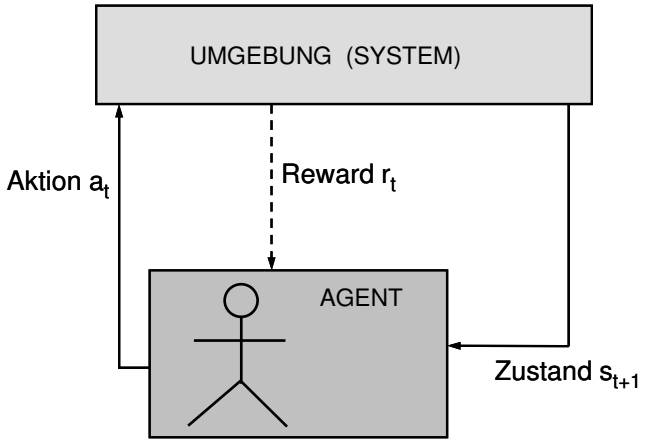
VII.A REINFORCEMENT LEARNING

1. Einleitung

Reinforcement Learning ist durch Lernprinzipien inspiriert, die man bei Menschen und Tieren beobachtet. Die Funktionsweise von Reinforcement Learning ist im folgenden Diagramm illustriert.



Ein Agent (Mensch, Roboter, Regelungssystem, Computerprogramm, etc.) beobachtet zu jedem (diskreten) Zeitpunkt t den Zustand s_t seiner Umgebung (oder eines Systems) und wählt dann entsprechend seiner gegenwärtigen Strategie (Policy) eine Aktion a_t, die seine Umgebung in einen neuen Zustand s_{t+1} überführt. Gleichzeitig erhält der Agent eine Bewertung r_t (Reward) seiner Aktion. Ein lernender Agent benutzt die Informationen über die induzierten Zustandsänderungen und die erhaltenen Rewards, um seine Aktions-Strategie zu optimieren. Beim Reinforcement Learning wird in den meisten Fällen die abdiskontierte Summe der zukünftigen Rewards als Mass für die Güte der Aktions-Strategie gewählt.

Definitionen und Notation:

- Zustand der Umgebung (oder des Systems) zur Zeit t: $s_t \in S$
- Mögliche Aktionen: $a_t \in A(s_t)$
- Übergangsfunktion der Systemzustände (ev. probabilistisch): $s_{t+1} = f(s_t, a_t)$
- Reward-Funktion (ev. probabilistisch): $r_t = \rho(s_t, a_t)$
- Zielfunktion für die Bewertung der Aktionsstrategie (= abdiskontierte Summe der zukünftigen Rewards): $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}, \quad 0 < \gamma \leq 1,$

  (Der Abdiskontierungsfaktor γ definiert einen "Vorausschau"-Horizont.)
- Aktionsstrategie π (= "Policy"): $a_t = \pi(s_t)$

  (π definiert, welche Aktion der Agent ausführt, wenn sich das System im Zustand s_t befindet.)

- "Value"-Funktion für eine gegebene Policy π: $V^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t r_t$,

  (= abdiskontierte Summe der zukünftigen Rewards wenn der Agent bei t = 0 im Zustand $s_0 = s$ startet und dann der Policy π folgt)

- Q-Funktion für eine gegebene Policy π: $Q^{\pi}(s,a) = \sum_{t=0}^{\infty} \gamma^t r_t$,

  (=abdiskontierte Summe der zukünftigen Rewards wenn der Agent bei t = 0 im Zustand $s_0 = s$ mit der Aktion $a_0 = a$ startet und dann der Policy π folgt)

Wenn die Übergangsfunktion $f(s_t, a_t)$ und/oder die Reward-Funktion $\rho(s_t, a_t)$ probabilistisch sind, werden die Ausdrücke für $V^{\pi}(s)$ and $Q^{\pi}(s,a)$ durch die entsprechenden Mittelwerte ersetzt.

$V^{\pi}(s)$ und $Q^{\pi}(s,a)$ erfüllen rekursive Relationen, die sogenannten Bellman Gleichungen:

$$V^{\pi}(s) = \rho(s, a) + \gamma V^{\pi}(s') , \quad \text{wobei } a = \pi(s) \text{ und } s' = f(s, a)$$

$$Q^{\pi}(s,a) = \rho(s, a) + \gamma Q^{\pi}(s',a') , \quad \text{wobei } s' = f(s, a) \text{ und } a' = \pi(s')$$

(Wenn sich $f(s, a)$ and $\rho(s, a)$ auf probabilistische Funktionen beziehen, werden die rechten Seiten der Bellman-Gleichungen durch entsprechende Mittelwerte ersetzt.)

2. Dynamic Programming

Das Ziel von "Dynamic Programming" ist die Bestimmung der optimalen Policy π^*, d.h. der Policy welche $V^{\pi}(s)$ maximiert,

$$V^*(s) = \max_{\pi} V^{\pi}(s),$$

und es kann gezeigt werden, dass π^* dann für alle Startwerte s optimal ist.

Für das on-line Lernen ist Q(s,a), oft besser geeignet als V(s) um die optimale Policy π^* zu finden. Die optimale Q-Funktion, Q* ist definiert als

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a),$$

und die optimale Policy π^* als die "greedy" Policy bezüglich Q*,

$$\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s,a).$$

$Q^*(s,a)$ erfüllt die Bellman'sche Optimalitäts-Gleichung (Bellman Optimality Equation):

$$Q^*(s,a) = \rho(s, a) + \gamma \max_{a' \in A(s')} Q^*(s',a') , \quad s' = f(s, a).$$

Für Zustands- und Aktionsräume, die diskret und endlich sind, wird die Bellman'sche Optimalitäts-Gleichung zu einem System von Gleichungen, das eine eindeutige Lösung hat und mit verschiedenen Methoden gelöst werden kann, vorausgesetzt dass die Dynamik der Umgebung (des Systems) und die Reward-Funktion, d.h.

$$s_{t+1} = f(s_t, a_t) \quad \text{und} \quad r_t = \rho(s_t, a_t),$$

vollständig bekannt sind.

3. Temporal-Difference Learning (TD-Learning)

In Wirklichkeit ist die Reaktion der Umgebung auf Aktionen oft nicht bekannt. Dann müssen wir die optimale Policy on-line durch Erfahrung "lernen", z.B. indem wir eine Explorations-Strategie verwenden und die entsprechenden Rewards beobachten.

Die am häufigsten verwendeten Explorations-Strategien sind

"ε-Greedy Exploration":

$$a(s_t) = \begin{cases} \arg\max_{a'} Q(s_t, a') & \text{with prob } 1-\varepsilon \\ a' \text{ random} \in A(s_t) & \text{with prob } \varepsilon \end{cases}$$

und

"Boltzmann Exploration":

$$\text{Prob}\left(a(s_t) = a\right) = \frac{e^{\beta Q(s_t,a)}}{\sum_{a'} e^{\beta Q(s_t,a')}},$$

wobei ε and β geeignet gewählte Parameter sind. Als Option können wir auch annehmen, dass während des Lernprozesses ε abnimmt, bzw. β zunimmt, z.B. gemäss

$$\varepsilon_t = \kappa \varepsilon_{t-1} \quad \text{oder} \quad \beta_t = \beta_{t-1} / \kappa \quad (\kappa < 1)$$

Die zentrale Idee beim on-line Reinforcement Learning ist das Konzept des TD-Learning (Temporal-Difference Learning). Dieses bezieht sich auf eine iterative Anpassung der V- oder Q-Werte um einen Betrag der proportional zum entsprechenden TD-Fehler (Temporal Difference Error) ist.

Der TD-Fehler ist die Differenz zwischen der rechten und linken Seite der entsprechenden Bellman-Gleichung, z.B.

$$\delta Q_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t),$$

mit $s_{t+1} = f(s_t, a_t)$, und wobei $a_t = a(s_t)$ und $a_{t+1} = a(s_{t+1})$ gemäss der verwendeten Explorations-Strategie.gewählt werden.

4. Tabellen-Basierte TD-Lernalgorithmen

Wenn die Anzahl Zustände und die Anzahl möglicher Aktionen beide endlich sind, basiert man die Implementierung von TD-Lernalgorithmen üblicherweise auf der iterativen Anpassung einer entsprechenden Tabelle von Q(s,a)-Werten. Durch die Verwendung geeigneter Diskretisierungs-Verfahren können tabellen-basierte Implementierungen im Prinzip auch auf Probleme mit kontinuierlichen Zustands- und Aktions-Räumen angewendet werde.

4.1 SARSA und Q-Learning

Die bekanntesten TD-Lernalgorithmen sind SARSA und Q-Learning.

SARSA-Lernalgorithmus:

Initialisierung $Q(s,a) = 0 \quad \forall s, \forall a$

Zu jedem Zeitpunkt t beobachtet der Agent den Zustand s_t des Systems und wählt eine Aktion $a_t = a(s_t)$ entsprechend der verwendeten Explorations-Strategie (z.B. ε-Greedy).

Dann beobachtet der Agent den Reward r_t und den neuen Zustand s_{t+1}, und ändert den Wert von $Q(s_t,a_t)$ gemäss

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right],$$

wobei α die Lernrate bezeichnet.

Da bei der Schätzung des TD-Fehlers die Aktion $a_{t+1} = a(s_{t+1})$ gemäss der auf den aktuellen Q-Werten basierenden Explorations-Strategie gewählt wird, bezeichnet man SARSA-Learning als einen "On-Policy TD-Algorithmus".

Q-Learning Algorithmus:

Hier bezieht sich der TD-Fehler auf die Bellman'sche Optimalitäts-Gleichung, d.h. $Q(s_t,a_t)$ wird wie folgt angepasst:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_{a' \in A(s')} Q(s_{t+1},a') - Q(s_t,a_t)\right].$$

Beim Q-Learning wird die Aktion $a_{t+1} = a(s_{t+1})$ gemäss $a_{t+1} = \arg\max_{a'} Q(s_{t+1},a')$ gewählt, und Q-Learning wird daher als "Off-Policy TD-Algorithmus" bezeichnet.

Man kann zeigen, dass Q-Learning immer gegen die optimale Q-Funktion $Q^*(s,a)$ konvergiert, falls die Dynamik des Systems Markov'sch und stationär ist.

Die Q-Werte, die man mit SARSA findet, stimmen nicht notwendigerweise mit den optimalen Q-Werten überein. SARSA-Laerning konvergiert deshalb nicht immer gegen die optimale Policy, findet aber in der Regel ebenfalls sehr gute Aktions-Strategien.

4.2 Eligibility Traces

In vielen Fällen, z.B. bei Problemen mit verzögerter Bewertung, kann der Lernprozess beschleunigt und verbessert werden, wenn bei jedem Zeitschritt nicht nur der Q-Werte für das aktuelle Zustands-Aktions-Paar angepasst wird, sondern auch die Q-Werte von Zustands-Aktions-Paaren, die in letzter Zeit vorgekommen sind.

Ein entsprechender Mechanismus kann in geeigneter Weise mit Hilfe von sogenannten "Eligibility Traces" $e_t(s,a)$ implementiert werden. Diese werden zu Beginn des Lernprozesses alle gleich Null gesetzt und dann wie folgt aktualisiert,

$$e_t(s,a) = \begin{cases} \gamma\lambda\, e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda\, e_{t-1}(s,a) & \text{otherwise} \end{cases} \qquad \forall\, s \text{ and } a\,,$$

wobei λ die Zerfallsrate der Eligibility Traces bezeichnet ($0 \le \lambda < 1$).

Eligibility Traces zeigen den Grad an, mit welchem ein Zustands-Aktions-Paar berechtigt ist, zum Lernprozess beizutragen, und können mit fast allen TD-Lernalgorithmen kombiniert werden, z.B. auch mit SARSA oder Q-Learning.

Die Kombination des oben beschriebenen Eligibility Trace Mechanismus mit SARSA-Learning nennt man zum Beispiel SARSA(λ)-Learning.

SARSA(λ)-Lernalgorithmus:

$$\delta Q_t \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$
$$e_t(s_t, a_t) \leftarrow e_t(s_t, a_t) + 1$$
$$\forall \text{ s and a}:$$
$$Q(s,a) \leftarrow Q(s,a) + \alpha\, \delta Q_t\, e_t(s,a)$$
$$e_t(s,a) \leftarrow \gamma\lambda\, e_t(s,a)$$

4.3 Actor-Critic Methoden

Actor-Critic Methoden sind TD-Lernverfahren, die für die Darstellung der Aktions-Strategie Policy) und der Value-Funktion separate Strukturen verwenden. Die Policy-Struktur nennt man "Actor", weil sie zur Wahl der Aktionen verwendet wird, und die Struktur, welche die Value-Funktion abschätzt, ist der Kritiker ("Critic").

Um die Aktionen zu kritisieren, die der Actor auswählt, benutzt der Kritiker den TD-Fehler δV_t ,

$$\delta V_t = r_t + \gamma V(s_{t+1}) - V(s_t)\ ,$$

und aktualisiert dann seine Schätzung der Value-Funktion wie folgt:

$$V(s_t) \leftarrow V(s_t) + \alpha_V \left[r_t + \gamma V(s_{t+1}) - V(s_t) \right] .$$

Der Actor wird üblicherweise durch Wahrscheinlichkeiten spezifiziert, mit denen er eine Aktion a wählt, wenn sich die Umgebung im Zustand s befindet, z.B.

$$\mathrm{Prob}\left(a(s_t) = a\right) = \frac{e^{\beta\, p(s_t,a)}}{\sum_{a'} e^{\beta\, p(s_t,a')}} .$$

Die Grössen $p(s,a)$ sind Policy-Parameter, die gemäss dem durch den Kritiker mitgeteilten TD-Fehler angepasst werden:

$$p(s_t,a_t) \leftarrow p(s_t,a_t) + \alpha_p \, \delta V_t \, .$$

Wie andere TD-Lernalgorithmen, können auch Actor-Critic Methoden mit einem Eligibility-Trace-Mechanismus kombiniert werden.

5. Probleme mit kontinuierlichen Zustands- und Aktionsräumen

Tabellen-basierte Implementierungen von TD-Lernalgorithmen eignen sich nur für Probleme mit einer relativ kleinen Anzahl von Zuständen und möglichen Aktionen. Um eine Explosion der Anzahl von Q-Werten bei grossen oder sogar kontinuierlichen (s, a)-Räumen zu vermeiden, muss die Funktion Q(s, a) approximiert werden, z.B. durch eine parametrisierte Funktion

$$Q(s,a\,;\theta),$$

wobei sich s and a auf die kontinuierlichen (eventuell multi-dimensionalen) Zustands- und Aktionsvariablen beziehen und θ auf einen endlichen Satz von Parametern.

Die Approximation von Q(s, a) wird oft durch ein neuronales Netzwerk mit Output $Q(s,a\,;W)$ realisiert, wobei $W = \{W_{ij}\}$ den Satz von Verbindungsgewichten bezeichnet.

Wenn nur der s-Raum kontinuierlich und die Zahl der möglichen Aktionen klein genug ist, kann es eventuell effizienter sein, für jede Aktion a_k ein separates neuronales Netzwerk zu verwenden.

Im Fall einer parametrisierten Q-Funktion konvergiert Q-Learning im Allgemeinen natürlich nicht gegen die optimale Lösung. In der Praxis scheinen solche Verfahren aber oft sehr gut zu funktionieren.

6. TD-Learning mit parametrisierten Value-Funktionen

6.1 SARSA und Q-Learning mit neuronalen Netzwerken

Die Q-Funktion wird durch ein neuronales Netzwerk mit Input (s, a) und Output Q(s, a; W) approximiert ($W = \{W_{ij}\}$ = Verbindungsgewichte). Während des Lernprozesses werden die Gewichte bei jedem Zeitschritt angepasst, und zwar entsprechend einem Gradientenverfahren ("error backpropapgation") in Bezug auf den jeweiligen TD-Fehler.

<u>NN-SARSA-Lernalgorithmus</u>:

$$\Delta W_{ij}(t) = \alpha \, \delta Q_t \, \frac{\partial Q(s_t,a_t\,;W)}{\partial W_{ij}},$$

wobei

$$\delta Q_t = r_t + \gamma Q(s_{t+1},a_{t+1}\,;W) - Q(s_t,a_t\,;W),$$

und wobei $a_t = a(s_t)$ und $a_{t+1} = a(s_{t+1})$ entsprechend der verwendeten Explorations-Strategie gewählt werden.

NN-Q-Learning Algorithmus:

$$\Delta W_{ij}(t) = \alpha\, \delta Q_t\, \frac{\partial Q(s_t, a_t; W)}{\partial W_{ij}},$$

wobei

$$\delta Q_t = r_t + \gamma \max_{a' \in A(s')} Q(s_{t+1}, a'; W) - Q(s_t, a_t; W).$$

Bemerkungen:

- Wenn der Aktions-Raum kontinuierlich ist, kann der "max"-Term beim Q-Learning im Allgemeinen nur approximativ (numerisch) ausgewertet werden.
- Auch TD-Lernverfahren, die auf neuronalen Netzwerken basieren, können mit einem "Eligibility Trace"-Mechanismus kombiniert werden. Dabei wird jedem Gewicht W_{ij} eine Eligibility Trace $e_t(W_{ij})$ zugeordnet.

6.2 Der Online Lernalgorithmus von Baxter et al

Der von Baxter et al [2] vorgeschlagene Lernalgorithmus bezieht sich auf ein parametrisiertes Actor-Critic System und kombiniert ein direktes Gradientenverfahren mit einem Eligibility-Trace-Mechanismus:

Der Actor ist durch parametrisierte Wahrscheinlichkeiten $P(a_k \mid s; \theta)$ charakterisiert, mit denen er eine Aktion a_k (k = 1, …, n) wählt, wenn sich die Umgebung im Zustand s befindet. Jedem Modellparameter $\theta_i \in \theta$ (i = 1, …, N) wird eine Eligibility Trace z_i zugeordnet.

Die Eligibility Traces und die Modelparameter werden wie folgt aktualisiert:

$$z_i(t+1) = \lambda\, z_i(t) + \frac{\partial P(a_t \mid s_t; \theta) \,/\, \partial \theta_i}{P(a_t \mid s_t; \theta)} = \lambda\, z_i(t) + \frac{\partial \ln P(a_t \mid s_t; \theta)}{\partial \theta_i},$$

$$\theta_i(t+1) = \theta_i(t) + \alpha_\theta\, r_t\, z_i(t+1),$$

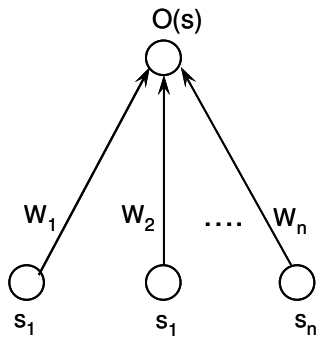
wobei $r_t = \rho(s_t, a_t)$ die Reward-Funktion bezeichnet und a_t gemäss den Aktionswahrscheinlichkeiten $P(a_k \mid s; \theta)$ gewählt wird. λ ist die Zerfallsrate der z_i und α_θ eine Lernrate.

Bei diesem Actor-Critic-Verfahren wird der Kritiker also einfach durch den instantanen Reward r_t dargestellt, d.h. der Actor versucht, den mittleren Reward zu maximieren (und nicht eine abdiskontierte Summe der zukünftigen Rewards). Zudem ist der Algorithmus nur für eine endliche Zahl von möglichen Aktionen formuliert.

Es ist aber möglich, den Algorithmus so zu verallgemeinern, dass er auch auf Probleme mit kontinuierlichen Aktionsräumen und abdiskontierten Rewards angewendet werden kann.

6.3 Das $A_{R\text{-}P}$ - Lernverfahren

Das $A_{R\text{-}P}$ - Lernverfahren für stochastische Neuronen (ohne Penalty-Term) kann als Spezialfall des Baxter-Algorithmus gedeutet werden:



$$O(s) = \begin{cases} 1 & \text{mit Wahrsch. } P(s) \\ 0 & \text{mit Wahrsch. } 1-P(s) \end{cases}$$

$$P(s) = \frac{1}{1+\exp(-\sum_{k=1}^{n} W_k s_k)}$$

→ 2 Aktionen: $a(s) = \begin{cases} a_1 & \text{falls } O(s) = 1 \\ a_2 & \text{falls } O(s) = 0 \end{cases}$

→ $P(a_1 \mid s) = P(s), \quad P(a_2 \mid s) = 1 - P(s)$

Baxter-Algorithmus mit $\lambda = 0$:

→ $\Delta W_k = W_k(t+1) - W_k(t) = \alpha r(t)\left[O(s) - P(s)\right]s_k$

→ Identisch mit $A_{R\text{-}P}$ – Regel für Gewichtsänderungen!

7. Beispiele

7.1 Das iterierte Prisoner's Dilemma

Bei diesem berühmten Spiel wählen zwei Spieler (A und B) simultan und unabhängig voneinander eine von zwei möglichen Aktionen: C ("Cooperate") oder D ("Defect"). Die entsprechende Reward-Matrix (r^A, r^B) ist wie folgt definiert:

		Player B	
		C	D
Player A	C	(0.3, 0.3)	(0, 0.5)
	D	(0.5, 0)	(0.1, 0.1)

Eine einfache, aber sehr effektive Strategie für dieses Spiel ist "Tit for Tat" (TfT). Diese bekannte Strategie hängt nur von der Aktion des Gegners ab, die dieser im letzten Spiel gewählt hat:

t–1			t
Agent	TfT		TfT
C	C		C
C	D	⟶	C
D	C		D
D	D		D

In den folgenden Tests spielt ein Agent wiederholte Spiele gegen TfT. Zu jedem Zeitpunkt t (d.h. bei jedem Spiel) kann er zwischen den beiden Aktionen $a_t = C$ oder $a_t = D$ wählen, und seine Zustandsinformation bezieht sich auf die Aktionen der beiden Spieler im letzten Spiel:

$$s_t = (a_{t-1}^{Agent}, a_{t-1}^{TfT})$$

Der Agent benutzt Reinforcement Learning, um die Summe seiner abdiskontierten Rewards, d.h. seine Value-Funktion zu maximieren

$$V^{Agent} = \sum_{t=0}^{\infty} \gamma^t r_t^{Agent} = \max .$$

Die optimale Strategie des Agenten hängt von γ ab:

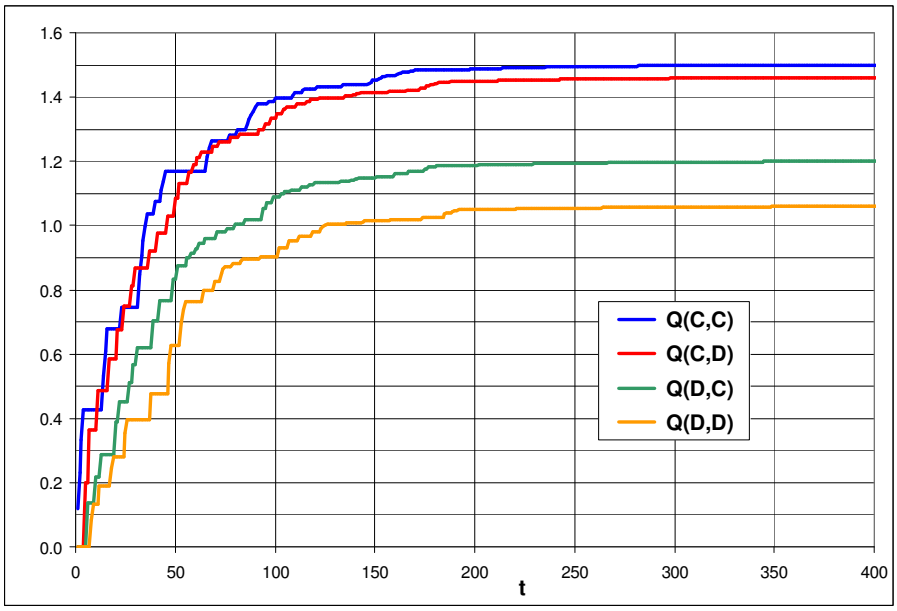
π_A: immer C $\rightarrow$ $V^A = 0.3 / (1-\gamma)$ $\rightarrow$ optimal für $\gamma \geq \frac{2}{3}$

π_B: alternierend CDCD… $\rightarrow$ $V^B = 0.5 / (1-\gamma^2)$ $\rightarrow$ optimal für $\frac{1}{4} \leq \gamma < \frac{2}{3}$

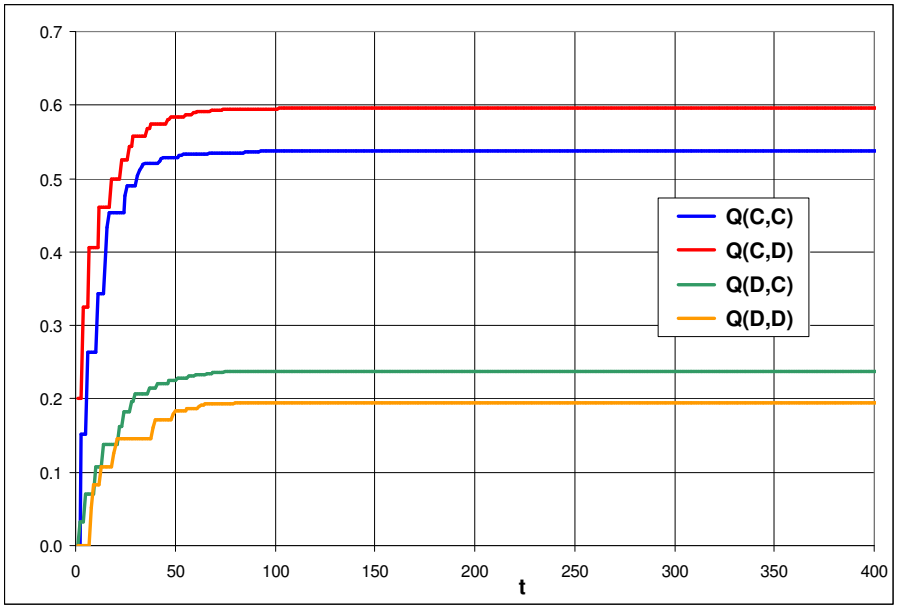
π_C: immer D $\rightarrow$ $V^C = 0.5 + 0.1 \cdot \gamma / (1-\gamma)$ $\rightarrow$ optimal für $\gamma < \frac{1}{4}$

Im Prinzip haben wir eigentlich acht Q-Werte: $Q(s_t ; a_t^{Agent}) = Q(a_{t-1}^{Agent}, a_{t-1}^{TfT} ; a_t^{Agent})$.
Da aber a_t^{TfT} nur von a_{t-1}^{Agent}, abhängt; gibt es nur vier verschiedene Q-Werte,
$Q(a_{t-1}^{Agent} ; a_t^{Agent})$, z.B. $Q(C,C;C) = Q(C,D;C) \equiv Q(C;C)$, etc.

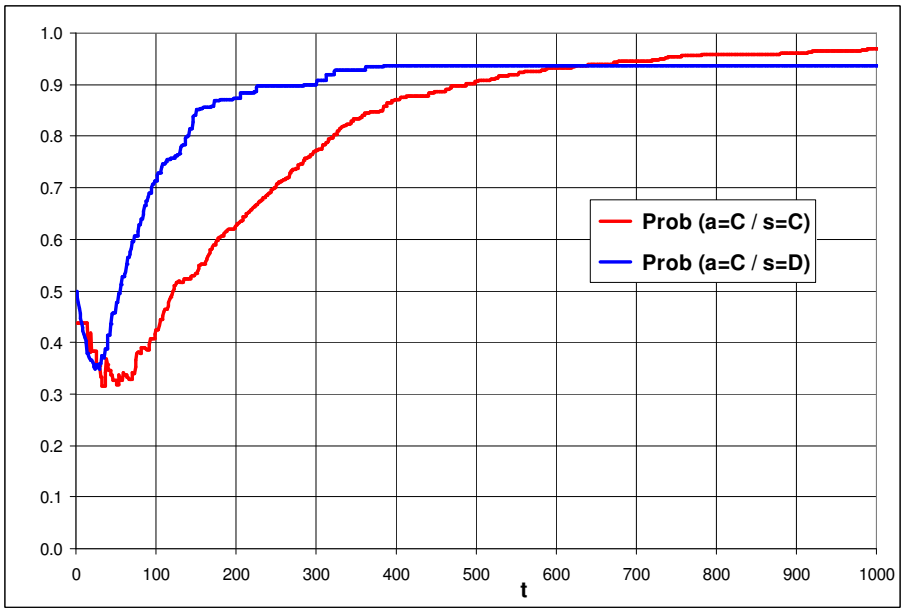
Einige Resultate dieser Tests sind in den Figuren 1 bis 4 zusammengefasst..



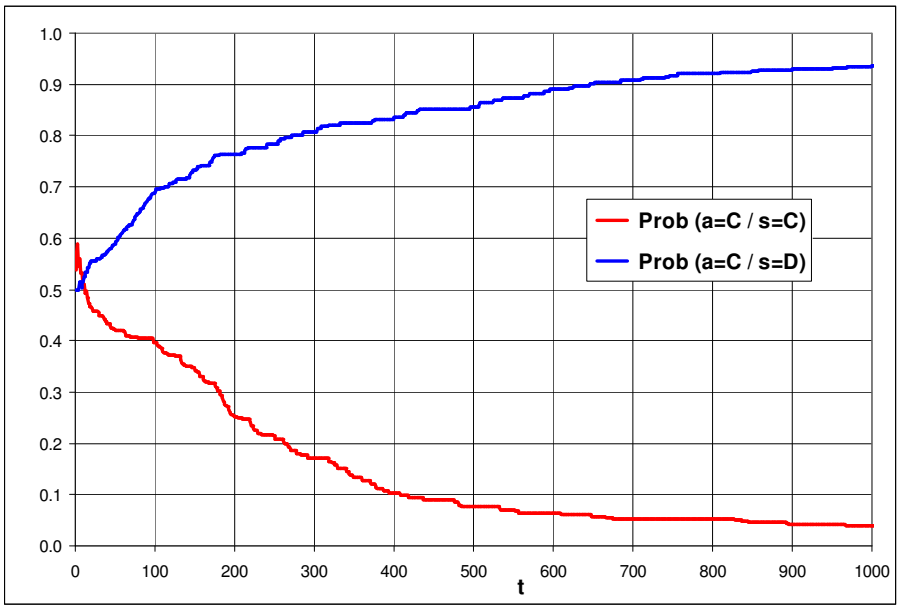
Figur 1: Iteriertes Prisoner's Dilemma, Agent gegen Tit for Tat ($\gamma = 0.8$)
Tabellen-Basiertes Q-Learning mit Boltzmann-Exploration
($\alpha = 0.4$, $\beta_0 = 0.5$, $\kappa = 0.997$)



Figur 2: Iteriertes Prisoner's Dilemma, Agent gegen Tit for Tat ($\gamma = 0.4$)
Tabellen-Basiertes Q-Learning mit Boltzmann-Exploration
($\alpha = 0.4$, $\beta_0 = 0.5$, $\kappa = 0.997$)



Figur 3: Iteriertes Prisoner's Dilemma, Agent gegen Tit for Tat ($\gamma = 0.8$)
Actor-Critic Learning (verallgemeinerter Baxter Algorithmus)
($\alpha_\theta = 1.0$, $\alpha_V = 0.25$, $\lambda = 0$)



Figur 4: Iteriertes Prisoner's Dilemma, Agent gegen Tit for Tat ($\gamma = 0.4$)
Actor-Critic Learning (verallgemeinerter Baxter Algorithmus)
($\alpha_\theta = 1.0$, $\alpha_V = 0.25$, $\lambda = 0$)

7.2 Eindimensionales Golfspiel

Als einfaches und illustratives Beispiel für ein Problem mit kontinuierlichen Zustands- und Aktionsräumen betrachten wir das folgende "eindimensionale Golfproblem":

- Eindimensionaler Golfplatz: $-5 \le s \le +5$ (z.B. -500 m bis +500 m)

- Zustand s = Lage des Golfballs (falls $|s| > 5$, dann Ball zurücklegen auf ± 5)

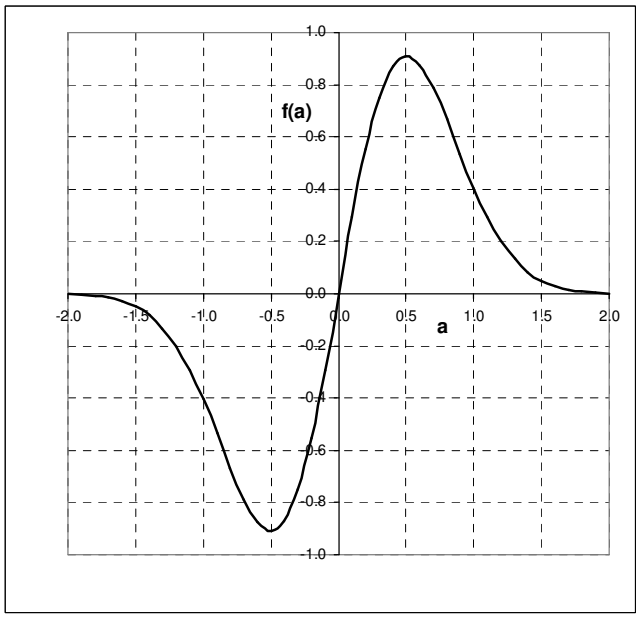
- Position des Lochs: $s_{Hole} = 0$

- Falls $|s| < 0.1$ dann Stopp (Ende der Episode, d.h. des Spiels)

- Aktion a: $|a|$ = Stärke des Schlages, sign(a) = Richtung des Schlages, $-\infty < a < +\infty$

- Wirkung der Aktion a auf den Zustand s:

$$s_{t+1} = s_t + f(a_t), \qquad f(a) = 3a\, e^{-2a^2}$$



Figur 5: Wirkung der Aktion a auf den Zustand s: $s_{t+1} = s_t + f(a_t)$

Aus Figur 5 ist ersichtlich, dass $|a| = 0.5$ den längsten Schlag bewirkt (etwa 90 m). Die Länge des Schlages nimmt sowohl mit abnehmender als auch zunehmender Schlagstärke $|a|$ ab.
("Wenn man zu stark schlägt, trifft man hauptsächlich in den Boden.")

Ein Agent (Golfspieler) soll die Aufgabe, den Ball aus jeder Distanz mit möglichst wenigen Schlägen ins Loch zu befördern, mit Hilfe des verallgemeinerten Baxter-Algorithmus lernen.
Die Modelle für den Actor (Agenten) und den Kritiker werden wie folgt gewählt:

Agenten-Modell:

Der lernende Agent ist durch eine Wahrscheinlichkeitsdichte

$$p(a \,|\, \mu(s);\beta(s)) = \frac{\beta e^{-\beta(a-\mu)}}{(1+e^{-\beta(a-\mu)})^2}$$

charakterisiert, gemäss der er im Zustand s eine Aktion a wählt, und in den folgenden Tests betrachten wir nur sehr einfache Modelle für die Parameter β and μ:

- $\beta = \beta_0$ (unabhängig von s)
- $\mu(s) = u_1 \cdot s$ ("linearer Spieler"), $\mu(s) = u_0 \dfrac{1 - e^{-u_1 \cdot s}}{1 + e^{-u_1 \cdot s}}$ ("sigmoider Spieler")

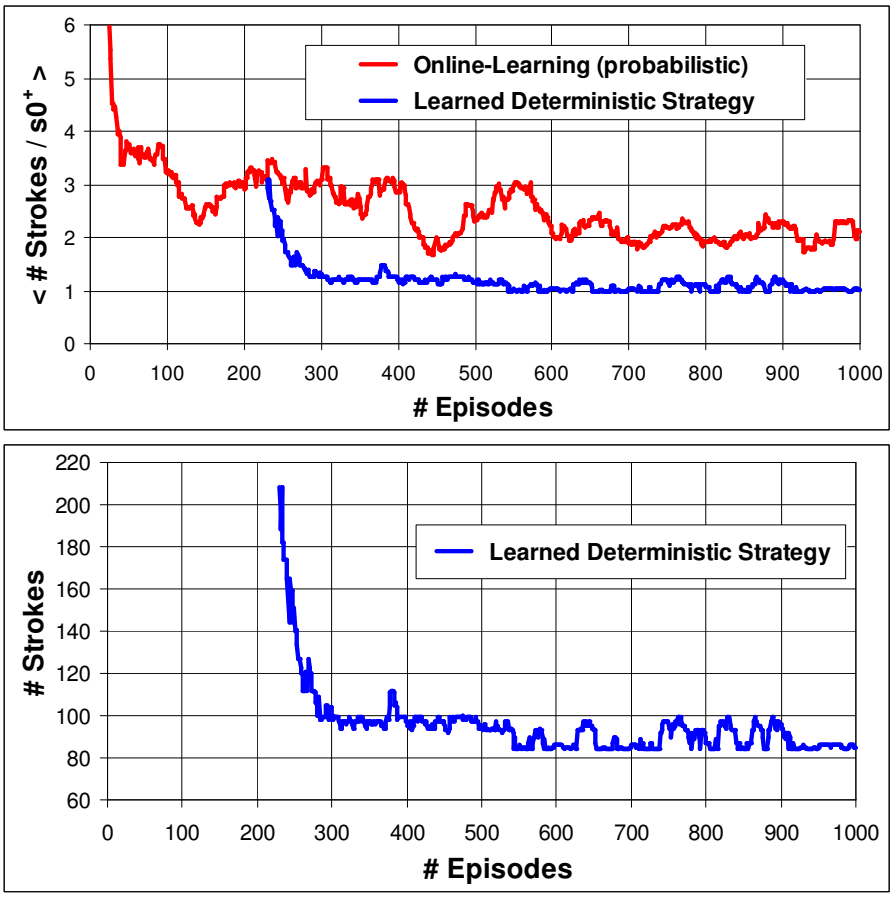
Kritiker-Modell:

Der Kritiker macht eine Abschätzung des aktuellen TD-Fehlers δV_t, und die entsprechende Value-Funktion V(s) wird durch eine einfache 2-Parameter-Funktion approximiert,

$$V(s) = v_0 + v_1 \cdot |s| \ .$$

Weiter verwenden wir einen episodischen Lernalgorithmus mit $\gamma = 1$ (keine Abdiskontierung) und $r_t = -1$ (jeder Schlag erhält einen "Reward" von -1).

Die entsprechende Evolution der Spielstrategie ist in Figur 6 dargestellt. Daraus ist ersichtlich, dass ein sigmoider Spieler nach etwa 500 Spielen eine nahezu optimale Strategie gelernt hat.



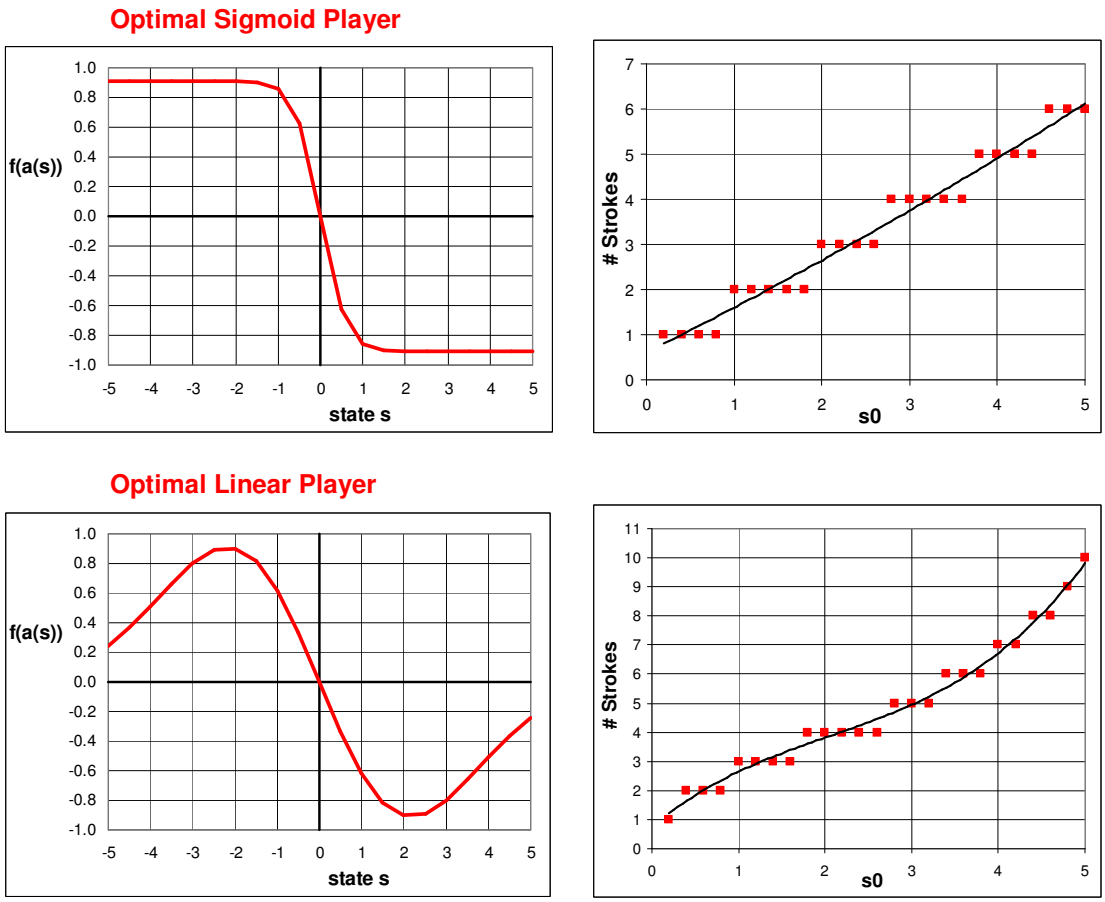
Figur 6: Evolution der Strategie eines sigmoiden Spielers im eindimensionalen Golf-Problem
Actor-Critic Learning (verallgemeinerter Baxter-Algorithmus)
($\gamma = 1.0$, $\alpha_\theta = 0.02$, $\alpha_V = 0.01$, $\lambda = 0$)

Der obere Graph bezieht sich auf die mittlere Anzahl Schläge dividiert durch die Anfangsdistanz zum Loch, gemessen in (aufgerundeten) 90m-Intervallen [$s_0^+ = \mathrm{Int}(|s_0|/0.9)+1$].
Die rote Kurve zeigt die Leistungsentwicklung während dem Lernprozess und beinhaltet deshalb auch Zufallseffekte der Explorationsstrategie. Die gelernte deterministische Strategie (blaue Kurve) kann aus dem aktuellen Agentenmodell extrahiert werden, indem man die Explorationswahrscheinlichkeit auf Null setzt.

Der untere Graph bezieht sich auf die totale Anzahl von Schlägen in einem Spiel über 25 Löcher (Distanzen zum Loch = 20m, 40m, 60m, …, 500m).

In Figur 7 wird der optimale sigmoide Spieler mit dem optimalen linearen Spieler verglichen. Die Graphen auf der linken Seite zeigen die aus der optimalen Strategie resultierenden Schlagdistanzen (dividiert durch 100m) als Funktion der Distanz zum Loch. Die Graphen auf der rechten Seite beziehen sich auf die Anzahl der benötigten Schläge, um das Loch aus einer Anfangsdistanz s_0 zu erreichen

Die betrachteten Spielertypen sind in ihren Möglichkeiten sehr eingeschränkt. Der lineare Spieler kann nur Strategien realisieren, bei denen die Schlagstärke proportional mit der Distanz zum Loch zunimmt. Beim sigmoiden Spieler ist die Schlagstärke mit zunehmender Distanz zum Loch immerhin nach oben beschränkt.



Figur 7: Gelernte optimale Strategien und entsprechende Resultate für den sigmoiden und für den linearen Golfspieler.

Die numerischen Resultate des Leistungsvergleichs sind in der folgenden Tabelle zusammengefasst, die zum Vergleich auch die Werte eines global optimalen Spielers enthält:

	⟨# Strokes / s_0^+⟩	Total # of Strokes
Globales Optimum	0.96	81
Optimaler sigmoider Spieler	1.00	84
Optimaler linear Spieler	1.49	121

Die Tabelle zeigt, dass der sigmoide Spieler sehr nahe an die maximal erreichbare Spielstärke herankommt, während der lineare Spieler auch im besten Fall noch ziemlich weit davon entfernt ist.

7.3 TD-Gammon und Liftsteuerung

Zwei besonders eindrückliche Beispiele für die praktische Anwendung von Reinforcement Learning beziehen sich auf das Backgammon-Spiel und auf die Steuerung von Liften in einem Hochhaus:

- TD-Gammon [3]
  Selbstlernendes Computerprogramm für das Backgammon-Spiel, das auf TD-Learning und auf der Approximation der Value-Funktion durch ein neuronales Netzwerk beruht.
  Die Version 3.0 erreicht die Spielstärke der weltbesten professionellen Backgammon-Spieler.
- Liftsteuerung [4]
  Optimierung der Steuerung eines Systems von 4 Liften in einem 10-stöckigen Hochhaus.
  Das Verfahren beruht ebenfalls auf TD-Learning mit Approximation der Value-Funktionen durch neuronale Netzwerke und findet bessere Lösungen als konventionelle Optimierungsstrategien.

Eine Zusammenfassung dieser Anwendungsbeispiele findet man im Buch von Sutton und Barto ([1], Kapitel 11).

8. REFERENZEN

[1] R.S. Sutton and A.G. Barto
Reinforcement Learning: An Introduction
MIT Press, Cambridge, Massachusetts, 1998
HTML-Version: http://www.cs.ualberta.ca/~sutton/book/the-book.html

[2] J. Baxter and P. Bartlett
Direct Gradient-Based Reinforcement Learning, II. Gradient Ascent Algorithms and Experiments
Technical Report, Research School of Information Sciences and Engineering,
Australian National University, 1999

[3] G.J.. Tesauro
TD-Gammon. A Self-Teaching Backgammon Program, Achieves Master Level Play
Neural Computation, 6(2), 1994, pp. 215-219

[4] R.H. Crites and A.G. Barto
Improving Elevator Performance Using Reinforcement Learning
In D.S. Touretzky et al, editors, *Advances in Neural Information Processing Systems, Proceedings of the 1995 Conference*, MIT Press, 1996, pp. 1017-1023