e-print http://www.gm.fh-koeln.de/ciopwebpub/Kone15a.d/TR-TDgame.pdf, Februar 2015

Reinforcement Learning für Brettspiele: Der Temporal Difference Algorithmus

Wolfgang Konen

Institut für Informatik,
Fakultät 10 der FH Köln
Steinmüllerallee 1, D-51643 Gummersbach,
Germany

http://www.gm.fh-koeln.de/~konen
wolfgang.konen@fh-koeln.de

Februar 2015
(Erstversion Oktober 2008)

Zusammenfassung

Dieser Technical Report erläutert, wie man die Ideen des Reinforcement-Lernens (TD-Lernens) auf Brettspiele anwenden kann. Dieser Technical Report trägt die wesentlichen Ideen aus Sutton and Barto [1998], Tesauro [1992] und Sutton and Bonde [1992] in kompakter Form zusammen und gibt Tipps für die praktische Umsetzung.

Neu gegenüber der vorherigen Version des Technical Reports aus dem Jahr 2008 ist: eine Aktualisierung und Vereinfachung der 'Self-Play' TD-Algorithmen, ein Abschnitt zu Game-Learning-Anwendungen von TD(λ) mit aktualisierten Literaturstellen, ferner eine Erweiterung um die neuen Anhänge Eligibility Traces und Typische Funktionsapproximatoren.

Inhaltsverzeichnis

1 Einführung

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

Temporal Difference Learning (TD) ist eine spezielle, weit verbreitete Form des Reinforcement Learning.

2 States und After-States

Eine Spielsituation in Brettspielen wie Schach, Go, Connect-4, 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** $\boldsymbol{s}_t$ (engl. *state*). Hierbei bezeichnet $t = 0, 1, 2, \ldots, N$ 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:

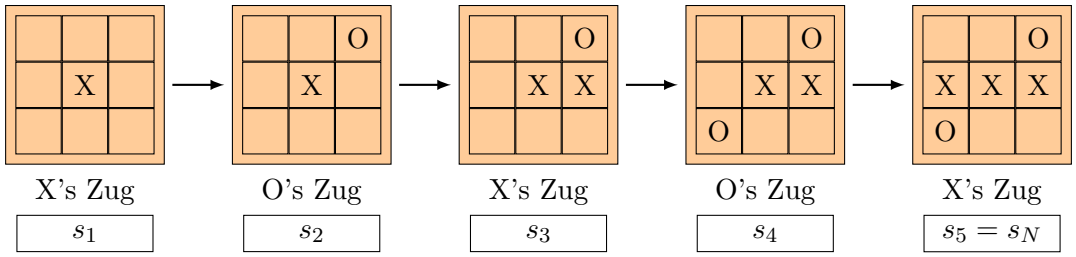

Abbildung 1: Beispielhafter Spielverlauf für TicTacToe. Das Spiel ist nach $N = 5$ Zügen beendet.

- 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 $\boldsymbol{s}_t$ = {After-State + wer hat gezogen} eindeutig repräsentiert werden. Fig. 1 zeigt einen solchen beispielhaften Spielverlauf für TicTacToe.

3 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 $\boldsymbol{s}_t$ die **Spielfunktion** $V(\boldsymbol{s}_t)$ (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.[1] Der Weiss-Agent befragt für alle möglichen After-States die Spielfunktion $V(\boldsymbol{s}_t)$ 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 jedoch dem *Endzustand* $\boldsymbol{s}_N$ eines konkreten Spielverlaufs jeweils eine Belohnung $r(\boldsymbol{s}_t)$ (engl. *reward*) zuweisen. Im obigen Beispiel von Fig. 1 ist der Reward $r(\boldsymbol{s}_5) = r(\boldsymbol{s}_N) = +1$, weil X gewonnen hat. Umgekehrt würde $r(\boldsymbol{s}_N) = 0$ zugewiesen, wenn O gewonnen hat und $r(\boldsymbol{s}_N) = 0.5$, wenn es ein Unentschieden ist. Der Wert des Endzustandes $\boldsymbol{s}_N$ in obiger Abbildung ist also $= +1$. Die Spielfunktion $V(\boldsymbol{s}_4)$ im obigen Beispiel sollte idealerweise auch $V(\boldsymbol{s}_4) = +1$

[1] 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.

werden, denn wenn O diesen After-State hinterläßt, wird ein optimal spielender X-Agent auf jeden Fall gewinnen. Ein lernendes System kann so bei Beobachtung vieler Spielverläufe sukzessive „von hinten“ deduzierend lernen, welche Positionen „gut“ sind. Dies ist die Kernidee des Temporal Difference Learnings.

Die ideale Spielfunktion $V(\boldsymbol{s}_t)$ enthielte zum Schluss **die Wahrscheinlichkeit, dass es von diesem Zustand $\boldsymbol{s}_t$ bei perfekt spielenden Spielern zu einem Sieg von X kommt** (Tesauro [1992]). Allgemeiner gesprochen – nicht nur auf Spiele bezogen – sollte die Value Function $V(\boldsymbol{s}_t)$ die ausgehend vom Zustand $\boldsymbol{s}_t$ in der Zukunft noch zu erwartenden Rewards (den Reward-Erwartungswert) angeben.

4 Der Temporal Difference (TD) Algorithmus

4.1 Die Grundideen

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

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

2. Für realistische Spiele ist der Raum der möglichen Zustände $\boldsymbol{s}_t$ 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 and Barto [1998] beschrieben.

4.1.1 Temporal Difference

Das 1. Problem bekommt man mit der Methode des Reinforcement Learning in den Griff, die ein Fehlersignal δ_t für den Zustand $V(\boldsymbol{s}_t)$ definiert:

$$\delta_t = r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}) - V(\boldsymbol{s}_t) \tag{1}$$

Das Fehlersignal verschwindet, wenn $V(\boldsymbol{s}_t)$ den Zielwert $r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1})$ erreicht. Man wartet also den nächsten Zustand $\boldsymbol{s}_{t+1}$ ab, schaut, ob man für diesen den Reward oder die Value Function kennt. Wenn ja, verändert man $V(\boldsymbol{s}_t)$ 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. $V(s_{t+10}) = 1$, zwar ein möglicher Nachfolger von $\boldsymbol{s}_t$ in einem Spielverlauf ist, aber es nicht sicher ist, ob immer (für alle Spielverläufe) von $\boldsymbol{s}_t$ der Weg zu s_{t+10} führt. Diese weit weg liegenden Spielfunktionswerte werden also in ihrer Wirkung auf $V(\boldsymbol{s}_t)$ „abgezinst“ (herabgesetzt, engl. *discount*).

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.

4.1.2 Funktionsapproximation

Das 2. Problem löst man durch Funktionsapproximation [Sutton and Barto, 1998, Kap. 8]: Anstatt alle möglichen Werte $V(\boldsymbol{s}_t)$ in einer gigantischen Tabelle abzulegen, definiert man eine Funktion $f(\boldsymbol{w}; \boldsymbol{s}_t)$ mit freien Parametern $\boldsymbol{w}$ (den Gewichten, engl. *weights*), so dass

$$V(\boldsymbol{s}_t) = f(\boldsymbol{w}; \boldsymbol{s}_t) \tag{2}$$

bestmöglich approximiert wird. Typische Realisationen für $f(\boldsymbol{w}; \boldsymbol{s}_t)$ sind:

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

$$f(\boldsymbol{w}; \boldsymbol{s}_t) = \boldsymbol{w} \cdot \boldsymbol{g}(\boldsymbol{s}_t) = \sum_k w_k g_k(\boldsymbol{s}_t)$$

Dann wird eine Änderung in $\boldsymbol{w}$, die das Fehlersignal $\delta_t = (V(\boldsymbol{s}_t) - f(\boldsymbol{w}; \boldsymbol{s}_t))^2$ verkleinert, natürlich auch andere Werte $f(\boldsymbol{w}; s_u), u \neq t$, beeinflussen, aber das ist u. U. sogar gewünscht, weil nicht alle Zustände besucht werden können. Man nennt diese Eigenschaft **Generalisierung**: von besuchten Zuständen und ihren Rewards auf andere, ähnlich gelagerte Zustände schließen.

Es ist noch eine Besonderheit zu beachten: Aufgrund der Generalisierung wird auch für einen Endzustand $\boldsymbol{s}_N$ die Spielfunktion $V(\boldsymbol{s}_N)$ in der Regel nicht gleich Null sein. Dies entspricht aber nicht dem Verständnis der Value-Function $V(\boldsymbol{s}_t)$, die für einen gegebenen Zustand $\boldsymbol{s}_t$ die zukünftig noch zu erwartende Summe an Rewards wiedergeben sollte. Wenn man nun den finalen Reward mit $r(\boldsymbol{s}_N) + \gamma V(\boldsymbol{s}_N)$ in Gl. (1) schon abgeschöpft hat, dann ist der Zustand $\boldsymbol{s}_N$ eigentlich wertlos, denn in ihm wird ja nicht mehr weitergespielt. Um nichts doppelt zu zählen, definiert man daher meist:

$$V(\boldsymbol{s}_t) = \begin{cases} 0, & \text{wenn } \boldsymbol{s}_t \text{ final} \\ f(\boldsymbol{w}; \boldsymbol{s}_t), & \text{sonst} \end{cases} \tag{3}$$

4.1.3 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 einigen Realisationen von $f(\boldsymbol{w}; \boldsymbol{s}_t)$ den Merkmals-Vektor (Feature-Vektor) $\boldsymbol{g}()$ einführt. Ein „schwieriges“ Input-Output-Mapping bei der Spielfunktion kann u. U. viel einfacher werden, wenn man die richtigen Merkmale $\boldsymbol{g}(\boldsymbol{s}_t)$ 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 Steinen schließlich den letzten Stein zu „ergattern“. Bekanntermaßen ist es die richtige Strategie, einen After-State anzustreben, der durch 4 teilbar ist. Deshalb ist das Feature

$$g_4(\boldsymbol{s}_t) = (\boldsymbol{s}_t \text{ durch 4 teilbar ? } 1 : 0) \tag{4}$$

sofort eine perfekte Kodierung der Spielfunktion für alle After-States $\boldsymbol{s}_t$ 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 $\boldsymbol{s}_t$ der After-State nach einem Spielzug von Schwarz, so sind 1 und 0 einfach zu vertauschen.)

Bei komplexeren Spielen wird es ein solch perfektes Feature in der Regel nicht geben oder es ist zum Zeitpunkt der Kodierung nicht bekannt. Wenn aber die Vermutung besteht, dass Teilbarkeit eine Rolle spielt, kann man einen Feature-Vektor $\boldsymbol{g}(\boldsymbol{s}_t) = (g_k(\boldsymbol{s}_t)), k = 2, \ldots, M$ mit

$$g_k(\boldsymbol{s}_t) = (\boldsymbol{s}_t \text{ durch } k \text{ teilbar } ? \ 1 : 0) \tag{5}$$

definieren. Dann sollte der Lernalgorithmus im Laufe der Zeit lernen, dass bei Nimm-3 nur g_4 wichtig und alle anderen $g_k, k \neq 4$, unwichtig sind.

4.2 Der TD(λ)-Algorithmus

Jetzt brauchen wir noch eine Änderungsvorschrift für die Gewichte $\boldsymbol{w}$. Eine entsprechende Regel findet sich in [Sutton and Barto, 1998, Kap. 8.2]:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{e}_t \tag{6}$$

$$\boldsymbol{e}_{t+1} = \gamma \lambda \boldsymbol{e}_t + \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_{t+1}; \boldsymbol{s}_{t+1}) \tag{7}$$

Der Vektor $\boldsymbol{e}_t$ besteht aus sogenannten **Eligibility Traces** für jedes Gewicht. Die Theorie, die dahinter steht, ist für $\lambda > 0$ nicht ganz einfach, einige Erklärungen hierzu finden sich in Anhang A. Beschränken wir uns hier auf den Fall $\lambda = 0$, dann ist $\boldsymbol{e}_t$ nichts anderes als der Gradient für die Funktion $V(\boldsymbol{s}_t) = f(\boldsymbol{w}_t; \boldsymbol{s}_t)$. Der Gradient weist in Richtung des steilsten Anstiegs dieser Funktion. Wenn das Fehlersignal δ_t in Gl. (1) positiv ist, marschieren wir also mit der Änderung gemäß Gl. (6) den Berg hinauf, d.h. $V(\boldsymbol{s}_t)$ wird angehoben. Ist δ_t negativ, so gehen wir den Berg hinab, d.h. $V(\boldsymbol{s}_t)$ wird verkleinert. Wenn wir nicht zu weit marschieren – dies kontrolliert die Lernschrittweite α – sollte also in jedem Schritt das Fehlersignal in Gl. (1) kleiner werden, man spricht deshalb auch von „**gradient descent**“.

Wir haben jetzt alles beisammen, um den TD(λ)-Algorithmus nach [Sutton and Barto, 1998, Kap. 8.2] bzw. Sutton and Bonde [1992] in Algorithm 1 aufzuschreiben.

Anmerkungen zu Algorithm 1:

- Durch jeden Lernschritt wird $V(\boldsymbol{s}_t) = f(\boldsymbol{w}_t, \boldsymbol{s}_t)$ näher an Zielwert $r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1})$ 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. Deshalb muss man V_{old} in jedem Durchlauf neu berechnen (und kann nicht das $V(\boldsymbol{s}_{t+1})$ vom vorherigen Schleifendurchlaufs übernehmen).
- Der Algorithmus wurde so aufgeschrieben, dass er fast Zeile für Zeile mit dem `main()` aus [SuttonBonde93] 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.

Algorithm 1 TD(λ)-Algorithmus. Input: Spielverlauf / Zustände $\{\boldsymbol{s}_t | \, t = 0, 1, \ldots, N\}$ und die zugehörigen Rewards $r(\boldsymbol{s}_t)$. Ferner eine Funktion $f(\boldsymbol{w}_0; \boldsymbol{s}_t)$ mit (teiltrainierten) Gewichten $\boldsymbol{w}_0$ zur Approximation der Spielfunktion $V(\boldsymbol{s}_t)$. Output: Verbesserte Gewichte $\boldsymbol{w}_N$.

function TDLTRAIN($\{s_t | \, t = 1, \ldots, N\}$, $\boldsymbol{w}_0$)
  $\boldsymbol{e}_0 \leftarrow \nabla_{\boldsymbol{w}_0} f(\boldsymbol{w}_0; \boldsymbol{s}_0)$ ▷ Initiale Eligibility Trace
  for $\big(t \leftarrow 0 \; ; \; t < N ; \; t \leftarrow t + 1\big)$ **do**
    $V_{old} \leftarrow f(\boldsymbol{w}_t; \boldsymbol{s}_t)$ ▷ Spielfunktion aktueller Zustand
    $V(\boldsymbol{s}_{t+1}) \leftarrow \begin{cases} 0, & \text{wenn } \boldsymbol{s}_{t+1} \text{ final} \\ f(\boldsymbol{w}_t; \boldsymbol{s}_{t+1}), & \text{sonst} \end{cases}$ ▷ Spielfunktion nächster Zustand
    $\delta_t \leftarrow r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}) - V_{old}$ ▷ Fehlersignal
    $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{e}_t$ ▷ Lernschritt
    $\boldsymbol{e}_{t+1} \leftarrow \gamma \lambda \boldsymbol{e}_t + \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_{t+1}; \boldsymbol{s}_{t+1})$ ▷ Eligibility Trace
  end for
  return $\boldsymbol{w}_N$
end function

4.3 „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 $\{\boldsymbol{s}_t | \, t = 0, 1, \ldots, N\}$ vorliegt, sondern dass diese im Laufe des Lernens inkrementell generiert wird. Der Self-Play-Algorithmus ist in Algorithm 2 dargestellt.

Die Funktion SELFPLAY wird wiederholt aufgerufen, jeweils mit einer Initialposition $\boldsymbol{s}_0$ und jeweils mit den Gewichten, die aus dem vorherigen Aufruf zurückgegeben wurden. (Beim ersten Aufruf werden zufällig initialisierte Gewichte genommen.) SELFPLAY spielt dann ein vollständiges Spiel durch, bis ein Endzustand $\boldsymbol{s}_t \in S_{Final}$ erreicht wird.

Die Funktion SELFPLAY ist in zwei Punkten gegenüber Algorithm 1 erweitert: 1) Der jeweils nächste Zustand $\boldsymbol{s}_{t+1}$ wird vom Algorithmus selbst gewählt (entweder als Random Move oder aus der bisher gelernten Spielfunktion). 2) Das Fehlersignal wird bei einem Random Move i. d. R. auf Null gesetzt.

Weitere Anmerkungen zu Algorithm 2:

- Typischerweise wird man die Funktion SELFPLAY immer mit dem leeren Brett als Startposition $\boldsymbol{s}_0$) starten. Es ist aber genausogut denkbar, dass man immer andere (zufällige) Startpositionen wählt, um die Exploration zu verbessern.

- Die Random Moves stellen sicher, dass nicht immer dasselbe Spiel abläuft. Um einen TD(λ)-Spielagenten zu trainieren, werden initial die Gewichte $\boldsymbol{w}$ zufällig initialisiert und dann die Funktion SELFPLAY 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.

- Die FOR-Schleife wird verlassen, sobald $\boldsymbol{s}_{t+1}$ ein Endzustand ist. Man beachte, dass der Funktionsapproximator nie darauf trainiert wird, $V(\boldsymbol{s}_N)$ in die Nähe von $r(\boldsymbol{s}_N)$ zu bewegen.

Algorithm 2 „Self-Play“: Inkrementeller $TD(\lambda)$-Algorithmus für Brettspiele. Input: Start-Player p [$= +1$ (Weiss) oder -1 (Schwarz)], Initialposition $\boldsymbol{s}_0$. Ferner (teiltrainierte) Gewichte $\boldsymbol{w}_0$ für die Funktion $f(\boldsymbol{w}; \boldsymbol{s}_t)$ zur Approximation der Spielfunktion $V(\boldsymbol{s}_t)$. Reward-Funktion $r(\boldsymbol{s}_t)$ aus der Spielumgebung. Output: Verbesserte Gewichte $\boldsymbol{w}_N$.

function SELFPLAY(p, $\boldsymbol{s}_0$, $\boldsymbol{w}_0$)
  $\boldsymbol{e}_0 \leftarrow \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_0; \boldsymbol{s}_0)$ ▷ Initiale Eligibility Trace
  for $\big(t \leftarrow 0$ **;** $\boldsymbol{s}_t \notin S_{Final}$ **;** $t \leftarrow t+1\big)$ **do** ▷ S_{Final}: Menge aller Endzustände
    $V_{old} \leftarrow f(\boldsymbol{w}_t, \boldsymbol{s}_t)$ ▷ Spielfunktion aktueller Zustand
    Wähle zufällig $q \in [0, 1]$
    if $(q < \epsilon)$ **then**
      Wähle ein zufälliges $\boldsymbol{s}_{t+1}$ ▷ Explorativer Zug (Random Move)
    else
      Wähle legalen After-State s_{t+1}, so dass ▷ Greedy Move
        $p \cdot (r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}))$
      maximiert wird.
    end if
    $V(\boldsymbol{s}_{t+1}) \leftarrow \begin{cases} 0, & \text{wenn } \boldsymbol{s}_{t+1} \in S_{Final} \\ f(\boldsymbol{w}_t; \boldsymbol{s}_{t+1}), & \text{sonst} \end{cases}$ ▷ Spielfunktion nächster Zustand
    $\delta_t \leftarrow \begin{cases} 0, & \text{wenn } q < \epsilon \wedge \boldsymbol{s}_{t+1} \notin S_{Final} \\ r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}) - V_{old}, & \text{sonst} \end{cases}$ ▷ Fehlersignal
    $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{e}_t$ ▷ Lernschritt
    $\boldsymbol{e}_{t+1} \leftarrow \gamma \lambda \boldsymbol{e}_t + \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_{t+1}; \boldsymbol{s}_{t+1})$ ▷ Eligibility Trace
    $p \leftarrow (-p)$
  end for
  return $\boldsymbol{w}_N$
end function

Es werden lediglich die Vorzustände von $V(\boldsymbol{s}_N)$ trainiert. Dadurch kann sich allerdings bei Funktionsapproximation auch die Antwort $V(\boldsymbol{s}_N)$ verändern.

- Wieso wird nach einem Random Move $\boldsymbol{s}_{t+1}$ nicht gelernt? – Weil der Random Move mit hoher Wahrscheinlichkeit nicht der beste Zug ist. Damit führt er möglicherweise auf ein für p schlecht bewertetes $V(\boldsymbol{s}_{t+1})$. Dies würde $V(\boldsymbol{s}_t)$ „herunterziehen“, auch wenn dies eigentlich ein Gewinnzustand wäre. Das ist sicher nicht wünschenswert. Die einzige Ausnahme von dieser Regel ist, wenn $\boldsymbol{s}_{t+1}$ ein Endstand mit Win für Player p ist. Denn dann war der Random Move ja offensichtlich die bestmögliche Wahl, und diese Information darf auch auf $V(\boldsymbol{s}_t)$ übertragen werden.

5 Tipps für die praktische Umsetzung

- Neben komplexeren Funktionsapproximatoren (wie Backprop) sollte man immer auch die lineare Funktion (evtl. mit Feature-Vektor)

$$f(\boldsymbol{w}; \boldsymbol{s}_t) = \boldsymbol{w} \cdot \boldsymbol{g}(\boldsymbol{s}_t) = \sum_k w_k g_k(\boldsymbol{s}_t) \tag{8}$$

  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, gut über den Feature-Vektor nachzudenken.

- Keine Sigmoid-Funktion im Output-Neuron kann manchmal günstiger sein.

6 Anwendungen für TD(λ) und Self-Play

Eine Anwendung der hier beschriebenen Algorithmen auf die einfachen Spiele **Nimm-3** und **Tic-TacToe**, verbunden mit einer Evaluation verschiedener Feature-Sets, wird in Konen and Bartz-Beielstein [2008, 2009] dargestellt.

Die zugehörige Java-Software wurde seitdem beständig weiterentwickelt und ist in naher Zukunft als Open Source von Github ... verfügbar. Zu den Erweiterungen gehören N-Tuple-Features, Temporal Coherence sowie eine MCTS-Variante.

Die erste Anwendung von N-Tuple-Features für das TD(λ)-basierte Game Learning wird in Lucas [2008a,b] für das komplexere Spiel **Othello** (auf Deutsch: **Reversi**) beschrieben.

Die ebenfalls komplexe Anwendung von TD(λ) auf das Spiel **Connect-4** wird in den Veröffentlichungen Thill et al. [2012], Thill [2012], Thill et al. [2014], Konen and Koch [2014], Bagheri et al. [2015], Thill [2015] beschrieben. Hier werden ebenfalls die N-Tuple-Features eingesetzt, dies führt auf große Netze mit mehreren Millionen Gewichten. Die Autoren zeigen, dass diese Netze dennoch mit TD(λ) und Eligibility Traces effizient trainiert werden können.

Die zugehörige Java-Software ist als Open Source von Github verfügbar: Thill and Konen [2014]. Die Software besitzt eine effiziente Implementierung der Eligibility Traces für Millionen Gewichte. Ferner erlaubt sie den Vergleich verschiedener online-adaptiver Lernverfahren (IDBD, Autostep, Temporal Coherence, K1 u.a.), siehe Bagheri et al. [2015], Konen and Koch [2014] für weitere Details.

7 Fazit

Der für die Praxis eines Brettspiel lernenden Agenten relevanteste Algorithmus ist die Variante „Self-Play“: Inkrementeller TD(λ)-Algorithmus (Kap. 4.3). Er ist so formuliert, dass sich der Pseudo-Code aus Sutton and Bonde [1992], 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-Vektor (s. Anhang B).

Wie die zahlreichen Anwendungen in Kap. 6 zeigen, stellt der TD(λ)-Algorithmus ein mächtiges Werkzeug im Bereich des Game Learnings dar.

Anhang

A Eligibility Traces

Dieser Anhang versucht, eine kurze, relativ verständliche Erklärung für die Methode der Eligibility Traces zu geben.

Ein Problem des Temporal Difference Learning ist, dass eine gute finale Bewertung $r(\boldsymbol{s}_N)$ erst durch Lernen auf $V(\boldsymbol{s}_{N-1})$ übertragen werden muss. Erst wenn dies erfolgt ist, kann $V(\boldsymbol{s}_{N-2})$ entsprechend lernen, usw. Dadurch wird das Lernen langsam und schwerfällig.

Es wären auch andere Wege denkbar. Einer wird von Sutton and Barto [1998] als sog. *Monte-Carlo-Methode* beschrieben (s. auch Thill et al. [2014]). Hier wartet man erst das Ende eines Spiels und den finalen Reward $r(\boldsymbol{s}_N)$ ab, definiert damit ein Fehlersignal $\delta_t = r(\boldsymbol{s}_N) - V(\boldsymbol{s}_t)$ für alle im Spiel durchlaufenen Zustände. Erst dann, wenn der finale Reward feststeht, führt man Lernschritte analog zu Gl. (6) für alle Zustände $\boldsymbol{s}_t$ mit diesem δ_t durch. Diese Methode ist prinzipiell gut, allein ist sie in der Praxis nur unhandlich umzusetzen, weil man sich alle Zustände und ihre Gradienten bis zum Ende eines Spiels merken müsste. Ferner erfolgen dann am Ende N Updates hintereinander und etwaige Auswirkungen, die das 1., 2., ... Update auf die nachfolgenden Updates haben würde, bleiben unberücksichtigt.

Schauen wir uns daher anhand des TD(0)-Lernens (Lernen ohne 'aktive' Eligibility Traces) noch einmal genauer an, warum diese Methode in der Regel nur „langsam und schwerfällig“ lernt. Betrachten wir dazu als Beispiel ein Spiel mit N Spielzügen, das schließlich mit einem Reward $r(\boldsymbol{s}_N)$ endet. Nehmen wir an, ein bestimmtes Gewicht w_i wäre an allen Spielzügen $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_N$ beteiligt.[2] Es müsste also eigentlich N-mal einen Update $u_t = \alpha\delta_t\nabla_w V$ erfahren. Wenn nun aber die Spielfunktion noch untrainiert[3] ist, lauten die Antworten $V(\boldsymbol{s}_0) = \ldots = V(\boldsymbol{s}_{N-1}) \approx 0$. Dann ist auch das Fehlersignal $\delta_t = V(\boldsymbol{s}_{t+1}) - V(\boldsymbol{s}_t)$ und damit das Update u_t gleich Null. Erst im letzten Schritt wird das Fehlersignal zu $\delta_N = r(\boldsymbol{s}_N) - V(\boldsymbol{s}_{N-1}) \approx r(\boldsymbol{s}_N)$ und es erfolgt **einmal** ein Update $u_N = \alpha\delta_N\nabla_w V$ mit $\delta_N \approx r(\boldsymbol{s}_N)$. Das bremst offensichtlich die Lerngeschwindigkeit ganz gewaltig aus.

[2] Mit 'beteiligt' meinen wir, dass das Gewicht in allen Spielzügen wesentlich aktiviert ist. Beispielsweise hat man im Game Learning oft binäre Features $x_i = \boldsymbol{g}_i(\boldsymbol{s}_t) \in \{0, 1\}$. Das zugehörige Gewicht w_i ist im Fall einer linearen Spielfunktion an einem Spielzug $\boldsymbol{s}_t$ entweder vollständig beteiligt ($\nabla_w V = x_i = 1$) oder unbeteiligt ($\nabla_w V = x_i = 0$).

[3] Dann besitzt die Spielfunktion nur kleine, zufällige Gewichte, und die Summe der Gewichte liegt mit hoher Wahrscheinlichkeit nahe 0.

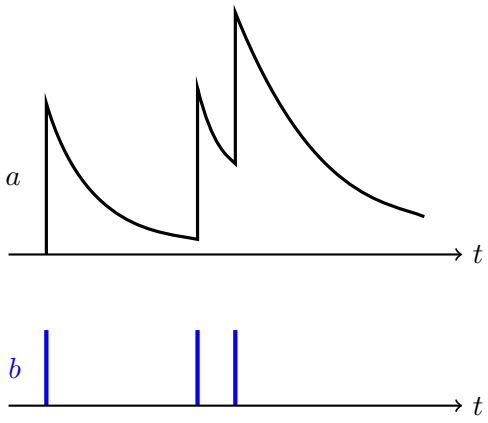

Abbildung 2: Beispielhafter Zeitverlauf einer Eligibility Trace (a), wenn das zugehörige Gewicht an einzelnen Zeitpunkten (b) eine Aktivierung erhält. Im hier dargestellten Fall für $\lambda < 1$ haben wir exponentiell abklingende Kurven, im Falle $\lambda = 1$ hätten wir eine ansteigende Stufenfunktion.

Die Eligibility Traces sind nun ein Weg, dies zu reparieren. Es ist zunächst festzuhalten, dass die Dimension des Vektors $\boldsymbol{e}_t$ der Eligibility Traces gleich der Dimension des Vektors $\boldsymbol{w}_t$ der Gewichte ist. Jedes Gewicht hat also seine eigene Eligibility Trace. Der Name, der übersetzt so viel wie „Qualifizierungs-Spur“ bedeutet, sagt aus, dass ein Gewicht durch vorherige Aktivierungen in einen „qualifizierten“[4] Zustand übergehen kann.

Um zu erklären, wie Eligibilty Traces wirken, betrachten wir den Spezialfall $\lambda = \gamma = 1$, dies entspricht am ehesten den sog. *Monte-Carlo-Methoden*. Weil das Gewicht w_i in unserem Beispiel an jedem Spielzug beteiligt ist, wird gemäß Gl. (7) in jedem Schritt ein $\nabla_w V$ auf *seine* Eligibility Trace aufaddiert:

$$\boldsymbol{e}_{t+1} = \boldsymbol{e}_t + \nabla_{\boldsymbol{w}} V$$

Wenn wir vereinfachend annehmen, dass der Gradient in jedem Schritt die gleiche Größe hätte,[5] erhalten wir also schließlich $e_{N-1} = N\nabla_{\boldsymbol{w}} V$. Wenn wir damit nun in den letzten Lernschritt nach Gl. (6) gehen und $\delta_N \approx r(\boldsymbol{s}_N)$ berücksichtigen:

$$\begin{aligned} \boldsymbol{w}_N &= \boldsymbol{w}_{N-1} + \alpha\delta_t e_{N-1} \\ &= \boldsymbol{w}_{N-1} + \alpha r_N N \nabla_{\boldsymbol{w}} V \end{aligned}$$

so erhalten wir in Summe über die gesamte Episode – wie gefordert und wie beim Monte-Carlo-Verfahren der Fall – einen Update von N-mal $u_t = \alpha r_N \nabla_w V$.

Dies war die Erklärung des TD(1)-Verfahrens. Das allgemeine TD(λ)-Verfahren bietet mit dem Parameter $\lambda \in [0, 1]$ nun die Möglichkeit, stufenlos zwischen klassischen TD-Methoden ($\lambda = 0$) und Monte-Carlo-Methoden ($\lambda = 1$) auszuwählen. Die TD(1)-Methode vermeidet gegenüber der ursprünglichen Monte-Carlo-Methode den Nachteil des komplizierten „Buchhaltens“ über vergangene Zustände. TD(1) sammelt die für die Gewichtsänderung notwendigen Änderungen auf und appliziert sie schließlich in einem Schritt (dem Schritt, in dem der Reward verfügbar wird).

Ist nun TD(1) unter allen möglichen TD(λ)-Methoden immer die beste Methode? – Das ist nicht der Fall, denn eine TD(1)-Methode summiert – ähnlich der Monte-Carlo-Methode – alle Änderungen

[4] im Sinne von „zur Änderung besonders berechtigten“ Zustand

[5] Dies ist bei binären Merkmalen $x_i = \boldsymbol{g}_i(\boldsymbol{s}_t) \in \{0, 1\}$ und linearer Spielfunktion auch exakt der Fall.

für ein Gewicht auf und appliziert sie in einem Schritt. Mögliche Querbeziehungen, die der 1., 2., ... Lernschritt auf die nachfolgenden Lernschritte haben würden, bleiben unberücksichtigt. In der Praxis ist es oft so, dass ein TD(λ)-Verfahren mit $\lambda < 1$ optimal ist. Welches das richtige λ ist, bleibt oft dem Experimentieren überlassen. In vielen Anwendungen werden die besten Ergebnisse mit einem $\lambda \in [0.7, 0.9]$ erzielt.

Bei einem TD(λ) mit $\lambda < 1$ werden weiter in der Vergangenheit liegende Aktivierungen eines Gewichtes zunehmend vergessen. Jede vergangene Aktivierung erzeugt eine exponentiell abklingende Spur in der jeweiligen Eligibility Trace (Fig. 2).

Weitere Erklärungen zu Eligibility Traces und ihre Anwendung auf ein komplexes Lernproblem (Connect-4) sind in Thill et al. [2014] zu finden. Hier werden auch weitere Eligibility-Trace-Varianten (*replacing traces*, *resetting traces*) erklärt.

B Typische Funktionsapproximatoren

Typische Funktionsapproximatoren für die Spielfunktion in Kap. 4.1.2 sind:

1. **Lineare Funktion ohne Output-Sigmoid**:

$$f(\boldsymbol{w}; \boldsymbol{s}_t) = \boldsymbol{w} \cdot \boldsymbol{g}(\boldsymbol{s}_t) = \sum_k w_k g_k(\boldsymbol{s}_t)$$

   - Der Gradient lautet in diesem Fall $\nabla_{\boldsymbol{w}} f(\boldsymbol{w}; \boldsymbol{s}_t) = \boldsymbol{g}(\boldsymbol{s}_t)$.
   - $\boldsymbol{g}(\boldsymbol{s}_t)$ ist entweder ein Feature-Vektor beliebiger Länge, der aus dem Zustand $\boldsymbol{s}_t$ gebildet wird, oder es ist einfach der Zustand $\boldsymbol{s}_t$ selbst.

2. **Lineare Funktion mit Output-Sigmoid**:

$$f(\boldsymbol{w}; \boldsymbol{s}_t) = \sigma(\boldsymbol{w} \cdot \boldsymbol{g}(\boldsymbol{s}_t)) = \sigma\left(\sum_k w_k g_k(\boldsymbol{s}_t)\right)$$

   - Die Output-Sigmoid ist z. B. die Fermi-Funktion

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

     Diese besitzt die Ableitung $\sigma'(y) = \sigma(y)(1 - \sigma(y))$.
   - Der Gradient lautet in diesem Fall $\nabla_{\boldsymbol{w}} f(\boldsymbol{w}; \boldsymbol{s}_t) = f(1 - f)\boldsymbol{g}(\boldsymbol{s}_t)$.

3. **Neuronales Netz mit Output-Sigmoid**:

$$f(w_j, v_{ji}; \boldsymbol{s}_t) = \sigma\left(\sum_{j=1}^{H} w_j h_j\right) \quad \text{mit} \quad h_j = \sigma\left(\sum_i v_{ji} g_i(\boldsymbol{s}_t)\right)$$

   - Das neuronale Netz besitzt H Hidden-Neurone, ein Output-Neuron, Hidden-zu-Output-Gewichte w_j und Input-zu-Hidden-Gewichte v_{ji}.

- Die Output-Sigmoid ist z. B. die Fermi-Funktion

$$\sigma(y) = \frac{1}{1+e^{-y}}$$

  Diese besitzt die Ableitung $\sigma'(y) = \sigma(y)(1-\sigma(y))$.
- Die Gradienten lautet in diesem Fall

$$\frac{\partial f}{\partial w_j} = f(1-f)h_j \tag{9}$$

$$\frac{\partial f}{\partial v_{ji}} = \frac{\partial f}{\partial h_j} \cdot \frac{\partial h_j}{\partial v_{ji}} = f(1-f)w_j \cdot h_j(1-h_j)x_i \tag{10}$$

Literatur

S. Bagheri, M. Thill, P. Koch, and W. Konen. Online adaptable learning rates for the game Connect-4. *IEEE Transactions on Computational Intelligence and AI in Games*, (accepted 11/2014):1, 2015. doi: http://dx.doi.org/10.1109/TCIAIG.2014.2367105. URL http://www.gm.fh-koeln.de/~konen/Publikationen/Bagh15.pdf. 9

W. Konen and T. Bartz-Beielstein. Reinforcement learning: Insights from interesting failures in parameter selection. In G. Rudolph, editor, *Proc. Parallel Problem Solving From Nature (PPSN'2008)*, pages 478–487. Springer, Berlin, 2008. 9

W. Konen and T. Bartz-Beielstein. Reinforcement learning for games: failures and successes – CMA-ES and TDL in comparision. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2641–2648, Montreal, Québec, Canada, 2009. ACM. URL http://www.gm.fh-koeln.de/~konen/Publikationen/evo-reinforce-GECCO2009.pdf. 9

W. Konen and P. Koch. Adaptation in nonlinear learning models for nonstationary tasks. In Bogdan Filipic, editor, *PPSN'2014: 13th International Conference on Parallel Problem Solving From Nature, Ljubljana*, Heidelberg, 2014. Springer. URL http://www.gm.fh-koeln.de/ciopwebpub/Kone14a.d/Kone14a.pdf. 9

S. Lucas. Investigating learning rates for evolution and temporal difference learning. In *Proc. IEEE Symposium on Computational Intelligence and Games CIG2008*, Perth, Australia, December 2008a. IEEE Press. 9

S. M. Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008b. 9

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. 1, 4, 5, 6, 10, 14

R. S. Sutton and A. Bonde. Nonlinear TD/Backprop pseudo C-code. Technical report, GTE Laboratories, 1992. URL http://webdocs.cs.ualberta.ca/~sutton/td-backprop-pseudo-code.text. 1, 6, 10, 14

G. Tesauro. Practical issues in temporal difference learning. *Mach. Learning*, 8:257–277, 1992. 1, 4

G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994. 14

M. Thill. Using n-tuple systems with TD learning for strategic board games (in German). CIOP Report 01/12, Cologne University of Applied Science, 2012. URL http://www.gm.fh-koeln.de/ciopwebpub/Thill12a.d/Thill12a.pdf. 9

M. Thill. Temporal difference learning methods with automatic step-size adaption for strategic board games: Connect-4 and Dots-and-Boxes. Master thesis, Cologne University of Applied Sciences, June 2015. URL http://www.gm.fh-koeln.de/~konen/research/PaperPDF/MT-Thill2015-final.pdf. 9

M. Thill and W. Konen. Connect-4 game playing framework (C4GPF), October 2014. URL http://github.com/MarkusThill/Connect-Four. 9

M. Thill, P. Koch, and W. Konen. Reinforcement learning with n-tuples on the game Connect-4. In Carlos Coello Coello, Vincenzo Cutello, et al., editors, *PPSN'2012: 12th International Conference on Parallel Problem Solving From Nature, Taormina*, pages 184–194, Heidelberg, 2012. Springer. URL http://www.gm.fh-koeln.de/ciopwebpub/Thi12.d/Thil12.pdf. 9

M. Thill, S. Bagheri, P. Koch, and W. Konen. Temporal difference learning with eligibility traces for the game Connect-4. In Mike Preuss and Günther Rudolph, editors, *CIG'2014, International Conference on Computational Intelligence in Games, Dortmund*, 2014. URL http://www.gm.fh-koeln.de/~konen/Publikationen/ThillCIG2014.pdf. 9, 10, 12

Sutton and Bonde [1992] ist eine 'fast' fertige TD(λ)-Implementierung, erstaunlich kurz (!), es fehlt nur I/O und Random Number Generator. Die Theorie dahinter ist in Sutton and Barto [1998], 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.

Tesauro [1994] ist 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.