Fachhochschule Köln
Cologne University of Applied Sciences

Reinforcement Learning für Brettspiele: Der Temporal Difference Algorithmus

Wolfgang Konen
FH Köln

Stand Oktober 2008

Abstract *Dieser kurze Technical Report erläutert, wie man die Ideen des Reinforcement-Lernens (TD-Lernens) auf Brettspiele anwenden kann. Dieser TR trägt im Wesentlichen Ideen aus [Sutton_Barto98], [Tesauro92] und [Sutton_Bonde93] in kompakter Form zusammen und gibt Tipps für die praktische Umsetzung.*

Einführung 1
States und After-States 1
Die Spielfunktion 2
Der Temporal Difference (TD) Algorithmus 2
Die Grundideen 2
Beispiel zum Feature-Vektor: Das Spiel Nimm-3 3
Der TD(λ)-Algorithmus 3
„Self-Play“: Inkrementeller TD(λ)-Algorithmus 4
„Human-Computer-Play“ 6
Tipps für die praktische Umsetzung 7
Fazit 7
Literatur 7

Einführung

Reinforcement Learning ist eine mächtige Optimierungsmethode für komplexe Probleme. Es hat besonders dann seine Vorteile, wenn nicht für jede einzelne Aktion eine Belohnung gegeben werden kann sondern erst später, nach einer Sequenz von Aktionen. Dies ist typischerweise bei Brettspielen der Fall.

States und After-States

Ein Brettspiel wie Schach, Go, Connect4, TicTacToe oder Nimm-3 wird in der Regel beschrieben durch

- die Information, welcher Spieler am Zug ist und
- die Position auf einem Spielbrett

Beides wird zusammengefaßt im **Zustand s_t** (engl. *state*). Hierbei bezeichnet t=0,1,2,…,N die Abfolge der Spiel(halb)-Züge. Bei TicTacToe ist N maximal 9, das Spiel kann aber auch nach weniger als 9 Zügen beendet sein.

Bei der Positionscodierung gibt es die State- und die After-State-Variante:
State-Codierung:

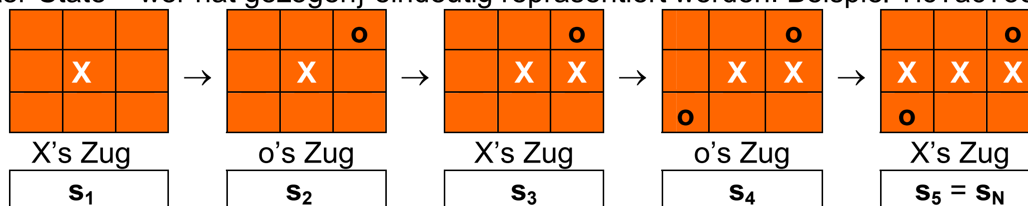
- Position, bevor der Spieler gezogen hat
- Information, welcher Spieler am Zug ist

After-State-Codierung:

- Position, nachdem der Spieler gezogen hat
- Information, welcher Spieler gerade gezogen hat

Meist ist die After-State-Codierung günstiger, weil es für Brettspiele i.d.R. nur zählt, welche Stellung nach dem Zug herauskommt. Mehrere verschiedene (State+Zug)-Kombinationen können zum gleichen After-State führen, aber eine (State+Zug)-Kombination führt nur zu genau einem After-State. Man spart also Komplexität, wenn man sich auf die Betrachtung der After-States stützt.

Nehmen wir einen konkreten Spielverlauf, so kann er durch eine Sequenz von Zuständen $\mathbf{s}_t$ = {After-State + wer hat gezogen} eindeutig repräsentiert werden. Beispiel TicTacToe:



Die Spielfunktion

Ziel des Reinforcement-Learnings ist es nun, aus vielen solchen Spielsequenzen einen Agenten zu trainieren, der möglichst optimal spielt. Wir hätten einen solchen optimalen Agenten, wenn wir für jeden Zustand $\mathbf{s}_t$ die **Spielfunktion** $V(\mathbf{s}_t)$ (engl. value function) kennen würden. Diese Spielfunktion soll Zuständen, die günstig für Weiss (**X**) sind, einen möglichst hohen Wert zuordnen und Zuständen, die günstig für Schwarz (**o**) sind, einen möglichst niedrigen Wert. (Wie hoch oder wie niedrig ist eigentlich ohne Belang, solange von den in einer Spielsituation möglichen After-States der Beste jeweils am höchsten / am niedrigsten ist.)
Der Weiss-Agent befragt für alle möglichen After-States die Spielfunktion $V(\mathbf{s}_t)$ und nimmt den höchsten Wert, der Schwarz-Agent geht genauso vor, nimmt aber den niedrigsten Wert. Nun haben wir aber diese Spielfunktion nicht und es gibt auch i.d.R. keine direkte Vorschrift, sie zu berechnen. Wir können aber dem Endzustand eines konkreten Spielverlaufs jeweils eine Belohnung r_t (engl. **R**eward) zuweisen, z.B. in obigem Beispiel r_5=+1, weil Weiss gewonnen hat. Umgekehrt würde r_N=0 zugewiesen, wenn Schwarz gewonnen hat und r_N=0.5, wenn es ein Unentschieden ist. Die Spielfunktion $V(\mathbf{s}_5)$ in obiger Abbildung wäre z.B. =+1.
Die Spielfunktion $V(\mathbf{s}_4)$ im obigen Beispiel sollte idealerweise auch $V(\mathbf{s}_4)$=+1 werden, denn wenn Schwarz diesen After-State hinterläßt, wird ein optimal spielender Weiss-Agent auf jeden Fall gewinnen. Ein lernendes System kann so bei Beobachtung vieler Spielverläufe sukzessive „von hinten" deduzierend lernen, welche Positionen „gut" sind.
Die ideale Spielfunktion $V(\mathbf{s}_t)$ enthielte zum Schluss **die Wahrscheinlichkeit, dass es von diesem Zustand $\mathbf{s}_t$ zu einem Sieg von Weiss kommt** [Tesauro92].

Der Temporal Difference (TD) Algorithmus

Die Grundideen

Das Ziel ist also, die Spielfunktion $V(\mathbf{s}_t)$ zu lernen. Wir stehen hier vor zwei erheblichen Problemen:

1. Ausser für den terminalen Zug $\mathbf{s}_N$ ist nicht bekannt, wie das Teacher-Signal für $V(\mathbf{s}_t)$ lauten soll.

2. Für realistische Spiele ist der Raum der möglichen Zustände $\mathbf{s}_t$ schnell viel zu riesig, als dass man alle in einer Tabelle ablegen könnte oder alle Zustände beim Lernen hinreichend oft besuchen könnte.

Lösungen für beide Probleme sind in Sutton&Barto's einflussreichem Buch [Sutton Barto98] beschrieben:
Das 1. Problem bekommt man mit der Methode des Reinforcement Learning in den Griff, die ein Fehlersignal δ_t für den Zustand $V(\mathbf{s}_t)$ definiert:

$$\delta_t = r_{t+1} + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)$$

Das Fehlersignal verschwindet, wenn $V(\mathbf{s}_t)$ den Zielwert $r_{t+1}+\gamma\, V(\mathbf{s}_{t+1})$ erreicht. Man wartet also den nächsten Zustand $\mathbf{s}_{t+1}$ ab, schaut, ob man für diesen den Reward oder die Value Function kennt. Wenn ja, verändert man $V(\mathbf{s}_t)$ in diese Richtung. Der Parameter γ (typischerweise =0.9) ist der sog. **Discount-Faktor**: Er berücksichtigt, dass eine weiter in der Zukunft liegende günstige Spielfunktion, z.B. $V(\mathbf{s}_{t+10})=1$, zwar ein möglicher Nachfolger von $\mathbf{s}_t$ in einem Spielverlauf ist, aber es nicht sicher ist, ob immer (für alle Spielverläufe) von $\mathbf{s}_t$ der Weg zu $\mathbf{s}_{t+10}$ führt. Diese weit weg liegenden Spielfunktionswerte werden also in ihrer Wirkung auf $V(\mathbf{s}_t)$ „abgezinst“ (abgeschwächt).
Zur Namensgebung „TD“: Weil für die meisten Spielzustände (in denen der Reward ja Null ist) das Fehlersignal im Wesentlichen die zeitliche Differenz in der Spielfunktion ist, nennt man den hierauf beruhenden Algorithmus **Temporal Difference (TD)** Algorithmus.

Das 2. Problem löst man durch Funktionsapproximation [Sutton Barto98, Kap. 8]: Eine Funktion $f(\mathbf{w};\mathbf{s}_t)$ mit freien Parametern $\mathbf{w}$ (den Gewichten) wird gesucht, um $V(\mathbf{s}_t)$ bestmöglich zu approximieren. Typische Realisationen für $f(\mathbf{w};\mathbf{s}_t)$ sind

- ein neuronales Netz mit $\mathbf{s}_t$ als Input, einer Hidden-Schicht, Gewichten $\mathbf{w}$ und einem Output-Neuron
- ein generalisiertes neuronales Netz mit Feature-Vektor $\mathbf{g}(\mathbf{s}_t)$ als Input
- eine lineare Funktion: $f(\mathbf{w};\mathbf{s}_t) = \mathbf{w} \cdot \mathbf{s}_t = \sum_k w_k s_{tk}$
- eine generalisierte lineare Funktion: $f(\mathbf{w};\mathbf{s}_t) = \mathbf{w} \cdot \mathbf{g}(\mathbf{s}_t) = \sum_k w_k g_k(\mathbf{s}_t)$

Dann wird natürlich eine Änderung in $\mathbf{w}$, die $|\, V(\mathbf{s}_t)^{(teach)} - f(\mathbf{w};\mathbf{s}_t)\,|$ verkleinert, auch andere Werte $f(\mathbf{w};\mathbf{s}_r)$, $r \neq t$, beeinflussen, aber das ist u.U. sogar gewünscht, weil nicht alle Zustände besucht werden können.
Feature-Vektor: Die Approximation von V durch f wird in der Regel besser gelingen, wenn benachbarte Zustände im Inputraum auch ähnliche Outputs haben. Das ist der Grund, weshalb man in den generalisierten Realisationen den Feature-Vektor $\mathbf{g}()$ einführt. Ein „schwieriges“ Input-Output-Mapping bei der Spielfunktion kann u.U. viel einfacher werden, wenn man die richtigen Features $g_k(\mathbf{s}_t)$ findet, wie das nachfolgende Beispiel zeigt.

Beispiel zum Feature-Vektor: Das Spiel Nimm-3

Beim Spiel Nimm-3 ist es das Ziel, durch wechselseitiges Wegnehmen von 1,2 oder 3 Hölzern schließlich das letzte Holz zu „ergattern“. Bekanntermaßen ist es die richtige Strategie, einen After-State anzustreben, der durch 4 teilbar ist. Deshalb ist das Feature

$g(\mathbf{s}_t)$ = ($\mathbf{s}_t$ durch 4 teilbar ? 1 : 0)

sofort eine perfekte Kodierung der Spielfunktion für alle After-States $\mathbf{s}_t$ nach einem Spielzug von Weiß. (Wir erinnern uns: Die Spielfunktion sollte ein Maß für die Wahrscheinlichkeit sein, dass Weiss von diesem Zustand aus bei optimalem Spiel gewinnt. Ist $\mathbf{s}_t$ der After-State nach einem Spielzug von Schwarz, so sind 1 und 0 einfach zu vertauschen.)

Der TD(λ)-Algorithmus

Jetzt brauchen wir noch eine Änderungsvorschrift für die Gewichte $\mathbf{w}$. Eine entsprechende Regel findet sich in [SuttonBarto98, Kap. 8.2]:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\mathbf{e}_t \quad \text{mit}$$

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \nabla_w f(\mathbf{w}_t; s_t)$$

Der Vektor $\mathbf{e}_t$ besteht aus sog. **Eligibility Traces** für jedes Gewicht. Die Theorie, die dahinter steht, ist für $\lambda>0$ nicht ganz einfach, und auch in der Praxis ist ein $\lambda>0$ längst nicht immer vorteilhaft. Beschränken wir uns deshalb hier auf den Fall $\lambda=0$, dann ist $\mathbf{e}_t$ nichts anderes als der Gradient für den Zustand $V(\mathbf{s}_t)$: Wenn das Fehlersignal δ_t positiv ist, marschieren wir den Berg hinauf, d.h. $V(\mathbf{s}_t)$ wird angehoben. Ist δ_t negativ, so gehen wir den Berg hinab, d.h. $V(\mathbf{s}_t)$ wird verkleinert. Wenn wir nicht zu weit marschieren – dies kontrolliert die Lernschrittweite α – sollte also in jedem Schritt das Fehlersignal kleiner werden, man spricht deshalb auch von **„gradient descent“**.

Wir haben jetzt alles beisammen, um den **TD(λ)-Algorithmus** nach [SuttonBarto98, Kap. 8.2] bzw. [Sutton_Bonde93] aufzuschreiben:

TD(λ)-Algorithmus

Input: Spielverlauf / Zustände $\mathbf{s}_0, \mathbf{s}_1, \ldots \mathbf{s}_N$ und die zugehörigen Rewards $r(\mathbf{s}_t)$.

{

Setze $V_{old} = V(\mathbf{s}_0)$ und $\mathbf{e}_0=\mathbf{0}$.

Setze $\mathbf{e}_1 = \gamma\lambda\mathbf{e}_0 + \nabla_{\mathbf{w}} f(\mathbf{w}; \mathbf{s}_0)$

Für t=0,…,N-1 {

Targetsignal $T_{t+1} = r_{t+1} + \gamma V(\mathbf{s}_{t+1})$ bestimmen bestehend aus

Response $V(\mathbf{s}_{t+1}) = f(\mathbf{w}; \mathbf{s}_{t+1})$ des neuronalen Netzes und

Reward $r_{t+1} = r(\mathbf{s}_{t+1})$ aus der Spielumgebung;

Fehlersignal $\delta_t = T_{t+1} - V_{old}$

Lernschritt: $\mathbf{w} = \mathbf{w} + \alpha\delta_t\mathbf{e}_t$

Response $V_{old} = f(\mathbf{w}; \mathbf{s}_{t+1})$ erneut berechnen (geändertes **w**!);

$\mathbf{e}_{t+1} = \gamma\lambda\mathbf{e}_t + \nabla_{\mathbf{w}} f(\mathbf{w}; \mathbf{s}_{t+1})$

}

} // Ende TD(λ)-Algorithmus

Anmerkungen:

- Durch jeden Lernschritt wird $V(\mathbf{s}_t) = f(\mathbf{w}; \mathbf{s}_t)$ näher an Zielwert $r_{t+1} + \gamma V(\mathbf{s}_{t+1})$ herangebracht.
- Es ist wichtig (und wird in manchen Implementierungen des TD(λ)-Algorithmus vergessen), dass die Response nach dem Lernschritt erneut berechnet wird, denn die Gewichte haben sich geändert, und deshalb muss man Gradient und V_{old} im Zustand $\mathbf{s}_{t+1}$ (der der Zustand $\mathbf{s}_t$ für die nächste Iteration wird) neu ausrechnen.

Der Algorithmus wurde so aufgeschrieben, dass er fast Zeile für Zeile mit dem main() aus [Sutton_Bonde93] korrespondiert.
Typische Werte für die Parameter sind $\gamma=0.9$, $\lambda<\gamma$, $\alpha=0.1$. Setzt man $\lambda=0$, so hat man wieder den üblichen Gradientenabstieg.

„Self-Play“: Inkrementeller TD(λ)-Algorithmus

Für den Einsatz in einem Brettspiel, in dem das Lernen inkrementell während des Spielens passiert, ist der Algorithmus ein wenig abzuändern. Der Unterschied ist, dass keine Zustandssequenz $\mathbf{s}_0, \mathbf{s}_1, \ldots \mathbf{s}_N$ vorliegt, sondern dass diese im Laufe des Lernens inkrementell generiert wird:

„Self-Play“: Inkrementeller TD(λ)-Algorithmus für Brettspiele

Input: Start-Player p [=+1 (Weiss) oder -1 (Schwarz)], Initialposition $\mathbf{s_0}$, sowie eine (teiltrainierte) Funktion $f(\mathbf{w};\mathbf{s}_t)$ zur Berechnung der Spielfunktion $V(\mathbf{s}_t)$

{

Setze $V_{old} = V(\mathbf{s_0})$ und $\mathbf{e}_0=\mathbf{0}$. [Beachte: zu $\mathbf{s_0}$ gehört Player –p]

Setze $\mathbf{e}_1 = \gamma\lambda\mathbf{e}_0 + \nabla_\mathbf{w} f(\mathbf{w};\mathbf{s}_0)$

Für t=0, …, maxN {

Mit Wahrsch. ε wähle einen für Player p erlaubten After-State $\mathbf{s}_{t+1}$ **zufällig** aus; [explorativer Move = Random Move]

Mit Wahrsch. (1-ε) suche den für Player p erlaubten After-State $\mathbf{s}_{t+1}$, der $p\cdot[f(\mathbf{w};\mathbf{s}_{t+1})+ r(\mathbf{s}_{t+1})]$ maximiert; [greedy Move]

Targetsignal $T_{t+1} = r_{t+1} + \gamma V(\mathbf{s}_{t+1})$ bestimmen bestehend aus

Response $V(\mathbf{s}_{t+1}) = f(\mathbf{w};\mathbf{s}_{t+1})$ des neuronalen Netzes und

Reward $r_{t+1}= r(\mathbf{s}_{t+1})$ aus der Spielumgebung;

Fehlersignal $\delta_t = T_{t+1} - V_{old}$

Wenn $\mathbf{s}_{t+1}$ kein Random Move ist:

Lernschritt: $\mathbf{w} = \mathbf{w} + \alpha\delta_t\mathbf{e}_t$

[bringt $V(\mathbf{s}_t) = f(\mathbf{w};\mathbf{s}_t)$ näher an Zielwert T_{t+1} heran]

Wenn $\mathbf{s}_{t+1}$ Spielendstand $\mathbf{s}_N$:

break; [For-Loop verlassen]

Response $V_{old} = f(\mathbf{w};\mathbf{s}_{t+1})$ erneut berechnen (geändertes **w**!);

$\mathbf{e}_{t+1} = \gamma\lambda\mathbf{e}_t + \nabla_\mathbf{w} f(\mathbf{w};\mathbf{s}_{t+1})$ [dies wird das $\mathbf{e}_t$ für die nächste Iteration]

Setze p = – p [Spielerwechsel]

}

} // Ende „Self-Play“

Anmerkungen:

- Der Reward $r(\mathbf{s}_{t+1})$ bestimmt sich nach folgender Tabelle:

$r(\mathbf{s}_{t+1})$	wenn …
1.0	… $\mathbf{s}_{t+1}$ ein Win für Player p=+1 ist
0.5	… $\mathbf{s}_{t+1}$ ein Remis-Endzustand (Tie, Draw) ist
0.0	… $\mathbf{s}_{t+1}$ ein Win für Player p=–1 ist oder wenn $\mathbf{s}_{t+1}$ kein Endzustand ist

- Die Spielfunktion $V(\mathbf{s}_t) = f(\mathbf{w};\mathbf{s}_t)$ repräsentiert idealerweise am Ende des Trainings die Wahrscheinlichkeit, dass Player +1 (Weiss) von diesem Zustand $\mathbf{s}_t$ aus gewinnt.
- Dies ist der Algorithmus für das Lernen aus <u>einem</u> Spielverlauf. O.B.d.A. können wir für die meisten Brettspiele den Start-Player immer mit p=+1 (Weiss) festlegen und als Initialposition das gleiche (leere bzw. mit fester Position belegte) Brett nehmen. Die Random Moves stellen sicher, dass nicht immer dasselbe Spiel abläuft. Um einen TD(λ)-Spielagenten zu trainieren, werden initial die Gewichte **w** zufällig initialisiert und dann der obige Algo viele Male (z.B. 10000 mal) aufgerufen.
- Typischerweise wird die Explorationsrate ε und die Lernrate α im Laufe des Trainings verkleinert, damit am Anfang viel ausprobiert, aber gegen Ende des Lernens die Konvergenz auf einen (hoffentlich guten) Spiel-Agenten erreicht wird.
- maxN ist die maximal mögliche Anzahl von Zügen in einem Spiel. Die Schleife wird aber verlassen, sobald $\mathbf{s}_{t+1}$ ein Endstand $\mathbf{s}_N$ ist. Man beachte, dass das Netz nie darauf trainiert wird, $V(\mathbf{s}_N)$ in die Nähe von $r(\mathbf{s}_N)$ zu bewegen. Es werden lediglich die Vorzustände zu $V(\mathbf{s}_N)$ trainiert (die allerdings über die Funktionsapproximation auch die Antwort $V(\mathbf{s}_N)$ verändern können).

- Das Targetsignal $T_{t+1} = r_{t+1} + \gamma V(\mathbf{s}_{t+1})$ sollte man „getrennt“ berechnen: Wenn $\mathbf{s}_{t+1}=\mathbf{s}_N$, also Spielendstand, dann $T_{t+1} = r_{t+1}$ (die ideale Spielfunktion hätte $V(\mathbf{s}_{t+1})$ =0, weil nie dieser Zustand gelernt wurde). Wenn kein Spielendstand, dann ist r_{t+1}=0, also $T_{t+1} = \gamma V(\mathbf{s}_{t+1})$. {TD_NNet.java::updateWeights}
  Ebenso macht es Sinn, im Greedy Move den Score $p\cdot[f(\mathbf{w};\mathbf{s}_{t+1}) + r(\mathbf{s}_{t+1})]$ getrennt zu berechnen. {N3TDPlayer.java::getBestTable}
  Der Grund ist in beiden Fällen, dass dann das Targetsignal für einen Spielendstand „sauber“ ist, d.h. genau dem Reward entspricht. (Das ideale $V(\mathbf{s}_{t+1})$ wäre zwar auch 0, aber die Approximation $f(\mathbf{w};\mathbf{s}_{t+1})$ kann Störungen enthalten, insbesondere weil ja für den Zustand $\mathbf{s}_{t+1} = \mathbf{s}_N$ nie gelernt wurde.)
- Wieso wird nach einem Random Move $\mathbf{s}_{t+1}$ nicht gelernt? – Weil der Random Move mit hoher Wahrscheinlichkeit nicht der beste Zug ist. Damit führt er möglicherweise auf ein schlecht bewertetes $V(\mathbf{s}_{t+1})$. Dies würde $V(\mathbf{s}_t)$ „herunterziehen“, auch wenn dies eigentlich ein Gewinnzustand wäre. Das ist sicher nicht wünschenswert.
- Ist ein Reward=0 sinnvoll für Win Player p = -1? Das bedeutet, dass der Reward für einen (-1)-Win um nichts anders ist als der (nicht vorhandene) Reward für alle intermediären Spielzustände. – Ja, trotzdem, denn: zufällig initialisierte Gewichte werden i.d.R. zufällige Outputs aus [0,1] mit Mittelwert 0.5 erzeugen. Wenn nun ein Reward am Ende ($\mathbf{s}_N$) ausbleibt, dann lernt das Netz die Vorzustände zu $\mathbf{s}_N$ auch auf 0 herunterzuziehen. Ein Reward=0 macht also Sinn.

„Human-Computer-Play“

Der Algorithmus aus dem vorigen Abschnitt ist für Self-Play („Computer gegen Computer“). Man könnte eine ähnliche Variante (wie bei Levkovich) auch für „Computer gegen Mensch“ schreiben. Auch hier wird die Zustandssequenz $\mathbf{s}_t$ für die Computerseite inkrementell während des Lernens generiert:

„Human-Computer-Play“: Inkrementeller TD(λ)-Algorithmus für Brettspiele

Input: Computer-Player p [=+1 (Weiss) oder -1 (Schwarz)], Initialposition $\mathbf{s_0}$, Flag **ComputerMove**, das =true ist, wenn der Computer einen Zug machen soll, sowie eine (teiltrainierte) Funktion $f(\mathbf{w};\mathbf{s}_t)$ zur Berechnung der Spielfunktion $V(\mathbf{s}_t)$

{
    Setze $V_{old} = V(\mathbf{s_0})$ und $\mathbf{e}_0=\mathbf{0}$.   [Beachte: zu $\mathbf{s_0}$ gehört Player –p]
    Setze $\mathbf{e}_1 = \gamma\lambda\mathbf{e}_0 + \nabla_{\mathbf{w}} f(\mathbf{w};\mathbf{s}_0)$
    Für t=0, …, maxN {
        Wenn (ComputerMove):
            suche den für Player p erlaubten After-State $\mathbf{s}_{t+1}$, der
            $p\cdot[f(\mathbf{w};\mathbf{s}_{t+1}) + r(\mathbf{s}_{t+1})]$ maximiert;   [greedy Move]
        sonst:
            warte auf $\mathbf{s}_{t+1}$ = Zustand nach Human Move
        Targetsignal $T_{t+1} = r_{t+1} + \gamma V(\mathbf{s}_{t+1})$ bestimmen bestehend aus
            Response $V(\mathbf{s}_{t+1}) = f(\mathbf{w};\mathbf{s}_{t+1})$ des neuronalen Netzes und
            Reward $r_{t+1}= r(\mathbf{s}_{t+1})$ aus der Spielumgebung;
        Fehlersignal $\delta_t = T_{t+1} - V_{old}$
        Wenn $\mathbf{s}_{t+1}$ ein Computer Move ist:
            Lernschritt: $\mathbf{w} = \mathbf{w} + \alpha\delta_t\mathbf{e}_t$
            [bringt $V(\mathbf{s}_t) = f(\mathbf{w};\mathbf{s}_t)$ näher an Zielwert $r_{t+1}+\gamma V(\mathbf{s}_{t+1})$ heran]
            ComputerMove=false;
        Wenn $\mathbf{s}_{t+1}$ Spielendstand $\mathbf{s}_N$ (egal ob durch Human oder Computer erreicht):
            break;   [For-Loop verlassen]

Response $V_{old} = f(\mathbf{w};\mathbf{s}_{t+1})$ erneut berechnen (geändertes **w**!);
$\mathbf{e}_{t+1} = \gamma\lambda\mathbf{e}_t + \nabla_{\mathbf{w}} f(\mathbf{w};\mathbf{s}_{t+1})$ [dies wird das $\mathbf{e}_t$ für die nächste Iteration]
}
} // Ende „Human-Computer-Play“

Anmerkungen:

- Gelernt wird also immer dann, wenn wir eine klare Bewertung für $\mathbf{s}_{t+1}$ haben:
  - entweder weil es ein ComputerMove ist (und $V(\mathbf{s}_{t+1})$ sollte zumindest asymptotisch verläßlich sein)
  - oder weil es ein HumanMove ist, der direkt zum Gewinn führte, in diesem Fall ist die Bewertung auch klar.
- Gelernt wird für den davorliegenden Zustand $\mathbf{s}_t$, also in der Regel für den Human Move.
- Der Algorithmus „Human-Computer-Play“ wird normalerweise für praktische Lernaufgaben keine Rolle spielen, weil man viel zu viele Runden spielen müsste, bis der Reinforcement-Learning-Agent (der Computer) seine Aufgabe wirklich gelernt hätte. Er kann aber für sehr einfache Lernaufgaben instruktiv sein, um zu sehen, wie der TD(λ)-Algorithmus auf einzelne Spielzüge reagiert.
- Die übrigen Anmerkungen zum Algorithmus „Self-Play“ gelten sinngemäß.

Tipps für die praktische Umsetzung

- Neben komplexeren Funktionsapproximatoren (wie Backprop) sollte man immer auch die lineare Funktion
  $$f(\mathbf{w};\mathbf{s}_t) = \mathbf{w} \cdot \mathbf{g}(\mathbf{s}_t) = \sum_k w_k g_k(\mathbf{s}_t)$$
  in Betracht ziehen. Der Grund: Sie lernt robuster (keine lokalen Nebenminima) und oftmals auch schneller. Zwar kann eine lineare Funktion nicht ein so komplexes I/O-Mapping realisieren, aber diesen Nachteil kann man oft dadurch kompensieren, dass man komplexere Features im Input anbietet (lieber ein paar Features zu viel als zu wenig!)
- Es lohnt sich, über den Feature-Vektor gut nachzudenken.
- Keine Sigmoid-Funktion im Output-Neuron.
- λ=0 (keine Eligibility Traces) ist für Brettspiele oftmals günstiger.

Fazit

Der für die Praxis eines Brettspiel lernenden Agenten relevanteste Algorithmus ist die Variante „Self-Play“: Inkrementeller TD(λ)-Algorithmus. Er ist so formuliert, dass sich der Pseudo-Code aus [Sutton_Bonde93], der zur Funktionsapproximation ein einfaches neuronales Netz (Backprop) implementiert, relativ leicht hier einpassen läßt.
Ebenso leicht läßt sich zur Funktionsapproximation eine lineare Funktion einbauen, der Gradient ist in diesem Fall nichts anderes als der Feature-Inputvektor.
Eine Anwendung der hier beschriebenen Algorithmen auf die Spiele Nim-3 und TicTacToe, verbunden mit einer Evaluation verschiedener Feature-Sets, wird in [Konen08] dargestellt.

Literatur

[Sutton&Barto98] Richard S. Sutton, Andrew G. Barto: Reinforcement Learning. MIT Press, Cambridge, 1998. Hervorragende Einführung ins Thema, auch mit sehr guten Erklärung und vielen Beispielen. Allerdings viele mathematische Details und viele RL-

Varianten, die es für den Beginner nicht ganz einfach machen. Aktuelle Adresse: http://www.cs.ualberta.ca/~sutton/book/the-book.html.

[Sutton_Bonde93] Richard Sutton and Allen Bonde Jr. (1992) *Nonlinear **TD/Backprop pseudo C-code***, GTE Laboratories. Ist eine „fast" fertige TD(λ)-Implementierung, erstaunlich kurz (!), es fehlt nur IO und Random Number Generator. Die Theorie dahinter ist in [Sutton&Barto98], Kap. 6.1-6.4 (TD), (6.8: after states), 7.1-7.3 (eligibility traces), 8.1-8.2 (function approximation, gradient descent) erklärt

[Tesauro95] Gerald Tesauro: Temporal Difference Learning and TD-Gammon, C*ommunications of the ACM,* March 1995 / Vol. 38, No. 3. Sehr guter Übersichtsartikel über Tesauro's Arbeiten von 1992.

[Tesauro92] Gerald Tesauro: Practical issues in temporal difference learning. *Mach. Learning 8,* (1992), 257-277. Die berühmte TD-Gammon-Veröffentlichung, in der Tesauro zeigt, dass ein RL-Spielagent mit erstaunlich wenig Vorwissen lernt, das Spiel Backgammon auf Weltklasseniveau zu spielen.

[Konen08] W. Konen, T. Bartz-Beielstein: Reinforcement Learning: Insights from Interesting Failures in Parameter Selection. In: G. Rudolph et al. (ed.), 10th International Conference on Parallel Problem Solving From Nature (PPSN2008), Dortmund, September 2008, p. 478-487. In: Lecture Notes in Computer Science, LNCS 5199, Springer, Berlin, 2008.