis zu 100 versteckten Neuronen, nicht in der Lage war innerhalb von 1000000 Schritten einen nMSE besser als 0.1 zu erreichen. Erst durch Erhöhung der Anzahl der Trainingsbeispiel auf 1000 und Verringerung des Rauschens auf N(0,0.0001) konnte ein nMSE von 0.02 nach 750000 Trainingsschritten mit 100 versteckten Neuronen erreicht werden.

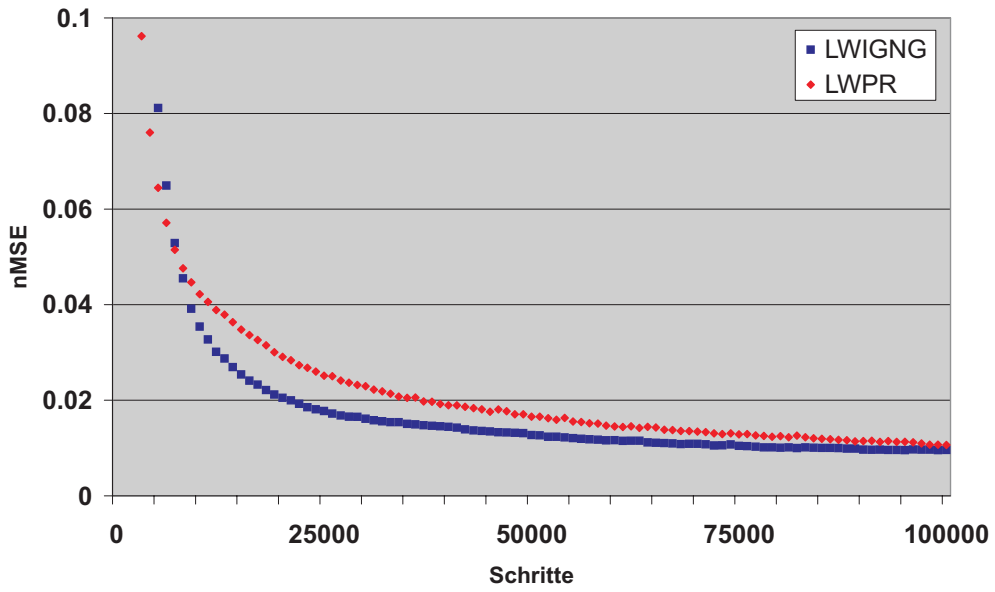


Abb. 4.11: Entwicklung des nMSE bei 2-dimensionalen Eingabevektoren.

ziert. Trainiert wurde dann mit diesem 10-dimensionalen Eingabevektor und dem Wert der Funktion für den Eingabevektor vor dem Hinzufügen der zusätzlichen Dimensionen und der Rotation. Damit soll getestet werden, ob die Verfahren in der Lage sind, die eigentlich 2-dimensionale Struktur dieser 10-dimensionalen Eingabedaten zu entdecken und auszunutzen. Es zeigt sich, dass dies sehr gut gelingt. Im Durchschnitt benötigt LWPR 65 Modelle, also wurde auch die maximale Neuronenanzahl auf diesen Wert gesetzt. Das Ergebnis (siehe Abbildung 4.12) gleicht im Wesentlichem dem des ersten Experiments.
Für ein drittes Experiment wurden dem 10-dimensionalen Eingabevektor zusätzlich 10 weitere Dimensionen mit einem $N(0, 0.0025)$ normalverteiltem Rauschen hinzugefügt, um zu testen, wie die Verfahren auf verrauschte Eingabedaten reagieren. Da LWPR dabei im Schnitt ebenfalls 65 Modelle benötigte, wurde wieder eine maximale Neuronenanzahl von 65 verwendet. Die Lernkurven (Abbildung 4.13) entsprechen wiederum fast den Vorhergehenden. Bemerkenswert ist, dass der nMSE für LWIGNG bei allen Experimenten am Ende der 100000 Schritte immer bei 0.009 liegt. LWPR schneidet im zweiten und dritten Experiment minimal schlechter als im ersten ab (0.012 gegenüber 0.010). LWIGNG besitzt also dieselben wünschenswerten Eigenschaften wie LWPR, wenn das Verfahren mit irrelevanten oder redundanten Eingabedaten konfrontiert wird. Problematisch wird es für beide Verfahren, wenn sie es im dritten Experiment mit einem stärkeren Rauschen zu tun haben. Bei einem Test mit einem Rauschen von N(0,0.1) benötigt LWPR über 12000 Modelle um einen nMSE von ungefähr 0.03 zu erreichen. LWIGNG erreicht mit 65 Modellen nur einen nMSE von 0.07. Auch eine drastische Erhöhung der maximalen Neuronenanzahl bringt dabei keine Verbesserung. Aufgrund der langen Laufzeiten (mehr als 1 Tag pro Lauf) durch die hohe Anzahl von Modellen wurden allerdings keine systematischen Experimente durchgeführt.
Abbildung 4.14 gibt einen Eindruck vom Laufzeitverhalten der Algorithmen für die einzelnen Experimente. Es werden die durchschnittlichen Zeiten für das Training und die Berechnung von jeweils 100000 Werten angegeben[8]. Die beiden Verfahren wurden

[8]Durchschnitte über 100 Läufe auf einem AMD Athlon XP 2400+, 2.0 GHz mit Windows XP

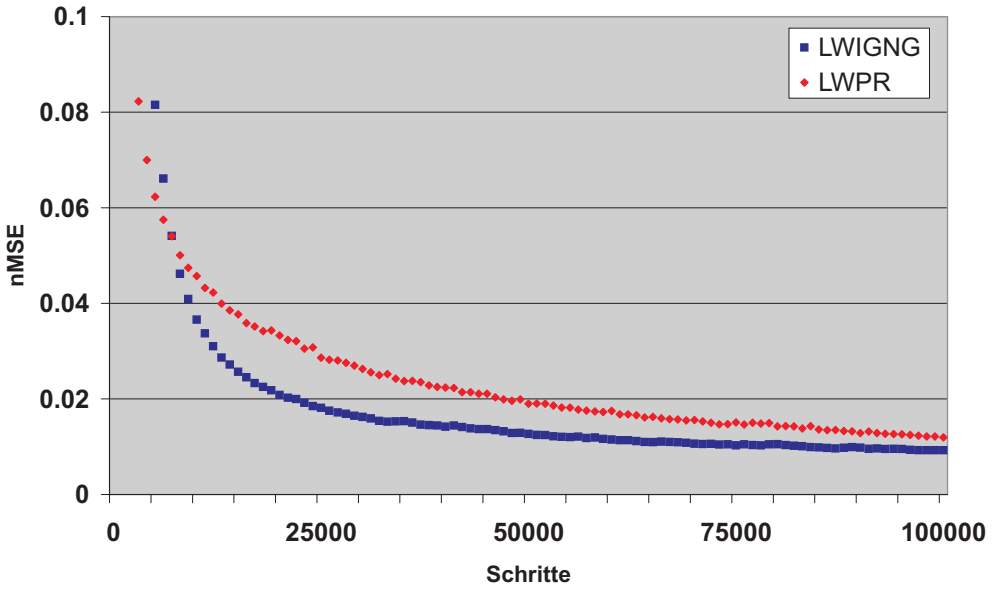


Abb. 4.12: Entwicklung des nMSE bei 10-dimensionalen Eingabevektoren.

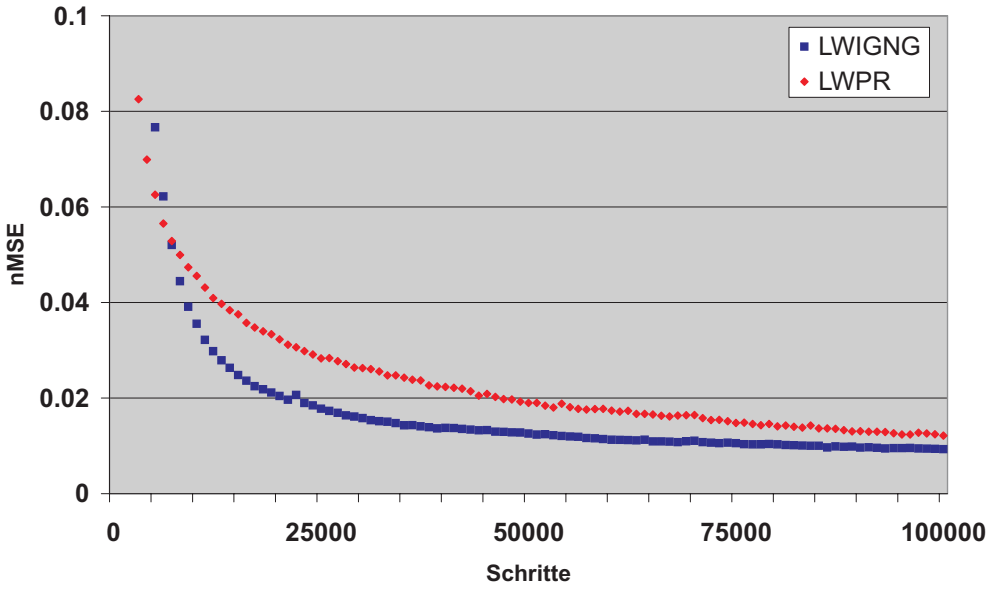


Abb. 4.13: Entwicklung des nMSE bei 20-dimensionalen Eingabevektoren.

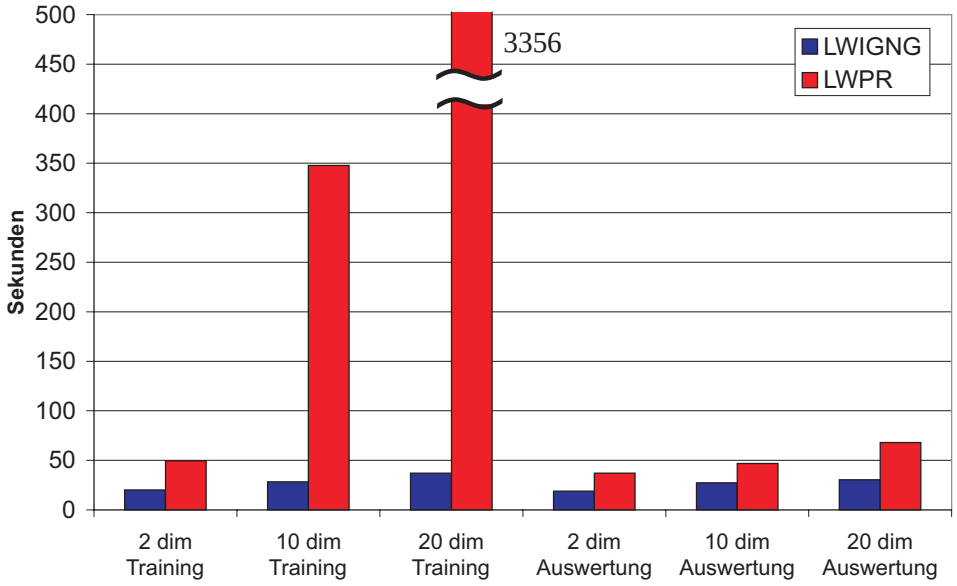


Abb. 4.14: Vergleich der Zeiten, die LWIGNG und LWPR für das Trainieren und das Auswerten von jeweils 100000 Beispielen brauchen (durchschnittliche Werte aus 100 Läufen).

nicht besonders stark in Hinblick auf Geschwindigkeit optimiert, aber dennoch können diese Daten einen ersten Eindruck vom Laufzeitverhalten der Algorithmen liefern. Bei LWPR werden lokale Modelle nur berechnet und trainiert, wenn deren Aktivierung größer als 0.001 ist. Bei LWIGNG gehen ohnehin jeweils nur die lokalen Modelle des Gewinnerneurons und der benachbarten Neuronen in die Berechnungen ein. Es ist deutlich, dass LWIGNG in allen Fällen schneller ist. Insbesondere beim Training mit hochdimensionalen Eingabevektoren braucht LWPR deutlich länger. Der Hauptgrund dafür dürfte darin liegen, dass LWPR mehr-dimensionale Gauß-Funktionen benutzt, um die Gewichte der lokalen Modelle zu berechnen und zu verändern, während LWIGNG mit ein-dimensionalen Gauß-Funktionen auskommt.

4.4.4 Approximation sich verändernder Zielfunktionen

Eine zweite Reihe von Experimenten wurde durchgeführt, um zu testen, wie gut LWIGNG und LWPR mit sich zeitlich verändernden Zielfunktionen und Eingabeverteilungen zurechtkommen, also mit Problemen, wie sie typischerweise beim Lernen von Aktionen auftreten.
In einem ersten Experiment wurde der verrauschte Ausgabewert von Testfunktion (4.1) während der ersten 50000 Schritte mit $\frac{t}{50000}$ (dabei ist t die Anzahl der bereits vergangenen Trainingsschritte) multipliziert. Es wurden die normalen 2-dimensionalen Eingabevektoren verwendet und während der zweiten 50000 Schritte wurde mit dem normalen Funktionswert trainiert. Das Ergebnis ist in Abbildung 4.15 zu sehen. Beim Berechnen des nMSE wurde dabei die aktuelle Zielfunktion (also $\frac{t}{50000} f(x)$ während der ersten 50000 Schritte) zugrunde gelegt. Wie zu sehen ist, gelingt es LWIGNG sehr gut, mit dieser sich schnell verändernden Funktion Schritt zu halten, während LWPR erst während der zweiten 50000 Schritte gute Ergebnisse erreicht. LWPR benötigte im Durchschnitt 57 Modelle[9],

[9]Das diese Anzahl geringer ist als beim Training mit der normalen Funktion, lässt sich wohl dadurch er-

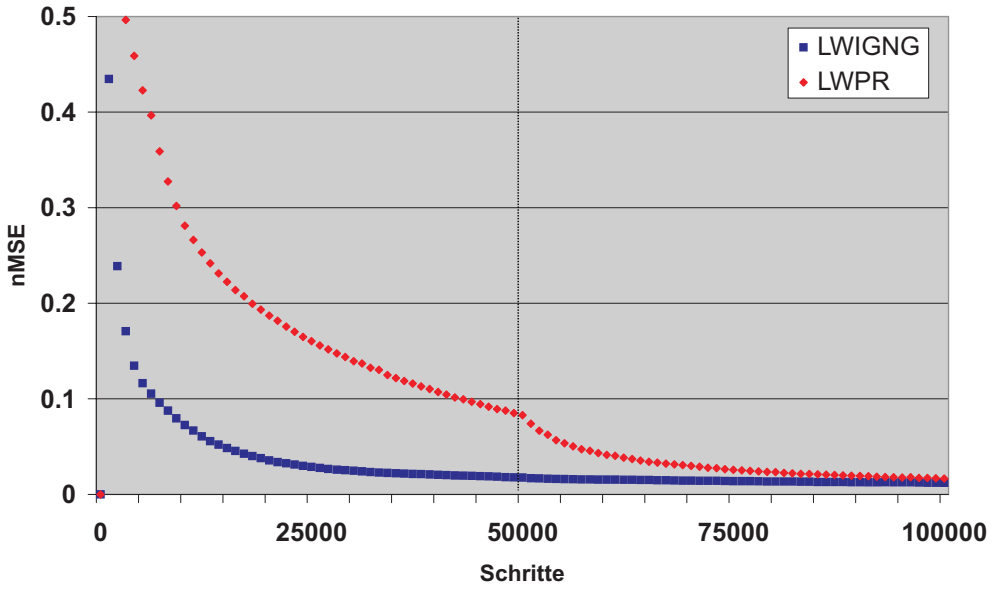


Abb. 4.15: Entwicklung des nMSE im Falle des „langsam ansteigenden Funktionswertes“. Der Strich bei 50000 Schritten markiert den Zeitpunkt, ab dem mit der nicht-modifizierten Testfunktion trainiert wurde.

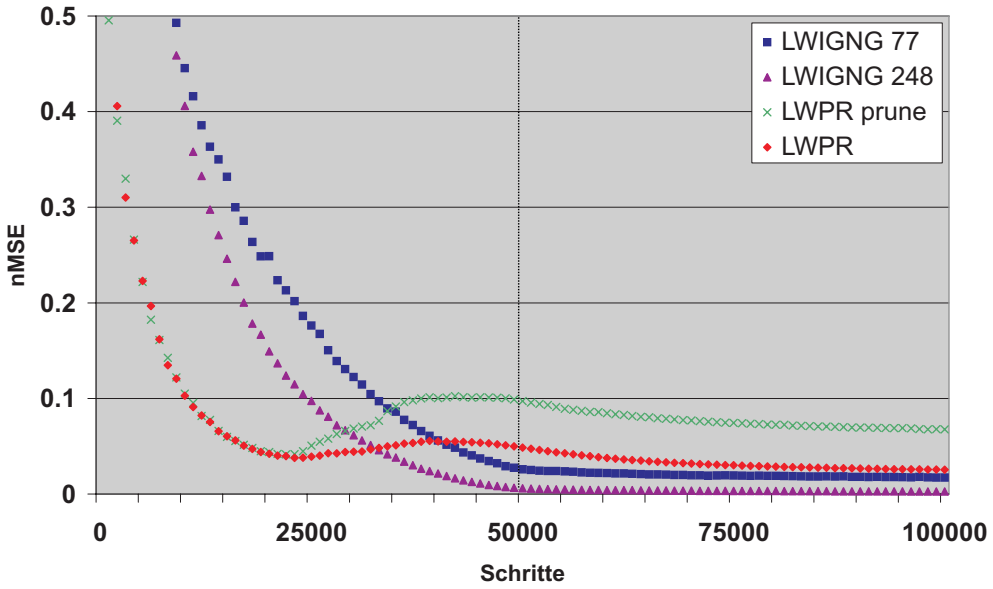


Abb. 4.16: Entwicklung des nMSE im Falle des „langsam abnehmenden Rauschens“ für LWIGNG mit verschiedenen maximalen Neuronenanzahlen und LWPR mit („LWPR prune“) und ohne mögliches Löschen von „stark überlappenden“ Modellen. Der Strich bei 50000 Schritten markiert den Zeitpunkt, ab dem mit der nicht-modifizierten Testfunktion trainiert wurde.

dementsprechend wurde auch die maximale Anzahl der Neuronen auf 57 gesetzt.
Beim nächsten Experiment wurde während der ersten 50000 Schritte zum normalen verrauschten Funktionswert zusätzlich ein normalverteiltes Rauschen mit Mittelwert 0 und langsam abnehmender Varianz von $1 - \frac{t}{50000}$ zum Funktionswert addiert. In diesem Fall wird der nMSE von Anfang an mit der nicht verrauschten Funktion berechnet. Bei diesem Experiment wurden zwei verschiedene Versionen von LWPR verwendet. Die erste Version ist die bisher verwendete, bei der zweiten werden lokale Modelle wieder gelöscht, wenn sie sich „zu stark“ überlappen. Wenn die Aktivierung bei demselben Eingabewert für verschiedene Modelle größer als 0.9 ist, wird das Modell gelöscht, bei dem die Determinante der Distanzmatrix am größten ist [Schaal und Atkeson, 1998][10]. Ohne das Löschen stark überlappender Modelle benötigt LWPR 248 Modelle, mit Löschen benutzt LWPR am Ende der 100000 Schritte 77 Modelle im Durchschnitt. Dementsprechend wurden auch für LWIGNG Läufe mit einer maximalen Neuronenanzahl von 77 und 248 durchgeführt. In Abbildung 4.16 sind die Ergebnisse zu sehen. LWPR lernt in beiden Fällen schneller. Interessant ist auch, dass LWPR nach ca. 25000 Schritten wieder etwas schlechter wird. Das könnte daran liegen, dass die Reichweiten der Modelle nun steigen und daher lokale Fehler einen größeren Einfluss auf die gesamte Approximation haben. Nach etwa 45000 Schritten nimmt der nMSE dann wieder ab. Bei LWIGNG hingegen nimmt der Fehler kontinuierlich ab und erreicht bessere Resultate als LWPR nach etwa 35000 (mit 248 Neuronen) bis 40000 (mit 77 Neuronen) Schritten. Die Version mit 248 Neuronen ist dabei die ganze Zeit besser als die mit 77 Neuronen und am Ende erreichen beide ein besseres Ergebnis als LWPR. Insbesondere LWPR mit dem Löschen stark überlappender Modelle schneidet im Vergleich recht schlecht ab.
Bei einem dritten Experiment wurde das Szenario mit den 10-dimensionalen Eingabevektoren aus dem vorhergehenden Abschnitt als Grundlage genommen. Im Unterschied zu vorher wurden jetzt allerdings für jede konstante Dimensionen zwei Konstanten c_1 und c_2 zwischen 0 und 1 zufällig gewählt. Dabei geht die schließlich verwendete Konstante c während der ersten 50000 Schritte langsam von c_1 in c_2 über. Für die zweiten 50000 Schritte wird c_2 verwendet.

$$c = \begin{cases} (1 - \frac{t}{50000})c_1 + \frac{t}{50000}c_2 & t \leq 50000 \\ c_2 & sonst \end{cases}$$

Diese Werte wurden dem 2-dimensionalen Vektor hinzugefügt und der entstandene 10-dimensionale Vektor schließlich noch mit einer Rotationsmatrix multipliziert. Es stellte sich heraus, dass es egal ist, ob LWPR mit oder ohne Löschen von Modellen benutzt wird, da die Modelle hier so weit auseinander liegen und die Reichweiten so gering sind, dass Löschen ohnehin nicht vorkommt. Insgesamt verwendet LWPR im Durchschnitt 316 Modelle. Für LWIGNG wurden Läufe mit einer maximalen Neuronenanzahl von 65 (wie im Experiment mit der statischen Funktion und den 10-dimensionalen Eingabevektoren) und von 316 durchgeführt. Wie man in Abbildung 4.17 sieht, ist LWIGNG nach etwa 5000 Schritten besser als LWPR, das während der ersten 50000 Schritte nur einen nMSE von ungefähr 0.09 erreichen kann und erst besser wird, wenn die Konstanten fest sind. LWPR ist am Ende leicht besser als LWIGNG mit 65 Neuronen, aber leicht schlechter als LWIGNG mit 316 Neuronen. Aufgrund der Bewegung der Neuronen können schon mit 65 Neuronen fast dieselben Ergebnisse erreicht werden wie im ersten Experiment mit der statischen Zielfunktion. Dies zeigt den Vorteil der verwendeten adaptiven Struktur.

klären, dass die Funktion gerade zu Beginn leicht zu approximieren ist (sie ist ja fast konstant Null) und sich deswegen die Reichweiten der vorhandenen rezeptiven Felder stark ausbreiten und so insgesamt weniger neue Zentren eingefügt werden.

[10] Bei allen bisher durchgeführten Experimenten kommt das praktisch nie vor.

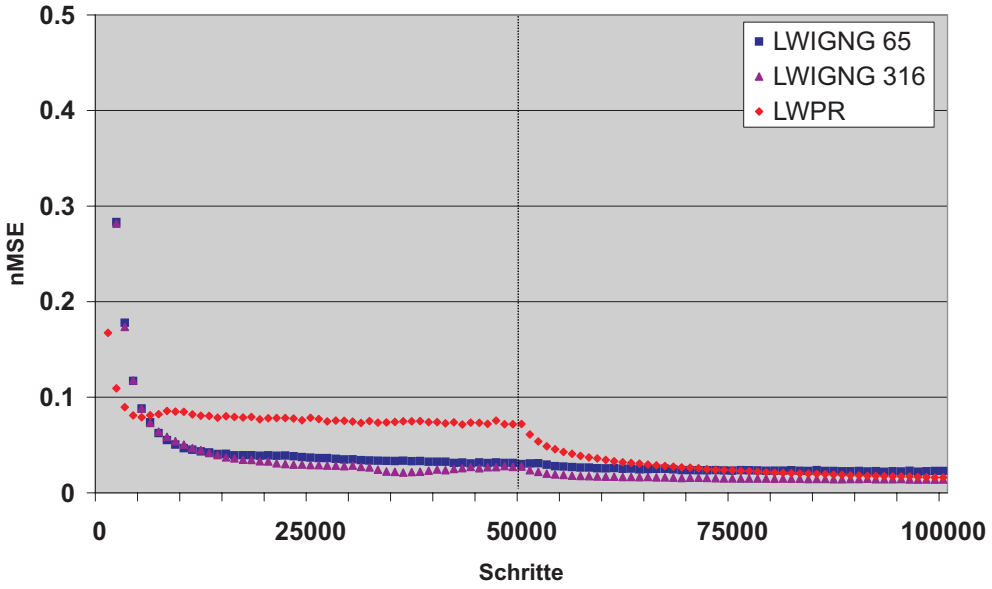


Abb. 4.17: Entwicklung des nMSE im Falle der „sich ändernden konstanten Dimensionen“ für LWPR und LWIGNG mit unterschiedlicher maximaler Neuronenanzahl. Der Strich bei 50000 Schritten markiert den Zeitpunkt, ab dem sich die Werte in den acht hinzugefügten Eingabedimensionen nicht mehr ändern.

Zusammenfassend lässt sich feststellen, dass LWIGNG wie LWPR gut zur inkrementellen Funktionsapproximation bei hoch-dimensionalen Eingaberäumen mit redundanten Eingabedimensionen geeignet ist. Verändert sich die Zielfunktion, so kann sich LWIGNG diesen Veränderungen besser anpassen. Bei Veränderungen in der Eingabeverteilung kann LWIGNG diesen aufgrund seiner flexiblen Strukturen gut folgen, während LWPR neue Modelle einfügen muss. Somit ist LWIGNG ein viel versprechender Kandidat für den Einsatz als Funktionsapproximator für Reinforcement-Learning-Verfahren.

Kapitel 5

Lernen von diskreten Aktionen: Q-LWIGNG

In diesem Kapitel geht es darum, das im vorigen Kapitel entwickelte Verfahren als Funktionsapproximator für Reinforcement-Learning-Algorithmen zum Lernen diskreter Aktionen einzusetzen. Dabei soll der Q-Learning-Algorithmus zum Einsatz kommen, da dieser als modellfreies Verfahren ein breites Anwendungsspektrum bietet. Zu diesem Zweck müssen noch einige kleinere Veränderungen an dem beschriebenen Verfahren vorgenommen werden. Zunächst werden diese Erweiterungen und der daraus resultierende Q-LWIGNG-Algorithmus beschrieben. Zur Evaluation wurden Experimente mit drei verschiedenen Szenarien durchgeführt, die als Standardprobleme des Reinforcement Learning gelten. Dementsprechend können die Ergebnisse dieser Experimente dann auch zumindest teilweise mit den in der Literatur angegebenen Resultaten verglichen werden.

5.1 Q-LWIGNG

Q-LWIGNG basiert auf dem im letzten Kapitel vorgestellten LWIGNG-Verfahren. Um dieses Verfahren als Funktionsapproximator für Q-Learning einsetzen zu können, müssen jedoch noch einige Erweiterungen durchgeführt werden. Zum einen muss bei Q-Learning für jede mögliche Aktion ein Wert approximiert werden, d.h. dass die zu approximierende Funktion nicht länger reellwertig, sondern vektorwertig ist. Die Dimension des Ausgabevektors entspricht dabei der Anzahl der möglichen Aktionen. Zum anderen haben die Eingabedaten im Lernprozess eine ganz bestimmte Reihenfolge, da sie aus den Trajektorien durch den Zustandsraum stammen. Diese Reihenfolge muss beim Training des zugrunde liegenden Wachsenden Neuralen Gases berücksichtigt werden, da es ansonsten zu einer ungünstigen Verteilung der Neuronen kommen kann.

5.1.1 Lernen aus Trajektorien

Bei den Experimenten zur Funktionsapproximation im letzten Kapitel waren die Eingabedaten zufällig verteilt. Geht es aber darum, Aktionen aus der Interaktion mit der Umwelt zu lernen, so besitzen die Eingabedaten eine bestimmte Reihenfolge und bestehen aus einzelnen Episoden, also Trajektorien durch den Zustandsraum. Diese beginnen mit einem Startzustand x_0 und enden mit einem terminalen Zustand x_T. Würde man den normalen GNG-Algorithmus verwenden, so würde diese Anordnung der Daten dazu führen, dass sich Neuronen in der Nähe der Terminalzustände konzentrieren, da die anfänglichen Positionskorrekturen von den späteren überlagert würden. Daher wird der Algorithmus so geändert, dass die Positionen der Neuronen während einer Episode (d.h. während der Zeit zwischen Startzustand x_0 und Endzustand x_T) nicht verändert und erst am Ende der Episode neu bestimmt werden. Dafür werden Hilfsvariablen $\widehat{w}_i$ zu jedem Neuron c_i hinzugefügt, in denen die Veränderungen der Position des jeweiligen Neurons während einer Episode aufaddiert werden. Zusätzlich erhält jedes Neuron einen Zähler z_i, in dem vermerkt wird, wie oft die entsprechende Hilfsvariable geändert wurde. Am Ende einer Episode werden dann die Neuronenpositionen mit Hilfe dieser Variablen auf den Durchschnitt aller akkumulierten Positionsveränderungen gesetzt:

$$\Delta w_i = \frac{\widehat{w}_i}{z_i} \text{ für alle Neuronen } c_i, i = 1, \ldots, N \text{ mit } z_i > 0 \tag{5.1}$$

Darüber hinaus wird mit dem Einfügen neuer Neuronen und Kanten und auch mit dem Löschen von Neuronen und Kanten bis zum Ende einer Episode gewartet. Zu diesem Zweck wird eine Liste P mit allen Paaren $(c_{b(x_t)}, c_{b'(x_t)})$ von nächsten und zweitnächsten Neuronen zu den Eingabevektoren x_t angelegt. Am Ende einer Episode werden dann entsprechend dieser Liste neue Kanten $e_{i,j}$ eingefügt und das Alter $age_{i,j}$ aller Kanten mit einem zugehörigen Paar in P auf Null gesetzt. Ebenso werden dann gegebenenfalls neue Neuronen eingefügt, bzw. zu alte Kanten oder Neuronen ohne Kanten gelöscht. In Abbildung 5.1 ist der gesamte Algorithmus für das Training eines solchen Netzes zu sehen. Für die Berechnung der Approximation genügt es, die Schritte 1 und 2 sowie die Schritte 8 bis 10 des ersten Abschnittes auszuführen; die restlichen Schritte betreffen nur das Training. Derselbe Algorithmus wird auch bei fehlerabhängiger Bewegung der Neuronen mit den entsprechenden Änderungen angewandt (Schritt 5 und Schritt 6 analog zu den Gleichungen (4.15) bzw. (4.17)).

5.1.2 Approximation von Aktions-Wertefunktionen

Bei Q-Learning muss die Aktions-Wertefunktion $Q(x, a)$ angenähert werden, d.h. für jeden Zustandswert x müssen mehrere Aktionswerte $Q(x, a)$ approximiert werden, je einer für jede mögliche Aktion a. Dabei sind verschiedene Möglichkeiten denkbar:

- LWIGNG wird so erweitert, dass mehrere Ausgabewerte für einen Zustand approximiert werden. Statt mit einem skalaren Wert v_i werden die einzelnen Neuronen mit Vektoren Q_i versehen. Bei der Berechnung wird dann der Vektor aller Aktionswerte $Q(x)$ für den jeweiligen Zustand x berechnet und ausgegeben.
- Die Q-Werte für die einzelnen Aktionen werden als getrennte Funktionen betrachtet. Für jede dieser Funktionen wird ein eigenes LWIGNG verwendet.
- Der Aktionswert wird einfach als zusätzliche Dimension dem Eingabewert x hinzugefügt und LWIGNG wird mit dem zusammengesetzten Vektor (x, a) trainiert.

Während einer Episode:

1. Erhalte Eingabepaar (x, y).
2. Bestimme das nächste Neuron c_b und das zweitnächste Neuron $c_{b'}$ (GNG-Algorithmus, Abb. 3.4, Schritt 2).
3. Erhöhe das Alter $age_{b,k}$ aller von c_b ausgehenden Kanten um 1.
4. Falls das Paar $(c_b, c_{b'})$ nicht in Liste P ist, füge es hinzu.
5. Akkumuliere den Quantisierungsfehler (GNG-Algorithmus, Abb. 3.4, Schritt 5).
6. Akkumuliere die Positionsveränderungen in den Hilfsvariablen $\widehat{w}$:
   $\Delta\widehat{w}_b = \alpha_b (x - w_b)$
   $\Delta\widehat{w}_{b,k} = \alpha_n (x - w_{b,k})$ für alle Nachbarn $c_{b,k}, k = 1, \ldots, N_b$ von c_b
7. Erhöhe die Zähler z_b und $z_{b,k}$ von c_b und allen Nachbarn $c_{b,k}$ um 1.
8. Berechne die lokalen Modelle $\widetilde{M}_{b,k}(x)$ gemäß (4.8).
9. Berechne die Gewichte $m_{b,k}(x)$ für die lokalen Modelle gemäß (4.11) (durchschnittliche Kantenlänge) bzw. (4.12) (zusätzliche Parameter $\lambda_{i,j}$).
10. Bestimme die endgültige Approximation $\widetilde{F}(x)$ gemäß (4.10).
11. Trainiere die lokalen Modelle $\widetilde{M}_{b,k}$ gemäß (4.13).
12. Trainiere gegebenenfalls die $\lambda_{b,k}$ gemäß (4.14).
13. Reduziere die Fehlervariablen err_i aller Neuronen (GNG-Algorithmus, Abb. 3.4, Schritt 11).
14. Falls $N + N_{insert} < N_{max}$, erhöhe den Zähler für das Einfügen neuer Neuronen N_{insert} alle N_{steps} Schritte um 1.

Am Ende einer Episode:

1. Berechne die neuen Neuronenpositionen w_i gemäß Gleichung (5.1).
2. Setze alle $\widehat{w}_i$ und alle Zähler z_i auf 0, $i = 1, \ldots, N$.
3. Falls es Paare (c_i, c_j) in P ohne gemeinsame Kante $e_{i,j}$ gibt, erzeuge diese.
4. Setze das Alter $age_{i,j}$ von Kanten $e_{i,j}$ mit Paar (c_i, c_j) in P auf 0 und lösche P.
5. Entferne alle Kanten mit $age_{i,j} > age_{max}$ und Neuronen ohne Kanten.
6. Füge N_{insert} neue Neuronen ein (analog zum GNG-Algorithmus, Abb. 3.4, Schritt 10) und setze N_{insert} gleich 0. Setze die Werte v_s der neu eingefügten Neuronen c_s gemäß der aktuellen Approximation an der jeweiligen Position w_s: $v_s = \widetilde{F}(w_s)$.

Abb. 5.1: Der erweiterte GNG-Algorithmus für Daten, die aus Trajektorien stammen.

Im Folgenden wird nur die erste Möglichkeit betrachtet. Die zweite Möglichkeit ist deutlich aufwändiger, weil zur Bestimmung der auszuführenden Aktion jeweils die Q-Werte aller möglichen Aktionen bestimmt werden müssen. Bei der zweiten Möglichkeit müssten dafür alle Gewichte und Interpolationsparameter für jedes LWIGNG gesondert berechnet werden. Die erste Möglichkeit ist da geschickter, da der komplette Q-Vektor in einem Durchgang bestimmt wird. Bei der dritten Möglichkeit besteht das Problem, dass nicht unbedingt Beziehungen zwischen den einzelnen Aktionswerten bestehen, so dass eine Abstandsmessung zwischen ihnen nicht unbedingt sinnvoll ist. Aber genau das würde passieren, betrachtete man den Aktionswert lediglich als zusätzliche Eingabedimension. Zudem wird das gesamte Funktionsapproximationsproblem durch diese zusätzliche Dimension schwieriger. Ein weiterer Vorteil der ersten Möglichkeit liegt darin, dass für das zugrunde liegende Wachsende Neurale Gas mehr Trainingsbeispiele zur Verteilung der Neuronen zur Verfügung stehen, da es unabhängig von der gewählten Aktion trainiert werden kann.
Zur Approximation der benötigten vektorwertigen Funktionen werden also die bisherigen Werte v_i an den Neuronen c_i durch Vektoren Q_i ersetzt. Die Dimension dieser Vektoren entspricht dabei der Anzahl der möglichen Aktionen N_a. Im Folgenden wird der Eintrag im Vektor Q_i, der für eine bestimmte Aktion a steht, mit $Q_i(a)$ bezeichnet.
Die Berechnung der Approximation des Zustands-Aktions-Wertes $\widetilde{Q}(x,a)$ zu einer bestimmten Aktion a in einem bestimmten Zustand x, geschieht genauso wie die Berechnung des Ausgabewertes $\widetilde{F}(x)$ in Kapitel 4. Auch das Training der Aktionswerte $Q_i(a)$ in den Neuronen kann genauso wie zuvor geschehen. Dabei werden die Trainingsbeispiele gemäß dem Q-Learning-Verfahren (vgl. Abschnitt 2.2.2) als

$$\left((x_{t-1}, a_{t-1}), r_t + \gamma \max_{a \in A} \widetilde{Q}(x_t, a) \right) \tag{5.2}$$

vorgegeben. Dabei steht x_{t-1} für den Zustand im letzten Zeitschritt (entspricht also dem Eingabevektor x bei dem LWIGNG-Verfahren) und a_{t-1} für die ausgeführte Aktion und gibt an, welcher Eintrag in den Vektoren Q_i geändert werden soll. Der Term $r_t + \gamma \max_{a \in A} \widetilde{Q}(x_t, a)$ ist eine Schätzung der zu erwartenden Belohnung[1], wenn Aktion a_{t-1} in Zustand x_{t-1} ausgeführt wird und dann die im Moment als am besten angesehenen Politik verfolgt wird. Auf die Verwendung einer Lernrate kann an dieser Stelle verzichtet werden, da das Training der Q_i in den Neuronen ohnehin mit der Lernrate α_v geschieht.

5.2 Experimente

Um das Q-LWIGNG-Verfahren zu testen, sollen Experimente mit drei unterschiedlichen Problemstellungen durchgeführt werden: mit dem Stabbalance-Problem, dem Mountain-Car-Problem und mit dem so genannten Acrobot-Problem. Diese drei Probleme sind in der Vergangenheit immer wieder als Testprobleme für Reinforcement-Learning-Verfahren verwendet worden und bieten sich daher an, um das hier vorgestellte Verfahren mit anderen Methoden vergleichen zu können. Leider unterscheiden sich sowohl die Durchführung als auch die Auswertung der Experimente in den einzelnen Studien teilweise stark voneinander. Die Vergleichbarkeit ist dadurch erheblich er-

[1] Ist dabei x_t ein Terminalzustand, so wird $\max_{a \in A} \widetilde{Q}(x_t, a)$ als Null angenommen.

schwert. Bei den Vergleichen, so überhaupt möglich, muss daher besonderes Augenmerk auf die Art der jeweiligen Ausführung der Experimente gelegt werden.
Bei der Wahl der Parameter für das zugrunde liegende Wachsende Neurale Gas wurden weitgehend die Parameter aus dem letzten Kapitel verwendet. Die meisten dieser Parameter sind für alle drei Probleme dieselben und werden in Tabelle 5.1 zusammengefasst. Als Initialwert für die Q-Werte wurde Null festgelegt, da in allen Experimenten nur negative Belohnungen verteilt werden und Null somit eine optimistische Schätzung darstellt. Dies entspricht der „exploring-starts"-Strategie aus Abschnitt 2.2.1 und führt dazu, dass zu Beginn der Läufe stark exploriert wird, da bereits ausgeführte Aktionen immer schlechter als noch nicht ausgeführte erscheinen. Parameter, die von Experiment zu Experiment variieren, wie z.B. die maximale Anzahl von Neuronen oder Lernraten, werden an der entsprechenden Stelle angegeben.
Aufgrund der hohen Anzahl der frei wählbaren Parameter, ist keine systematische Studie zu den Auswirkungen einzelner Parameter möglich. Die Parameterwerte wurden auch nicht gezielt optimiert, sondern nach einigen Vorexperimenten festgelegt. Dabei ist klar, dass die erzielten Ergebnisse natürlich sehr stark von den verwendeten Parametern abhängen und Methoden, die bei den verwendeten Parametern im Vergleich zu anderen sehr gut abschneiden, bei einer anderen Parameterwahl eventuell viel schlechter sind. Trotzdem dürften zumindest generelle Tendenzen erkennbar sein. Natürlich wäre auch eine gründliche mathematische Analyse des Verfahrens wünschenswert. Diese kann aber im Rahmen der vorliegenden Arbeit aufgrund der Komplexität des entwickelten Verfahrens nicht geleistet werden[2].

Parameter	**Wert**	**Beschreibung**
N_{steps}	100	Anzahl der Schritte zwischen dem Einfügen von Neuronen
age_{max}	100	maximales Alter für Kanten
δ_{err}	0.001	Fehlerreduktionsrate pro Schritt
δ_{new}	0.01	Fehlerreduktionsrate beim Einfügen neuer Neuronen
α_b	0.025	Lernrate für die Bewegung des Gewinnerneurons
α_n	0.01	Lernrate für die Bewegungen der Nachbarneuronen
μ	0.1	Regularisierungsparameter
λ_{start}	30	Initialwert für die $\lambda_{i,j}$ an den Neuronen und Kanten
α_l	1000	Lernrate für die Anpassung der $\lambda_{i,j}$ an den Neuronen und Kanten
β_{err}	0.05	Gewicht des aktuellen Fehlers
α_r	0.025	Lernrate für die fehlerabhängige Anpassung des Gewinnerneurons
α_q	0.01	Lernrate für die fehlerabhängige Anpassung der Nachbarneuronen
Q_0	0	Initialwert für die Q-Werte

Tabelle 5.1: Standardparameter für Q-LWIGNG bei den durchgeführten Experimenten.

Neben dem Vergleich mit anderen Verfahren aus der Literatur, soll in den Experimenten die Untersuchung der verschiedenen Aspekte der LWIGNG-Methode aus dem letzten Kapitel im Vordergrund stehen. So wurden die Experimente vor allem in Hinblick auf folgende Fragen angelegt:

1. Wie wirkt sich die Art der Bestimmung der Gewichte $m_{b,k}(x)$ für die einzelnen lokalen Modelle $\widetilde{M}_{b,k}$ bei der Approximation auf das Lernverhalten aus (vgl. Glei-

[2] Teilweise fehlen auch noch die mathematischen Werkzeuge für eine genaue Analyse der Kombination von Reinforcement Learning und Funktionsapproximation [Sutton, 1999].

chung(4.10))?
Dazu werden Untersuchungen mit unterschiedlichen Methoden zur Bestimmung dieser Gewichte gemacht:

- Nearest Neighbour („NN“): Hierbei wird nur das lokale Modell des nächsten Neurons berücksichtigt.
- Durchschnittliche Kantenlänge („Kante“): Berechnung der Gewichte $m_{b,k}(x)$ nach Gleichung (4.11).
- Zusätzliche Parameter $\lambda_{i,j}$ an den Neuronen und Kanten („Lambda“): Berechnung der Gewichte $m_{b,k}(x)$ nach Gleichung (4.12).

2. Wie wirkt sich die Bewegung der Neuronen aus? Ist es beim Lernen von diskreten Aktionen mit Q-Learning sinnvoll die Bewegung fehlerabhängig zu machen?
   Dazu wird die Art der Bewegung der Neuronen variiert:
   - Häufigkeitsabhängig: Bewegung der Neuronen wie beim gewöhnlichen GNG (siehe Abbildung 3.4) mit den Änderungen für das Lernen aus Trajektorien (Abbildung 5.1).
   - Fehlerabhängig („err.“): Akkumulation des Approximationsfehlers nach (4.15) und Bewegung der Neuronen gemäß Gleichung (4.17) unter Berücksichtigung der Änderungen für das Lernen aus Trajektorien (siehe Abbildung 5.1).
   - Keine Bewegung der Neuronen(„fix“): Alle Neuronen werden zu Beginn in einer regelmäßigen Gitterstruktur angeordnet und bewegen sich dann nicht mehr.
3. Welche Rolle spielt die maximale Anzahl von Neuronen?
   Dazu werden Experimente mit verschiedenen maximalen Neuronenanzahlen durchgeführt.
4. Wie wirkt sich die Interpolation von Q-Werten zwischen Neuronen aus?
   Die Ergebnisse einzelner Verfahren mit und ohne („const“) lokal linearer Interpolation zwischen benachbarten Neuronen werden miteinander verglichen.

Im Folgenden werden die in den Klammern angegebenen Kurzbezeichnungen zusammen mit der maximalen Neuronenanzahl als Bezeichner für die einzelnen Verfahren verwendet (z.B. „Kante err. 32“ für die Berechnung der Gewichte über die durchschnittliche Kantenlänge mit fehlerabhängiger Bewegung der Neuronen bei einer maximalen Neuronenanzahl von 32).

5.2.1 Das Stabbalance-Problem

Beim Stabbalance-Problem (auch pole balancing problem oder cart pole problem) ist ein beweglicher Stab mit einem Ende so auf einem Wagen befestigt, dass er nach rechts und links pendeln kann (siehe Abbildung 5.2). Der Wagen kann sich auf einer begrenzten Strecke nach links oder rechts bewegen. Durch die Bewegung des Wagens kippt der Stab jeweils in die Gegenrichtung. Ziel ist es, den Wagen so zu bewegen, dass der Stab balanciert wird, d.h. dass der Winkel θ zwischen der Senkrechten und dem Stab nicht zu groß wird. Zudem darf der Wagen die festgelegte Strecke nicht verlassen. Als Aktion kann der Wagen mit einer bestimmten Kraft nach rechts oder links beschleunigt werden.
Beschrieben wurde das Problem bereits in den 60er Jahren [Cannon, 1967]. Die ersten Versuche, Kontrollstrategien für dieses Problem zu lernen, stellt das *BOXES*-System dar

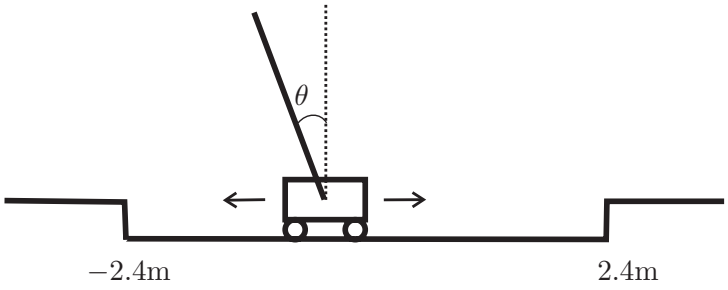


Abb. 5.2: Skizze des Stabbalance-Problems.

[Michie und Chambers, 1968a,b]. Dabei wurde der Zustandsraum diskretisiert und eine Aktions-Wertefunktion gelernt, die ähnlich wie bei Monte-Carlo-Methoden die durchschnittliche Episodenlänge für ein Zustands-Aktions-Paar approximierte. Darauf aufbauend entwickelten Barto, Sutton und Anderson ein Lernsystem, welches aus einem so genannten *associative search element* (ASE) und einem *adaptive critic element* (ACE) besteht und ein wesentlich besseres Lernverhalten aufweisen soll [Barto u. a., 1983]. Das Problem dabei liegt darin, dass ein Vorzeichenfehler bei der mathematischen Modellierung der Gravitation das Problem erheblich vereinfacht hat[3]. Dieses Lernsystem kann als eine der ersten Anwendungen der Actor-Critic-Architektur (siehe Abschnitt 2.5.2) gelten. In der Folge wurde das Problem in einer Reihe weiterer Arbeiten als Testproblem für Reinforcement-Learning-Verfahren verwendet, wobei für die Simulation meist dieselben Parameter verwendet wurden wie in [Barto u. a., 1983][4]. Allerdings gibt es erhebliche Unterschiede darin, wie lange trainiert wird und wann Episoden abgebrochen werden, sowie in der Art der Auswertung.
Die Beschreibung des Problems in der vorliegenden Arbeit folgt der Darstellung in [Barto u. a., 1983] (der Vorzeichenfehler bei der Gravitation wurde natürlich korrigiert). Der Zustand des Systems zu einem Zeitpunkt t kann durch vier Variablen beschrieben werden:

$$\begin{aligned} h_t &= \text{die Position des Wagens auf der Strecke in Metern} \\ \dot{h}_t &= \text{die Geschwindigkeit des Wagens in } \mathrm{m/s} \\ \theta_t &= \text{der Winkel zwischen Stab und der Senkrechten im Bogenmaß} \\ \dot{\theta}_t &= \text{die Winkelgeschwindigkeit des Winkels zwischen Stab und der Senkrechten} \end{aligned} \tag{5.3}$$

Die Dynamik des Systems ist durch die folgenden Bewegungsgleichungen gegeben[5]:

$$\begin{aligned} \ddot{\theta}_t &= \frac{g \sin\theta_t + \cos\theta_t \left(\frac{-a_t - m_p l \dot{\theta}^2 \sin\theta_t}{m_c + m_p} \right)}{l \left(\frac{4}{3} - \frac{m_p \cos^2\theta_t}{m_c + m_p} \right)} \\ \ddot{h}_t &= \frac{a_t + m_p l \left(\dot{\theta}_t^{\,2} \sin\theta_t - \ddot{\theta}_t \cos\theta_t \right)}{m_c + m_p} \end{aligned} \tag{5.4}$$

[3] Im Prinzip hängt der Stab durch den Vorzeichenfehler nach unten.

[4] Der Vorzeichenfehler bei der Modellierung der Gravitation scheint bei den meisten Arbeiten korrigiert zu sein. Zum Teil findet sich der Fehler aber auch in der neueren Literatur (z.B. [Paraskevopoulos u. a., 1999; Xu u. a., 2002]).

[5] Die Reibung des Wagens auf der Strecke und die Reibung des Stabes werden dabei nicht berücksichtigt. In den Arbeiten, in denen auch die Reibung simuliert wird, wird diese stets auf einen so kleinen Wert gesetzt, dass sie praktisch keine Rolle spielt (z.B. in [Barto u. a., 1983]).

Dabei haben die Parameter dieselben Werte wie in [Barto u. a., 1983] und den meisten anderen Arbeiten und werden in Tabelle 5.2 angegebenen. Die Simulation des Stabbalance-Problems erfolgt in Zeitschritten von 0.02 Sekunden, wobei der Agent in jedem Zeitschritt eine neue Aktion wählen kann. Die Bahn, auf der sich der Wagen bewegen kann, reicht von -2.4 m bis +2.4 m.

Parameter	**Wert**	**Beschreibung**
a_t	$\pm$ 10 N	die gewählte Aktion
m_c	1.0 kg	Masse des Wagens
m_p	0.1 kg	Masse des Stabes
l	0.5 m	halbe Länge des Stabes
g	9.8 m/s^2	Gravitation

Tabelle 5.2: Werte für die Simulation des Stabbalance-Problems.

Beim Stabbalance-Problem wurden zwei unterschiedliche Versionen untersucht: Bei der ersten ist der Stab zu Beginn einer neuen Episode senkrecht und der Wagen genau in der Mitte der Bahn, bei der zweiten werden für Position und Winkel zufällige Werte gewählt. Die Winkelgeschwindigkeit und die Wagengeschwindigkeit werden immer zu Beginn einer neuen Episode auf Null gesetzt. Wenn der absolute Winkel zwischen Stab und der Senkrechten größer als $12°$ ist oder der Wagen die Bahn verlässt, wird die laufende Episode abgebrochen. Bei Abbruch einer Episode gibt es eine Belohnung von -1, ansonsten ist die Belohnung in jedem Zeitschritt Null.
Die Zustandsvariablen in (5.3) werden normiert, so dass sie hauptsächlich zwischen Null und Eins liegen, bevor das Netz mit ihnen trainiert wird[6]:

$$x_t = \begin{pmatrix} (h_t + 2.4) \,/\, 4.8 \\ (\dot{h}_t + 0.6) / \, 1.2 \\ (\theta_t + 0.2) \,/\, 0.4 \\ (\dot{\theta}_t + 0.3) / \, 0.6 \end{pmatrix} \tag{5.5}$$

Als Parameterwerte für das zugrunde liegende Wachsende Neurale Gas wurden die Werte aus Tabelle 5.1 verwendet. Die Parameter für Q-Learning wurden nach einigen vorläufigen Experimenten auf bestimmte Werte festgelegt und im weiteren Verlauf nicht weiter variiert. Als Lernrate α_v für das Training der Vektoren Q_i an den einzelnen Neuronen wurde 0.1 verwendet. Der Diskontierungsfaktor γ wurde auf 0.9 gesetzt. Der Explorationsparameter ε betrug 0. Es wurde also immer diejenige Aktion ausgeführt, die gerade als beste Aktion galt. Eine ausreichende Exploration wurde durch die „exploring-starts“-Strategie (siehe Abschnitt 2.2.1) sichergestellt, indem alle Q-Werte zu Beginn auf Null gesetzt wurden. Es wurden Experimente mit maximalen Neuronenanzahlen zwischen 16 und 256 durchgeführt. Bei jedem Experiment wurden jeweils 100 Läufe durchgeführt und die durchschnittliche Anzahl von Episoden bis es gelang, den Stab 1000, 10000 und 100000 Schritte zu balancieren, berechnet. Gelang es innerhalb von 10000 Episoden nicht, den Stab 100000 Schritte zu balancieren, wurde der komplette Lauf abgebrochen.

[6] Alle Winkel werden im Gradmaß angegeben. Die maximal auftretenden Wagen- bzw. Winkelgeschwindigkeiten $\dot{h}_t$ und $\dot{\theta}_t$ wurden auf 0.6 bzw. 0.3 geschätzt.

Stabbalance mit festen Startbedingungen

Bei dieser Versuchsreihe wurden der Winkel und die Position des Wagens zu Beginn jeder Episode auf Null gesetzt. Dies entspricht dem Problem wie es u.a. in [Sutton und Barto, 1998] beschrieben ist. Da als Aktion nur die Beschleunigung nach rechts oder links zur Verfügung steht, gibt es nicht die Möglichkeit einfach stehen zu bleiben, um den Stab zu balancieren. Zunächst wurden Experimente mit verschiedenen Verfahren und einer maximalen Neuronenanzahl von 32 ausgeführt. Da bei 32 Neuronen Kanten-Verfahren am besten abschnitten, wurde bei diesen noch die maximale Neuronenanzahl variiert. Bei maximal 16 Neuronen wurde mit 2 Neuronen gestartet, bei maximal 32 oder 81 Neuronen mit 16 gleichmäßig verteilten Startneuronen.
Tabelle 5.3 zeigt den prozentualen Anteil der abgebrochenen Läufe (also Läufe ohne Erfolg bei 10000 Episoden) bei 100 Läufen und die durchschnittliche Anzahl von Episoden, die benötigt wurde, bis der Stab das erste mal 100000 Schritte balanciert wurde. Die Durchschnittswerte werden auch in den Abbildungen 5.3 und 5.4 graphisch dargestellt. Die Durchschnitte wurden jeweils nur für nicht abgebrochene Läufe berechnet.

Methode	**abgebrochene Läufe**	**Anzahl Episoden bis 100000**
Kante 16	62 %	349.5
Kante err. 16	43 %	260.4
Kante fix 16	100 %	-
NN const 32	7 %	278.9
NN 32	8 %	343.4
NN err. 32	0 %	703.5
Kante const 32	0 %	547.3
Kante 32	0 %	146.7
Kante err. 32	2 %	105.9
Lambda const 32	0 %	186.2
Lambda 32	0 %	479.6
Lambda err. 32	3 %	340.0
Kante 81	4 %	332.7
Kante err. 81	1 %	135.7
NN fix 81	21 %	665.1
Kante fix 81	5 %	791.5
Lambda fix 81	2 %	545.1
Kante fix 256	11 %	770.3

Tabelle 5.3: Prozentualer Anteil der abgebrochenen Läufe bei insgesamt 100 Läufen und durchschnittliche Anzahl von Episoden pro Lauf bis der Stab 100000 Schritte balanciert wird (Stabbalance-Problem mit festen Startbedingungen).

Werden lediglich 16 Neuronen verwendet, so kommt es zu einer großen Anzahl von Abbrüchen aufgrund der 10000-Episoden-Grenze. Bei 32 Neuronen und Kanten- oder Lambda-Verfahren kommt es zu überhaupt keinen Abbrüchen, solange keine fehlerabhängige Bewegung der Neuronen verwendet wird. Fehlerabhängige Bewegung führt zwar zu schnellerem Lernen, es kann aber, wenn auch selten, zu Abbrüchen kommen. Interessanterweise ist dies bei NN-Verfahren genau andersherum: Hier kommt es

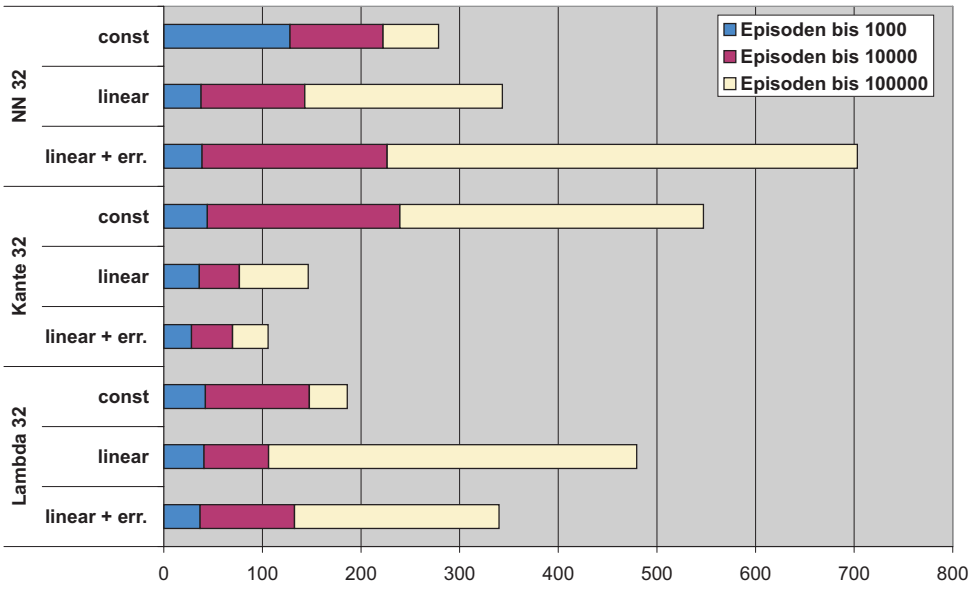


Abb. 5.3: Durchschnittliche Anzahl von Episoden bis 1000, 10000 bzw. 100000 Schritte balanciert wird für 32 Neuronen und unterschiedliche Methoden (Stabbalance-Problem mit festen Startbedingungen).

nur mit der fehlerabhängigen Bewegung zu keinen Abbrüchen, aber dafür dauert es länger, bis der Stab 100000 Schritte balanciert wird.
Betrachtet man die durchschnittliche Anzahl von Episoden bis 100000 Schritte erreicht werden für 32 Neuronen (Abbildung 5.3), so kann man feststellen, dass Kanten-Verfahren mit lokal linearer Interpolation mit Abstand am besten abschneiden. Am schnellsten wird dabei bei fehlerabhängiger Bewegung der Neuronen gelernt. Auch bei den Lambda-Verfahren bringt die fehlerabhängige Bewegung der Neuronen Vorteile bei der Lerngeschwindigkeit. Allerdings sind Lambda-Verfahren, außer bei Verwendung der konstanten Approximation, deutlich schlechter als Kanten-Verfahren. Sowohl bei NN-Verfahren als auch bei Lambda-Verfahren ist die konstante Approximation deutlich besser als die lineare Interpolation. Es fällt außerdem auf, dass NN- und Lambda-Verfahren mit konstanter Approximation zwar länger als mit linearer Interpolation brauchen, um den Stab 10000 Schritte zu balancieren, dann aber auch sehr schnell 100000 Schritte balancieren können. Das könnte daran liegen, dass diese Verfahren weniger stark generalisieren und somit lange brauchen, um überhaupt Lösungen zu finden, diese dann aber gleich eine gute Qualität aufweisen. Bei dem NN-Verfahren und bei dem Lambda-Verfahren mit konstanter Approximation ist der Einfluss auf benachbarte Neuronen entweder gar nicht vorhanden oder wird gesondert über die Parameter $\lambda_{i,j}$ geregelt. Daher braucht „NN const 32“ auch lange, bis der Stab überhaupt 1000 Schritte balanciert wird.
In Abbildung 5.4 ist zu sehen, dass 32 eine gute Wahl für die maximale Anzahl von Neuronen ist. Bei der Verwendung von 16 oder 81 bzw. 256 Neuronen brauchen die Kanten-Verfahren deutlich länger, um die 100000 Schritte zu erreichen. Bei 81 Neuronen verschlechtert die fehlerabhängige Bewegung der Neuronen das Ergebnis sogar noch weiter. Noch schlechter ist das Ergebnis allerdings, wenn sich die Neuronen gar nicht bewegen. Bei 16 Neuronen und festem Netz gelingt es überhaupt nicht, den Stab 100000 Schritte zu balancieren. Bei 81 oder 256 Neuronen dauert es deutlich länger als bei allen anderen getesteten Verfahren. NN- und Lambda-Verfahren sind zwar bei festem Netz und 81 Neuronen etwas schneller als das Kanten-Verfahren, aber die Ergebnisse sind immer noch

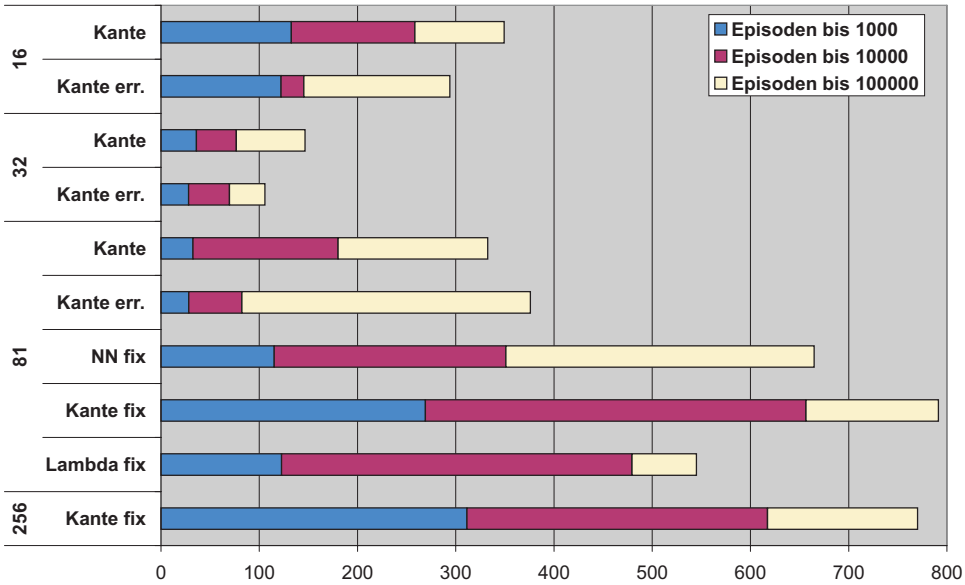


Abb. 5.4: Durchschnittliche Anzahl von Episoden bis 1000, 10000 bzw. 100000 Schritte balanciert wird für unterschiedliche Neuronenanzahlen und unterschiedliches Vorgehen bei der Bewegung der Neuronen (Stabbalance-Problem mit festen Startbedingungen).

deutlich schlechter als für Kanten-Verfahren mit beweglichen Neuronen. Wahrscheinlich liegen die festen Neuronen an ungünstigen Stellen für eine gute Approximation der Aktions-Wertefunktion. Hier bringt die adaptive Struktur des zugrunde liegenden Wachsenden Neuralen Gases also deutliche Vorteile.

Stabbalance mit zufälligen Startbedingungen

Dieselben Experimente wie im letzten Abschnitt wurden auch mit zufälligen Startbedingungen durchgeführt, d.h. dass die Position des Wagens und der Winkel des Stabes zu Beginn zufällig innerhalb der erlaubten Grenzen gewählt werden. Im Unterschied zu den Experimenten mit festen Startbedingungen gibt es Abbrüche aufgrund der 10000-Episoden-Grenze diesmal nur bei 16 Neuronen oder festem Netz (siehe Tabelle 5.4). Durch die unterschiedlichen Startbedingungen wird der Algorithmus mit einer größeren Anzahl von unterschiedlichen Zuständen konfrontiert und exploriert somit stärker. Insgesamt schneiden die Kanten-Verfahren mit linearer Interpolation etwas schlechter, die anderen Verfahren eher etwas besser als bei festen Startbedingungen ab. Wie in Abbildung 5.5 zu sehen ist, sind die Kanten-Methoden aber immer noch am besten. Diesmal ist die Kanten-Methode mit häufigkeitsabhängiger Bewegung der Neuronen etwas schneller als die mit fehlerabhängiger Bewegung. Auch bei den anderen Verfahren führt die fehlerabhängige Bewegung zu schlechteren Ergebnissen. Wahrscheinlich werden die Neuronen dabei nicht ausreichend im Zustandsraum verteilt. Die Bandbreite der überhaupt auftretenden Zustände dürfte durch die zufälligen Startbedingungen etwas größer als bei festen Startbedingungen sein[7]. Es kommt hinzu, dass beim Q-Learning Fehler in der Aktions-Wertefunktion nicht unbedingt negative Auswirkungen haben müssen, solange die Rangfolge der besten Aktionen nicht vertauscht wird. Bei der konstanten Approxi-

[7] Ist bei festen Startbedingungen einmal eine gute Politik gefunden, treten viele Zustände gar nicht mehr auf, weil immer mit demselben Startzustand begonnen wird und dann wieder einer ähnlichen Trajektorie durch den Zustandsraum wie beim letzten Mal gefolgt wird.

Methode	**abgebrochene Läufe**	**Anzahl Episoden bis 100000**
Kante 16	42 %	453.3
Kante err. 16	49 %	408.3
Kante fix 16	100 %	-
NN const 32	0 %	199.0
NN 32	0 %	311.6
NN err. 32	0 %	432.7
Kante const 32	0 %	255.6
Kante 32	0 %	157.2
Kante err. 32	0 %	169.3
Lambda const 32	0 %	211.9
Lambda 32	0 %	253.7
Lambda err. 32	0 %	342.9
Kante 81	0 %	164.8
Kante err. 81	0 %	175.3
NN fix 81	0 %	520.9
Kante fix 81	3 %	1193.2
Lambda fix 81	1 %	672.1
Kante fix 256	6 %	712.6

Tabelle 5.4: Prozentualer Anteil abgebrochener Läufe bei insgesamt 100 Läufen und durchschnittliche Anzahl von Episoden pro Lauf bis der Stab 100000 Schritte balanciert wird (Stabbalance-Problem mit zufälligen Startbedingungen).

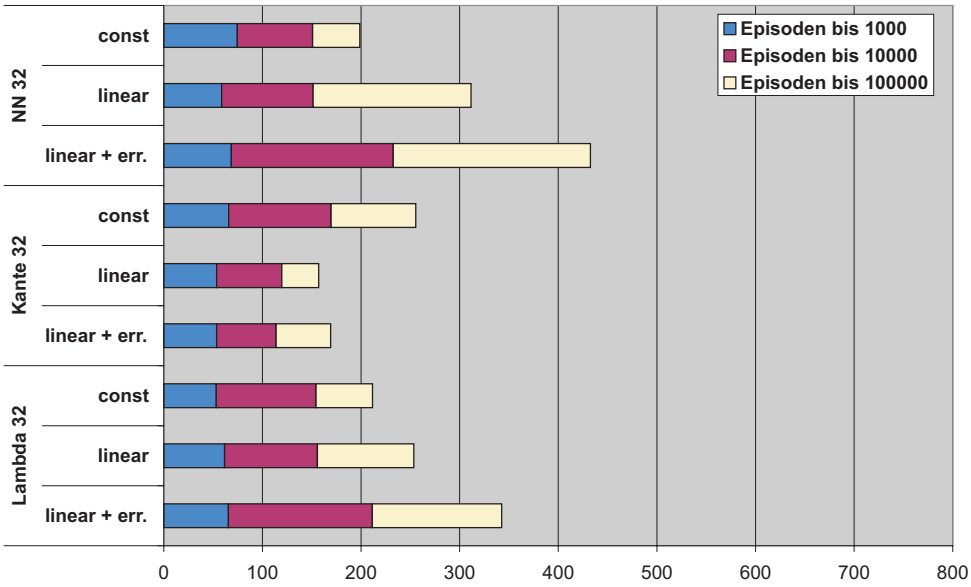


Abb. 5.5: Durchschnittliche Anzahl von Episoden bis 1000, 10000 bzw. 100000 Schritte balanciert wird für 32 Neuronen und unterschiedliche Methoden (Stabbalance-Problem mit zufälligen Startbedingungen).

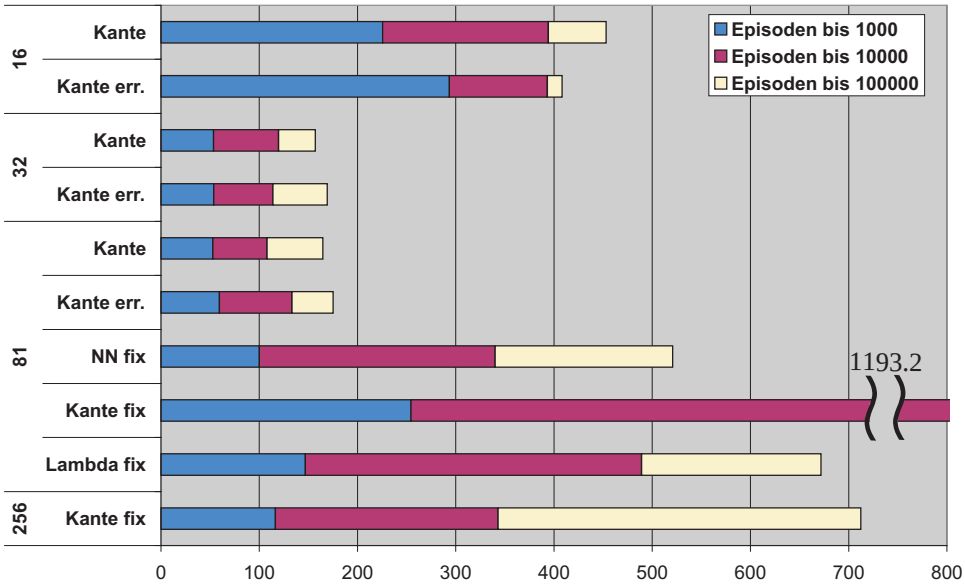


Abb. 5.6: Durchschnittliche Anzahl von Episoden bis 1000, 10000 bzw. 100000 Schritte balanciert wird für unterschiedliche Neuronenanzahlen und unterschiedliches Vorgehen bei der Bewegung der Neuronen (Stabbalance-Problem mit zufälligen Startbedingungen).

mation gibt es einen ähnlichen Effekt wie bei festen Startbedingungen: Wieder schneidet die konstante Approximation bei den NN-Verfahren und den Lambda-Verfahren besser und bei den Kanten-Verfahren schlechter als die lineare Interpolation ab.
Auch mit 81 Neuronen erreichen die Kanten-Verfahren diesmal gute Werte (vergleiche Abbildung 5.6). Bei nur 16 Neuronen sind die Werte hingegen etwas schlechter als bei festen Startbedingungen, was wieder daran liegen dürfte, dass sich die auftretenden Zustände bei zufälligen Startbedingungen stärker unterscheiden. Bei festem Netz sind die durchschnittlichen Werte wieder äußerst schlecht. Mit 16 Neuronen konnte wiederum nicht gelernt werden, den Stab 100000 Schritte zu balancieren und bei 81 und 256 Neuronen brauchen die Verfahren mit festem Netz wieder deutlich länger als alle anderen getesteten Verfahren.

Vergleich mit anderen Verfahren

Im Folgenden sollen nun die hier erreichten Ergebnisse mit denen aus der Literatur verglichen werden. Leider ist bei einem Teil der Ergebnisse in der Literatur kein Vergleich möglich, da sich die dort durchgeführten Experimente erheblich von den hier durchgeführten unterscheiden. So sind etwa Lernziele oder Belohnungsfunktionen anders gewählt, so dass beispielsweise der Betrag des Winkels θ mit in die Strafe einfließt, was das Problem vereinfacht. Zum Teil wurden auch ganz andere Parameter für die Massen oder für die Stablänge gewählt. Zudem ist die Art der Auswertung der Experimente keineswegs einheitlich[8]. Der Vergleich muss sich daher auf den Teil der Literatur beschränken, in denen Experimente zumindest hinreichend ähnlich durchgeführt wurden.

- In [Anderson, 1993] werden *Resource-Allocation*-Netzwerke [Platt, 1991a,b], eine Variante von Backpropagation-Netzen, benutzt, um die Zustands-Aktions-Werte der

[8] Zum Teil ist die Beschreibung auch unvollständig, so dass nicht genau klar ist, wie die Experimente überhaupt durchgeführt wurden.

Q-Funktion zu lernen. Ziel war, den Stab 90000 Schritte zu balancieren. Gestartet wurde mit aufrechtem Stab und einer Winkelgeschwindigkeit von Null. Die beste Methode benötigte im Durchschnitt, gemittelt über 30 Läufe, ungefähr 3300 Episoden bis das Ziel erreicht wurde. Im Gegensatz dazu benötigen die hier verwendeten Verfahren ungefähr zwischen 100 und 800 Episoden.

- Ein ähnlicher Ansatz wie bei Selbstorganisierenden Karten wird in [Finton und Yu, 1994] benutzt, um „wichtige“ Merkmale im Eingangsraum zu entdecken. Die Aktions-Wertefunktion wird dabei mit einer „Fuzzy“-Variante von Q-Learning gelernt. Dabei wurden durchschnittlich 458 Episoden benötigt, um den Stab 100000 Schritte zu balancieren[9]. Dieser Wert wird bei allen hier durchgeführten Experimenten mit beweglichen Neuronen unterschritten.

- In [Pyeatt und Howe, 1998] wird eine *Decision-Tree-Reinforcement-Learning*-Methode verwendet. Diese wird mit Neuronalen Netzen und einer einfachen Tabellen-Methode verglichen[10]. Der Decision-Tree-Methode gelingt es nach weniger als 2000 Episoden den Stab „immer“ 2000 Schritte zu balancieren. Mit der tabellierten Aktions-Wertefunktion werden dafür etwa 5000 Episoden benötigt. Mit dem Neuronalen Netz wird dieses Ziel zumindest innerhalb von 20000 Episoden nicht erreicht. Bei diesen Experimenten geht es also auch um die Stabilität der erreichten Lösung und sie sind deswegen schlecht mit den hier durchgeführten Experimenten vergleichbar, bei denen der Stab einmal 100000 Schritte balanciert werden musste.

- In [Anderson, 1987] werden zwei Neuronale Netze mit Backpropagation-Lernregel verwendet: eines zur Evaluation des Zustandes und eines zur Auswahl der Aktionen. Sind die Netze lediglich einlagig, so werden keine befriedigenden Resultate erreicht. Mit zweilagigen Netzen gelingt es nach etwa 5000 Episoden den Stab 100000 Schritte zu balancieren. Das ist deutlich länger als mit den in der vorliegenden Arbeit entwickelten Verfahren.

- In [Barreca und Buttazzo, 1993] wird eine Kohonenkarte zur Klassifizierung der Zustände in einer Actor-Critic-Architektur benutzt. Die Parameter der Kohonenkarte werden dabei so gewählt, dass diese sich nach ungefähr 50000 Schritten nicht mehr ändert. Die Anfangsbedingungen waren zufällig. Die besten Ergebnisse ergaben sich, wenn erfolglose Episoden nach 10000 Schritten abgebrochen wurden. Nach 700 Episoden war die beste Methode in der Lage den Stab für „eine ziemlich lange“ Zeit zu balancieren.

- In [Gullapalli, 1992a] werden *Stochastic Real-Valued Units (SRV)* benutzt, die eine kontinuierliche Ausgabe produzieren. Es ist nicht klar, ob die Verwendung von kontinuierlichen Aktionen das Problem verschärft oder vereinfacht. Die Startbedingungen sind zufällig. Der Stab soll lediglich 1000 Schritte am Stück balanciert werden. Dennoch benötigen die dort benutzten Reinforcement-Learning-Methoden dafür deutlich mehr Schritte als Q-LWIGNG bis zum Erreichen der 100000 Schritte braucht. So benötigt das beste der dort eingesetzten Verfahren im Durchschnitt von 100 Trainingsläufen über 2300 Episoden, um den Stab die 1000 Schritte zu balancieren.

[9] Anscheinend bei festen Startbedingungen.

[10] Leider werden weder zum genauen Ablauf des Experiments noch zu den Methoden genaue Angaben gemacht. Da [Barto u. a., 1983] zitiert wird, kann man annehmen, dass feste Startbedingungen und die üblichen Parameter benutzt wurden.

- In [Kimura und Kobayashi, 1998] wird ein REINFORCE-Verfahren [Williams, 1992] benutzt, um kontinuierliche Aktionen für das Stabbalance-Problem zu lernen. Dem Verfahren gelingt es nach etwa 140 Episoden den Stab mehr als 4000 Schritte zu balancieren. Die Lerngeschwindigkeit scheint mit unseren Ergebnissen vergleichbar, allerdings können die dabei verwendeten kontinuierlichen Aktionen im Bereich $[-20N, 20N]$ liegen, während in den hier durchgeführten Experimenten lediglich $\pm 10N$ benutzt werden, was das hier bearbeitete Problem etwas schwieriger machen dürfte.

- In [Likas und Lagaris, 1999] wird eine direkte Reinforcement-Learning-Methode zusammen mit einer nicht-linearen Optimierungsmethode verwendet. Ziel ist es, den Stab 120000 Schritte zu balancieren. Die Startbedingungen sind zufällig. Der Algorithmus benötigt im Durchschnitt von 50 Läufen 2250 Episoden, bis dieses Ziel erreicht wird. Das ist zwar wesentlich besser, als bei den in [Likas und Lagaris, 1999] zum Vergleich herangezogenen Methoden mit einem *adaptive heuristic critic (AHC)* [Anderson, 1987] (6175 Episoden) und mit genetischem Reinforcement Learning (4097 Episoden), aber doch deutlich langsamer, als die hier vorgestellten Methoden.

- Ein spezieller neuro-genetischer Algorithmus (*Symbiotic Adaptive Neuro-Evolution*, SANE) wird in [Moriarty und Mikkulainen, 1996] verwendet. Ziel ist wiederum, den Stab 120000 Schritte zu balancieren. Es werden sowohl Experimente mit festen als auch mit zufälligen Startbedingungen durchgeführt. Der Algorithmus benötigt im Durchschnitt von 50 Läufen 535 Episoden bei festen und 1691 Episoden bei zufälligen Startbedingungen. Der verwendete Algorithmus ist jedoch besonders in Hinblick auf Recheneffizienz entwickelt worden. Möglicherweise ist dieser Algorithmus was die Rechenzeit anbelangt schneller als Q-LWIGNG, braucht dafür aber mehr Episoden.

Wie zu sehen ist, erzielt Q-LWIGNG also deutlich bessere Resultate bei der durchschnittlichen Anzahl der benötigten Episoden als vergleichbare Verfahren aus der Literatur. So braucht das „Kante err. 32“-Verfahren bei festen Startbedingungen durchschnittlich nur 105.9 Episoden, um den Stab 100000 Schritte zu balancieren. Bei zufälligen Startbedingungen benötigt „Kante 32“ lediglich 157.2 Episoden im Durchschnitt. Die hohe Lerngeschwindigkeit von Q-LWIGNG könnte daraus resultieren, dass Q-LWIGNG mit einer geringen Anzahl von Neuronen auskommt und somit von Anfang an eine gute Generalisierung erreicht.

5.2.2 Das Mountain-Car-Problem

Beim Mountain-Car-Problem soll ein Fahrzeug einen steilen Berg hochfahren (siehe Abbildung 5.7). Das Problem besteht darin, dass der Antrieb nicht ausreicht, um direkt vom Tal auf den Berg zu fahren. D.h. das Fahrzeug muss zunächst in die entgegengesetzte Richtung beschleunigen und durch hin- und herschaukeln genug Energie ansammeln, um schließlich auf den Gipfel zu gelangen. Im Gegensatz zum Stabbalance-Problem muss also kein Zustand vermieden werden, sondern ein bestimmtes Ziel erreicht werden. Das macht das Problem etwas schwieriger, weil dieses Ziel zunächst einmal mehr oder weniger zufällig erreicht werden muss[11], bevor es dafür eine Belohnung geben kann. Es

[11] Dabei hilft die „exploring-starts“-Strategie ein wenig, weil unbekannte Aktionen bevorzugt werden.

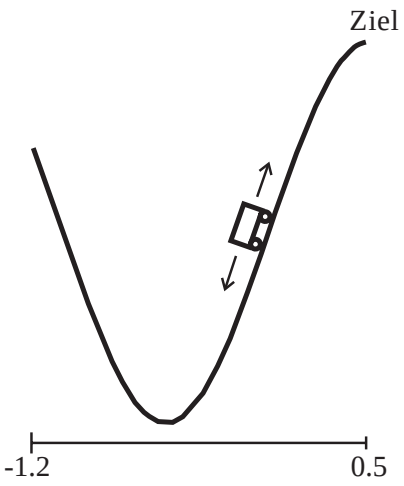


Abb. 5.7: Skizze des Mountain-Car-Problems.

stehen drei Aktionen a zur Verfügung: nach links beschleunigen (-1), stehen bleiben (0) und nach rechts beschleunigen (+1). Die Position h_t und die Geschwindigkeit $\dot{h}_t$ werden wie folgt berechnet [Sutton und Barto, 1998]:

$$\begin{aligned} h_{t+1} &= h_t + \dot{h}_{t+1} \\ \dot{h}_{t+1} &= \dot{h}_t + 0.001 a_t - 0.0025 \cos(3 h_t) \end{aligned} \tag{5.6}$$

Das Problem ist gelöst, wenn $h_t \geq 0.5$ ist. Ist $h_t \leq -1.2$, wird h_t auf -1.2 und $\dot{h}_t$ auf Null gesetzt. Die Geschwindigkeit wird auf Werte aus dem Intervall $[-0.07, 0.07]$ beschränkt. Die Belohnung ist -1 in jedem Schritt und Null bei Erreichen des Ziels. Wenn bei einer Episode innerhalb von 100000 Schritten das Ziel nicht erreicht wird, wird die laufende Episode abgebrochen. Gestartet wird jeweils mit zufälligen Startwerten aus den angegebenen Intervallen.
Der Eingabevektor x_t für die Verfahren besteht aus der normierten Position h_t und der normierten Geschwindigkeit $\dot{h}_t$:

$$x_t = \begin{pmatrix} (h_t + 1.2) \,/\, 1.7 \\ (\dot{h}_t + 0.07) / \, 0.14 \end{pmatrix} \tag{5.7}$$

Der Diskontierungsfaktor γ wurde auf 1 gesetzt, d.h. die Q-Funktion gibt einfach die negative Zeit bis zum Erreichen des Ziels an. Der Explorationsparameter ε wurde wieder auf 0 gesetzt. Als Lernrate α_v wurde 0.7 verwendet, da dieser Wert in [Sutton und Barto, 1998] die besten Ergebnisse erzielte. Die verschiedenen Methoden wurden mit einer maximalen Neuronenanzahl von 16 getestet. Zusätzlich wurden bei den Kanten-Verfahren auch Experimente mit einer maximalen Neuronenanzahl von 9 und von 25 durchgeführt. Dabei wurde jeweils schon mit der maximalen Anzahl von Neuronen gestartet, die auf einem regelmäßigen Gitter im Eingaberaum verteilt wurden.
Insgesamt wurden für jedes Experiment 100 Läufe durchgeführt. Tabelle 5.5 zeigt den prozentualen Anteil der abgebrochenen Episoden und die durchschnittliche Anzahl der benötigten Schritte pro Episode über alle 20 Episoden[12]. Zusätzlich wird auch die durchschnittliche Anzahl der Schritte pro Episode für die letzten 5 Episoden getrennt angegeben, um zu zeigen, wie gut die Verfahren gelernt haben. Die Ergebnisse werden auch in den Abbildungen 5.8 und 5.9 graphisch dargestellt.

[12] Dieses Maß für den Erfolg des Lernprozesses wurde auch von [Sutton und Barto, 1998] gewählt.

Methode	abgebrochene Episoden	Schritte / Episode alle 20 Episoden	Schritte / Episode letzte 5 Episoden
Kante 9	0.00 %	118.0	61.1
Kante err. 9	0.00 %	118.7	63.2
Kante fix 9	0.00 %	117.4	60.5
NN const 16	10.80 %	12226.2	5971.8
NN 16	0.00 %	164.2	82.9
NN err. 16	0.00 %	159.2	77.4
NN fix 16	0.00 %	151.1	76.8
Kante const 16	0.00 %	208.1	100.2
Kante 16	0.00 %	117.2	59.1
Kante err. 16	0.00 %	124.4	61.5
Kante fix 16	0.00 %	118.8	62.0
Lambda const 16	11.95 %	13503.1	4402.3
Lambda 16	17.20 %	17655.3	21812.8
Lambda err. 16	16.65 %	17253.6	20521.8
Lambda fix 16	15.75 %	16303.7	15337.2
Kante 25	0.00 %	154.4	72.0
Kante err. 25	0.00 %	152.9	62.6
Kante fix 25	0.00 %	146.3	61.1

Tabelle 5.5: Der prozentuale Anteil von abgebrochenen Episoden aufgrund der 100000-Schritte-Grenze in 100 Läufen und die durchschnittliche Anzahl der Schritte pro Episode während aller 20 und der letzten 5 Episoden (gemittelt über 100 Läufe) beim Mountain-Car-Problem.

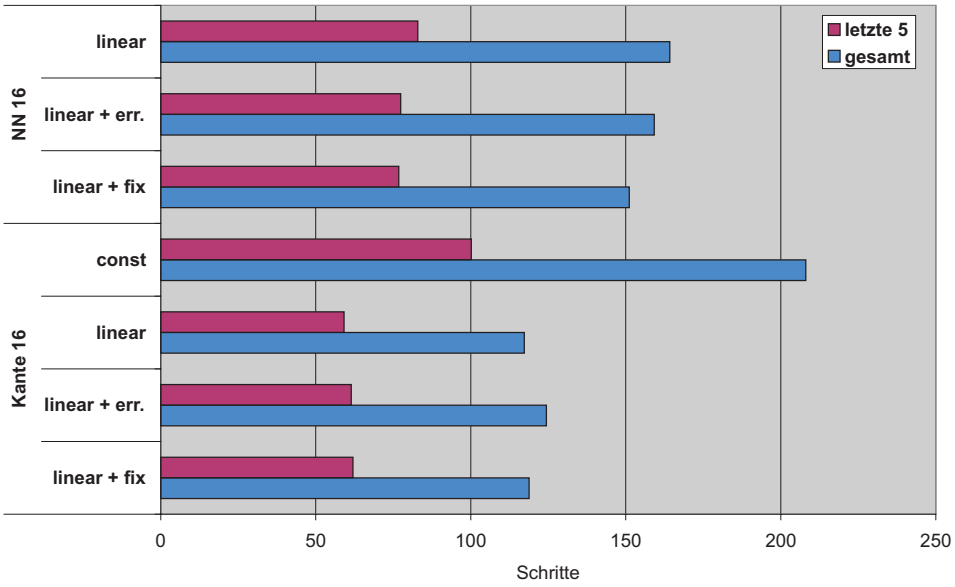


Abb. 5.8: Durchschnittliche Anzahl von Schritten pro Episode (gemittelt über 100 Läufe) bei 16 Neuronen und unterschiedlichen Methoden (Mountain-Car-Problem).

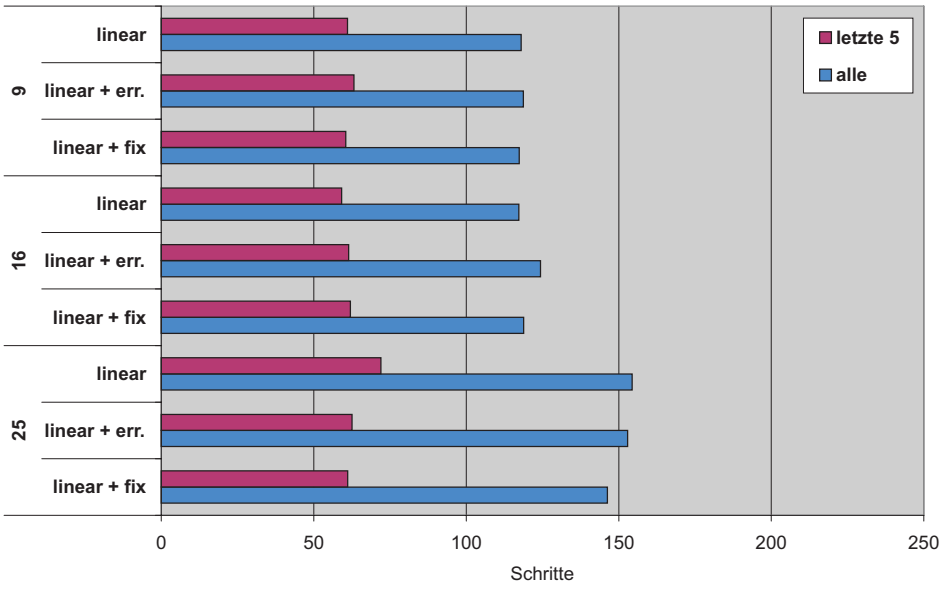


Abb. 5.9: Durchschnittliche Anzahl von Schritten pro Episode (gemittelt über 100 Läufe) bei Kanten-Verfahren für unterschiedliche Neuronenanzahlen und unterschiedliches Vorgehen bei der Bewegung der Neuronen (Mountain-Car-Problem).

Wie man in Tabelle 5.5 sieht, kommt es nur bei „NN const 16“ und bei den Lambda-Verfahren zu Abbrüchen. Allen anderen Verfahren gelingt es, das Problem nach nur 20 Episoden gut zu lösen. Das Problem bei den Lambda-Verfahren könnte darin bestehen, dass die Einstellung der $\lambda_{i,j}$ an den Neuronen und Kanten nicht gelingt, weil sich die zu approximierende Aktions-Wertefunktion zu schnell ändert. „NN const 16“ versagt wohl deshalb, weil die konstante Approximation bei nur 16 Neuronen zu wenig Variationsmöglichkeiten bietet, um die Wertefunktion hinreichend genau zu approximieren.
In Abbildung 5.8 werden diejenigen Verfahren mit maximal 16 Neuronen verglichen, bei denen es zu keinen Abbrüchen kam. Wie schon beim Stabbalance-Problem schneiden die Kanten-Verfahren mit lokal linearer Interpolation am besten ab. Dabei ist die Art der Bewegung der Neuronen hier ziemlich unerheblich: Fehlerabhängige, häufigkeitsabhängige und keine Bewegung der Neuronen führen zu fast gleichen Ergebnissen. Das könnte damit zu tun haben, dass sich wegen der geringen Anzahl von nur 20 Episoden pro Lauf die Neuronenpositionen nicht stark verändern. Die NN-Verfahren sind nur etwas langsamer als die Kanten-Verfahren. Auch bei diesen spielt die Art der Neuronenbewegung keine große Rolle. Abbildung 5.9 zeigt eine graphische Darstellung der Werte aus Tabelle 5.5 für Kanten-Verfahren mit unterschiedlicher Neuronenanzahl. Die Neuronenanzahl hat einen geringen Einfluss auf das Lernverhalten. Nur bei 25 Neuronen sind die Werte für den Durchschnitt über alle 20 Episoden schlechter als bei weniger Neuronen, was daran liegen dürfte, dass bei 25 Neuronen nicht so stark generalisiert wird und daher das Lernen etwas langsamer vonstatten geht. In den Lernkurven in Abbildung 5.10 kann man gut sehen, wie rasch die Kanten-Verfahren bei 9 oder 16 Neuronen lernen. Nach weniger als 10 Episoden scheint bereits eine optimale Politik gefunden worden zu sein.
Wie die vom „Kante 16“-Verfahren gelernte Wertefunktion $V(x) = \max_a Q(x,a)$ nach 20 Episoden aussieht, ist in Abbildung 5.11 zu sehen. Erwartungsgemäß hat diese hohe Werte bei einer Position von 0.5, da dort das Ziel ja erreicht ist. Überraschend ist vielleicht, dass auch in den Ecken bei -1.2 hohe Werte erreicht werden. Das dürfte aber daran liegen, dass diese Bereiche gar nicht oder nur selten besucht wurden und sich daher dort

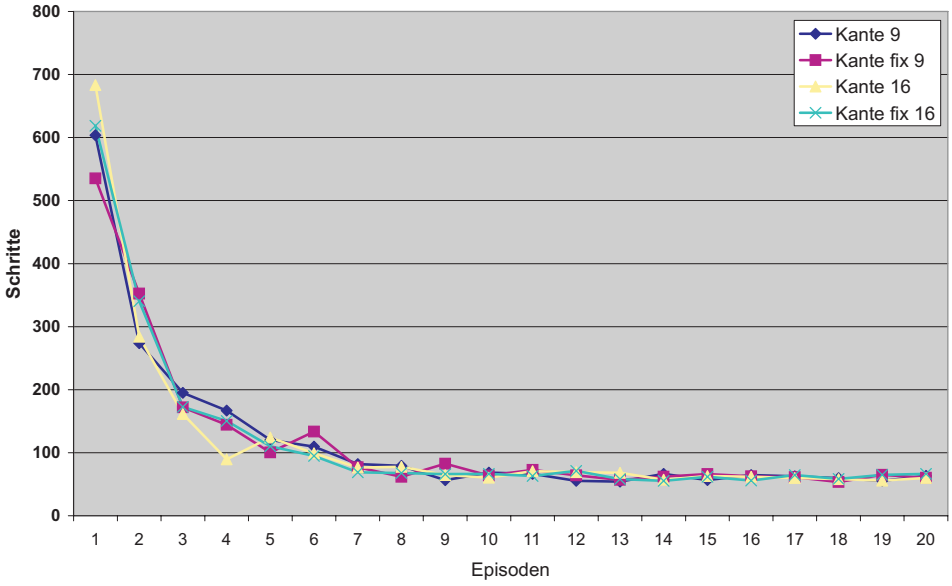


Abb. 5.10: Die Entwicklung der durchschnittlichen Schritteanzahl pro Episode (gemittelt über 100 Läufe) für verschiedene Kanten-Verfahren (Mountain-Car-Problem).

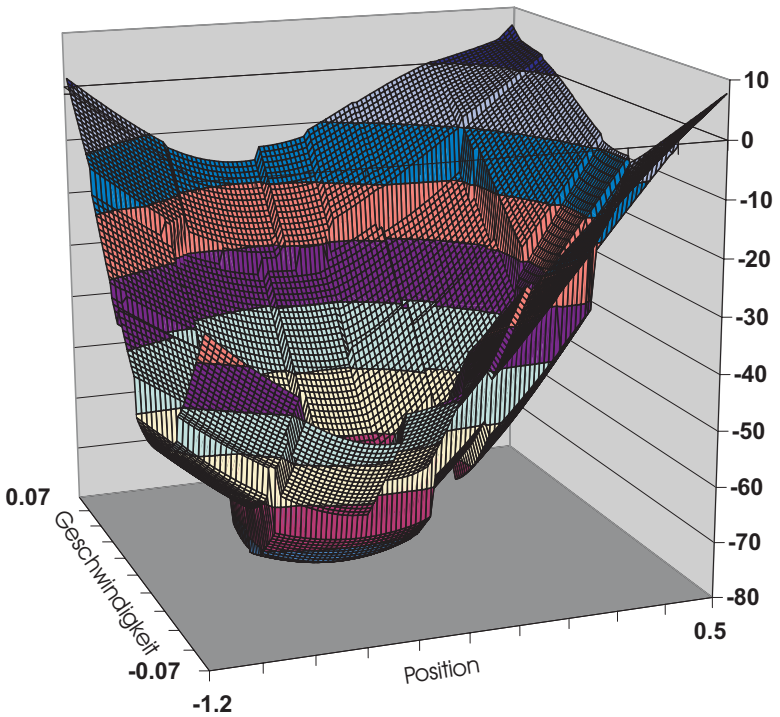


Abb. 5.11: Eine von einem „Kante 16“-Verfahren gelernte Wertefunktion nach 20 Episoden (Mountain-Car-Problem).

noch nicht die „wahren“ Werte finden. Ihr Minimum erreicht die Wertefunktion bei geringer Geschwindigkeit und mittleren Positionen, weil in solchen Zuständen erst noch Geschwindigkeit aufgebaut werden muss, um das Ziel zu erreichen.

Vergleich mit anderen Verfahren

Auch beim Mountain-Car-Problem gibt es verschiedene Varianten, was einen Vergleich oft erschwert oder unmöglich macht. Daher ist eine Beschränkung auf Arbeiten nötig, bei denen das Ziel und die Parameter der Simulation dieselben waren wie in unseren Experimenten und die Auswertung zumindest so ähnlich, dass sie einen Vergleich zulässt.

- In [Sutton und Barto, 1998; Sutton, 1996] wird ein SARSA-Algorithmus zusammen mit einem CMAC-Funktionsapproximator mit zehn 9×9 Gittern benutzt. Die durchschnittliche Anzahl der Schritte während der ersten 20 Episoden (gemittelt über 30 Läufe) beträgt dabei etwa 450 (bei der günstigsten Parameterwahl)[13]. Diese Verfahren sind also deutlich langsamer als die Q-LWIGNG-Verfahren.

- Bei der Decision-Tree-Reinforcement-Learning-Methode in [Pyeatt und Howe, 1998] werden ungefähr 1000 Episoden benötigt, um eine Politik zu lernen, die um die 100 Schritte benötigt (Durchschnitt über 10 Läufe). Die dort zum Vergleich herangezogenen Methoden mit tabellierter Q-Funktion und neuronaler Funktionsapproximation benötigen ähnlich lange. Wie in 5.10 zu sehen ist, erreichen die Kanten-Verfahren solche Werte schon nach weniger als 10 Episoden, wobei die besten dabei gefundenen Politiken nur um die 60 Schritte im Durchschnitt brauchen, um das Ziel zu erreichen.

- [Reynolds, 2000] benutzt eine adaptive Partitionierung des Zustandsraums in Zellen. Dabei wird eine Zelle geteilt, wenn in benachbarten Zellen unterschiedliche Aktionen favorisiert werden, die Q-Werte einigermaßen bekannt sind und sich die Q-Werte von Aktionen in benachbarten Zellen deutlich unterscheiden. Dieses Verfahren benötigt etwa 300 - 400 Episoden, um eine gute Politik zu lernen.

- In [Kretchmar und Anderson, 1997] werden CMAC, radiale Basisfunktionen und normalisierte radiale Basisfunktionen als Funktionsapproximator für Q-Learning eingesetzt. Dabei stellt sich heraus, dass bei der Verwendung von CMAC- und RBF-Approximatoren am schnellsten gelernt wird. Nach etwa 20 Episoden scheinen diese Verfahren bei geeigneter Parameterwahl Ergebnisse (durchschnittliche Anzahl von Schritten pro Episode über 10 Läufe) zu erreichen, die mit den unseren vergleichbar sind.

Auch beim Mountain-Car-Problem erreichen die hier vorgestellten Verfahren sowohl bei der Lerngeschwindigkeit als auch bei den gefundenen Politiken sehr gute Ergebnisse. Das „Kante 16“-Verfahren braucht im Durchschnitt der 20 Episoden lediglich 117.2 Schritte und während der letzten 5 Schritte durchschnittlich nur 59.1 Schritte pro Episode, um das Ziel zu erreichen. Diese Ergebnisse übertreffen die meisten in der Literatur zu findenden Resultate oder sind diesen zumindest ebenbürtig. Der Grund für den raschen Lernerfolg dürfte auch in der relativ einfachen Struktur des Problems liegen. Gute Politiken teilen den lediglich 2-dimensionalen Zustandsraum in klar umgrenzte Gebiete auf, in denen entweder nach rechts oder nach links beschleunigt wird. Das heißt, dass die

[13] Es ist nicht klar, ob und wann Episoden, die zu lange dauern, abgebrochen wurden. Solche Abbrüche können natürlich einen großen Einfluss auf den Durchschnittswert haben.

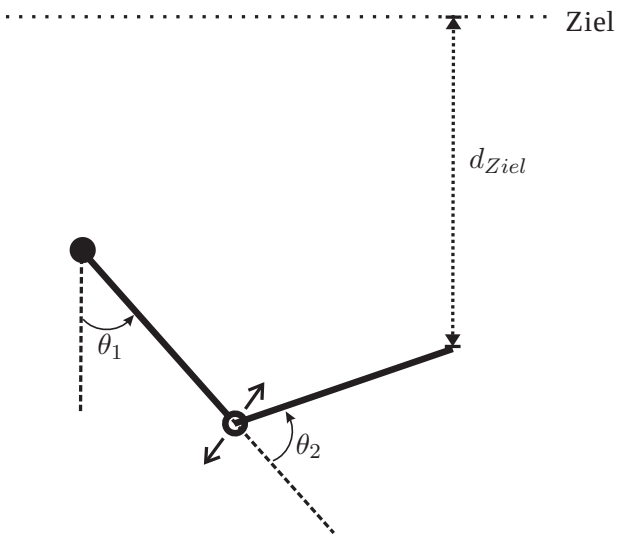


Abb. 5.12: Skizze des Acrobot-Problems.

zu einer Aktion gehörende Wertefunktion in einem großen, homogenen Gebiet einfach größer als die Wertefunktion, die zu der anderen Aktion gehört, sein muss. Die Kombination von lokal linearer Interpolation und Gewichtung der lokalen Modelle über die Kanten, erreicht das ziemlich schnell.

5.2.3 Das Acrobot-Problem

Der Acrobot bildet grob einen Turner nach, der sich mit den Händen an einer Reckstange festhält und versucht, sich mit Bewegungen aus dem Hüftgelenk aufzuschwingen. Er besteht aus zwei Stäben, die durch ein Gelenk an ihren Enden miteinander verbunden sind. Das eine Ende des ersten Stabes ist drehbar an einem festen Punkt befestigt und das andere Ende des zweiten Stabes soll eine gewisse Höhe erreichen, indem eine Kraft auf das gemeinsame Gelenk gegeben wird (siehe Abbildung 5.12). Dieses Problem ist deutlich schwieriger als die vorhergehenden. Wie auch beim Mountain-Car-Problem geht es darum, einen bestimmten Zustand zu erreichen, wobei die Strategie zum Erreichen dieses Zustandes wesentlich komplizierter als beim Mountain-Car-Problem ist. Zudem hat der Zustandsvektor nun 4 Dimensionen, wobei wohl eine große Anzahl von auftretenden Zuständen unterschieden werden muss.
Die Systemdynamik ist durch folgende Gleichungen gegeben [Sutton und Barto, 1998]:

$$
\begin{aligned}
\ddot{\theta}_1 &= -d_1^{-1}\left(d_2\ddot{\theta}_2 + \varphi_1\right) \qquad (5.8)\\
\ddot{\theta}_2 &= \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1}\left(\tau + \frac{d_2}{d_1}\varphi_1 - m_2 l_1 l_{c2}\dot{\theta}_1^2 \sin\theta_2 - \varphi_2\right)\\
d_1 &= m_1 l_{c1}^2 + m_2\left(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2}\cos\theta_2\right) + I_1 + I_2\\
d_2 &= m_2\left(l_{c2}^2 + l_1 l_{c2}\cos\theta_2\right) + I_2\\
\varphi_1 &= -m_2 l_1 l_{c2}\dot{\theta}_2^2\sin\theta_2 - 2 m_2 l_1 l_{c2}\dot{\theta}_2\dot{\theta}_1\sin\theta_2 + \left(m_1 l_{c1} + m_2 l_1\right) g\cos\left(\theta_1 - \pi/2\right) + \varphi_2\\
\varphi_2 &= m_2 l_{c2} g\cos\left(\theta_1 + \theta_2 - \pi/2\right)
\end{aligned}
$$

Die Simulation erfolgt in Zeitschritten von 0.05 Sekunden, wobei die Aktion $\tau \in \{-1, 0, 1\}$ alle 4 Zeitschritte geändert werden kann. Die Winkelgeschwindigkeiten $\dot{\theta}_1$ und

$\dot{\theta}_2$ werden dabei auf die Intervalle $[-4\pi, 4\pi]$ bzw. $[-9\pi, 9\pi]$ begrenzt. Tabelle 5.6 gibt die verwendeten Parameter wieder. Diese Werte wurden auch in [Sutton und Barto, 1998] verwendet.

Parameter	**Wert**	**Beschreibung**
m_1, m_2	1	Masse der Stäbe
l_1, l_2	1	Länge der Stäbe
l_{c1}, l_{c2}	0.5	Entfernung zum Masseschwerpunkt der Stäbe
I_1, I_2	1	Trägheitsmoment der Stäbe
g	9.8	Gravitation

Tabelle 5.6: Werte für die Simulation des Acrobots.

Das Ziel besteht darin, das Ende des zweiten Stabes mindestens eine Stablänge über den Aufhängepunkt des ersten Stabes zu schwingen, also:

$$d_{Ziel} = l_1 - \left(l_1 \sin\left(\theta_1 - \frac{\pi}{2}\right) + l_2 \sin\left(\theta_1 + \theta_2 - \frac{\pi}{2}\right)\right) \leq 0 \tag{5.9}$$

Die Strafe ist -1 in jedem Schritt bis dieses Ziel erreicht wird. Wird dieses Ziel nach 10000 Episoden nicht erreicht, wird der Lauf abgebrochen. Gestartet wird immer mit herunterhängendem Acrobot, d.h. im Zustand $\dot{\theta}_1 = \dot{\theta}_2 = \theta_1 = \theta_2 = 0$.
Die 4 Eingabedimensionen wurden wieder so normiert, dass sie ungefähr zwischen 0 und 1 liegen.

$$x_t = \begin{pmatrix} (\theta_{1,t} + \pi)_{[0,2\pi]} / \, 2\pi \\ (\theta_{2,t} + \pi)_{[0,2\pi]} / \, 2\pi \\ (\dot{\theta}_{1,t} + 4\pi) / \, 8\pi \\ (\dot{\theta}_{2,t} + 9\pi) / \, 18\pi \end{pmatrix} \tag{5.10}$$

Dabei wurde darauf geachtet, dass der Zustand, in dem die Stäbe herabhängen, einem Eingabewert von 0.5 für die Winkel entspricht, da ansonsten minimale Änderungen im Winkel zu extremen Änderungen in deren Kodierung führen können (ein Winkel von -0.1 würde als 0.984 kodiert, während 0.1 als 0.016 kodiert werden würde). Deswegen wird zu den Winkeln π addiert. Falls der resultierende Winkel dann größer als 2π ist, wird 2π abgezogen, um im Intervall $[0, 2\pi]$ zu bleiben[14].
Beim Acrobot-Problem wurden für die verschiedenen Methoden jeweils nur 10 Läufe mit 500 Episoden durchgeführt, da das Problem doch etwas aufwändiger als die anderen ist. So mussten diesmal auch zwischen 81 und 625 Neuronen eingesetzt werden, um einigermaßen befriedigende Ergebnisse zu erreichen. Bei maximal 81 Neuronen wurde mit 16, bei 256 mit 81 und bei 625 mit 256 Startneuronen begonnen. Der Lernparameter α_v wurde genauso wie in [Sutton und Barto, 1998] auf 0.2 und der Diskontierungsfaktor γ auf 1 gesetzt. Der Explorationsparameter ε für die Aktionsauswahl wurde auf 0 gesetzt, da schon eine einzelne falsche Aktion die Lösung des Problems erheblich verlängern kann.
Zur Auswertung wurden beim Acrobot-Problem die prozentualen Anteile der abgebrochenen Episoden bestimmt. Dabei wurde dieser Anteil einmal für alle 500 Episoden insgesamt, aber auch für die letzten 100 Episoden alleine betrachtet, da Abbrüche aufgrund der 10000-Schritte-Grenze gerade zu Beginn doch recht häufig sind. Außerdem wurde auch der Durchschnitt der Schritte pro Episode für alle 500 Episoden und für die letzten

[14] Eine andere Möglichkeit besteht darin, bei der Entfernungsberechnung im Wachsenden Neuralen Gas die Tatsache zu berücksichtigen, dass es sich um Winkel handelt. Das hätte jedoch eine Änderung im GNG-Algorithmus notwendig gemacht.

100 Episoden alleine berechnet. Schließlich wurde noch die Länge der kürzesten gefundenen Lösung bestimmt. Tabelle 5.7 zeigt die Ergebnisse. Dort finden sich jedoch keine Ergebnisse für „NN 256“, „NN err. 256“, „Lambda 256“ und „Lambda err. 256“. In den durchgeführten Experimenten waren diese Verfahren nämlich in der überwiegenden Anzahl der Episoden überhaupt nicht in der Lage, den Stab innerhalb der erlaubten 10000 Schritte aufzuschwingen.

Methode	**abgebrochene Episoden gesamt**	**abgebrochene Episoden letzte 100**	**Schritte / Episode gesamt**	**Schritte / Episode letzte 100**	**Schritte kürzeste Lösung**
Kante 81	3.38 %	0.20 %	736.81	287.46	76
Kante err. 81	1.58 %	0.50 %	450.06	293.80	75
Kante fix 81	11.12 %	2.76 %	2222.17	3519.22	82
NN const 256	0.38 %	0.00 %	644.98	317.02	111
NN fix 256	0.00 %	0.00 %	162.50	151.81	70
Kante const 256	2.26 %	0.20 %	795.70	215.16	82
Kante 256	1.18 %	0.00 %	329.08	131.67	76
Kante err. 256	1.02 %	0.50 %	366.14	279.63	75
Kante fix 256	0.06 %	0.00 %	167.48	143.51	76
Lambda const 256	1.98 %	1.00 %	1272.38	870.12	88
Lambda fix 256	0.32 %	0.40 %	381.69	411.19	83
Kante 625	0.30 %	0.20 %	338.84	262.71	71
Kante err. 625	0.12 %	0.00 %	283.29	234.78	77
Kante fix 625	0.62 %	0.10 %	322.34	201.26	71

Tabelle 5.7: Ergebnisse für das Acrobot-Problem. Der prozentuale Anteil von abgebrochenen Episoden wird einmal für alle 500 Episoden und zusätzlich für die letzten 100 Episoden alleine angegeben. Auch bei der durchschnittlichen Anzahl von Schritten pro Episode wurden die Durchschnittswerte einmal über alle 500 Episoden und zusätzlich für die letzten 100 Episoden alleine berechnet, um den Lernerfolg besser beurteilen zu können. Außerdem wurde das Minimum der kürzesten Lösung aus allen Läufen bestimmt.

Nur „NN fix 256“ gelingt es, den Stab in jeder Episode aufzuschwingen. Aber bei einer Reihe von Verfahren kommt es immerhin während der letzten 100 Episoden zu keinen oder nur sehr wenigen Abbrüchen. Bei den NN- und den Lambda-Verfahren gelingt es nur den Verfahren mit konstanter Approximation und den Verfahren ohne Bewegung der Neuronen das Aufschwingen des Acrobots zu lernen. Wieder sind die Kanten-Verfahren diejenigen, die am schnellsten lernen (vgl. Abbildung 5.13). Nur „NN fix 256“ erreicht ähnlich gute Werte. Die gleichmäßige Verteilung der 256 Neuronen stellt offensichtlich eine gute Grundlage dar, um die benötigte Aktions-Wertefunktion zu approximieren. Auffällig ist, dass alle Verfahren ohne Bewegung der Neuronen bei dem Gesamtdurchschnitt fast dieselben Werte wie beim Durchschnitt über die letzten 100 Episoden alleine haben. Diese Verfahren scheinen also ziemlich schnell zu lernen. Diese Vermutung wird durch Abbildung 5.14 bestätigt. Dort ist die Entwicklung der durchschnittlichen Schritteanzahl pro Episode für die drei besten Verfahren zu sehen. Die Verfahren ohne Bewegung der Neuronen erreichen dabei schneller als „Kante 256“ niedrige Werte. „Kante 256 fix“ scheint zudem auch äußerst stabil zu lernen. Bei „NN fix 256“ und besonders bei „Kante 256“ sind deutlich mehr „Ausreißer“ in den Kurven zu sehen. Diese kommen daher, dass es zumindest in einer Episode nicht gelang, den Stab erfolgreich aufzusch-

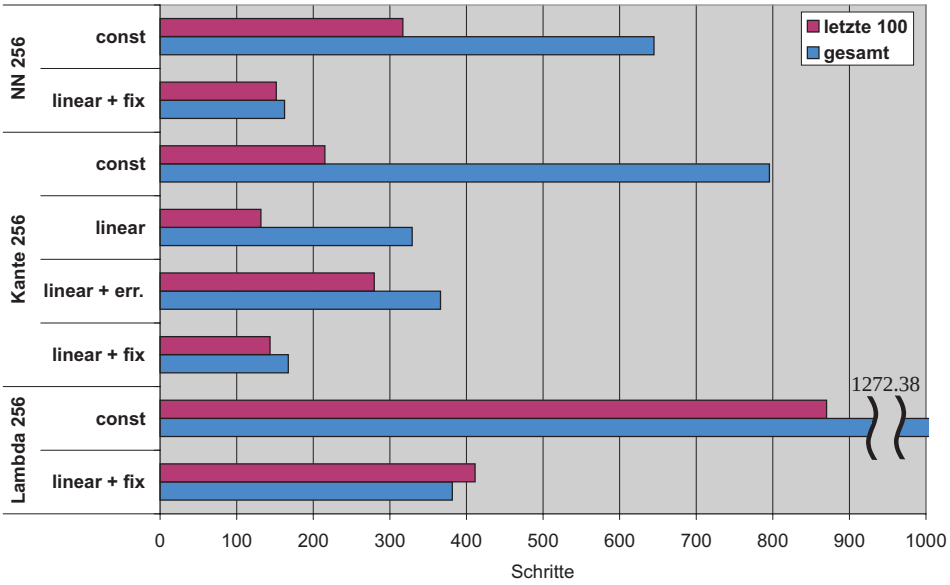


Abb. 5.13: Durchschnittliche Anzahl von Schritten pro Episode für verschiedene Verfahren bei einer maximalen Anzahl von 256 Neuronen (Acrobot-Problem).

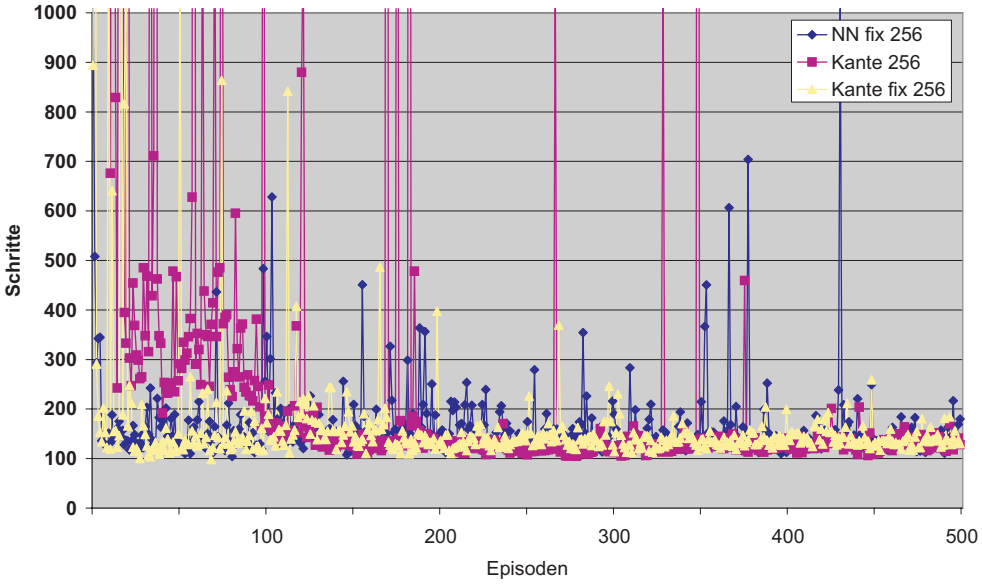


Abb. 5.14: Durchschnittliche Anzahl von Schritten (10 Läufe) bis zum Aufschwingen des Acrobots pro Episode für die drei besten Verfahren mit einer maximalen Anzahl von 256 Neuronen (Acrobot-Problem).

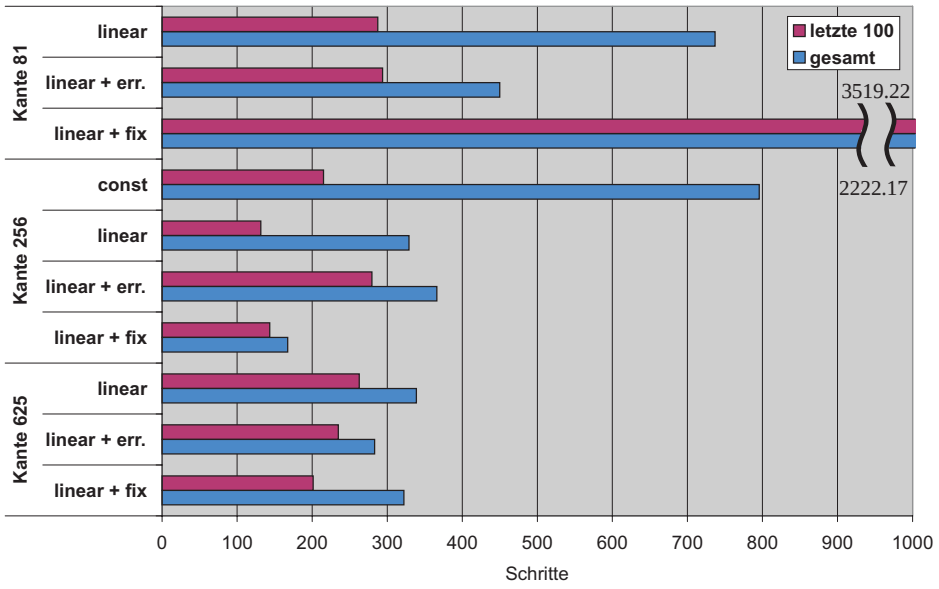


Abb. 5.15: Durchschnittliche Anzahl von Schritten pro Episode bei Kanten-Verfahren für unterschiedliche Neuronenanzahlen und unterschiedliches Vorgehen bei der Bewegung der Neuronen (Acrobot-Problem).

wingen. Die gefundenen Lösungen sind also nicht besonders stabil. Wenn es gelingt den Stab einmal schnell aufzuschwingen, heißt das nicht, dass das in der nächsten Episode wieder so gut gelingt. Das liegt auch daran, dass zwischen zwei Episoden die Neuronen bewegt werden und sich die Wertefunktion auch dadurch ändert. Auf der anderen Seite dürfte dies aber auch zu einer stärkeren Exploration führen, was das Finden von guten Lösungen vereinfacht.

In Abbildung 5.15 werden Kanten-Verfahren mit unterschiedlicher Maximalanzahl von Neuronen und unterschiedlicher Art der Bewegung der Neuronen gegenüber gestellt. Die besten Werte werden bei 256 Neuronen erzielt. Nur bei fehlerabhängiger Bewegung ist das Ergebnis mit 625 Neuronen etwas besser als bei 256 Neuronen. Während bei 256 Neuronen ohne Bewegung fast dasselbe Ergebnis für die letzten 100 Episoden wie mit häufigkeitabhängiger Bewegung erzielt wird und bei 625 Neuronen sogar ein besseres, sind bei nur 81 Neuronen die Verfahren mit Bewegung deutlich überlegen. Es gelingt dabei die Neuronen so zu verteilen, dass eine bessere Approximation als mit den festen Positionen möglich ist.

Bei Problemen, die wie das Acrobot-Problem einen festen Startzustand besitzen, ist die Stabilität der gefundenen Lösung oft nicht so entscheidend. Es genügt ja, einmal eine möglichst gute Politik zu finden, die dann gespeichert werden kann. Wie gut oder schlecht die darauf folgenden Episoden gelingen, interessiert dann nicht. Daher werden auch die kürzesten Lösungen, die durch die einzelnen Verfahren gefunden werden, miteinander zu vergleichen (Tabelle 5.7, letzte Spalte). Den besten Wert erreicht dabei „NN fix 256“, aber die Kanten-Verfahren sind auch nicht viel schlechter. Insbesondere sind mit einer höheren Neuronenanzahl etwas bessere Lösungen möglich als mit weniger Neuronen, auch wenn dafür die Durchschnittswerte für die Anzahl von Schritten pro Episode schlechter werden. Viele Neuronen erlauben eine bessere Spezialisierung und damit eine feinere „Auflösung“ bei der Politik, während wenige Neuronen mehr generalisieren und daher etwas schneller lernen und bessere Durchschnittswerte erreichen.

Vergleich mit anderen Verfahren

Auch beim Acrobot-Problem sollen Vergleiche mit Ergebnissen aus der Literatur angestellt werden. Vor allem bei den Zielen und der Belohnungsfunktion unterscheiden sich die dort durchgeführten Experimente. So wird in einigen Arbeiten angestrebt, den Acrobot nicht nur aufzuschwingen, sondern auch an der höchsten Stelle zu balancieren. In die Belohnungsfunktion geht dabei meist die Höhe des Stabendes mit ein. Allerdings gelingt es bis jetzt keinem Verfahren, dieses Ziel 100-prozentig zu erreichen. Die Güte der Lösung wird dabei eher durch Anschauung als durch genau quantifizierte Werte bestimmt. Für die einfachere Variante, bei der es nur darum geht, eine bestimmte Höhe zu erreichen, finden sich jedoch eher vergleichbare Ergebnisse. Daher wurde in dieser Arbeit auch nur die einfachere Variante betrachtet.

- In [Sutton und Barto, 1998; Sutton, 1996] wird ein SARSA(λ)-Verfahren zusammen mit einem CMAC-Funktionsapproximator verwendet. Die Zerlegung des Zustandsraums gestaltet sich dabei relativ kompliziert und basiert auf insgesamt 48 tilings entlang verschiedener Dimensionen. Diese Zerlegung führt zu etwa 25000 unterscheidbaren Zuständen für jede Aktion. Nach ungefähr 400 - 500 Episoden führt dieses Verfahren zu einer stabilen Politik, die etwa 75 Schritte benötigt (Durchschnitt aus 10 Läufen). Dies ist zumindest was die Durchschnittswerte betrifft besser als die besten hier erzielten Resultate. Dafür sind die hier verwendeten Kanten-Verfahren mit festem Netz aber etwas schneller, wie an einer ähnlichen Abbildung wie 5.14 in [Sutton und Barto, 1998] zu sehen ist. Zudem ist bei den hier benutzen Verfahren keine besondere Aufmerksamkeit bei der Zerlegung des Zustandsraums nötig. Bezüglich der gefundenen kürzesten Lösungen dürften die Verfahren in etwa vergleichbar sein.

- In [Coulom, 2002, 2004] wird versucht, das Acrobot-Problem mit einem modellbasierten, kontinuierlichen TD(λ)-Verfahren zu lösen. Als Funktionsapproximator wird ein Neuronales Netz verwendet. Es wird eine maximale Kraft von $\pm 2\mathrm{N}$ benutzt, was das Problem etwas vereinfacht. Das Ziel ist, den Acrobot in die aufrechte Position zu bringen und zu balancieren. Die Belohnungsfunktion gibt die Höhe des Stabendes zurück. Trainiert wurden 1000000 Episoden. Das Ziel wird nicht ganz erreicht, der Acrobot kommt zwar in die aufrechte Position mit geringer Geschwindigkeit, kann aber nicht balanciert werden.

- In [Munos und Moore, 1999] wird ebenfalls das Ziel verfolgt, den Acrobot zu balancieren. Dabei werden Techniken benutzt, die nach verschiedenen Kriterien den Zustandsraum mit einem so genannten kd-trie (im Prinzip ein binärer Entscheidungsbaum) immer weiter unterteilen und die Ausgabewerte mit Hilfe der Werte in den Blättern interpolieren. Um das Problem gut zu lösen, müssen bei den besten Verfahren ungefähr 5000 Zustände unterschieden werden[15].

- In [Davies, 1997] werden verschiedene Interpolationstechniken auf gitterartigen Strukturen zur Funktionsapproximation benutzt. Es werden Experimente mit modellbasierter Werte-Iteration und Q-Learning durchgeführt. Die besten gelernten Politiken benötigen 1136 (Werte-Iteration) bzw. 1529 Schritte (Q-Learning) im Durchschnitt. Um diese Ergebnisse zu erreichen, werden jeweils Hunderttausende von Werte-Backups benötigt. Dabei wurden erfolglose Episoden jedoch nicht

[15] Leider wird ein nicht näher beschriebenes Maß für die Güte der Lösung verwendet, so dass keine weitergehenden Vergleiche möglich sind. Zudem sind die Parameter der Simulation unklar.

nach 10000 Schritten abgebrochen. Trotzdem kann man schließen, dass die dort verwendeten Verfahren deutlich langsamer als die in der vorliegenden Arbeit entwickelten Verfahren beim Aufschwingen des Acrobot sind. So benötigt „NN fix 256“ nur etwa 152 Schritten im Durchschnitt und würde denselben Wert auch ohne 10000-Schritte-Grenze erreichen, da es bei diesem Verfahren zu überhaupt keinen Abbrüchen kommt.

- In [Boone, 1997] werden verschiedene Verfahren hinsichtlich der minimal benötigten Schritteanzahl verglichen. Bei Verfahren, die ein perfektes Modell besitzen, erreicht ein heuristisches Suchverfahren eine minimale Schritteanzahl von 63. Bei Verfahren, die nur ein Modell lernen und dann Suchalgorithmen verwenden, erreicht der A*-Algorithmus [Hart u. a., 1968] zusammen mit einem linearen Modell eine kürzeste Länge von 71. Bei den modellfreien Verfahren werden bei einer SARSA-CMAC-Kombination minimal 76 Schritte und bei Q-Learning auf einer Tabelle 116 Schritte minimal benötigt. Die beste von unseren Verfahren gefundene Lösung benötigt 70 Schritte. Das ist zwar schlechter als die mit einem perfekten Modell gefundene Lösung, aber besser als die Lösungen der modellfreien Verfahren.

Die in den hier vorgestellten Experimenten erreichten Ergebnisse sind hinsichtlich der Durchschnitte schlechter als die in [Sutton und Barto, 1998; Sutton, 1996] erreichten. Dennoch finden sie gute Einzellösungen und sind relativ schnell. Dies könnte daran liegen, dass die verwendeten Q-LWIGNG-Verfahren in Bezug auf das Acrobot-Problem zwar nicht sonderlich stabil sind, dafür aber gut explorieren und dabei gute Lösungen finden.

5.2.4 Zusammenfassung der Ergebnisse

Um die zu Beginn des Abschnitts 5.2 aufgeworfenen Fragen beantworten zu können, werden in Tabelle 5.8 die Ergebnisse der vier Experimente überblickartig zusammengefasst. Die Ergebnisse der Lernverfahren werden dabei nach einem groben Raster in sieben verschiedene Klassen eingeteilt (- - -, - -, -, o, +, ++, +++). Es wurde darauf geachtet, dass ähnliche Ergebnisse derselben Klasse zugeordnet werden. Bei der Einteilung wurden jeweils die Ergebnisse für die Neuronenanzahlen betrachtet, bei denen die besten Ergebnisse erreicht wurden. Beim Stabbalance-Problem wurde dabei die durchschnittliche Anzahl von Episoden bis der Stab 100000 Schritte balanciert wurde zugrunde gelegt. Das Kriterium beim Mountain-Car-Problem war die durchschnittliche Dauer bis das Ziel während der letzten 5 Episoden erreicht wurde. Beim Acrobot wurde die durchschnittliche Dauer pro Episode während der letzten 100 Episoden berücksichtigt.
Betrachtet man Tabelle 5.8, so fällt auf, dass immer Kanten-Verfahren das beste Lernverhalten aufweisen. Lambda-Verfahren schneiden meist recht schlecht ab, was wohl daran liegt, dass sich die zu approximierende Aktions-Wertefunktion zu rasch ändert, um die Parameter $\lambda_{i,j}$ an den Neuronen und Kanten geeignet einzustellen. Bei den NN-Verfahren hängt der Erfolg vom betrachteten Problem ab. So werden beim Mountain-Car-Problem gute Ergebnisse erreicht, was aber auch daran liegen dürfte, dass gute Politiken dabei recht einfach strukturiert sind. Insgesamt scheint es aber am besten zu sein, die Gewichte über die durchschnittliche Kantenlänge zu bestimmen. Ein Vorteil der Kanten-Verfahren liegt darin, dass die Entfernungen der Neuronen zueinander und zum Eingabevektor mit in die Approximation eingehen, ohne dass zusätzliche Parameter erforderlich sind.

Methode	**Stabbalance (fest)**	**Stabbalance (zufällig)**	**Mountain-Car**	**Acrobot**
NN const	0	++	- - -	0
NN	-	0	++	- - -
NN err.	- - -	-	++	- - -
NN fix	- - -	- -	++	++
Kante const	- -	+	+	+
Kante	++	+++	+++	+++
Kante err.	+++	+++	+++	+
Kante fix	- - -	- - -	+++	++
Lambda const	+	++	- - -	–
Lambda	- -	+	- - -	- - -
Lambda err.	-	0	- - -	- - -
Lambda fix	- -	- -	- - -	-

Tabelle 5.8: Zusammenfassung der erzielen Ergebnisse aus allen mit Q-LWIGNG durchgeführten Experimenten.

Die Bewegung der Neuronen wirkt sich bei Kanten-Verfahren positiv auf das Lernverhalten aus. Dabei kann aber nicht endgültig geklärt werden, ob häufigkeits- oder fehlerabhängige Bewegung besser ist. Ohne Bewegung lernen NN- und Kanten-Verfahren entweder sehr gut (Mountain-Car, Acrobot) oder aber sehr schlecht (Stabbalance). Es kommt sehr darauf an, dass die zu Beginn festgelegten Neuronenpositionen eine gute Approximation für das jeweilige Problem ermöglichen. Um gute Ergebnisse zu erreichen, benötigen Verfahren mit festem Netz aber meist eine größere Anzahl von Neuronen als Kanten-Verfahren mit Bewegung.
Bei Experimenten mit verschiedenen Maximalanzahlen von Neuronen stellte sich aber auch heraus, dass eine Erhöhung der Neuronenanzahl nicht unbedingt mit einer Verbesserung des Lernverhaltens einhergeht. Sind viele Neuronen vorhanden, wird weniger stark generalisiert. Allerdings darf die Anzahl der Neuronen auch nicht zu gering sein, weil dann überhaupt nicht gelernt werden kann.
Die Verwendung von lokal linearer Interpolation ist bei den Kanten-Verfahren durchweg besser als die konstante Approximation. Bei NN- oder Lambda-Verfahren ist dies jedoch eher umgekehrt. Hier werden mit der konstanten Approximation oft bessere Ergebnisse als bei der lokal linearen Interpolation erreicht. Dennoch bleiben diese hinter den Ergebnissen zurück, die mit Kanten-Verfahren möglich sind.
Zusammenfassend lässt sich feststellen, dass die Kombination von Q-Learning und Kanten-Verfahren mit beweglichen Neuronen und lokal linearer Interpolation ein äußerst gutes Lernverhalten aufweist. Beim Stabbalance- und beim Mountain-Car-Problem erreichen diese Verfahren Ergebnisse, die mit den besten in der Literatur beschriebenen vergleichbar sind oder diese gar noch übertreffen. Beim Acrobot werden zwar bei den Durchschnittswerten nicht ganz die Ergebnisse erreicht, wie sie teilweise in der Literatur zu finden sind, aber dennoch gute Einzellösungen gefunden.

Kapitel 6

Lernen von kontinuierlichen Aktionen: ReinforceLWIGNG

Während im letzten Kapitel LWIGNG in Verbindung mit Q-Learning erfolgreich zum Lernen diskreter Aktionen eingesetzt wurde, geht es in diesem Kapitel darum kontinuierliche Aktionen zu lernen. Dazu wird ein REINFORCE-Algorithmus in einer Actor-Critic-Architektur eingesetzt und LWIGNG wiederum als Funktionsapproximator benutzt. Das entwickelte Verfahren wird als *ReinforceLWIGNG* bezeichnet. Nach der Beschreibung von ReinforceLWIGNG wird das Problem betrachtet, Aktionen für einen simulierten 2-rädrigen Roboter zu lernen. Der Roboter hat dabei die Aufgabe, einen rollenden Ball so anzufahren, dass er in Richtung eines vorgegebenen Ziels geschossen werden kann. Dazu muss sich der Ball in einem geringen Abstand vor dem Roboter befindet. Da der Ball immer in die Richtung des Vektors vom Roboter zum Ball geschossen wird, muss zusätzlich der Winkel zwischen diesem Vektor und dem Vektor vom Roboter zum Ziel möglichst klein sein, um das Ziel zu treffen. Der Roboter wird dabei durch das Setzen der Geschwindigkeiten der beiden Räder gesteuert. Es wird gelernt, welche Differenzen in den Geschwindigkeiten dabei angewendet werden sollen: Eines der Räder wird auf maximale Geschwindigkeit, das andere aufgrund der gelernten Geschwindigkeitsdifferenz gesetzt. Es werden wieder Experimente mit verschiedenen Varianten des Verfahrens gemacht, um zu sehen, wie sich die einzelnen Aspekte von LWIGNG auf das Lernen auswirken. Zusätzlich wird untersucht, was geschieht, wenn die Wahrnehmung von Zuständen oder die Ausführung von Aktionen durch Rauschen gestört wird.

6.1 ReinforceLWIGNG

Da REINFORCE-Algorithmen direkte Belohnungen benötigen, muss noch das Problem gelöst werden, wie aus zeitverzögerten Belohnungen direkte Belohnungen generiert werden können. Dies geschieht innerhalb der im nächsten Abschnitt beschriebenen Actor-Critic-Architektur. Anschließend muss geklärt werden, wie die Politik des Actor und die

Wertefunktion des Critic approximiert werden können, da sowohl von kontinuierlichen Aktionen als auch von kontinuierlichen Zuständen ausgegangen wird. Dies geschieht wieder mit Hilfe des im Kapitel 4 vorgestellten LWIGNG-Verfahrens, wobei auch die Erweiterungen zum Lernen aus Trajektorien aus Abschnitt 5.1.1 berücksichtigt werden müssen.

6.1.1 REINFORCE-Algorithmen für zeitverzögerte Belohnungen

Grundlage von ReinforceLWIGNG ist der in Abschnitt 2.5.1 hergeleitete spezielle REINFORCE-Algorithmus. Dabei wird die auszuführende skalare Aktion a_t mit Hilfe einer von der aktuellen Situation x_t abhängigen Normalverteilung bestimmt:

$$a_t \sim N\left(\mu(x_t), \sigma(x_t)\right) \tag{6.1}$$

Falls es bestimmte Minimal- und Maximalwerte a_{min} und a_{max} für Aktionen gibt, muss sichergestellt werden, dass a_t innerhalb dieser Grenzen liegt. Deswegen wird a_t auf a_{min} bzw. a_{max} gesetzt, falls a_t außerhalb der Grenzen liegt. Durch die Funktionen $\mu(x)$ und $\sigma(x)$ wird die vom Actor verfolgte Politik π vollständig beschrieben.
Das Problem der REINFORCE-Algorithmen besteht darin, dass diese eine direkte Belohnung zur Anpassung der Parameter der Politik brauchen, wie in den Anpassungsformeln (2.39) und (2.40) zu sehen ist. Würde man zum Zeitpunkt dieser Anpassungen schon den Return kennen, also die Summe aller zukünftigen Belohnungen, könnte man einfach diesen als direkte Belohnung verwenden. Denn dann würden die Aktionen bevorzugt, die zu einem hohen Return führen. Da der Return aber erst am Ende einer Episode feststeht, soll stattdessen eine Schätzung aus direkter Belohung und dem zu erwartenden Return verwendet werden. Zustands-Wertefunktionen stellen gerade solche Schätzungen für den zu erwartenden Return dar. Daher kann

$$\widetilde{R_t} = r_{t+1} + \gamma \widetilde{V}(x_{t+1}) \tag{6.2}$$

in den Anpassungsformeln (2.39) und (2.40) anstelle der direkten Belohnung verwendet werden. Die Aufgabe des Critic besteht also darin, eine Zustands-Wertefunktion zu lernen, um daraus die Belohnungen für den Actor zu generieren. REINFORCE-Algorithmen erfordern auch die Wahl einer „reinforcement baseline“. Dafür sollte eine Schätzung für die durchschnittlich in einem Zustand zu erwartende Belohnung verwendet werden. Auch hier kann die Zustands-Wertefunktion weiter helfen. Da der geschätzte Return als direkte Belohnung verwendet werden soll, stellt der aktuelle Wert eines Zustandes $\widetilde{V}(x_t)$ gerade eine solche Schätzung aufgrund der bisher gemachten Erfahrungen dar und kann daher für $\overline{r}$ verwendet werden. Damit ergeben sich dann die folgenden Anpassungsformeln für die Parameter der Normalverteilung[1]:

$$\Delta\mu(x_t) = \alpha(r_{t+1} + \gamma\widetilde{V}(x_{t+1}) - \widetilde{V}(x_t))(a_t - \mu(x_t)) \tag{6.3}$$

$$\Delta\sigma(x_t) = \alpha(r_{t+1} + \gamma\widetilde{V}(x_{t+1}) - \widetilde{V}(x_t))\frac{(a_t - \mu(x_t))^2 - \sigma(x_t)^2}{\sigma(x_t)} \tag{6.4}$$

Der Ausdruck $r_{t+1} + \gamma\widetilde{V}(x_{t+1}) - \widetilde{V}(x_t)$ ist dabei gerade der Temporal-Difference-Fehler δ_t. Dabei ist $\delta_t > 0$, falls die gerade ausgeführte Aktion a_t besser als erwartet ist. Dementsprechend wird dann $\mu(x_t)$ in Richtung der aktuellen Aktion a_t verschoben. Ist a_t außerdem mehr als eine Standardabweichung $\sigma(x_t)$ von $\mu(x_t)$ entfernt, so wird $\sigma(x_t)$ erhöht,

[1] Im Prinzip kann jeder REINFORCE-Algorithmus auf diese Weise erweitert werden, um mit zeitverzögerten Belohnungen umzugehen.

um das Auftreten von a_t wahrscheinlicher zu machen. Ist a_t innerhalb einer Standardabweichung von $\mu(x_t)$, so wird $\sigma(x_t)$ verkleinert, was in diesem Fall das Auftreten von a_t ebenfalls wahrscheinlicher macht. Ist $\delta_t < 0$ so führen dieselben Mechanismen dazu, dass das Auftreten von a_t unwahrscheinlicher wird.

6.1.2 Approximation von Politiken und Zustands-Wertefunktionen

Nun muss noch geklärt werden, wie die Politik und die Zustands-Wertefunktion approximiert und gelernt werden können. Dafür soll wieder LWIGNG zum Einsatz kommen. Da auch diesmal die Eingabedaten von Trajektorien durch den Zustandsraum stammen, wird der Algorithmus aus Abbildung 5.1 verwendet, der diese Tatsache berücksichtigt.
Insgesamt müssen diesmal drei Funktionen approximiert werden: die Funktionen für die Parameter $\mu(x)$ und $\sigma(x)$ der Normalverteilung und die Wertefunktion $V(x)$. Für gewöhnlich werden in Actor-Critic-Architekturen Wertefunktion und Politik getrennt voneinander approximiert. Auch $\mu(x)$ und $\sigma(x)$ könnten jeweils mit eigenen Funktionsapproximatoren gelernt werden. Im Folgenden wird jedoch nur ein einziges LWIGNG benutzt, um alle drei Funktionen zu approximieren. Da nämlich die Eingabevektoren x bei allen drei Funktionen immer die gleichen sind, würde sich bei LWIGNG das zugrunde liegende Wachsende Neurale Gas zumindest bei häufigkeitsabhängiger Bewegung ohnehin immer gleich verhalten[2]. Bei fehlerabhängiger Bewegung wird der Fehler bei den Werten v_i der Wertefunktion benutzt.
Daher speichert jedes Neuron diesmal einen dreidimensionalen Vektor $(v_i, \mu_i, \sigma_i)^T$. Die Berechnung der Approximationen $\widetilde{V}(x_t)$, $\widetilde{\mu}(x_t)$ und $\widetilde{\sigma}(x_t)$ für einen Zustand x_t geschieht analog zu Gleichung (4.10) als gewichtete Summe, wobei bei der Berechnung der lokalen Modelle die jeweiligen v_i, μ_i, bzw. σ_i verwendet werden. Trainiert wird das LWIGNG analog zu den Anpassungsformeln (4.13) mit Trainingsbeispielen (x_t, y_t), wobei diesmal $y_t = (y_t^V, y_t^\mu, y_t^\sigma)^T$ ein dreidimensionaler Vektor ist, der die Ziele der Anpassung für $V(x_t)$, $\mu(x_t)$ und $\sigma(x_t)$ angibt. Für y_t^V wird dabei gemäß der gewöhnlichen Temporal-Difference-Methode (siehe Abschnitt 2.2)

$$y_t^V = r_{t+1} + \gamma \widetilde{V}(x_{t+1}) \tag{6.5}$$

gewählt. Die Ziele y_t^μ und y_t^σ ergeben sich aus den Anpassungsformeln (6.3) und (6.4)[3]:

$$y_t^\mu = \widetilde{\mu}(x_t) + \left(r_{t+1} + \gamma \widetilde{V}(x_{t+1}) - \widetilde{V}(x_t)\right)(a_t - \widetilde{\mu}(x_t)) \tag{6.6}$$

$$y_t^\sigma = \widetilde{\sigma}(x_t) + \left(r_{t+1} + \gamma \widetilde{V}(x_{t+1}) - \widetilde{V}(x_t)\right) \frac{(a_t - \widetilde{\mu}(x_t))^2 - \widetilde{\sigma}(x_t)^2}{\widetilde{\sigma}(x_t)} \tag{6.7}$$

Da in diesen Ausdrücken die zukünftige Belohnung r_{t+1} und der Nachfolgezustand x_{t+1} vorkommen, können die Anpassungen für einen Zustand x_t jeweils erst zum Zeitpunkt $t+1$ stattfinden. Falls es minimale und maximale Werte a_{min} und a_{max} für Aktionen gibt, sollte dafür gesorgt werden, dass die resultierenden μ_i innerhalb dieser Grenzen bleiben, indem sie gegebenenfalls einfach auf a_{min} bzw. a_{max} gesetzt werden. Um ein geeignetes Explorationsverhalten sicherzustellen, sollten auch für σ_i Grenzen σ_{min} und σ_{max} gewählt werden, innerhalb derer sich die σ_i dann bewegen können.
In Abbildung 6.1 ist der ReinforceLWIGNG-Algorithmus überblickartig dargestellt. Soll nicht gelernt, sondern nur eine Aktion a_t bestimmt werden, so reicht es aus $\mu(x_t)$ zu berechnen und als Aktion zu benutzen.

[2] Bei gleicher Initialisierung.

[3] Da es sich um die Ziele der Anpassung handelt, kommen in diesen Ausdrücken keine Lernraten vor. Die Vektoren in den Neuronen werden jedoch wie üblich mit einer bestimmten Lernrate diesen Zielen angepasst.

1. Erhalte Zustand x_{t+1} und Belohnung r_{t+1}
2. Berechne $\widetilde{V}(x_{t+1})$, $\widetilde{\mu}(x_{t+1})$ und $\widetilde{\sigma}(x_{t+1})$ als gewichtete Summen nach (4.10)
3. Bestimme den Trainingsvektor $y_t = (y_t^V, y_t^\mu, y_t^\sigma)^T$ gemäß den Gleichungen (6.5), (6.6) und (6.7)
4. Trainiere das erweiterte GNG mit (x_t, y_t) analog zum erweiterten GNG-Algorithmus für Daten aus Trajektorien aus Abbildung 5.1
5. Bestimme Aktion $a_{t+1} \sim N\left(\widetilde{\mu}(x_{t+1}), \widetilde{\sigma}(x_{t+1})\right)$
   falls $a_{t+1} < a_{min} \Rightarrow a_{t+1} = a_{min}$; falls $a_{t+1} > a_{max} \Rightarrow a_{t+1} = a_{max}$

Abb. 6.1: Der ReinforceLWIGNG-Algorithmus.

6.2 Experimente

ReinforceLWIGNG soll eingesetzt werden, um das Abfangen eines Balls mit einem simulierten 2-rädrigen Roboter zu lernen. Der Ball soll dabei so abgefangen werden, dass er direkt zu einem vorgegebenen Ziel geschossen werden kann. Ursprünglich sollte als Simulationsumgebung die 3-dimensionale Fußballsimulation „rcssserver3d“ [Obst und Rollmann, 2004; Kögler und Obst, 2004] benutzt werden, die auch in der 3D-Simulationsliga der jährlichen RoboCup-Wettbewerbe verwendet wird. Leider waren zum Zeitpunkt der hier durchgeführten Experimente noch keine simulierten 2-rädrigen Roboter für diese Umgebung verfügbar[4]. Daher musste eine eigene Simulationsumgebung geschaffen werden[5]. Dabei wurde aber darauf geachtet, die Parameter der Simulation möglichst ähnlich zu denen der 3D-Simulationsliga des RoboCup zu wählen, um einen späteren Einsatz des verwendeten Verfahrens in der 3D-Simulationsliga zu vereinfachen.

6.2.1 Beschreibung des Ball-Abfang-Problems

Das Ziel beim Ball-Abfang-Problem besteht darin, einen 2-rädrigen Roboter so zu einem Ball zu positionieren, dass dieser in der Lage ist, den Ball in Richtung eines vorgegebenen Ziels zu schießen. Die Situation ist in Abbildung 6.2 dargestellt. Um den Ball schießen zu können, darf der Abstand zwischen Roboter und Ball höchstens $0.07\ \mathrm{m}$ betragen. Der Winkel zwischen der Orientierung des Roboters und dem Vektor zum Ball α_t^B darf dabei nicht größer als $22.5°$ sein. Der Roboter kann also den Ball nur dann schießen, wenn dieser vor ihm liegt. Der Ball wird immer in die Richtung des Vektors vom Zentrum des Roboters zum Zentrum des Balls geschossen. Daher darf der Winkel zwischen dieser Richtung und der Richtung zum Ziel α_t^Z nicht größer als $11.25°$ werden, um das Ziel einigermaßen zu treffen. Falls alle diese Bedingungen erfüllt sind, hat der Roboter sein Ziel erreicht und die Episode wird als erfolgreich betrachtet. Gelingt es dem Roboter jedoch nicht, den Ball innerhalb von 25 Sekunden simulierter Zeit unter diesen Bedingungen zu erreichen, so wird die Episode abgebrochen.
Es sollte erwähnt werden, dass keineswegs klar ist, wie eine optimale Lösung für dieses Problem aussieht. Für statische Positionen, die mit 2-rädrigen Robotern angefahren

[4] Es ist anzunehmen, dass simulierte 2-rädrige Roboter erstmals 2006 in der 3D-Simulationsliga des RoboCup zum Einsatz kommen werden.

[5] Die Verwendung einer eigenen Simulationsumgebung hat außerdem den Vorteil, dass darin Experimente erheblich schneller ablaufen können, als im komplexen „rcssserver3d“.

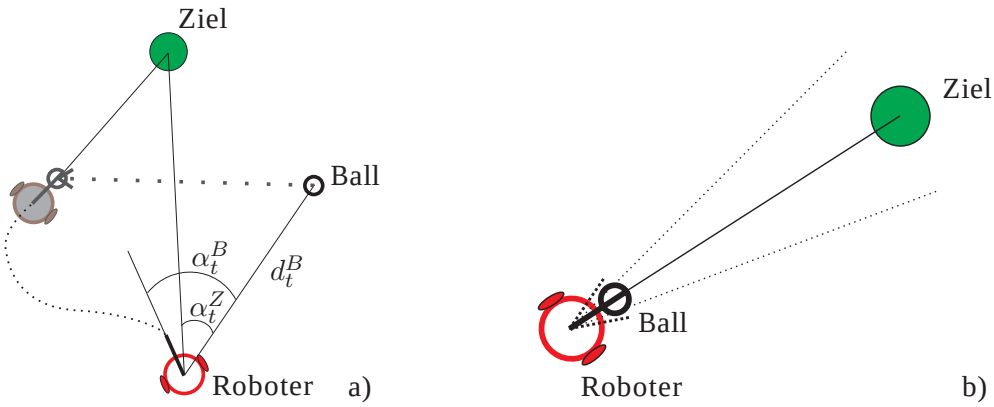


Abb. 6.2: Skizze verschiedener Zustände mit den Winkeln und Entfernungen. a) Zufälliger Startzustand mit einer möglichen Lösungstrajektorie. b) Der zu erreichende Zielzustand. Die gepunkteten Linien zeigen die Grenzen an, die bei den Winkeln eingehalten werden müssen.

werden sollen, können optimale Trajektorien angegeben werden [Balkcom und Mason, 2002]. Dabei dreht sich der Roboter entweder auf der Stelle oder fährt so schnell er kann nach vorne. Das Problem wird schwieriger, wenn sich die Position, die angefahren werden soll, verändert und noch dazu unter bestimmten Bedingungen erreicht werden soll. Um dabei die optimalen Trajektorien zu verwenden, muss zunächst der Punkt bestimmt werden, an dem der Roboter unter den richtigen Bedingungen auf den Ball treffen kann. D.h. die Bewegung des Balls und die möglichen Bewegungen des Roboters müssen voraus simuliert werden. Noch schwieriger ist es, wenn die Wahrnehmung des Roboters oder das Ausführen von Aktionen durch Rauschen gestört ist. Dann kann nämlich der Punkt, an dem der Roboter auf den Ball trifft, nicht mehr ganz genau bestimmt werden. Algorithmen, die die optimalen Trajektorien benutzen, wären dann hauptsächlich damit beschäftigt, den Roboter zu drehen, um ihn auf einen Treffpunkt auszurichten, der sich ständig leicht ändert. In diesem Fall ist es wohl besser immer vorwärts zu fahren und kleine Korrekturen während der Fahrt auszuführen. Da nicht unbedingt klar ist, wie eine solche Lösung aussehen sollte[6], bietet es sich an, ein Lernverfahren zur Lösung dieses Problems einzusetzen.

Der simulierte Roboter ist rund und besitzt mit einem Durchmesser von $0.44\ \mathrm{m}$ dieselbe Größe wie in der 3D-Simulationsliga des RoboCup. Die maximale Geschwindigkeit des Roboters beträgt $2\ \frac{\mathrm{m}}{\mathrm{s}}$. Der simulierte Roboter ist in der Lage, die Geschwindigkeiten der Räder alle 100 Millisekunden zu ändern. Bei der Simulation des Roboters werden keine Beschleunigungs- oder Bremsvorgänge berücksichtigt. Ebenso wird Reibung vernachlässigt und es werden auch keine Kollisionen simuliert. Natürlich spielen diese Effekte in der Realität und auch in der 3D-Simulationsliga eine große Rolle. Die prinzipiellen Schwierigkeiten bei der betrachteten Aufgabe verringern sich durch diese Vereinfachungen jedoch nicht. So erfordert der Einsatz in der 3D-Simulationsliga keine zusätzlichen Eingabedimensionen und anstelle der Geschwindigkeiten könnten Beschleunigungen oder Kräfte gelernt werden. Gelingt es, in der hier verwendeten Simulationsumgebung ein erfolgreiches Verhalten zu lernen, so sollte dasselbe Lernverfahren auch in der 3D-Simulationsliga anwendbar sein.

[6] Gegenwärtig werden in der 3D-Simulationsliga bei dem Problem, einen Ball so anzufahren, dass er in Richtung eines vorgegebenen Ziels geschossen werden kann wohl mehr oder weniger geschickte Heuristiken verwendet, wobei oft vereinfacht von einem ruhenden Ball ausgegangen wird. Zudem handelt es sich bei den derzeit simulierten Robotern einfach um Kugeln, die keine Orientierung besitzen und den Ball immer schießen können, wenn dieser nahe genug ist.

Der Ball hat einen Durchmesser von 0.22 m. Beim Ball ist es nötig, eine Art Reibung mitzusimulieren, da die Geschwindigkeit des Balls höher als die Geschwindigkeit des Roboters sein kann. Daher nimmt die Ballgeschwindigkeit mit $v^B_{t+1} = 0.995 v^B_t$ in jedem Simulationsschritt von 10 Millisekunden ab, d.h. dass der Ball jede Sekunde etwa 40 Prozent seiner Geschwindigkeit verliert.
Beim Lernen wird davon ausgegangen, dass sich der Ball auf dem Boden befindet und nicht allzu schnell ist. Ansonsten wäre es wahrscheinlich so gut wie unmöglich, den Ball unter den richtigen Bedingungen abzufangen. Außerdem wird davon ausgegangen, dass der Ball irgendwo in der Nähe des Roboters ist, da dies sehr einfach mit einem hand-kodierten Verhalten erreicht werden kann und deswegen nicht extra gelernt werden braucht. Daher wird der Ball zu Beginn einer Episode auf einen zufälligen Punkt bis zu 5 m entfernt vom Roboter gesetzt und erhält eine zufällige Geschwindigkeit von bis zu $4\ \frac{\text{m}}{\text{s}}$. Das Ziel, zu dem der Ball geschossen werden soll, wird zufällig in 5–15 m Entfernung vom Ball gewählt.
Die kontinuierlichen Aktionen a, die der Roboter lernt, können Werte zwischen -1 und 1 annehmen und kodieren die Geschwindigkeiten, die auf die Räder angewendet werden sollen. Falls $a > 0$, wird das linke Rad auf eine Geschwindigkeit von 1 und das rechte Rad auf $1 - 2\,|a|$ gesetzt. Ist $a > 0$, so wird genau umgekehrt verfahren. Der Roboter versucht sich also immer so schnell wie möglich unter Einhaltung der gelernten Geschwindigkeitsdifferenz zu bewegen. Der Zustand wird in einem 5-dimensionalen Eingabevektor mit folgenden Dimensionen kodiert (vgl. Abbildung 6.2):

- Der Abstand d^B_t vom Roboter zum Ball.
- Der Winkel α^B_t zwischen der aktuellen Orientierung des Roboters und der Richtung zum Ball.
- Der Winkel α^Z_t zwischen der Richtung zum Ziel und der Richtung vom Roboter zum Ball.
- Die Differenz $\Delta\alpha^B_t = \alpha^B_t - \alpha^B_{t-1}$ in α^B_t zwischen letzten und aktuellen Zeitschritt.
- Die Differenz $\Delta\alpha^Z_t = \alpha^Z_t - \alpha^Z_{t-1}$ in α^Z_t zwischen letzten und aktuellen Zeitschritt.

Die letzten beiden Dimensionen erlauben dem Roboter dabei eine Einschätzung, wie sich die relevanten Winkel α^B_t und α^Z_t aufgrund der Bewegungen des Balles und des Roboters ändern. Denn diese beiden Winkel sollen ja möglichst nahe bei Null sein, wenn der Roboter den Ball erreicht. Der Eingabevektor wird wie üblich so normiert, dass die einzelnen Einträge zwischen 0 und 1 liegen:

$$x_t = \begin{pmatrix} d^B_t \,/\, 5.0 \\ \left(\alpha^B_t + \pi\right) / \, 2\pi \\ \left(\alpha^Z_t + \pi\right) / \, 2\pi \\ \left(\Delta\alpha^B_t + 0.5\pi\right) / \, \pi \\ \left(\Delta\alpha^Z_t + 0.5\pi\right) / \, \pi \end{pmatrix} \tag{6.8}$$

Der Roboter erhält in jedem Zeitschritt, in dem er die erforderlichen Bedingungen nicht erreicht, eine vom Abstand zum Ball abhängige Bestrafung von $-0.01 d^B_t$. Dadurch sollte er lernen, dass Ziel möglichst schnell zu erreichen. Außerdem werden dadurch erfolgreiche Episoden zu Beginn des Lernprozesses etwas wahrscheinlicher, da so der Roboter von Anfang an versucht, die Entfernung zum Ball zu minimieren. Wenn er den Ball unter den richtigen Bedingungen erreicht, erhält er eine Belohnung von 100. Dieser Wert wurde deshalb so hoch gewählt, damit der Unterschied zu der entfernungsabhängigen

Strafe pro Zeitschritt groß genug ist. Andernfalls besteht die Gefahr, dass der Roboter zwar lernt den Abstand zum Ball schnell zu minimieren, nicht aber die richtigen Winkel einzuhalten.

6.2.2 Ergebnisse

Bei der Auswertung steht diesmal nicht nur die Lerngeschwindigkeit der einzelnen Verfahren im Vordergrund, sondern vor allem auch die Qualität der gefundenen Lösungen, also die Frage, wie schnell es einer bestimmten Politik gelingt, den Ball abzufangen. Anders als beim Acrobot-Problem kann aber diesmal die Qualität der Lösung nicht einfach während des Trainings bestimmt werden. Weil beim Acrobot immer von demselben Startzustand ausgegangen wurde, konnte einfach die bis zum Aufschwingen benötigte Zeit als Maß für die Qualität der Lösung benutzt werden. Beim Abfangen des Balls gibt es aber unterschiedliche Startzustände, weil der Ball aus allen möglichen Situationen gut abgefangen werden soll. Um die Qualität einer Politik festzustellen, wird diese daher auf einer vordefinierten Menge unterschiedlicher Startzustände getestet. Die Startzustände wurden dabei so ausgewählt, dass sie ein großes Spektrum der möglichen Zustände abdecken. Der Ball ist dabei jeweils $2\,\mathrm{m}$ vor dem Roboter. Ziele werden in $10\,\mathrm{m}$ Entfernung zum Ball in acht verschiedenen Winkeln ($-180°, -135°, -90°, \ldots, 90°, 135°$) gesetzt. Der Ball hat eine Geschwindigkeit von $4\,\frac{\mathrm{m}}{\mathrm{s}}$ in eine von vier Richtungen ($-180°, -90°, 0°, 90°$) (siehe Abbildung 6.3). Insgesamt ergeben sich so 32 verschiedene Tests. Alle 100 Episoden wird die aktuelle Politik getestet, indem der Roboter mit allen 32 unterschiedlichen Startzuständen konfrontiert wird. Dabei wird die durchschnittliche Zeit bestimmt, die der Roboter pro Test benötigt, um den Ball abzufangen. Er hat dafür in jeder dieser Testepisoden maximal 25 Sekunden der simulierten Zeit zur Verfügung. Gelingt es innerhalb dieser Zeit nicht, den Ball richtig abzufangen, so wird die Episode als Fehlversuch gewertet und geht mit einer Zeit von 25 Sekunden in den Durchschnitt ein. Während der Testepisoden wird nicht gelernt und sie werden auch nicht bei der Gesamtanzahl der Episoden mitgezählt. Ein kompletter Durchgang mit allen 32 Tests wird im Folgenden als *Evaluation* bezeichnet. Während eines Laufs merkt sich der Roboter den minimalen bisher bei den Evaluationen erreichten Durchschnittswert und speichert die aktuelle Politik ab, wenn dieser Wert unterschritten wird. Die beste abgespeicherte Politik würde dann als Politik im Roboter eingesetzt werden.
Für das zugrunde liegende Wachsende Neurale Gas wurden dieselben Parameter wie im letzten Kapitel verwendet (siehe Tabelle 5.1). Für die Auswahl der übrigen Lernparameter wurden wieder einige vorläufige Tests durchgeführt. Es fand aber keine systematische Optimierung der Parameter statt. Alle μ_i und v_i werden mit Null initialisiert, die σ_i werden auf 0.25 gesetzt, um am Anfang stark zu explorieren. Als minimaler und maximaler Wert für σ_i wurde $\sigma_{min} = 0.1$ und $\sigma_{max} = 0.25$ gewählt. Als Lernrate für die μ_i, die σ_i und die Werte v_i in den Neuronen wurde 0.1 benutzt. Es wurde ein Diskontierungsfaktor γ von 0.9 verwendet. Bei allen Experimenten wurden jeweils 10 Läufe mit 20000 Episoden durchgeführt. Es wurden zunächst Experimente ohne Rauschen ausgeführt, um die verschiedenen Varianten des Verfahrens zu beurteilen. In einer zweiten Reihe von Experimenten wurde dann eines der zuvor getesteten Verfahren mit verrauschter Wahrnehmung und Rauschen in der Aktionsausführung konfrontiert.

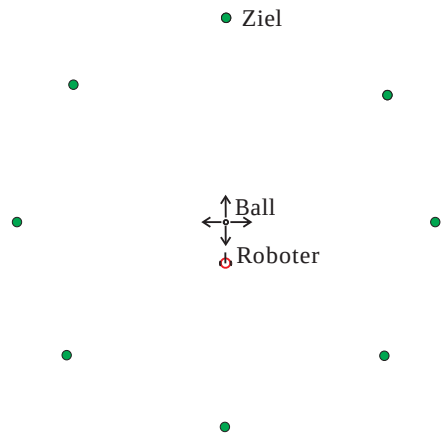


Abb. 6.3: Skizze der zum Testen benutzten Startzustände. Der Ball bewegt sich mit $4\ \frac{\mathrm{m}}{\mathrm{s}}$ in eine der vier Richtungen und der Roboter muss ihn so abfangen, dass er zum jeweiligen Ziel geschossen werden kann.

Experimente ohne Rauschen

Es wurden wieder Experimente mit NN-, Kanten- und Lambda-Verfahren mit verschiedenen Arten der Neuronenbewegung und konstanter Approximation bzw. lokal linearer Interpolation durchgeführt. Für alle Verfahren wurden Experimente mit maximal 1024 Neuronen durchgeführt (bei 243 Startneuronen) und für die Kanten-Verfahren zusätzlich Experimente mit maximal 243 Neuronen (32 Startneuronen) und 1536 Neuronen (1024 Startneuronen). Die Kanten-Verfahren wurden für die Experimente mit unterschiedlichen Neuronenanzahlen ausgewählt, weil die Lambda-Verfahren, die bei maximal 1024 Neuronen ähnlich gut abschneiden, aufwändiger sind. NN-Verfahren sind bei maximal 1024 Neuronen nicht in der Lage eine befriedigende Politik zu lernen, so dass sie für weitergehende Experimente ohnehin nicht in Frage kommen. Da die Verfahren mit konstanter Approximation diesmal sehr gut abschnitten, wurden auch Experimente mit konstanter Approximation und fehlerabhängiger Bewegung der Neuronen durchgeführt.
Zur Auswertung wurde der durchschnittliche Anteil von Fehlversuchen für alle 200 pro Lauf durchgeführten Evaluationen und zusätzlich für die letzten 50 Evaluationen alleine bestimmt. Genauso wurde die durchschnittlich pro Testepisode benötigte Zeit für alle Evaluationen und die letzten 50 Evaluationen berechnet. Zusätzlich wurde noch das Minimum der durchschnittlichen Zeit pro Testepisode über alle durchgeführten Evaluationen und alle Läufe bestimmt, um die Qualität der besten von den jeweiligen Verfahren gelernten Politiken zu beurteilen. Tabelle 6.1 zeigt alle Werte. Zur besseren Übersicht wird ein Teil der Ergebnisse für maximal 1024 Neuronen in den Abbildungen 6.4 und 6.5 und ein Teil der Ergebnisse für die Kanten-Verfahren mit unterschiedlichen Neuronenanzahlen in den Abbildungen 6.7 und 6.8 graphisch dargestellt.
Betrachtet man den prozentualen Anteil der Fehlversuche bei maximal 1024 Neuronen für die NN-Verfahren in Tabelle 6.1, so kann man sofort feststellen, dass die NN-Verfahren diesmal äußerst ungeeignet sind. Am besten schneidet noch das NN-Verfahren mit fehlerabhängiger Bewegung ab, aber auch dabei sind um die 70 % Fehlversuche zu verzeichnen. Sehr gut hingegen schneiden die Kanten- und Lambda-Verfahren mit konstanter Approximation und beweglichen Neuronen ab (siehe Abbildung 6.4). Bei diesen kommt es bei den letzten 50 Evaluationen zu keinen oder nur äußerst wenigen Fehlversuchen. Die Verfahren mit lokal linearer Interpolation sind diesmal schlechter als die Verfahren mit konstanter Approximation. Dabei können die Verfahren mit fehlerabhängi-

Methode	**Anteil Fehlvers. gesamt**	**Anteil Fehlvers. letzte 50**	**Zeit / Test gesamt**	**Zeit / Test letzte 50**	**Zeit / Test Minimum**
Kante const 243	3.99 %	0.29 %	6.94	5.80	4.84
Kante const err. 243	12.54 %	1.27 %	8.93	5.95	4.89
Kante 243	81.23 %	65.82 %	22.17	19.88	5.75
Kante err. 243	80.27 %	67.82 %	22.02	20.11	5.53
Kante const fix 243	96.84 %	95.98 %	24.59	24.51	14.42
Kante fix 243	97.63 %	97.50 %	24.79	24.78	15.86
NN const 1024	95.28 %	96.80 %	24.14	24.39	14.09
NN const err. 1024	88.72 %	94.95 %	23.34	24.23	10.53
NN 1024	95.09 %	95.73 %	24.31	24.41	15.24
NN err. 1024	72.95 %	68.49 %	21.41	20.76	11.88
NN const fix 1024	99.50 %	99.50 %	24.90	24.88	21.69
NN fix 1024	99.63 %	99.44 %	24.92	24.87	22.84
Kante const 1024	1.13 %	0.00 %	5.53	5.04	4.68
Kante const err. 1024	1.97 %	0.08 %	5.71	5.03	4.79
Kante 1024	10.96 %	3.21 %	9.61	7.57	4.93
Kante err. 1024	2.78 %	0.32 %	6.57	5.57	4.78
Kante const fix 1024	30.29 %	23.98 %	16.02	15.08	8.16
Kante fix 1024	58.84 %	58.21 %	18.96	18.93	5.96
Lambda const 1024	0.86 %	0.10 %	5.72	5.31	4.71
Lambda const err. 1024	2.13 %	0.00 %	5.62	5.08	4.75
Lambda 1024	32.22 %	18.03 %	13.73	11.22	5.12
Lambda err. 1024	2.43 %	1.43 %	6.25	5.76	4.62
Lambda const fix 1024	19.91 %	19.71 %	12.94	12.98	5.86
Lambda fix 1024	98.76 %	99.70 %	24.81	24.94	10.55
Kante const 1536	1.41 %	0.00 %	5.68	5.10	4.74
Kante const err. 1536	1.36 %	0.00 %	5.62	5.08	4.75
Kante 1536	12.03 %	1.71 %	9.81	7.05	5.14
Kante err. 1536	7.72 %	1.94 %	8.27	6.99	4.98

Tabelle 6.1: Testergebnisse für das Abfangen des Balls. Die ersten beiden Spalten geben den prozentualen Anteil von Fehlversuchen pro Test für alle und die jeweils letzten 50 Evaluationen in einem Lauf an. Die nächsten beiden Spalten beinhalten die durchschnittlich benötigte Zeit pro Test ebenfalls für alle Evaluationen und die jeweils letzten 50 Evaluationen pro Lauf. Die letzte Spalte gibt das Minimum der durchschnittlichen Zeit pro Test über alle Evaluationen in allen 10 Läufen an.

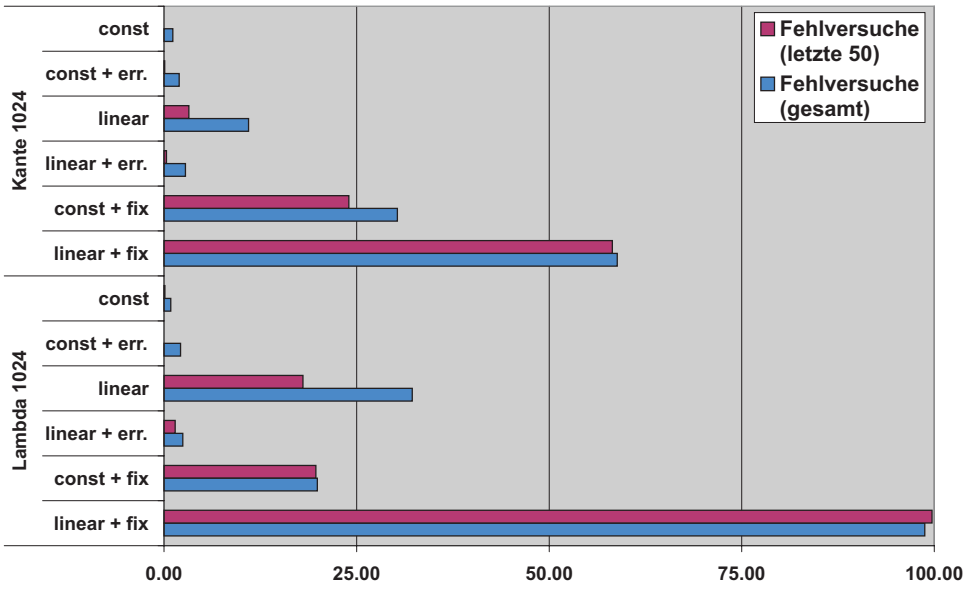


Abb. 6.4: Prozentualer Anteil von Fehlversuchen für alle Evaluationen und die jeweils 50 letzten Evaluationen pro Lauf für die verschiedenen Verfahren bei einer maximalen Anzahl von 1024 Neuronen.

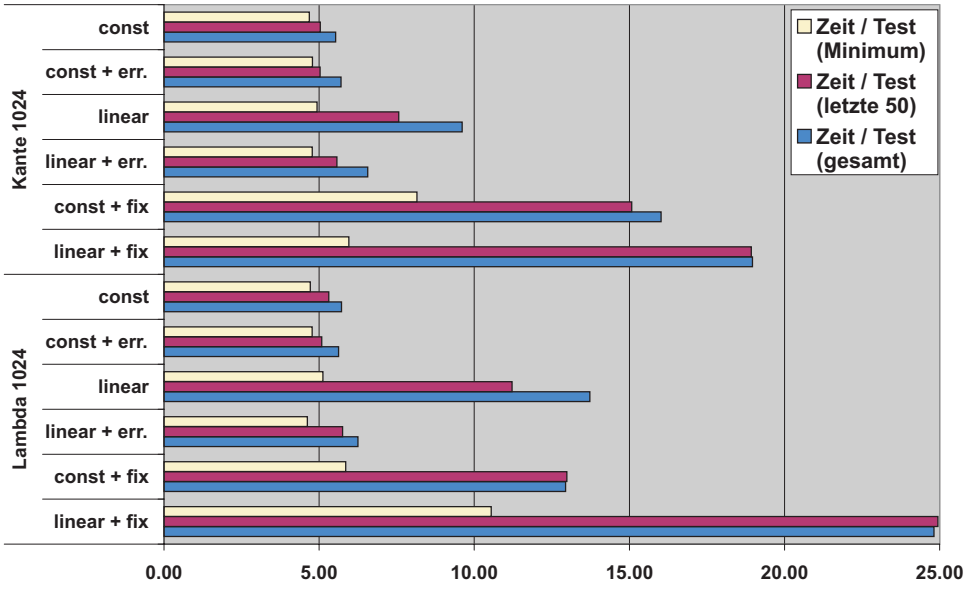


Abb. 6.5: Die durchschnittlichen Zeiten pro Test bei der besten Evaluation und die Durchschnitte über die jeweils 50 letzten Evaluationen pro Lauf und über alle Evaluationen für die verschiedenen Verfahren bei einer maximalen Anzahl von 1024 Neuronen.

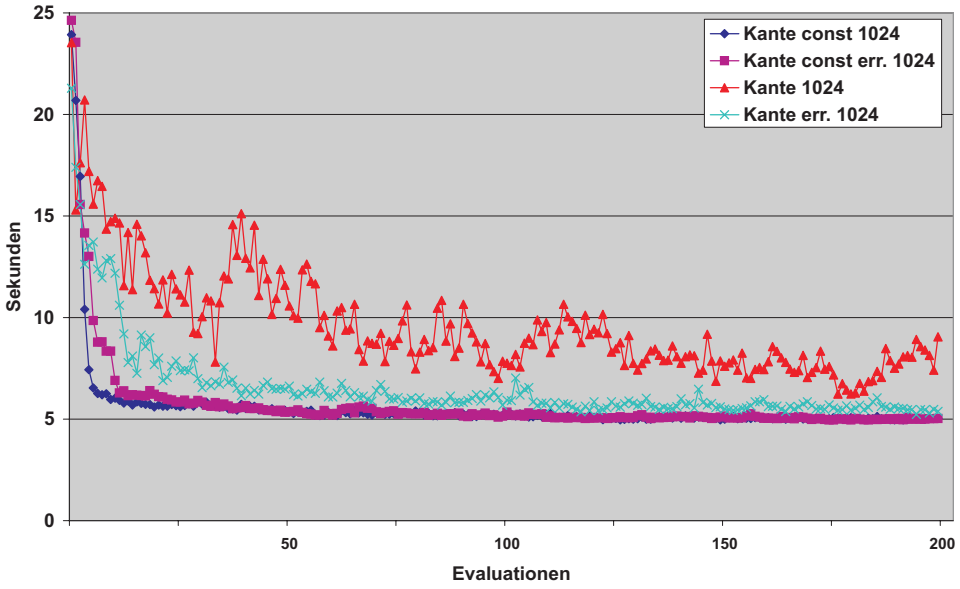


Abb. 6.6: Entwicklung der im Durchschnitt pro Test benötigten Zeit für die vier Kanten-Verfahren mit beweglichen Neuronen und einer maximalen Anzahl von 1024 Neuronen (Durchschnitt aus 10 Läufen).

ger Bewegung und lokal linearer Interpolation fast noch die Ergebnisse der konstanten Approximation erreichen. Deutlich mehr Fehlversuche haben die Verfahren mit häufigkeitsabhängiger Bewegung und lokal linearer Interpolation. Bei der konstanten Approximation hingegen hat die Art der Bewegung keine großen Auswirkungen. Sehr viele Fehlversuche treten bei allen Verfahren mit festem Netz auf. Auch dabei sind Verfahren mit konstanter Approximation besser als Verfahren mit lokal linearer Interpolation. Aber selbst bei dem besten Verfahren mit konstanter Approximation kommt es ohne Bewegung der Neuronen in fast 20 % aller Tests zu Fehlversuchen.
Auch bei der Qualität der gefundenen Lösungen sind die Kanten- und Lambda-Verfahren mit beweglichen Neuronen besser als alle Verfahren mit festem Netz, die deutlich schlechtere Lösungen liefern (siehe Abbildung 6.5). Die allerbeste Lösung wird dabei vom „Lambda err. 1024“-Verfahren mit einer durchschnittlichen Zeit von 4.62 Sekunden pro Testepisode gefunden, dicht gefolgt vom „Kante const 1024“-Verfahren mit 4.68 Sekunden. Bei der durchschnittlich benötigten Zeit pro Test sind die Verfahren mit konstanter Approximation wieder etwas besser als Verfahren mit lokal linearer Interpolation. Bei lokal linearer Interpolation ist häufigkeitsabhängige Bewegung wieder schlechter als fehlerabhängige Bewegung der Neuronen. Am schlechtesten schneiden wieder die Verfahren ohne Bewegung der Neuronen ab. Bewegliche Neuronen sind also mitentscheidend für den Lernerfolg. Bei fast allen Verfahren sind die Durchschnittswerte für die letzten 50 Evaluationen dem Mittel über alle Evaluationen sehr ähnlich, was dafür spricht, dass recht schnell gelernt wird. Diese Vermutung wird durch Abbildung 6.6 bestätigt, die Lernkurven für Kanten-Verfahren mit beweglichen Neuronen zeigt. Kanten-Verfahren mit konstanter Approximation lernen sehr schnell, wobei die häufigkeitsabhängige Bewegung noch etwas schneller als die fehlerabhängige Bewegung ist. Lokal lineare Interpolation mit fehlerabhängiger Bewegung ist über den ganzen Lauf etwas und lokal lineare Interpolation mit häufigkeitsabhängiger Bewegung sogar deutlich schlechter. Bei häufigkeitsabhängiger Bewegung schwanken die Durchschnittswerte stark, was andeutet, dass der Lernprozess dabei nicht sonderlich stabil ist.

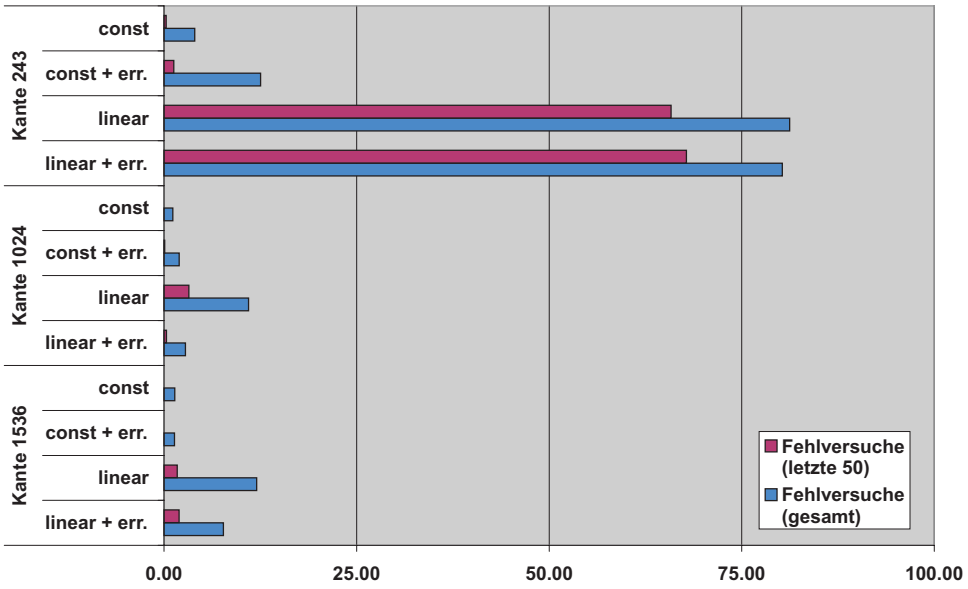


Abb. 6.7: Prozentualer Anteil von Fehlversuchen für alle Evaluationen und die jeweils 50 letzten Evaluationen pro Lauf für Kanten-Verfahren mit beweglichen Neuronen bei verschiedenen Maximalanzahlen von Neuronen.

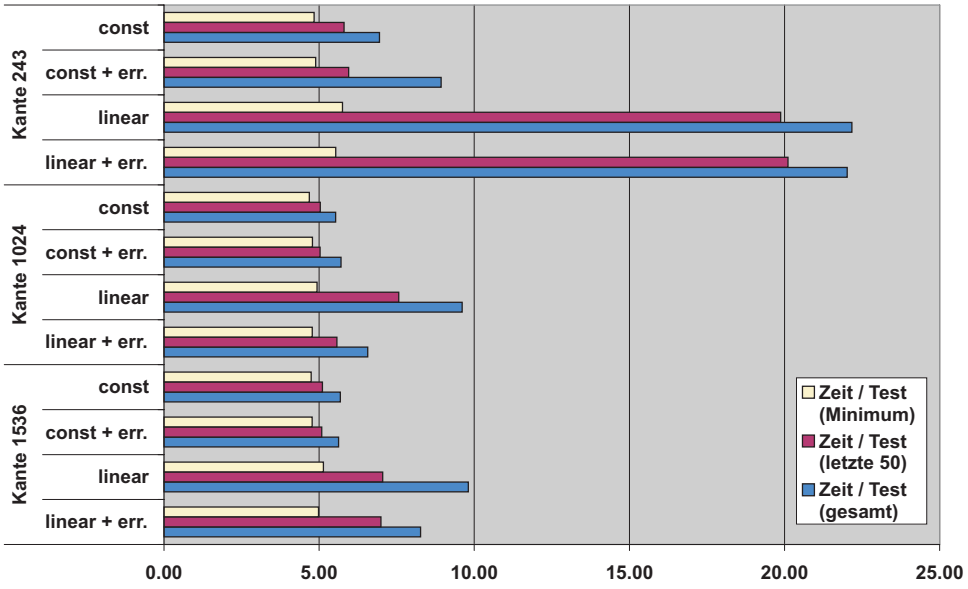


Abb. 6.8: Die durchschnittlichen Zeiten pro Test bei der besten Evaluation und die Durchschnitte über die jeweils 50 letzten Evaluationen pro Lauf und über alle Evaluationen für Kanten-Verfahren mit beweglichen Neuronen bei verschiedenen Maximalanzahlen von Neuronen.

Für die Kanten-Verfahren wurden auch die Auswirkungen der maximalen Anzahl von Neuronen untersucht. Dabei wurden Experimente mit maximal 243, 1024 und 1536 Neuronen durchgeführt, wobei bei 243 ($= 3^5$) und 1024 ($= 4^5$) Neuronen auch Vergleiche mit Verfahren ohne Bewegung der Neuronen angestellt werden konnten. Da es diesen bei 243 Neuronen jedoch nicht gelingt, eine befriedigende Politik zu lernen (siehe Tabelle 6.1) und sie auch bei 1024 Neuronen deutlich schlechter als die anderen Verfahren sind (siehe Abbildungen 6.4 und 6.5), wurden diese Ergebnisse nicht in die Abbildungen 6.7 und 6.8 aufgenommen. Betrachtet man den prozentualen Anteil von Fehlversuchen für Kanten-Verfahren in Abbildung 6.7, so überrascht das gute Abschneiden der Kanten-Verfahren mit konstanter Approximation bei nur 243 Neuronen. Es kommt während der jeweils letzten 50 Evaluationen in einem Lauf zu fast keinen Fehlversuchen, während Kanten-Verfahren mit lokal linearer Interpolation mindestens 65 % Fehlversuche haben. Bei 1024 Neuronen sinkt der Anteil der Fehlversuche für alle Kanten-Verfahren, während er bei einer weiteren Erhöhung der maximalen Neuronenanzahl auf 1536 für Verfahren mit lokal linearer Interpolation wieder etwas steigt.
Die Überlegenheit der Kanten-Verfahren mit konstanter Approximation spiegelt sich auch in der durchschnittlichen Anzahl von Schritten pro Test in Abbildung 6.8 wider. Bei maximal 243 Neuronen gelingt es nur diesen Verfahren, gute Durchschnittswerte zu erreichen. Aber auch sonst ist die konstante Approximation besser als die lokal lineare Interpolation. Bei den besten gefundenen Politiken sind die Unterschiede nicht so groß, aber die von den Verfahren mit konstanter Approximation gefundenen Politiken sind fast immer etwas besser. Die Art der Bewegung der Neuronen spielt keine besonders große Rolle. Es werden meist sehr ähnliche Werte bei häufigkeitsabhängiger und bei fehlerabhängiger Bewegung erreicht. Mit maximal 1024 Neuronen werden insgesamt bessere Resultate als mit 243 Neuronen erreicht. In den allermeisten Fällen sind die Resultate mit 1024 Neuronen auch etwas besser als die mit 1536 Neuronen erzielten. Bei diesem Problem ist 1024 also eine gute Wahl für die maximale Anzahl von Neuronen.
Abbildung 6.9 zeigt einige Beispieltrajektorien für das Abfangen des Balls. Dabei wurde die beste gefundene Politik des „Kante const 1024“-Verfahrens benutzt. Die entstandenen Trajektorien sind relativ glatt und der Ball wird fast immer so angefahren, dass das Ziel mit hoher Präzision getroffen werden kann. Die Trajektorien zeigen, dass sich das vorgeschlagene Lernverfahren gut eignet, um das Abfangen des Balls unter den geforderten Bedingungen zu lernen.
Zusammenfassend lässt sich feststellen, dass sich lokal lineare Interpolation in diesem Fall negativ auf den Lernerfolg auswirkt und deswegen konstante Approximation verwendet werden sollte. Das könnte damit zusammenhängen, dass bei lokal linearer Interpolation kleine Neuronenbewegungen schon eine große Auswirkung auf die Approximation haben können. Dennoch hat die verwendete adaptive Struktur des zugrunde liegenden Wachsenden Neuralen Gases deutliche Vorteile: Alle Verfahren ohne bewegliche Neuronen schneiden nach allen Kriterien deutlich schlechter als die Verfahren mit beweglichen Neuronen ab. Es hat sich auch herausgestellt, dass die Gewichtung mehrerer lokaler Modelle entscheidend ist. Keinem NN-Verfahren gelingt es das Problem auch nur einigermaßen befriedigend zu lösen. Da Lambda-Verfahren in etwa dieselben Werte wie Kanten-Verfahren erreichen, aber deutlich komplizierter sind, bietet sich das Kanten-Verfahren mit konstanter Approximation in den Neuronen als Standardverfahren an und soll daher einfach mit *ReinforceGNG* bezeichnet werden.

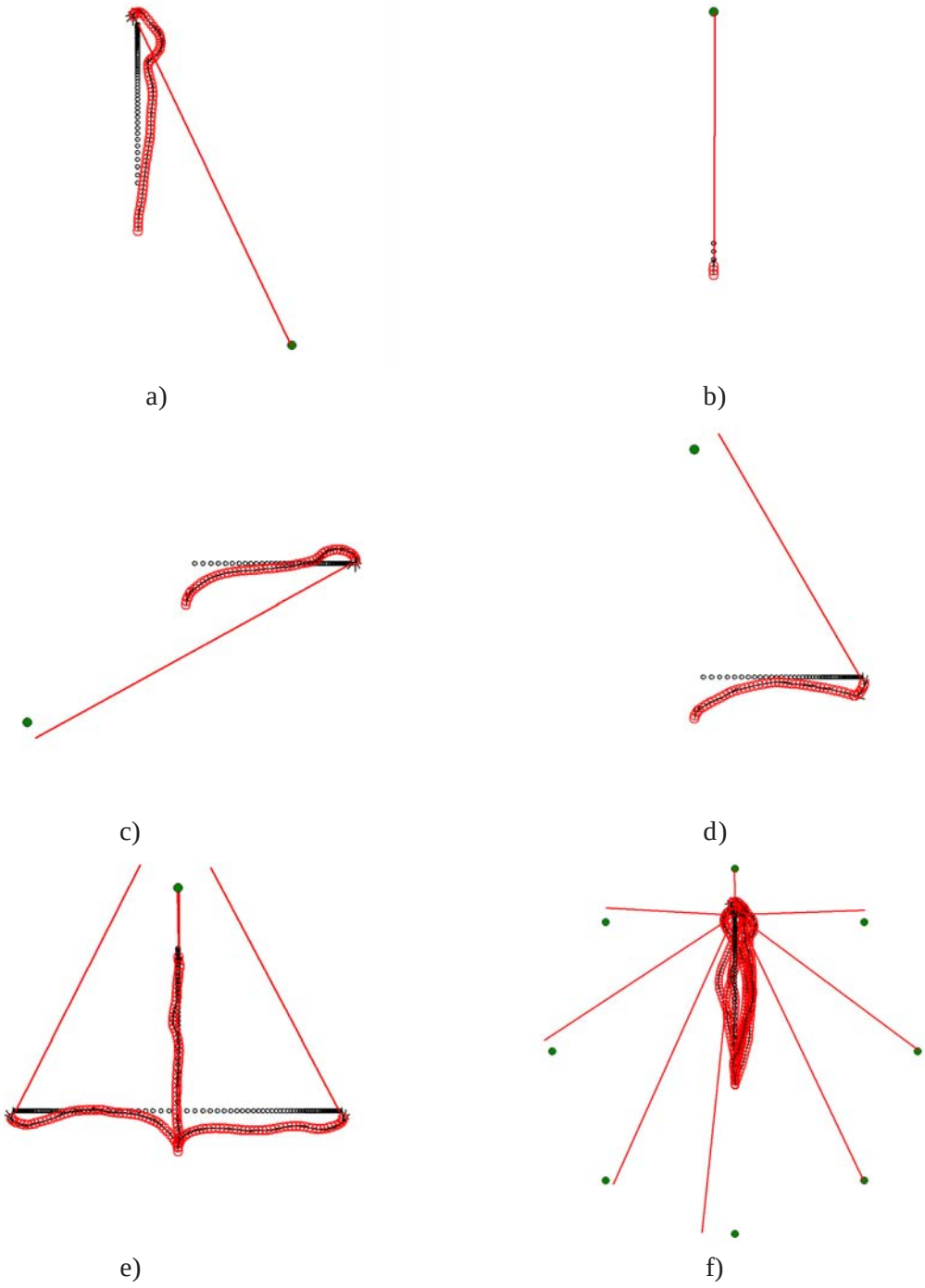


Abb. 6.9: Trajektorien des mit dem „Kante const 1024“-Verfahren gelernten Verhaltens. Der Roboter wird durch den großen, roten Kreis, der Ball durch den kleinen, schwarzen Kreis und das Ziel durch den ausgefüllten, grünen Kreis visualisiert. Alle Objekte wurden alle 100 Millisekunden gezeichnet. Die roten Linien zeigen die Richtung an, in die der Ball am Ende der Episode geschossen werden würde. a) - d) Jeweils eine Trajektorie für einen bestimmten Startzustand. e) Ein festes Ziel zusammen mit den vier verschiedenen Richtungen, in die sich der Ball bewegen kann. Der Ball der sich auf den Roboter zu bewegt, ist kaum zu sehen, da er sehr schnell abgefangen wird. f) Trajektorien für einen sich nach oben bewegenden Ball mit Schüssen zu allen acht verschiedenen Zielen.

Experimente mit Rauschen

In einer zweiten Reihe von Experimenten wurde der Einfluss von Rauschen in der Wahrnehmung und in der Ausführung von Aktionen auf das Lernen untersucht. Bei der Wahrnehmung wurde normalverteiltes Rauschen von $N(0, 0.0965)$ den Distanzinformationen und normalverteiltes Rauschen von $N(0, 0.1225)$ allen Winkelinformation hinzugefügt. Diese Werte sind dieselben wie die in der Simulationsliga des RoboCup 2004 verwendeten. Bei der Ausführung von Aktionen wurde normalverteiltes Rauschen von $N(0, 0.05)$ auf die Geschwindigkeiten der Räder in jedem Simulationsschritt gegeben.
Diesmal wurden nur Experimente mit dem ReinforceGNG-Verfahren durchgeführt, d.h. es wurde das Kanten-Verfahren mit konstanter Approximation und häufigkeitsabhängiger Bewegung verwendet. Die Maximalanzahl der Neuronen wurde von 243 bis 1536 variiert. Wieder wurden jeweils 10 Läufe mit 20000 Episoden durchgeführt. Wie Tabelle 6.2 zeigt, gelingt es auch bei verrauschter Wahrnehmung und verrauschter Aktionsausführung gute Politiken zu lernen, um den rollenden Ball unter den richtigen Bedingungen abzufangen.

Methode	**W**	**A**	**Anteil Fehlvers. gesamt**	**Anteil Fehlvers. letzte 50**	**Zeit / Test gesamt**	**Zeit / Test letzte 50**	**Zeit / Test Minimum**
Kante const 243			3.99 %	0.29 %	6.94	5.80	4.84
Kante const 1024			1.13 %	0.00 %	5.53	5.04	4.68
Kante const 1536			1.41 %	0.00 %	5.68	5.10	4.74
Kante const 243		✓	1.67 %	5.81 %	8.63	7.07	4.98
Kante const 1024		✓	1.93 %	0.00 %	5.74	5.10	4.70
Kante const 1536		✓	1.22 %	0.00 %	5.69	5.17	4.78
Kante const 243	✓		10.35 %	2.99 %	9.61	8.11	6.21
Kante const 1024	✓		2.93 %	0.45 %	7.46	6.66	5.63
Kante const 1536	✓		1.97 %	0.07 %	7.06	6.28	5.51
Kante const 243	✓	✓	6.31 %	2.96 %	8.82	8.08	6.02
Kante const 1024	✓	✓	3.32 %	0.61 %	7.80	7.03	5.70
Kante const 1536	✓	✓	2.39 %	0.07 %	7.12	6.31	5.54

Tabelle 6.2: Testergebnisse für das Abfangen des Balls mit und ohne Rauschen auf die Wahrnehmung von Zuständen (W) und Ausführung von Aktionen (A). Die ersten beiden Spalten geben den prozentualen Anteil von Fehlversuchen pro Test für alle und die jeweils letzten 50 Evaluationen in einem Lauf an. Die nächsten beiden Spalten beinhalten die durchschnittlich benötigte Zeit pro Test ebenfalls für alle Evaluationen und die jeweils letzten 50 Evaluationen pro Lauf. Die letzte Spalte gibt das Minimum der durchschnittlichen Zeit pro Test über alle Evaluationen in allen 10 Läufen an.

Abbildung 6.10 zeigt den prozentualen Anteil von Fehlversuchen für die verschiedenen Arten von Rauschen und verschiedene Maximalanzahlen von Neuronen. Meist verschlechtert Rauschen bei der Wahrnehmung der Zustände das Ergebnis stärker als Rauschen bei der Aktionsausführung. Beides zusammen führt zu einer weiteren Erhöhung des Anteils von Fehlversuche. Interessanterweise ist dieses Verhältnis bei 243 Neuronen jedoch gerade umgekehrt. Hier kommt es bei Rauschen in der Aktionsausführung zu einem höheren Anteil von Fehlversuchen als bei Rauschen in der Wahrnehmung. Das kann daran liegen, dass die Approximation ohnehin nicht so genau ist, da nur wenige Neuronen vorhanden sind. So scheint die zusätzliche Verschlechterung durch die verrausch-

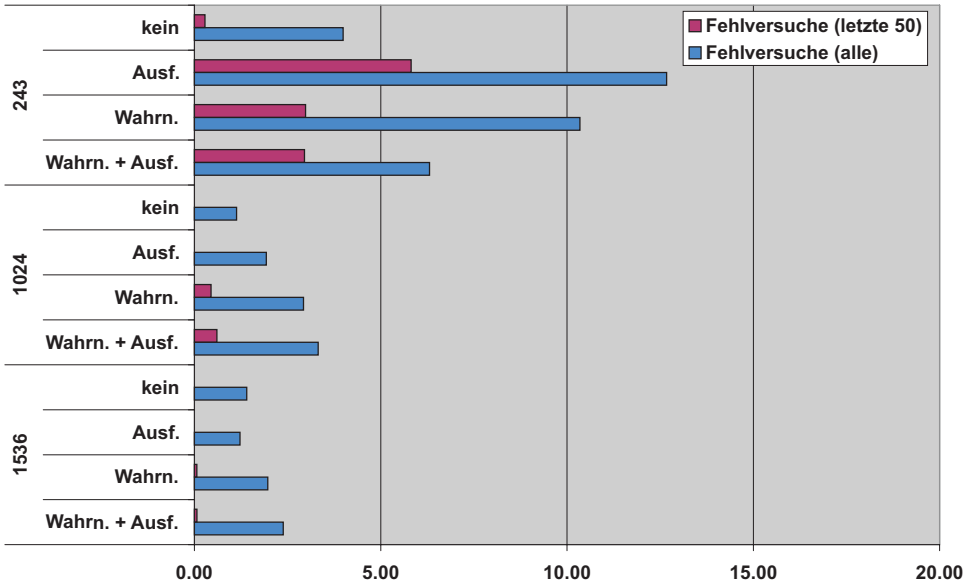


Abb. 6.10: Prozentualer Anteil von Fehlversuchen für alle Evaluationen und die jeweils 50 letzten Evaluationen pro Lauf für ReinforceGNG mit verschiedenen Maximalanzahlen von Neuronen und verschiedenen Arten von Rauschen. Beachte, dass die Skala im Gegensatz zu den Abbildungen 6.4 und 6.7 diesmal nur bis 20 % reicht.

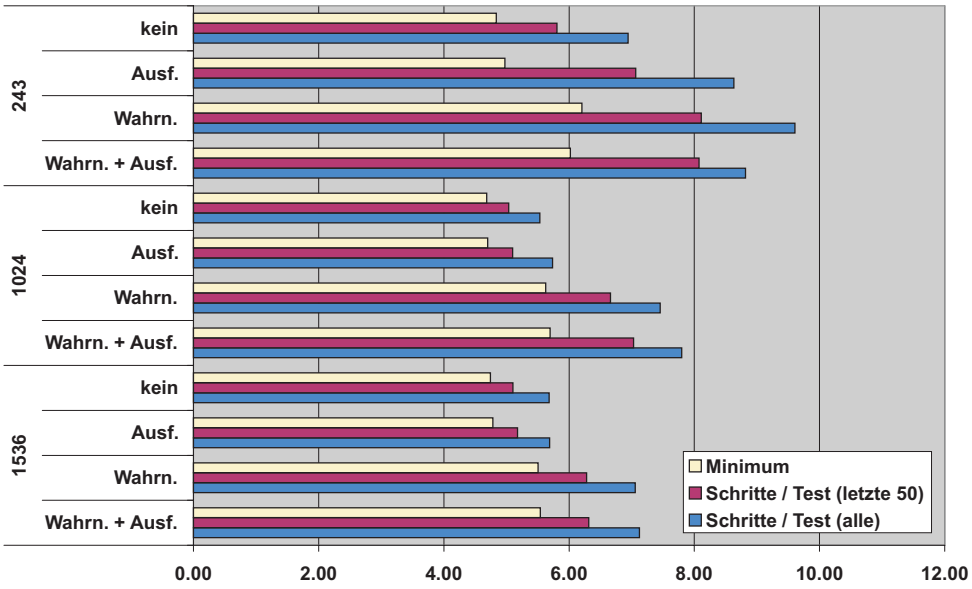


Abb. 6.11: Die durchschnittlichen Zeiten pro Test bei der besten Evaluation und die Durchschnitte über die jeweils 50 letzten Evaluationen pro Lauf und über alle Evaluationen für ReinforceGNG mit verschiedenen Maximalanzahlen von Neuronen und verschiedenen Arten von Rauschen. Beachte, dass die Skala im Gegensatz zu den Abbildungen 6.5 und 6.8 diesmal nur bis 12 Sekunden reicht.

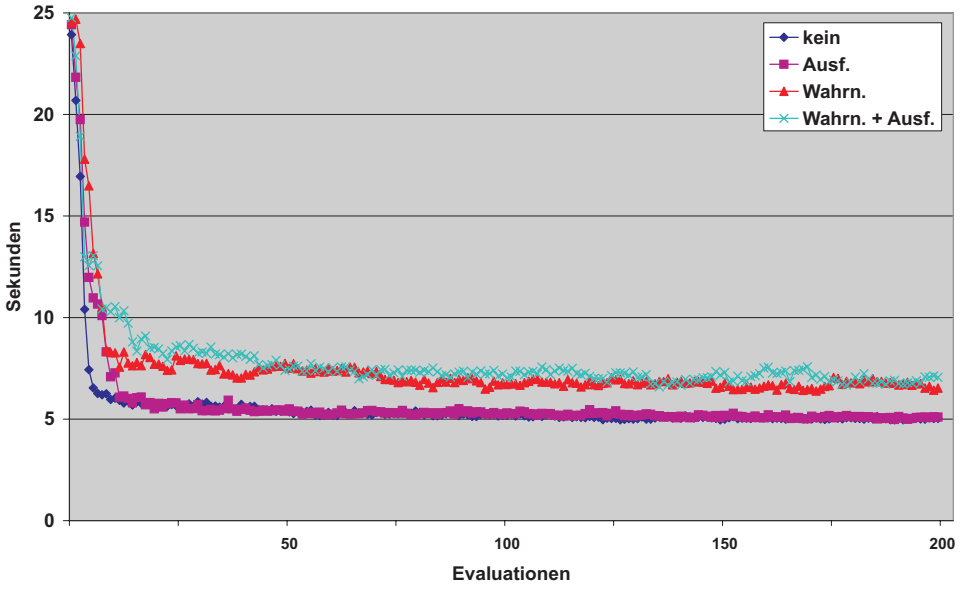


Abb. 6.12: Entwicklung der im Durchschnitt pro Test benötigten Zeit für ReinforceGNG mit maximal 1024 Neuronen und verschiedenen Arten von Rauschen (Durchschnitt aus 10 Läufen).

te Wahrnehmung nicht so stark ins Gewicht zu fallen wie die verrauschte Ausführung der grob approximierten Politik. Mit steigender Neuronenanzahl sinkt der Anteil von Fehlversuchen bei allen Arten von Rauschen. Ohne Rauschen steigt dieser Anteil beim Übergang von 1024 auf 1536 Neuronen ganz minimal an. Während der letzten 50 Evaluationen kommt es bei 1536 Neuronen unabhängig vom Rauschen zu so gut wie keinen Fehlversuchen.

Auf die minimale durchschnittlich benötigte Zeit der jeweils besten gefundenen Politiken hat Rauschen in der Aktionsausführung fast keine Auswirkungen (siehe Abbildung 6.11). Die gefundenen Minima bei den Evaluationen gleichen den Werten, die ohne Rauschen erzielt werden. Sobald Rauschen in der Wahrnehmung hinzukommt, verschlechtern sich diese Ergebnisse etwas. Dabei ist es egal, ob auch die Aktionsausführung verrauscht ist. Dasselbe gilt auch für die erreichten Durchschnittswerte über alle und die letzten 50 Evaluationen. Der Unterschied zwischen den verschiedenen Maximalanzahlen von Neuronen ist nicht besonders ausgeprägt. Schon mit 243 Neuronen werden gute Politiken gelernt, die sich beim Übergang auf 1024 Neuronen minimal verbessern. Zwischen maximal 1024 Neuronen und 1536 Neuronen besteht kaum ein Unterschied.

Abbildung 6.12 zeigt Lernkurven für ReinforceGNG mit maximal 1024 Neuronen bei den verschiedenen Arten von Rauschen. Dabei werden die bisherigen Ergebnisse bestätigt. Zwischen keinem Rauschen und Rauschen in der Aktionsausführung besteht kaum ein Unterschied. Rauschen in der Aktionsausführung führt lediglich zu einem minimal langsameren Lernen. Rauschen in der Wahrnehmung verschlechtert die Durchschnitte um etwa eine Sekunde. Zusätzliches Rauschen in der Aktionsausführung verschlechtern diese Werte nur noch minimal. Dass sich Rauschen in der Aktionsausführung kaum auswirkt, dürfte daran liegen, dass sich die Fehler in der Geschwindigkeit über die Zeit einigermaßen „herausmitteln“. Rauschen in der Wahrnehmung erschwert das Lernen, weil dadurch zum einen nicht die bisher gelernten Aktionen für den tatsächlichen Zustand ausgeführt werden, zum anderen aber auch die Auswirkungen der ausgeführten Aktion nicht dem tatsächlichen Zustand, sondern dem falsch wahrgenommenen Zustand zuge-

schrieben werden. Insgesamt zeigen diese Ergebnisse aber, dass auch bei verrauschter Wahrnehmung und Rauschen in der Aktionsausführung gute Politiken gelernt werden können und das Verfahren auch bei solchen Problemen einsetzbar ist.

6.2.3 Vergleich mit ähnlichen Problemen

So weit uns bekannt ist, stellen die vorgestellten Experimente die ersten Versuche dar, kontinuierliche Aktionen für einen 2-rädrigen Roboter in einem so komplexen Szenario, wie dem hier betrachteten, zu lernen. Daher können die erzielten Ergebnisse nur qualitativ mit ähnliche Problemen verglichen werden, in denen ebenfalls Aktionen für 2-rädrige Roboter gelernt wurden oder bewegliche Objekte berücksichtigt werden mussten. In der Darstellung und der Beurteilung der Schwierigkeit der betrachteten Probleme wird dabei vor allem die Art des zu lösenden Problems berücksichtigt. Die tatsächliche Schwierigkeit des jeweils bearbeiteten Lernproblems hängt natürlich noch von einer ganzen Reihe weiterer Faktoren ab, wie z.B. der Wahl der Belohnungsfunktion, der Kodierung des Eingabezustandes oder der Art und der Stärke von eventuell vorhandenem Rauschen. So hat der Zustandsraum in einigen der im folgenden beschriebenen Arbeiten deutlich mehr Dimensionen als in unserem Problem, da direkt auf Sensordaten gelernt wird[7]. Im Folgenden kann es also nicht um einen Vergleich der Leistungsfähigkeit der verwendeten Verfahren gehen, sondern nur um den grundsätzlichen Schwierigkeitsgrad der bearbeiteten Probleme.

- In [Riedmiller und Janusz, 1995] lernt ein Roboterarm diskrete Aktionen, um einen rollenden Ball zu greifen. Dabei wird immer von demselben Startzustand und derselben Ballgeschwindigkeit ausgegangen. Es reicht aus, den Arm irgendwie in die Nähe des Balls zu bringen und es müssen keine besonderen Bedingungen eingehalten werden. Das dort betrachtete Problem ist also erheblich einfacher, als das in unseren Experimenten verwendete.

- In [Gaskett u. a., 1999a,b] werden mit dem Wire-Fitting-Ansatz (siehe 2.4) kontinuierliche Aktionen für einen Unterwasser-Roboter gelernt. Dabei wird mit einer 2-dimensionalen Simulation gearbeitet und die Steuerung des Roboters erfolgt ähnlich wie bei dem hier simulierten 2-rädrigem Roboter. Es wird gelernt, bestimmte Punkte der Reihe nach anzufahren. Diese Punkte sind statisch und es ist egal, von welcher Seite sich der Roboter ihnen nähert. Auch dieses Problem ist deutlich einfacher, als das Abfangen eines sich bewegenden Balls. Zudem sehen die gelernten Trajektorien nicht besonders zeit-optimal aus.

- In [Großmann und Poli, 2000] lernt ein 2-rädriger Roboter diskrete Aktionen, um verschiedene Ziele anzufahren. Das Problem wird dabei dadurch verschärft, dass sich der Roboter in einer durch Wände unregelmäßig begrenzten Arena befindet (es gibt viele Ecken, in denen der Roboter hängen bleiben kann). Der Zustandsraum ist 4-dimensional (2D-Position des Roboters, Orientierung, Winkel zum Ziel). Das Problem ist etwas schwieriger als das Anfahren eines statischen Ziels, da das Ziel nicht immer auf direktem Weg angefahren werden kann, aber immer noch erheblich leichter als das Anfahren eines beweglichen Ziels. Ein ähnliches Problem, wobei allerdings Startposition und Zielposition immer dieselben sind, wird auch in [Provost u. a., 2004] betrachtet.

[7] Dabei bestehen jedoch zwischen den einzelnen Dimensionen sehr starke Abhängigkeiten und nicht alle Daten sind relevant in Hinblick auf die Lösung des jeweiligen Problems, so dass der zur Lösung eigentlich notwendige Zustandsraum doch relativ niedrig-dimensional ist.

- Auch in [Smart und Kaelbling, 2001] geht es darum, ein statisches Ziel zu erreichen. Es muss allerdings zusätzlich ein Hindernis vermieden werden, das auf dem Weg liegt. Der Zustand besteht dabei aus den Entfernungen und Winkeln zum Ziel und zum Hindernis. Dieses Problem ist etwas schwieriger als das Anfahren eines Ziels ohne Hindernis, aber ebenfalls erheblich leichter als das in der vorliegenden Arbeit betrachtete.

- Bei dem „Wall-following"-Problem muss ein Roboter den Wänden eines Korridors oder einer umgrenzten Arena folgen [Smart und Kaelbling, 2001, 2000; Buck u. a., 2002a; Millán u. a., 2002]. Die Aktionen geben dabei die gewünschte Rotationsgeschwindigkeit des Roboters an. Auch bei diesem Problem gibt es weder bewegliche Ziele, noch müssen bestimmte Bedingungen beim Erreichen des Ziels eingehalten werden.

- In verschiedenen weiteren Arbeiten lernen Roboter Hindernisse zu vermeiden [Janusz und Riedmiller, 1995; Touzet, 1997; Gross u. a., 1996; Aljibury u. a., 2002; Buck u. a., 2002b]. Allerdings soll dabei kein bestimmtes Ziel angefahren werden, sondern einfach möglichst geradeaus gefahren werden. Hindernisvermeidung ist ein deutlich einfacheres Problem als das Abfangen eines sich bewegenden Balls, da kein bestimmter Zielzustand unter vorgegebenen Bedingungen erreicht werden muss und sich die Hindernisse, die vermieden werden sollen, nicht bewegen.

- In [Gross u. a., 1997] lernt ein mit einer Farbkamera ausgerüsteter Khepera-Miniatur-Roboter ein Andock-Manöver, bei dem ein vorgegebenes Ziel unter einem bestimmten Winkel erreicht werden muss. Aktionen bestehen aus kontinuierlichen Werten für den Winkel, um den sich der Roboter drehen soll, und aus der kontinuierlichen Geschwindigkeit. Der Aktionsraum ist also deutlich komplizierter als bei unseren Experimenten. Das Ziel, das angefahren werden soll, bewegt sich aber nicht, was das Problem deutlich einfacher macht.

- In [Buck u. a., 2002a] wird eine Zustands-Wertefunktion für das Problem bestimmt, dass ein Roboter an einem anderen Roboter vorbeifahren muss. Der andere Roboter folgt einer festen Politik und versucht den ersten Roboter abzufangen. Bei diesem Problem muss also die Bewegung des anderen Roboters berücksichtigt werden. Ein ähnliches Problem mit beweglichen Zielen wird auch in [Buck u. a., 2002b] betrachtet. Auch dabei reicht es aus, irgendwie geeignete Ausweichbewegungen auszuführen, was das Problem etwas einfacher macht, als das gezielte Abfangen eines sich bewegenden Balls. Außerdem wird bei diesen Arbeiten ein Modell der Umwelt benötigt, da nur eine Zustands-Wertefunktion gelernt wird und mögliche Aktionen deswegen simuliert werden müssen, um die auszuführende Aktion zu bestimmen.

Insgesamt erscheinen alle beschriebenen Probleme einfacher als das in der vorliegenden Arbeit betrachtete. Dabei kommen die in [Gross u. a., 1997] und [Buck u. a., 2002a] beschriebenen Probleme, dem hier verwendeten Problem noch am nächsten. In [Gross u. a., 1997] muss ein festes Ziel unter bestimmten Bedingungen angefahren werden und in [Buck u. a., 2002a] müssen bewegliche Objekte in der Umwelt berücksichtigt werden, allerdings ohne das ein genau vorgegebenes Ziel erreicht werden muss. Das in der vorliegenden Arbeit betrachtete Problem beinhaltet beide Aspekte. Da bei diesem komplexen Problem gute Ergebnisse erreicht werden, kann man davon ausgehen, dass sich ReinforceGNG auch bei ähnlich komplexen Problemen erfolgreich anwenden lässt.

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde mit LWIGNG zunächst ein allgemein einsetzbarer Funktionsapproximator auf Basis eines Wachsenden Neuralen Gases entwickelt, der auf die speziellen Anforderungen beim Lernen von Aktionen eingeht. Wachsende Neurale Gase wurden deshalb ausgewählt, weil sie in der Lage sind, sich der lokalen Häufigkeit und der lokalen Struktur einer unbekannten Eingabeverteilung anzupassen. Zur Approximation von Funktionen erhalten die einzelnen Neuronen zusätzliche Werte, mit deren Hilfe dann ein Ausgabewert berechnet werden kann. Bei dieser Berechnung wird die Nachbarschaftsstruktur des Wachsenden Neuralen Gases explizit berücksichtigt und zwischen den Werten benachbarter Neuronen interpoliert. Dafür wurden bereits bestehende Verfahren, bei denen aber die verwendeten Selbstorganisierenden Karten bestimmten Anforderungen genügen mussten, verallgemeinert und vereinfacht. Dabei wird nicht nur für das Gewinnerneuron eine solche Interpolation berechnet, sondern auch für alle benachbarten Neuronen. Die Ausgabewerte dieser so genannten lokal linearen Modelle gehen dann in eine gewichtete Summe ein, um eine kontinuierlichere Approximation zu erreichen. Es werden zwei verschiedene Möglichkeiten der Gewichtung benutzt: Die eine verwendet die durchschnittliche Kantenlänge aller von einem Neuron ausgehenden Kanten (Kanten-Verfahren), die andere führt zusätzliche Parameter ein, um den wechselseitigen Einfluss der Modelle aufeinander zu steuern (Lambda-Verfahren). Schließlich wurde noch die Bewegung der Neuronen so geändert, dass sie sich auch nach dem Approximationsfehler richtet.
In verschiedenen Experimenten wurde die Leistungsfähigkeit des entwickelten Funktionsapproximators demonstriert. Diese Experimente wurden so konstruiert, dass ähnliche Schwierigkeiten wie beim Lernen von Aktionen bestehen. LWIGNG wurde dabei mit LWPR, einem state-of-the-art Verfahren zur inkrementellen Funktionsapproximation, verglichen. Zunächst wurden Experimente mit einer 2-dimensionalen Testfunktion durchgeführt. In weitergehenden Experimenten wurde der 2-dimensionale Eingabevektor durch Hinzufügen von acht konstanten Dimensionen und Multiplikation mit einer Rotationsmatrix zu einem 10-dimensionalen und schließlich durch Hinzufügen von 10 weiteren leicht verrauschten Dimensionen auch zu einem 20-dimensionalen Eingabevektor erweitert. Der Vorteil der verwendeten adaptiven Struktur zeigt sich vor allem darin, dass trotz des wachsenden Schwierigkeitsgrades bei allen diesen Experimenten diesel-

be Approximationsgüte mit fast derselben geringen Anzahl von Neuronen erreicht wird. Die Anpassung an die lokale Struktur der Eingabedaten ermöglicht es, Ressourcen zu sparen und auch in höher-dimensionalen Räumen zu lernen. In allen Experimenten erreicht LWIGNG eine etwas bessere Approximationsgüte als LWPR. Es zeigt sich auch, dass LWIGNG gerade bei vielen Eingabedimensionen hinsichtlich der Laufzeit erheblich schneller als LWPR ist.
In einer zweiten Reihe von Experimenten wurde getestet, wie das Verfahren auf sich verändernde Zielfunktionen, stark verrauschte Eingabedaten zu Beginn des Trainings und auf sich verändernde Eingabeverteilungen reagiert, auf Bedingungen also, wie sie für das Lernen von Aktionen typisch sind. Auch dabei konnte sich LWIGNG bewähren und sehr gute Ergebnisse erzielen. Verändert sich die Zielfunktion, so kann sich LWIGNG diesen Veränderungen besser als LWPR anpassen. Bei stark verrauschten Eingabedaten erreicht LWPR zwar zunächst eine bessere Approximationsgüte, wird mit der Zeit aber schlechter und kann bei abnehmendem Rauschen die Funktion nicht so gut wiedergeben wie LWIGNG. Bei Veränderungen in der Eingabeverteilung kann LWIGNG diesen aufgrund seiner adaptiven Struktur gut folgen, während LWPR neue Modelle einfügen muss. Gerade aufgrund dieser Eigenschaften sollte LWIGNG gut als Funktionsapproximator für Reinforcement-Learning-Verfahren geeignet sein.
Im weiteren Verlauf der Arbeit wurde LWIGNG dann mit Q-Learning zum Q-LWIGNG-Verfahren gekoppelt und zur Approximation einer Aktions-Wertefunktion eingesetzt. Dazu musste der zugrunde liegende GNG-Algorithmus abgewandelt werden, um die spezifische Reihenfolge der Daten beim Lernen von Aktionen zu berücksichtigen. Vor allem in diesem Punkt und darin, dass die topologische Nachbarschaft in der Neuronenstruktur mit in die Approximation der Aktions-Wertefunktion eingeht, unterscheidet sich Q-LWIGNG von allen bisher entwickelten Verfahren zum Reinforcement Learning auf Basis von Selbstorganisierenden Karten.
Q-LWIGNG wurde anhand von verschiedenen Standardproblemen des Reinforcement Learning getestet. Beim Stabbalance-Problem erzielt Q-LWIGNG deutlich bessere Werte bei der durchschnittlichen Anzahl der benötigten Episoden bis der Stab lange balanciert wird als vergleichbare Ergebnisse aus der Literatur. Auch beim Mountain-Car-Problem erreichen die hier vorgestellten Verfahren sowohl bei der Lerngeschwindigkeit als auch bei den gefundenen Politiken sehr gute Resultate, die mit den besten Ergebnissen aus der Literatur vergleichbar sind. Nur beim Acrobot-Problem konnten die besten bekannten Ergebnisse bei der durchschnittlichen Anzahl von Schritten pro Episode nicht ganz erreicht werden. Q-LWIGNG war aber in der Lage sehr gute Einzellösungen zu finden, die nur von modellbasierten Verfahren übertroffen werden.
Bei allen Experimenten schneiden vor allem die Kanten-Verfahren sehr gut ab. Lambda-Verfahren und zum Vergleich herangezogene Verfahren ohne eine Gewichtung von mehreren lokalen Modellen (NN-Verfahren) sind in den allermeisten Fällen deutlich schlechter und oft nicht in der Lage, befriedigende Politiken zu lernen. Lokal lineare Interpolation wirkt sich bei den Kanten-Verfahren äußerst positiv auf das Lernverhalten aus und ist in allen Experimenten besser als die Verwendung von konstanter Approximation. Verfahren ohne Bewegung von Neuronen konnten nur teilweise gute Ergebnisse erreichen. Es scheint immer darauf anzukommen, ob die Neuronen für das jeweilige Problem vorab gut im Eingaberaum verteilt sind. Ist dies nicht der Fall, so bringt die verwendete adaptive Struktur des zugrunde liegenden Wachsenden Neuralen Gases eindeutig Vorteile. Zwischen fehlerabhängiger und häufigkeitsabhängiger Neuronenbewegung besteht meist kein großer Unterschied und es hängt vom jeweiligen Problem ab, welche Methode besser geeignet ist. Meist genügen relativ wenige Neuronen, um ein gutes Lernverhalten zu erreichen. Werden zu viele Neuronen eingesetzt, wird oft langsamer gelernt, da

nicht so stark generalisiert wird. Abschließend lässt sich festhalten, dass die Kombination von Q-Learning und Kanten-Verfahren mit beweglichen Neuronen und lokal linearer Interpolation bei den betrachteten Problemen ein sehr gutes Lernverhalten aufweist und daher auch gut bei anderen Problemen einsetzbar sein sollte, um diskrete Aktionen zu lernen.

Zum Lernen von kontinuierlichen Aktionen wurde LWIGNG mit einem REINFORCE-Algorithmus zum ReinforceLWIGNG- bzw. ReinforceGNG-Verfahren verbunden. Dafür musste ein REINFORCE-Algorithmus so erweitert werden, dass auch aus zeitverzögerten Belohnungen gelernt werden kann. LWIGNG wird in einer Actor-Critic-Architektur sowohl zum Lernen einer Zustands-Wertefunktion, als auch zur Approximation der Politik eingesetzt. Dabei lernt LWIGNG situationsabhängige Erwartungswerte und Standardabweichungen für eine Normalverteilung, aus der dann die auszuführende Aktion bestimmt wird. Das ist unseres Wissens das erste Mal, dass ein auf einer Selbstorganisierenden Karte basierendes Verfahren auf diese Weise eingesetzt wird.

Um dieses Verfahren zu testen, wurden Aktionen für einen 2-rädrigen Roboter gelernt. Dieser hatte die Aufgabe einen sich bewegenden Ball so anzufahren, dass er in Richtung eines vorgegebenen Ziels geschossen werden kann. Dieses Problem ist komplexer als die sonst in der Literatur betrachteten Probleme, bei denen ebenfalls Aktionen für einen 2-rädrigen Roboter gelernt oder beim Lernen bewegliche Objekte berücksichtigt werden müssen. ReinforceLWIGNG war in der Lage, dieses komplexe Problem zufrieden stellend zu lösen, wenn eine lokale Gewichtung mehrerer Modelle, wie bei Lambda- und Kanten-Verfahren, verwendet wurde. Verfahren mit konstanter Approximation sind dabei meist erfolgreicher als Verfahren mit lokal linearer Interpolation. Es zeigt sich wieder, dass sich die Bewegung der Neuronen äußerst positiv sowohl auf die durchschnittlich pro Testepisode benötigte Zeit als auch auf die Qualität der gefundenen Lösungen auswirkt. Alle Verfahren mit festem Netz schneiden schlechter als vergleichbare Verfahren mit beweglichen Neuronen ab. Ob die Neuronen dabei häufigkeitsabhängig oder auch fehlerabhängig bewegt werden, ist hingegen nicht so entscheidend. Das Kanten-Verfahren mit konstanter Approximation und beweglichen Neuronen bietet sich als Standardverfahren an und wird als ReinforceGNG bezeichnet. ReinforceGNG bewährte sich auch in Experimenten mit Rauschen in der Wahrnehmung von Zuständen oder der Ausführung von Aktionen und soll in der Simulationsliga des RoboCup angewendet werden, sobald simulierte 2-rädrige Roboter verfügbar sind.

Für die Zukunft sind verschiedene Richtungen denkbar, in die weiter gearbeitet werden könnte: so könnte etwa die Komplexität der betrachteten Probleme weiter erhöht werden, die praktische Anwendbarkeit durch eine Verringerung der einstellbaren Parameter erleichtert oder das Verfahren anderweitig weiterentwickelt werden. Eine interessante Frage betrifft beispielsweise das Ausnutzen der speziellen rekursiven Struktur der Wertefunktionen beim Reinforcement Learning. Die Bellmann-Gleichungen beschreiben einen Zusammenhang zwischen dem Wert eines Zustandes und dem Wert des Nachfolgezustandes. Diese Struktur könnte eventuell beim Reinforcement Learning mit Selbstorganisierenden Karten ausgenutzt werden, etwa indem Kanten entlang der typischen Zustandsübergänge eingefügt werden.

Bei den in der vorliegenden Arbeit entwickelten Verfahren könnten auch Aktivierungsspuren (eligibility traces) eingesetzt werden. Damit könnten bei der Anpassung der Werte auch weiter in der Zukunft liegende Belohnungen berücksichtigt werden. Auch die Bewegungsregeln für die Neuronen bieten noch Spielraum für Verbesserungen. Gegenwärtig orientieren sich diese an der lokalen Häufigkeit der Eingabevektoren oder an dem geschätzten Approximationsfehler. Gerade beim Lernen von Aktionen sind aber auch andere Kriterien denkbar. So könnte etwa die aktuelle Politik bei der Neuronen-

bewegung beachtet werden. Bei Q-LWIGNG könnten Unterschiede in den Q-Werten für die einzelnen Aktionen, bei ReinforceLWIGNG auch Erwartungswert und Standardabweichung der Normalverteilung berücksichtigt werden. So sind in Regionen, wo die Politik relativ sicher ist und sich die Aktionen gleichen oder zumindest sehr ähnlich sind vielleicht weniger Neuronen nötig, als in Gebieten, in denen das nicht der Fall ist.
Insgesamt wurde in der vorliegenden Arbeit neben dem allgemein einsetzbaren Funktionsapproximator LWIGNG ein ganzes Spektrum von Methoden zum Lernen von diskreten und kontinuierlichen Aktionen auf Basis eines Wachsenden Neuralen Gases entwickelt. Dabei konnte gezeigt werden, dass die zugrunde liegende adaptive Struktur eindeutige Vorteile hat, und dass vor allem Kanten-Verfahren gut zum Lernen von Aktionen geeignet sind. Q-LWIGNG mit lokal linearer Interpolation eignet sich sehr gut zum Lernen diskreter Aktionen und mit ReinforceGNG steht ein Verfahren zur Verfügung, welches für das Lernen von kontinuierlichen Aktionen breit einsetzbar ist.

Literaturverzeichnis

[Albus 1971] ALBUS, J.S.: A theory of cerebellar function. In: *Mathematical Biosciences* (1971), Nr. 10, S. 25–61

[Albus 1981] ALBUS, J.S.: *Brain, Behavior, and Robotics*. Peterborough, NH : Byte Books, 1981

[Aljibury u. a. 2002] ALJIBURY, H. ; ARROYO, A. ; NECHYBA, M.: Using Locally Weighted Regression to Enhance Q-Learning. In: *2002 Florida Conference on Recent Advances in Robotics*, 2002

[Anderson 1987] ANDERSON, C.W.: Strategy Learning with Multilayer Connectionist Representations. In: *Proc. of the 4th International Workshop on Machine Learning*, 1987, S. 103–114

[Anderson 1993] ANDERSON, C.W.: Q-Learning with Hidden-Unit Restarting. In: *Advances in Neural Information Processing Systems* (1993), Nr. 5, S. 81–88

[Atkeson u. a. 1997] ATKESON, C.G. ; MOORE, A. ; SCHAAL, S.: Locally Weighted Learning. In: *Artificial Intelligence Review* 11 (1997), Nr. 1-5, S. 11–73

[Aupetit u. a. 2000a] AUPETIT, M. ; COUTURIER, P. ; MASSOTTE, P.: Function Approximation with Continuous Self-Organizing Maps using Neighbouring Influence Interpolation. In: *Proc. of Neural Computation (NC 2000)*, 2000

[Aupetit u. a. 2000b] AUPETIT, M. ; COUTURIER, P. ; MASSOTTE, P.: A 'Recruiting Neural-Gas' for Function Approximation. In: *Proc. International Joint Conference on Neural Networks (IJCNN 2000)*, 2000

[Aupetit u. a. 1999a] AUPETIT, M. ; MASSOTTE, P. ; COUTURIER, P.: C-SOM: A Continuous Self-Organizing Map For Function Approximation. In: *Proc. of Intelligent System and Control (ISC 99)*, 1999

[Aupetit u. a. 1999b] AUPETIT, M. ; MASSOTTE, P. ; COUTURIER, P.: A Continuous Self-Organizing Map using Spline Technique for Function Approximation. In: *Proc. of Artificial Intelligence Control and Systems (AICS 99)*, 1999

[Baird 1995] BAIRD, L.C.: Residual Algorithms: Reinforcement Learning with Function Approximation. In: *Proceedings of the Twelfth International Conference on Machine Learning*, 1995, S. 30–37

[Baird und Klopf 1993] BAIRD, L.C. ; KLOPF, H.: Reinforcement Learning with high-dimensional continuous actions / Wright-Patterson Air Force Base, Ohio. 1993 (WL-TR-93-1147). – Forschungsbericht

[Balkcom und Mason 2002] BALKCOM, D. ; MASON, M.: Time Optimal Trajectories for Bounded Velocity Differential Drive Vehicles. In: *International Journal of Robotics Research* 21 (2002), Nr. 3, S. 199–217

[Barreca und Buttazzo 1993] BARRECA, D.M. ; BUTTAZZO, G.C.: Learning Control Tasks by Failure Signals. In: *Proceedings of the 1993 World Congress on Neural Networks* Bd. 4, 1993, S. 518–522

[Barto u. a. 1993] BARTO, A.G. ; BRADTKE, S.J. ; SINGH, S.P.: Learning to Act using Real-Time Dynamic Programming / University of Massachusetts, Amherst. 1993 (UM-CS-1993-002). – Forschungsbericht

[Barto u. a. 1983] BARTO, A.G. ; SUTTON, R.S. ; ANDERSON, C.W.: Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. In: *IEEE Trans. on Systems, Man, and Cybernetics* 13 (1983), Nr. 5

[Bertsekas und Tsitsiklis 1996] BERTSEKAS, D. ; TSITSIKLIS, J.: *Neuro-dynamic programming*. Athena Scientific, Belmont, Massachusetts, 1996

[Bohn 1997] BOHN, C.-A.: An incremental unsupervised learning scheme for function approximation. In: *Proceedings of the 1997 IEEE International Conference on Neural Networks*, North-Holland, 1997, S. 1792–1797

[Boone 1997] BOONE, G.: Efficient Reinforcement Learning: Model-Based Acrobot Control. In: *International Conference on Robotics and Automation*, 1997

[Boyan und Moore 1995] BOYAN, J.A. ; MOORE, A.: Generalization in Reinforcement Learning: Safely Approximating the Value Function. In: TESAURO, G. (Hrsg.) ; TOURETZKY, D.S. (Hrsg.) ; LEEN, T. (Hrsg.): *Advances in Neural Information Processing Systems 7*. Cambridge, MA : The MIT Press, 1995, S. 369–376

[Bruske und Sommer 1995] BRUSKE, J. ; SOMMER, G.: Dynamic cell structure learns perfectly topology preserving map. In: *Neural Computation* 7 (1995), Nr. 4, S. 845–865

[Buck u. a. 2002a] BUCK, S. ; BEETZ, M. ; SCHMITT, T.: Approximating the Value Function for Continuous Space Reinforcement Learning in Robot Control. In: *Proceedings of the IEEE/RSJ IROS 2002*. Lausanne, Switzerland, 2002

[Buck u. a. 2002b] BUCK, S. ; STULP, F. ; BEETZ, M. ; SCHMITT, T.: Machine Control Using Radial Basis Value Functions and Inverse State Projection. In: *IEEE International Conference on Automation, Robotics, Control, and Vision*. Singapore, 2002

[Cannon 1967] CANNON, R.H.: *Dynamics of Physical Systems*. New York, 1967

[Coulom 2002] COULOM, R.: *Reinforcement Learning using Neural Networks with Applications to Motor Control*, Institute National Polytechnique de Grenoble, PhD Thesis, 2002

[Coulom 2004] COULOM, R.: High-Accuracy Value-Function Approximation with Neural Networks Applied to the Acrobot. In: VERLEYSEN, M. (Hrsg.): *European Symposium on Artificial Neural Networks*, 2004

[Davies 1997] DAVIES, S.: Multidimensional Triangulation and Interpolation for Reinforcement Learning. In: MOZER, M.C. (Hrsg.) ; JORDAN, M.I. (Hrsg.) ; PETSCHE, T. (Hrsg.): *Advances in Neural Information Processing Systems 9*, The MIT Press, 1997, S. 1005

[Dayan 1992] DAYAN, P.: The convergence of TD(λ) for general λ. In: *Machine Learning* 8 (1992), S. 341–362

[Finton und Yu 1994] FINTON, D. J. ; YU, H.H.: An Application of Importance-Based Feature Extraction in Reinforcement Learning. In: *Neural Networks for Signal Processing*, 1994

[Frank und Friedman 1993] FRANK, I. ; FRIEDMAN, J.: A Statistical View of Some Chemometrics Regression Tools. In: *Technometrics* 35 (1993), Nr. 2, S. 109–147

[Fritzke 1992] FRITZKE, B.: Growing Cell Structures—a Self-Organizing Network in *k* Dimensions. In: ALEKSANDER, I. (Hrsg.) ; TAYLOR, J. (Hrsg.): *Artificial Neural Networks II*. Amsterdam : North-Holland, 1992, S. 1051–1056

[Fritzke 1993] FRITZKE, B.: Growing Cell Structures – a self-organizing network for unsupervised and supervised learning / International Computer Science Institute, Berkeley. 1993 (TR-93-026). – Forschungsbericht

[Fritzke 1994a] FRITZKE, B.: Fast learning with incremental RBF networks. In: *Neural Processing Letters* 1 (1994), Nr. 1, S. 2–5

[Fritzke 1994b] FRITZKE, B.: Supervised Learning with Growing Cell Structures. In: COWAN, J. (Hrsg.) ; TESAURO, G. (Hrsg.) ; ALSPECTOR, J. (Hrsg.): *Advances in Neural Information Processing Systems 6*. San Mateo, CA : Morgan Kaufmann Publishers, 1994, S. 255–262

[Fritzke 1995a] FRITZKE, B.: A growing neural gas network learns topologies. In: TESAURO, G. (Hrsg.) ; TOURETZKY, D. (Hrsg.) ; LEEN, T. (Hrsg.): *Advances in Neural Information Processing Systems 7*. Cambridge, MA, 1995

[Fritzke 1995b] FRITZKE, B.: Incremental learning of local linear mappings. In: FOGELMAN, F. (Hrsg.) ; GALLINARI, P. (Hrsg.): *ICANN'95: International Conference on Artificial Neural Networks*. Paris, France : EC2 & Cie, 1995, S. 217–222

[Fritzke 1996] FRITZKE, B.: Unsupervised ontogenetic Networks. In: FIESLER, E. (Hrsg.) ; BEALE, R. (Hrsg.): *Handbook of Neural Computation*. Institute of Physics Publishing and Oxford University Press, 1996

[Gaskett u. a. 1999a] GASKETT, C. ; WETTERGREEN, D. ; ZELINSKY, A.: Q-Learning in Continuous State and Action Spaces. In: *Australian Joint Conference on Artificial Intelligence*, 1999, S. 417–428

[Gaskett u. a. 1999b] GASKETT, C. ; WETTERGREEN, D. ; ZELINSKY, A.: Reinforcement Learning applied to the control of an Autonomous Underwater Vehicle. In: *Proceedings of the Australian Conference on Robotics and Automation (AuCRA99)*, 1999

[Gibbs und MacKay 1997] GIBBS, M. ; MACKAY, D.: Efficient implementation of Gaussian processes / Cavendish Laboratory, Cambridge, UK. 1997. – Forschungsbericht

[Göppert und Rosenstiel 1993a] GÖPPERT, J. ; ROSENSTIEL, W.: Self-organizing maps vs. backpropagation: An experimental study. In: *Proc. of Workshop on Design Methodologies for Microelectronis and Signal Processing*, 1993, S. 153–162

[Göppert und Rosenstiel 1993b] GÖPPERT, J. ; ROSENSTIEL, W.: Topology-preserving interpolation in self-organizing maps. In: *Proceedings of NeuroNimes 93*, 1993, S. 425–434

[Göppert und Rosenstiel 1995a] GÖPPERT, J. ; ROSENSTIEL, W.: Interpolation in SOM: Improved generalization by iterative methods. In: *Proceedings of ICANN'95*, 1995, S. 425–434

[Göppert und Rosenstiel 1995b] GÖPPERT, J. ; ROSENSTIEL, W.: *Interpolierte Selbstorganisierende Karte in der Funktionsapproximation*. S. 77–86. In: BOCKLISCH, S. F. (Hrsg.): *Technologien und Neuronale Netze in der Praxis*. Aachen, Germany : Shaker Verlag, 1995

[Göppert und Rosenstiel 1995c] GÖPPERT, J. ; ROSENSTIEL, W.: Topological interpolation in SOM by affine transformations. In: *Proceedings of ESANN'95*. Brussels, 1995

[Göppert und Rosenstiel 1997] GÖPPERT, J. ; ROSENSTIEL, W.: The continuous interpolating self-organizing map. In: *Neural Processing Letters* 5 (1997), Nr. 3, S. 185–192

[Großmann 2001] GROSSMANN, A.: *Continual learning for mobile robots*, University of Birmingham, Birmingham, UK, PhD Thesis, 2001

[Großmann und Poli 2000] GROSSMANN, A. ; POLI, R.: Learning a Navigation Task in Changing Environments by Multi-Task Reinforcement Learning. In: *Advances in Robot Learning. Proceedings of the Eighth European Workshop (EWLR-99)*, 2000, S. 23–43

[Gross u. a. 1996] GROSS, H.-M. ; STEPHAN, V. ; BOEHME, H.-J.: Sensory-based Robot Navigation using Self-Organizing Networks and Q-Learning. In: *Proceedings of World Congress on Neural Networks WCNN'98*, 1996, S. 94–99

[Gross u. a. 1997] GROSS, H.-M. ; STEPHAN, V. ; KRABBES, M.: A Neural Field Approach to Topological Reinforcement Learning in Continuous Action Spaces. In: *IEEE World Congress on Computational Intelligence WCCI'98 - IJCNN'98*, 1997

[Gullapalli 1990] GULLAPALLI, V.: Associative Reinforcement Learning of Real-valued Functions / University of Massachusetts, Amherst, MA. 1990 (UM-CS-1990-129). – Forschungsbericht

[Gullapalli 1992a] GULLAPALLI, V.: Dynamic systems control via associative reinforcement learning. In: *Dynamic, Genetic, and Chaotic Programming: The Sixth Generation*, 1992

[Gullapalli 1992b] GULLAPALLI, V.: *Reinforcement Learning and its Application to Control*, University of Massachusetts, Amherst, MA, PhD Thesis, 1992

[Hart u. a. 1968] HART, P.E. ; NILSSON, N.J. ; RAPHAEL, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *EEE Transactions on Systems Science and Cybernetics* SSC4 (1968), Nr. 2, S. 100–107

[Hastie u. a. 2001] HASTIE, T. ; TIBSHIRANI, R. ; FRIEDMAN, J.: *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. New York : Springer, 2001

[Hawlitzky 2001] HAWLITZKY, M.: *Untersuchungen zu dynamischen Erweiterungen von Kohonen-Karten*, Johannes Gutenberg-Universität, Mainz, Diplomarbeit, 2001

[Hebb 1949] HEBB, D.: *Organisation of Behavior*. New York : Wiley, 1949

[Herrmann und Der 1995] HERRMANN, M. ; DER, R.: Efficient Q-Learning by division of labour. In: *Proceedings International Conference on Artificial Neural Networks - ICANN'95* Bd. 2. Paris, 1995, S. 129–134

[Jaakkola u. a. 1994] JAAKKOLA, T. ; JORDAN, M.I. ; SINGH, S.P.: On the convergence of stochastic iterative dynamic programming algorithms. In: *Neural Computation* 6 (1994), S. 1185–1201

[Janusz und Riedmiller 1995] JANUSZ, B. ; RIEDMILLER, M.: Self-Learning neural control of a mobile robot. In: *Proceedings of the IEEE ICNN 95*. Perth, 1995

[Kaelbling u. a. 1996] KAELBLING, L.P. ; LITTMAN, M.L. ; MOORE, A.: Reinforcement Learning: A Survey. In: *Journal of Artificial Intelligence Research* 4 (1996), S. 237–285

[Kimura und Kobayashi 1998] KIMURA, H. ; KOBAYASHI, S.: Reinforcement Learning for Continous Action using Stochastic Gradient Ascent. In: *The 5th International Conference on Intelligent Autonomous Systems (IAS-5)*, 1998, S. 288–295

[Kögler und Obst 2004] KÖGLER, M. ; OBST, O.: Simulation League: The Next Generation. In: POLANI, D. (Hrsg.) ; BONARINI, A. (Hrsg.) ; BROWNING, B. (Hrsg.) ; YOSHIDA, K. (Hrsg.): *RoboCup2003* Bd. 3020. Berlin, Heidelberg, New York : Springer, 2004, S. 458 – 469

[Kohonen 1982] KOHONEN, T.: Self-organized formation of topologically correct feature maps. In: *Biol. Cybern.* 43 (1982), S. 59–69

[Kretchmar und Anderson 1997] KRETCHMAR, R. ; ANDERSON, C.: Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning. In: *Proceedings of the IEEE International Conference on Neural Networks*, 1997, S. 834–837

[Likas und Lagaris 1999] LIKAS, A. ; LAGARIS, I.: Training Reinforcement Neurocontrollers Using the Polytope Algorithm. In: *Neural Processing Letters* 9 (1999), Nr. 2, S. 119 – 127

[Linde u. a. 1980] LINDE, Y. ; BUZO, A. ; GRAY, R.: An Algorithm for Vector Quantizer Design. In: *IEEE Trans. Communications*, 1980, S. 84–95

[Lovejoy 1991] LOVEJOY, W.S.: A survey of algorithmic methods for partially observed Markov decision processes. In: *Annals of Operations Research* (1991), Nr. 28, S. 47–66

[MacQueen 1967] MACQUEEN, J.B.: Some Methods for classification and Analysis of Multivariate Observations. In: *roceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*. Berkeley, 1967, S. 281–297

[Malcoci u. a. 1998] MALCOCI, A. ; JACQUET, W. ; BRETON, S. ; URBAN, J.-P.: Growing Hyperplan-Based Self-Organizing Maps for Function Approximation. In: *Proceedings of the Symposium of Electronics and Telecommunications (ETc'98)*. Timisoara, Romania, 1998

[Martinetz 1993] MARTINETZ, T.: Competitive hebbian learning rule forms perfectly topology preserving maps. In: *Proceedings of ICANN'93*, Springer, 1993, S. 427–434

[Martinetz u. a. 1993] MARTINETZ, T. ; BERKOVICH, S. ; SCHULTEN, K.: "Neural-gas" network for vector quantization and its application to time-series prediction. In: *IEEE-Transactions on Neural Networks* 4 (1993), Nr. 4, S. 558–569

[Martinetz und Schulten 1991] MARTINETZ, T. ; SCHULTEN, K.: A "neural gas" network learns topologies. In: *Artificial Neural Networks* I (1991), S. 397–402

[Martinetz und Schulten 1994] MARTINETZ, T. ; SCHULTEN, K.: Topology representing networks. In: *Neural Networks* 7 (1994), Nr. 3, S. 507–522

[Michie und Chambers 1968a] MICHIE, D. ; CHAMBERS, R.A.: BOXES: An Experiment in adaptive control. In: DALE, E. (Hrsg.) ; MICHIE, D. (Hrsg.): *Machine Intelligence 2*. 1968, S. 137–152

[Michie und Chambers 1968b] MICHIE, D. ; CHAMBERS, R.A.: 'Boxes' as a model of pattern-formation. In: *Towards a Theoretical Biology* 1 (1968), S. 206–215

[Millán u. a. 2002] MILLÁN, J. ; POSENATO, D. ; DEDIEU, E.: Continuous-Action Q-Learning. In: *Machine Learning* 49 (2002), S. 247–265

[Moriarty und Mikkulainen 1996] MORIARTY, D.E. ; MIKKULAINEN, R.: Efficient Reinforcement Learning through Symbiotic Evolution. In: *Machine Learning* 22 (1996), S. 11–33

[Munos 1997] MUNOS, R.: Finite-Element methods with local triangulation refinement for continuous Reinforcement Learning problems. In: VAN SOMEREN, M. (Hrsg.) ; WIDMER, G. (Hrsg.): *European Conference on Machine Learning*, 1997

[Munos und Moore 1999] MUNOS, R. ; MOORE, A.: Variable Resolution Discretization for High-Accuracy Solutions of Optimal Control Problems. In: *Proceedings of IJCAI*, 1999, S. 1348–1355

[Munos und Moore 2002] MUNOS, R. ; MOORE, A.: Variable Resolution Discretization in Optimal Control. In: *Machine Learning* 49, Numbers 2/3 (2002), November/December, S. 291–323

[Murphy 2000] MURPHY, K.P.: A Survey of POMDP Solution Techniques / U.C. Berkeley. 2000. – Forschungsbericht

[Obst und Rollmann 2004] OBST, O. ; ROLLMANN, M.: SPARK – A Generic Simulator for Physical Multiagent Simulations. In: LINDEMANN, G. (Hrsg.) ; DENZINGER, J. (Hrsg.) ; TIMM, I. (Hrsg.) ; UNLAND, R. (Hrsg.): *Proc. MATES 2004*. Springer, September 2004 (Lecture Notes in Artificial Intelligence), S. 243–257

[Omohundro 1990] OMOHUNDRO, S.: The Delaunay triangulation and function learning / International Computer Science Institute, Berkeley. 1990 (Tr-90-001). – Forschungsbericht

[Paraskevopoulos u. a. 1999] PARASKEVOPOULOS, V. ; HEYWOOD, M.I. ; CHATWIN, C.R.: Modular SRV Reinforcement Learning Architectures for Non-Linear Control. In: *International Journal on Intelligent Control and Systems* (1999)

[Perl 2001] PERL, J.: DyCoN: Ein dynamisch gesteuertes Neuronales Netz zur Modellierung und Analyse von Prozessen im Sport. In: PERL, J. (Hrsg.): *Sport & Informatik VIII*. Köln: Strauß, 2001

[Platt 1991a] PLATT, J.C.: Learning by combining memorization and gradient descent. In: *Advances in Neural Information Processing Systems* (1991), Nr. 3, S. 714–720

[Platt 1991b] PLATT, J.C.: A resource-allocating network for function interpolation. In: *Neural Computation* (1991), Nr. 3, S. 213–225

[Polani 1995] POLANI, D.: On the choice of organization measures for self-organizing feature maps / Johannes Gutenberg-Universität, Institut für Informatik, Mainz. 1995 (95,1). – Forschungsbericht

[Prescott 1994] PRESCOTT, A.: *Explorations in Reinforcement and Model-based Learning*, University of Sheffield, PhD Thesis, 1994

[Provost u. a. 2004] PROVOST, J. ; KUIPERS, B. J. ; MIIKKULAINEN, R.: Self-Organizing Perceptual and Temporal Abstraction for Robot Reinforcement Learning. In: *AAAI-04 Workshop on Learning and Planning in Markov Processes*, 2004

[Pyeatt und Howe 1998] PYEATT, L.D. ; HOWE, A.E.: Decision Tree Function Approximation in Reinforcement Learning. Fort Collins, Colorado, 1998 (TR CS-98-112). – Forschungsbericht

[Reynolds 2000] REYNOLDS, S.I.: Adaptive Resolution Model-Free Reinforcement Learning: Decision Boundary Partitioning. In: *Proc. 17th International Conf. on Machine Learning*, Morgan Kaufmann, San Francisco, CA, 2000, S. 783–790

[Riedmiller und Janusz 1995] RIEDMILLER, M. ; JANUSZ, B.: Using Neural Reinforcement Controllers in Robotics. In: *Proc. 8th Australian Conference on Artificial Intelligence*. Canberra, 1995

[Ritter u. a. 1990] RITTER, H. ; MARTINETZ, T. ; SCHULTEN, K.: *Neuronale Netze: eine Einführung in die Neuroinformatik selbstorganisierter Netzwerke*. Bonn : Addison Wesley, 1990

[Ritter und Schulten 1986] RITTER, H. ; SCHULTEN, K.: Topology conserving mappings for learning motor tasks. In: *AIP Conference Proceedings 151 on Neural Networks for Computing*, American Institute of Physics Inc., 1986, S. 376–380

[Rummery 1995] RUMMERY, G.A.: *Problem solving with reinforcement learning*, Cambridge University, PhD Thesis, 1995

[Rummery und Niranjan 1994] RUMMERY, G.A. ; NIRANJAN, M.: On-line Q-Learning using connectionist systems / Engineering Department, Cambridge University. 1994 (CUED / F-INFENG / TR 166). – Forschungsbericht

[Santamaría u. a. 1997] SANTAMARÍA, J. ; SUTTON, R.S. ; RAM, A.: Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces. In: *Adaptive Behaviour* 6 (1997), Nr. 2, S. 163–217

[Saunders u. a. 1998] SAUNDERS, R. ; STITSON, C. ; WESTON, J. ; BOTTOU, L. ; SCHÖLKOPF, B. ; SMOLA, A.: Support Vector Machine reference manual / University of London. 1998 (CSD–TR–98–03). – Forschungsbericht

[Schaal und Atkeson 1998] SCHAAL, S. ; ATKESON, C.G.: Constructive Incremental Learning from Only Local Information. In: *Neural Computation* 10 (1998), Nr. 8, S. 2047–2084

[Schaal u. a. 2002] SCHAAL, S. ; ATKESON, C.G. ; VIJAYAKUMAR, S.: Scalable Techniques from Nonparametric Statistics for Real Time Robot Learning. In: *Applied Intelligence* 17(1) (2002), S. 49–60

[Smart und Kaelbling 2000] SMART, W.D. ; KAELBLING, L.P.: Practical Reinforcement Learning in Continuous Spaces. In: *Proc. 17th International Conf. on Machine Learning*, Morgan Kaufmann, San Francisco, CA, 2000, S. 903–910

[Smart und Kaelbling 2001] SMART, W.D. ; KAELBLING, L.P.: Reinforcement Learning for Robot Control. In: *Mobile Robots XVI (Proc. SPIE 4573)*, 2001

[Smith 2001] SMITH, A.J.: *Dynamic generalisation of Continuous Action Spaces in Reinforcement Learning: A Neurally Inspired Approach*, University of Edinburgh, PhD Thesis, 2001

[Smith 2002] SMITH, A.J.: Applications of the self-organising map to reinforcement learning. In: *Neural Networks* 15 (2002), S. 1107–1124

[Smola und Schölkopf 1998] SMOLA, A. ; SCHÖLKOPF, B.: A tutorial on support vector regression / University of London. 1998 (NC2-TR-1998-030). – Forschungsbericht

[Stephan u. a. 2003] STEPHAN, V. ; SAUPE, M. ; BISCHOFF, M. ; REINDANZ, H. ; GROSS, H.-M.: Is Reinforcement-Learning Able to Solve Real-World Challenges? In: *Proceedings of ICANN 2003*, 2003, S. 346–349

[Sutton 1988] SUTTON, R.S.: Learning to predict by the method of temporal differences. In: *Machine Learning* 3 (1988), S. 9–44

[Sutton 1996] SUTTON, R.S.: Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In: *Advances in Neural Information Processing Systems* 8 (1996), S. 1038–1044

[Sutton 1999] SUTTON, R.S.: Open Theoretical Questions in Reinforcement Learning. In: FISCHER, P. (Hrsg.) ; SIMON, H.U. (Hrsg.): *EuroCOLT '99*, 1999, S. 11–17

[Sutton und Barto 1998] SUTTON, R.S. ; BARTO, A.G.: *Reinforcement Learning. An Introduction*. MIT Press, Cambridge, Massachusetts, 1998

[Sutton u. a. 2000] SUTTON, R.S. ; MCALLESTER, D. ; SINGH, S. ; MANSOUR, Y.: Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: *Advances in Neural Information Processing Systems* 12 (2000), S. 1057–1063

[ten Hagen und Kröse 2000] TEN HAGEN, S. ; KRÖSE, B.: Q-learning for systems with continuous state and action spaces. In: *BENELEARN 2000, 10th Belgian-Dutch Conference on Machine Learning*, 2000

[Tesauro 1992] TESAURO, G.J.: Practical issues in temporal difference learning. In: *Machine Learning* 8 (1992), S. 257–277

[Tesauro 1994] TESAURO, G.J.: TD-Gammon, a self-teaching backgammon programm, achieves master-level play. In: *Neural Computation* 6 (1994), Nr. 2, S. 215–219

[Tesauro 1995] TESAURO, G.J.: Temporal difference learning and TD-Gammon. In: *Communications of the ACM* 38 (1995), S. 58–68

[Touzet 1997] TOUZET, C.: Neural Reinforcement Learning for Behavior Synthesis. In: *Robotics and Autonomous Systems* Special Issue on Learning Robot: the New Wave (1997)

[Tsitsiklis 1994] TSITSIKLIS, J.N.: Asynchronous stochastic approximation and Q-Learning. In: *Machine Learning* 16 (1994), S. 185–202

[Tsitsiklis und Van Roy 1997] TSITSIKLIS, J.N. ; VAN ROY, B.: An Analysis of Temporal-Difference Learning with Function Approximation. In: *IEEE Transactions on Automatic Control* 42 (1997), Nr. 5, S. 674–690

[Van Hulle 2000] VAN HULLE, M.: *Faithful Representations And Topographic Maps: From Distortion- to Information-Based Self-Organization*. New York : Wiley-Interscience, 2000

[Vijayakumar u. a. 2005] VIJAYAKUMAR, S. ; D'SOUZA, A. ; SCHAAL, S.: Incremental Online Learning in High Dimensions. In: *Neural Computation* (to appear) (2005)

[Vijayakumar u. a. 2002] VIJAYAKUMAR, S. ; D'SOUZA, A. ; SHIBATA, T. ; CONRADT, J. ; SCHAAL, S.: Statistical Learning for Humanoid Robots. In: *Autonomous Robot* 12(1) (2002), S. 55–69

[Vijayakumar und Schaal 1998] VIJAYAKUMAR, S. ; SCHAAL, S.: Local Adaptive Subspace Regression. In: *Neural Processing Letters* 7(3) (1998), S. 139–149

[Vijayakumar und Schaal 2000] VIJAYAKUMAR, S. ; SCHAAL, S.: Fast and Efficient Incremental Learning for High-dimensional Movement Systems. In: *Proc. International Conference on Robotics and Automation (ICRA2000)* Bd. 2, 2000, S. 1894–1899

[Watkins 1989] WATKINS, C.J.: *Learning from Delayed Rewards*, Cambridge University, PhD Thesis, 1989

[Watkins und Dayan 1992] WATKINS, C.J. ; DAYAN, P.: Q-Learning. In: *Machine Learning* 8 (1992), S. 279–292

[Wedel und Polani 1996] WEDEL, J. ; POLANI, D.: Critic-based Learning of Actions with Self-Organising Feature Maps / Johannes Gutenberg-Universität, Institut für Informatik, Mainz. 1996 (96,5). – Forschungsbericht

[Williams und Rasmussen 1995] WILLIAMS, C.K. ; RASMUSSEN, C.E.: Gaussian Processes for Regression. In: TOURETZKY, D.S. (Hrsg.) ; MOZER, M.C. (Hrsg.) ; HASSELMO, M.E. (Hrsg.): *Advances in Neural Information Processing Systems 8*, MIT Press, 1995

[Williams 1992] WILLIAMS, R.J.: Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. In: *Machine Learning* 8 (1992), S. 229–256

[Winter u. a. 2000] WINTER, M. ; METTA, G. ; SANDINI, G.: Neural-Gas for Function Approximation: a Heuristic for Minimizing the Local Estimation Error. In: *International Joint Conference on Neural Network (IJCNN 2000)*, 2000

[Wold 1975] WOLD, H.: Soft modeling by latent variables: the non-linear iterative partial least squares approach. In: *Perspectives in Probability and Statistics* (1975), S. 117–144

[Xu u. a. 2002] XU, X. ; HAN-GEN, H. ; DEWEN, H.: Efficient Reinforcement Learning Using Recursive Least-Squares Methods. In: *Journal of Artificial Intelligent Research* 16 (2002), S. 259–292

Überblick über die verwendete Notation

Im Folgenden wird die verwendete Notation kurz dargestellt. Die Aufzählung wurde nach den Kapiteln, in denen die Bezeichner das erste Mal auftreten, gegliedert. Zum Teil treten in den verschiedenen Kapiteln dieselben Bezeichner in unterschiedlichen Bedeutungen auf. Das ist leider nicht immer zu vermeiden, wenn man bestrebt ist, die in der Literatur üblichen Bezeichner zu verwenden. Aus dem Kontext geht aber in jedem Fall hervor, welche Bedeutung gemeint ist.

Reinforcement Learning (Kapitel 2)

t	diskreter Zeitschritt
s	(diskrete) Zustände
S	Menge (diskreter) Zustände s
a	Aktion
$A(s)$	Menge möglicher Aktionen a für einen Zustand s
π	Politik eines Agenten
$\Pr(\cdot\|\cdot)$	bedingte Wahrscheinlichkeit
r	Belohnung
$P^a_{ss'}$	Übergangswahrscheinlichkeit für einen Zustand s zu einem Zustand s' unter der Aktion a
$R^a_{ss'}$	Erwartungswert für die Belohnung beim Übergang vom Zustand s zum Zustand s' unter der Aktion a
T	Zeitpunkt beim Erreichen eines Terminalzustandes
R_t	Return, der auf den Zeitschritt t folgt
V^π	Zustands-Wertefunktion zur Politik π
Q^π	Aktions-Wertefunktion zur Politik π
E	Erwartungswert einer Zufallsvariablen
E_π	von der Politik π abhängiger Erwartungswert einer Zufallsvariablen
γ	Diskontierungsfaktor
V^*	optimale Zustands-Wertefunktion
Q^*	optimale Aktions-Wertefunktion
π^*	optimale Politik
$V(s)$	geschätzter Wert eines Zustandes s
$Q(s,a)$	geschätzter Wert einer Aktion a für einen Zustand s
α	Lernrate
ε	Explorationsparameter

x	Eingabevektor / kontinuierlicher Zustand
$\tilde{\cdot}$	Approximation
(x, y)	Ein-Ausgabe-Paar mit Eingabevektor x und Zielwert y
θ	Parametervektor bei Funktionsapproximation
θ^*	optimaler Parametervektor bei Funktionsapproximation
$E(\cdot)$	Fehlerfunktion
$\nabla_x f$	Gradient einer Funktion f an der Stelle x
$\Delta\cdot$	Differenz einer Größe zwischen zwei Zeitschritten
$N(\mu, \sigma)$	Normalverteilung mit Mittelwert μ und Standardabweichung σ
g	Wahrscheinlichkeitsfunktion / Dichte
u	stochastische Einheit bei REINFORCE-Algorithmen
Θ	Matrix der Gewichtsvektoren
$\overline{r}$	Schätzung der durchschnittlich zu erwartenden Belohnung
$\mu(x)$	von einer Situation x abhängiger Mittelwert der Normalverteilung
$\sigma(x)$	von einer Situation x abhängige Standardabweichung der Normalverteilung
e	„characteristic eligibilty“ bei REINFORCE-Algorithmen
b	„reinforcement baseline“ bei REINFORCE-Algorithmen
δ_t	„Temporal-Difference“-Fehler zum Zeitpunkt t

Selbstorganisierende Karten (Kapitel 3)

c_i	Neuron einer Selbstorganisierenden Karte
A	Menge von Neuronen
w_i	Position eines Neurons im Eingaberaum
$e_{i,j}$	Kante zwischen den Neuronen c_i und c_j
E	Menge von Kanten
$c_{i,j}$	j-tes Nachbarneuron von Neuron c_i
$c_{b(x)}$	Gewinnerneuron zum Eingabevektor x
$d(x, y)$	Metrik
$h(d, t)$	Nachbarschafts- oder Aktivierungsfunktion
err_i	Fehlervariable eines Neurons
$age_{i,j}$	Alter einer Kante
N	Anzahl von Neuronen
δ	Reduktionsrate für Fehlervariablen
$D_i(x)$	Aktivierungsfunktion bei Funktionsapproximation
v_i	in einem Neuron gespeicherter Wert
a_i	Koeffizient bei der linearen Interpolation
μ	Regularisierungsparameter
l_i	Differenzvektoren
L	Matrix von Differenzvektoren

LWIGNG (Kapitel 4)

$\delta(i)$	Index des zum Interpolationskoeffizienten a_i gehörenden Neurons
$\widetilde{M}_{i,j}(x)$	Wert eines lokal linearen Modells an der Stelle x
$m_{i,j}(x)$	Gewicht eines lokal linearen Modells für die Stelle x
$\lambda_{i,j}$	Reichweitenparameter an den Neuronen und den Kanten
$\widetilde{F}$	endgültige Approximation
β_{err}	Gewicht des aktuellen Fehlers

Q-LWIGNG (Kapitel 5)

$\hat{w}_i$	Summe der Positionsveränderungen beim Lernen aus Trajektorien
z_i	Zähler für die Anzahl der Positionsveränderungen beim Lernen aus Trajektorien
Q_i	Vektor zur Approximation der Q-Werte in den Neuronen

ReinforceLWIGNG (Kapitel 6)

v_i	Wert für die Approximation der Zustands-Wertefunktion V in den Neuronen
μ_i	Wert für die Approximation des Mittelwertes μ in den Neuronen
σ_i	Wert für die Approximation der Standardabweichung σ in den Neuronen
y^V	Zielwert für die Anpassung der Zustands-Wertefunktion V
y^μ	Zielwert für die Anpassung des Mittelwertes μ der Normalverteilung
y^σ	Zielwert für die Anpassung der Standardabweichung σ der Normalverteilung

Lebenslauf

8.11.1972	geboren in Wiesbaden
1979–1983	Friedrich von Schiller-Grundschule, Wiesbaden
1983–1992	Dilthey-Gymnasium, Wiesbaden
1992–1993	Zivildienst bei der ev. Ringkirchengemeinde, Wiesbaden
1993–2000	Studium Mathematik und Sozialkunde für das Lehramt an Gymnasien, Johannes Gutenberg-Universität Mainz
2000	1. Staatsexamen für das Lehramt an Gymnasien Examensarbeit in Informatik über „Agentenbasierte Simulation der Entstehung von Eigentumsnormen“
August 2000 bis Juli 2005	wissenschaftlicher Mitarbeiter am Institut für Informatik am Fachbereich Mathematik und Informatik der Johannes Gutenberg-Universität Mainz
Juli 2001 bis Juni 2005	wissenschaftlicher Mitarbeiter im Projekt „Dynamisch-adaptive Situationserkennung und Lernverfahren zum Erwerb individueller und kooperativer Fähigkeiten in Multi-Agentensystemen“ im Rahmen des DFG-Schwerpunktprogramms „Kooperierende Teams mobiler Roboter in dynamischen Umgebungen“