e-print http://www.gm.fh-koeln.de/ciopwebpub/Kone15c.d/TR-TDgame_EN.pdf, July 2015

Reinforcement Learning for Board Games: The Temporal Difference Algorithm

Wolfgang Konen

Cologne University of Applied Sciences
Steinmüllerallee 1, D-51643 Gummersbach,
Germany

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

July 2015 (Update May 2022)

Abstract

This technical report shows how the ideas of reinforcement learning (RL) and temporal difference (TD) learning can be applied to board games. This report collects the main ideas from Sutton and Barto [1998], Tesauro [1992] and Sutton and Bonde [1992] in a compact form and gives hints for the practical application.

It contains a section on 'Self-Play' TD algorithms, a section on game-learning applications for TD(λ), furthermore appendices on eligibility traces and typical function approximators.

Contents

1 Introduction **2**

2 States und After-States **2**

3 The Value Function **3**

4 The Temporal Difference (TD) Algorithm **4**

4.1 Basic ideas 4

4.1.1 Temporal Difference 4

4.1.2 Function Approximation 4

4.1.3 Feature Vector 5

4.2 The TD(λ) Algorithm 6

4.3 „Self-Play“ : Incremental TD(λ) Algorithm 7

5 Hints for Practical Applications **8**

6 Applications of TD(λ) and Self-Play **8**

7 Conclusion **9**

A Eligibility Traces **9**

B Typical Function Approximators **11**

C Q-Self-Play **12**

1 Introduction

Reinforcement learning (RL) is a powerful optimization method for complex problems. It has special advantages if not every single action is accompanied by a reward, but only after a sequence of actions a certain reward (or punishment) can be given. This is the typical situation in board games.

Temporal difference (TD) learning is a popular technique in reinforcement learning.

2 States und After-States

A game situation in board games like chess, go, Connect-4, TicTacToe or Nimm-3 is usually described by

- the information who's move it is
- the position on the board.

Both elements together form the **state** s_t of the game. Here $t = 0, 1, 2, \ldots, N$ denotes the sequence of moves (more correctly half-moves, a. k. a. **ply**). In TicTacToe N is at most 9, but the game can be over in less than 9 moves as well.

When coding the game situation we can distinguish the state and the after-state variant:

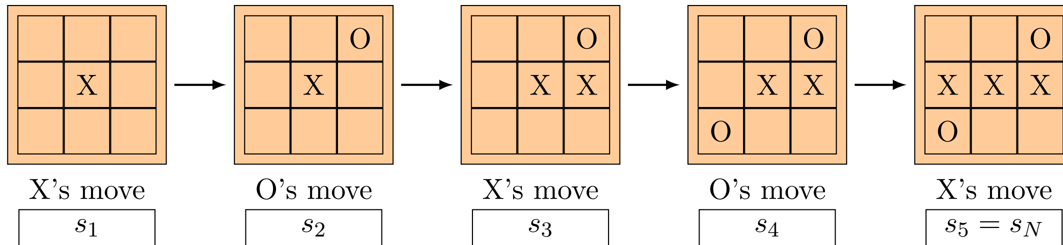

Figure 1: Exemplary course of a game for TicTacToe. The game is finished after $N = 5$ moves.

- State coding:
  - position before the player has made his move
  - information who's move it is
- **After-State** coding:
  - position after the player has made his move
  - information who made the move.

In most cases the after-state coding is advantageous, because for board games it is usually only important which is the board **after** the move. Multiple (state+move)-combinations may lead to the same after-state, but each (state+move)-combination leads exactly to one after-state. The complexity is reduced if we take the after-state perspective.

Let us take a specific course of a game: It can always be represented as a sequence of states $\boldsymbol{s}_t$ = {after-state + who made the move}. Fig. 1 shows a TicTacToe-example for a course of a game.

3 The Value Function

We cover here 1- and 2-player games. In 1-player games we have only $p = 1$ (White), in 2-player games we have $p = 1$ (White) and $p = -1$ (Black).

The goal of reinforcement learning is to train an agent from a (usually large) set of such game sequences. We would have an optimal agent, if we knew for every state $\boldsymbol{s}_t$ the **value function** $V(\boldsymbol{s}_t)$. This value function should give those states being advantageous for White (X) a high value and those states being advantageous for Black (O) a low value.[1] White will evaluate for all possible after-states $\boldsymbol{s}_{t+1}$ the value function $V(\boldsymbol{s}_{t+1})$ and takes the one with the highest value. Black evaluates the same value function, but she takes the lowest value.

The problem is: We do not have this ideal value function, and there is usually no direct approach to calculate it. However, we can assign to every *final state* $\boldsymbol{s}_N$ of a specific game the reward belonging to it. In the example Fig. 1 above it is the reward $r(\boldsymbol{s}_5) = r(\boldsymbol{s}_N) = +1$, because X has won. In the opposite case, if O had won, the reward would be $r(\boldsymbol{s}_N) = 0$ and $r(\boldsymbol{s}_N) = 0.5$, if it is a draw. The value of the finale state $\boldsymbol{s}_N$ in the figure above is thus $= +1$. The value function $V(\boldsymbol{s}_4)$ in the example above should be ideally $V(\boldsymbol{s}_4) = +1$ as well, since if O leaves this after-state to his opponent, an optimally playing agent X will win for sure. A learning system can in this way

[1] How high or how low is usually irrelevant as long as the best after-state has the highest / lowest score.

learn in a deductive way successively „from behind“ which positions are „good “ and which are „bad “. This is the central idea of temporal difference learning.

The ideal value function $V(\boldsymbol{s}_t)$ contains in the end **the probabilty that perfectly playing agents will reach from $\boldsymbol{s}_t$ a win for X** (Tesauro [1992]). More generally speaking – not only for games – the value function $V(\boldsymbol{s}_t)$ should code the **sum of the future rewards** in this episode starting from state $\boldsymbol{s}_t$ (the expected value of the sum of those rewards).

4 The Temporal Difference (TD) Algorithm

4.1 Basic ideas

The goal is to learn the value function $V(\boldsymbol{s}_t)$. We are facing here two major challenges:

1. Except for the terminal state s_N, it is not known what the teacher signal for $V(\boldsymbol{s}_t)$ should be.

2. For non-trivial games the space of possible states $\{\boldsymbol{s}_t\}$ is vastly big, so that it is not possible to have them all in a big table or to visit them all sufficiently often.

Solutions for both problems are described in Sutton & Barto's influential book Sutton and Barto [1998].

4.1.1 Temporal Difference

Thw first problem is solved by reinforcement learning, which defines an error signal δ_t for state $V(\boldsymbol{s}_t)$:

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

The error signal vanishes if $V(\boldsymbol{s}_t)$ reaches the target $r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1})$. The algorithm waits for the next state $\boldsymbol{s}_{t+1}$ and looks if for that state the reward or the value function is known. If so, it changes $V(\boldsymbol{s}_t)$ in that direction. Parameter γ (typically 0.9) is the so-called **discount factor**: It takes into account that a state further away in the future, say $V(s_{t+10}) = 1$, might be a possible successor of $\boldsymbol{s}_t$ in the course of the game, but it is not guarenteed that the game path (for all course of games) leads from $\boldsymbol{s}_t$ to s_{t+10}. Thus the game states more distant in the future are *discounted* in their effect on $V(\boldsymbol{s}_t)$.

This clarifies the name „TD“: Since for most states the reward is zero, the error signal is in most cases the temporal difference in the value function and the algorithm based on that is called the **temporal difference (TD)** algorithm.

4.1.2 Function Approximation

The second problem is solved by function approximation [Sutton and Barto, 1998, Sec. 8]: Instead of putting all values $V(\boldsymbol{s}_t)$ into a (gigantic) table, a function $f(\boldsymbol{w}; \boldsymbol{s}_t)$ with free parameters $\boldsymbol{w}$ (the *weights*) is defined such that

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

is approximated in the best possible way. Typical realisations for $f(\boldsymbol{w}; \boldsymbol{s}_t)$ are:

- a neural net with $\boldsymbol{s}_t$ as input, a hidden layer, weights $\boldsymbol{w}$ and one output neuron

- a neural net with feature vector $\boldsymbol{g}(\boldsymbol{s}_t)$ as input
- a linear function: $f(\boldsymbol{w};\boldsymbol{s}_t) = \boldsymbol{w}\cdot\boldsymbol{s}_t = \sum_k w_k s_{tk}$
- a linear function with feature vector $\boldsymbol{g}(\boldsymbol{s}_t)$ as input:

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

With function approximation, a change in $\boldsymbol{w}$ which diminuishes the error signal $\delta_t = (V(\boldsymbol{s}_t) - f(\boldsymbol{w};\boldsymbol{s}_t))^2$ for a certain $\boldsymbol{s}_t$ will at the same time change other values $f(\boldsymbol{w};s_u), u \neq t$. But this is often a desirable effect, becaus not all states can be visited anyway. This property is known as **generalization**: transfer knowledge from visited states and their rewards to other, similar, non-visited states.

There is a peculiarity to be pointed out: Due to the generalization effect it will be usually the case that for a final state $\boldsymbol{s}_N$ the value function $V(\boldsymbol{s}_N)$ is not zero. This contradicts the notion of the value function $V(\boldsymbol{s}_t)$ which should reflect for a given state $\boldsymbol{s}_t$ the sum of rewards to be expected in the future. Since it is a *final* state, the sum has to be zero. In order not to count something twice, one defines usually:

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

4.1.3 Feature Vector

The approximation of V by f will be usually easier if neighboring states in input space have similar outputs. This is the reason why often **feature vectors $\boldsymbol{g}()$** are introduced in the realization of $f(\boldsymbol{w};\boldsymbol{s}_t)$. A „difficult“ input-output mapping may become much easier if the right feature is found, as the following example demonstrates.

Example feature vector: The game Nimm-3. The game Nimm-3 has the goal to capture the final piece from a pile of pieces, when each player in turn has the right to grab 1, 2 or 3 pieces. The well-known optimal strategy is to seek for an after-state with the sum of pieces divisible by 4. This can be expressed with the feature

$$g_4(\boldsymbol{s}_t) = (\boldsymbol{s}_t \text{ is divisible by } 4\ ?\ 1 : 0). \tag{4}$$

This feature in itself is already a perfect coding of the value function after a move of White. (We recall: The value function should code the probability that White will win from that state. If on the other hand $\boldsymbol{s}_t$ is the after-state for Black, one has only to swap 1 and 0 in Eq. (4).)

For more complex games such a perfect feature is normally not possible or it is not known at the time of coding. If one has the assumption that divisibility plays a role in general for a game, one can define the feature vector $\boldsymbol{g}(\boldsymbol{s}_t) = (g_k(\boldsymbol{s}_t)), k = 2,\ldots,M$ with

$$g_k(\boldsymbol{s}_t) = (\boldsymbol{s}_t \text{ is divisible by } k\ ?\ 1 : 0) \tag{5}$$

Then the learning algorithm should learn for Nimm-3 in the course of time that only g_4 is important and all other $g_k, k \neq 4$ are of no importance.

Algorithm 1 TD(λ) algorithm. Input: The course of the game $\{\boldsymbol{s}_t | t = 0, 1, \ldots, N\}$ and the associated rewards $r(\boldsymbol{s}_t)$. A function $f(\boldsymbol{w}_0; \boldsymbol{s}_t)$ with (partially trained) weights $\boldsymbol{w}_0$ to approximate the value function $V(\boldsymbol{s}_t)$. Output: improved weights $\boldsymbol{w}_N$.

```
function TDLTrain({s_t | t = 1, ..., N}, w_0)
    e_0 ← ∇_{w_0} f(w_0; s_0)                              ▷ Initial eligibility traces
    for (t ← 0 ; t < N; t ← t + 1) do
        V_old ← f(w_t; s_t)                                 ▷ Value function current state
        V(s_{t+1}) ← { 0,                 if s_{t+1} final   ▷ Value function next state
                     { f(w_t; s_{t+1}),   otherwise
        δ_t ← r(s_{t+1}) + γV(s_{t+1}) − V_old             ▷ Error signal
        w_{t+1} ← w_t + αδ_t e_t                            ▷ Learning step
        e_{t+1} ← γλe_t + ∇_w f(w_{t+1}; s_{t+1})           ▷ Eligibility traces
    end for
    return w_N
end function
```

4.2 The TD(λ) Algorithm

Finally we need an update rule (learning rule) for the weights $\boldsymbol{w}$. Such a rule is given in [Sutton and Barto, 1998, Sec. 8.2]:

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

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

The vector $\boldsymbol{e}_t$ contains the so-called **eligibility traces** for each weight. The theory behind eligibility traces is not that easy for $\lambda > 0$. Some explanations are found in Appendix A. Let us restrict ourselves here to the case $\lambda = 0$: Then $\boldsymbol{e}_t$ is nothing else than the gradient of the function $V(\boldsymbol{s}_t) = f(\boldsymbol{w}_t; \boldsymbol{s}_t)$. The gradient points into the direction of steepest ascent. If the error signal δ_t in Eq. (1) is positive then we move with a change according to Eq. (6) up the hill, i. e. $V(\boldsymbol{s}_t)$ is increased. If δ_t is negative we move down the hill, i. e. $V(\boldsymbol{s}_t)$ is reduced. If we do not move too far – this is controlled by the learning step size α – then the error signal in Eq. (1) should be reduced in every step. This technique is therefore known as „**gradient descent**".

We now have everything together to formulate the TD(λ) algorithm according to [Sutton and Barto, 1998, Sec. 8.2] and Sutton and Bonde [1992] in Algorithm 1.

Annotations for Algorithm 1:

- Each learning step will move $V(\boldsymbol{s}_t) = f(\boldsymbol{w}_t, \boldsymbol{s}_t)$ closer to the target $r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1})$ (if α is not too big).
- It is important (and sometimes forgotten in implementations of TD(λ)), that the response has to be recalculated after each learning step because the weights have changed. Therefore we recalculate V_{old} in every pass (and do not re-use $V(\boldsymbol{s}_{t+1})$ from the previous pass through the loop).
- Algorithm 1 is formulated in such a way that it corresponds nearly line-by-line with the `main()` function from [SuttonBonde93]. Typical parameters are $\gamma = 0.9, \lambda < \gamma, \alpha = 0.1$.
- In case $\lambda = 0$ the algorithm reduces to the usual gradient descent.

4.3 „Self-Play“ : Incremental TD(λ) Algorithm

We now consider the case where only the game environment is given, but no courses of games from an external source are provided. In that case we have to change the algorithm in such a way that the algorithm *generates* many courses of games $\{\boldsymbol{s}_t | \, t = 0, 1, \ldots, N\}$ as the learning incrementally progresses. This self-play algorithm is shown in Algorithm 2.

Function SELFPLAY is called repeatedly, each time with an initial position $\boldsymbol{s}_0$ and with the learned weights returned from the previous call. (In the first call we start with randomly initialised weights.) SELFPLAY performs a full game until a final state $\boldsymbol{s}_t \in S_{Final}$ is reached.

Function SELFPLAY is extended in two points with respect to Algorithm 1: 1) The next state $\boldsymbol{s}_{t+1}$ is choosen by the algorithm itself (either a random move or a greedy move exploiting the value function learned so far). 2) The error signal is usually set to zero in case of a random move.

Algorithm 2 „Self-Play“: Incremental TD(λ) algorithm for 1- and 2-player board games. Input: Starting player p [$= +1$ (White) or -1 (Black)], R: number of players, $\boldsymbol{s}_0$: initial position. Furthermore, (partially trained) weights $\boldsymbol{w}_0$ for function $f(\boldsymbol{w}; \boldsymbol{s}_t)$ approximating the value function $V(\boldsymbol{s}_t)$. Reward function $r(\boldsymbol{s}_t)$ comes from the game-playing environment. Output: Improved weights $\boldsymbol{w}_{t+1}$ at end of episode.

---

function SELFPLAY(p, R, $\boldsymbol{s}_0$, $\boldsymbol{w}_0$)
    $\boldsymbol{e}_0 \leftarrow \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_0; \boldsymbol{s}_0)$ ▷ Initial eligibility traces
    for $\left(t \leftarrow 0 \; ; \; \boldsymbol{s}_t \notin S_{Final} \; ; \; t \leftarrow t+1\right)$ **do** ▷ S_{Final}: set of all final states
        $V_{old} \leftarrow f(\boldsymbol{w}_t, \boldsymbol{s}_t)$ ▷ Value function current state
        Choose randomly $q \in [0, 1]$
        if $(q < \epsilon)$ **then**
            Choose a random $\boldsymbol{s}_{t+1}$ ▷ Explorative move (random move)
        else
            Choose a legal after-state s_{t+1} such that ▷ Greedy move
                $p \cdot (r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}))$
            is maximized.
        end if

$$V(\boldsymbol{s}_{t+1}) \leftarrow \begin{cases} 0, & \text{if } \boldsymbol{s}_{t+1} \in S_{Final} \\ f(\boldsymbol{w}_t; \boldsymbol{s}_{t+1}), & \text{otherwise} \end{cases}$$ ▷ Value function next state

$$\delta_t \leftarrow \begin{cases} 0, & \text{if } q < \epsilon \wedge \boldsymbol{s}_{t+1} \notin S_{Final} \\ r(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}) - V_{old}, & \text{else} \end{cases}$$ ▷ Error signal

        $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{e}_t$ ▷ Learning step
        $\boldsymbol{e}_{t+1} \leftarrow \gamma \lambda \boldsymbol{e}_t + \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_{t+1}; \boldsymbol{s}_{t+1})$ ▷ Eligibility traces
        $p \leftarrow (-p)^R$
    end for
    return $\boldsymbol{w}_{t+1}$
end function

Annotations for Algorithm 2:

- Typically, function SELFPLAY is started with an empty board as starting position $\boldsymbol{s}_0$. Equally well another (randomly) starting position can be choosen to improve exploration.
- The random moves guarantee that not always the same course of game is undertaken. To

train a TD(λ) agent, the start weights are initialised randomly and function SELFPLAY is called many (e. g. 10000) times.

- Typically, the exploration rate ϵ and the learning rate α will be diminuished during training. This enforces that there is more exploration in the beginning, while in the end we have convergence to a (hopefully well-playing) agent.
- The FOR-loop will be left as soon as $\boldsymbol{s}_{t+1}$ is a final state. Note that the function approximator is never trained to move $V(\boldsymbol{s}_N)$ closer to $r(\boldsymbol{s}_N)$. Only the predecessors of $V(\boldsymbol{s}_N)$ are trained. The generalization effect might indirectly change $V(\boldsymbol{s}_N)$ as well, but this value is never used by the algorithm.
- Why is learning inhibited after a random move? – Because the random move is with high probability not the best move. Therefore it leads with high probabilty to a „bad“ $V(\boldsymbol{s}_{t+1})$ („bad“ for p). This would deteriorate $V(\boldsymbol{s}_t)$ if it were a winning state. This is certainly not a desired behaviour. Therefore we skip the learning step after a random move. The only exception from this rule is if $\boldsymbol{s}_{t+1}$ is a final state with a win for player p: Then the random move was obviously the best – or one of the best – choices. Therefore $V(\boldsymbol{s}_t)$ should be trained.

5 Hints for Practical Applications

- Apart from complex function approximators (e. g. backprop) one should always try the simple linear function approximator (possibly with feature vector) as well:

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

  The reason: The linear approximator learns more robustly (has no local minima) and often learns faster. Although a linear function cannot realize complex I/O-mappings, this disadvantage is often compensated when offering complex features in the input. (It is often more advisable to start with too many features than with too few!)
- It pays off to think thoroughly about the coding of the feature vector.
- Function approximators with no sigmoid function for the output neuron are sometimes better.

6 Applications of TD(λ) and Self-Play

The following overview of applications for TD-learning in games is by no means comprehensive. We list here only some of the literature relatively close to the algorithms presented.

An application of the Algorithms 1 and 2 presented here to simple games like **Nimm-3** or **TicTacToe**, including the evaluation of different feature vectors, is shown in Konen and Bartz-Beielstein [2008, 2009].

The associated Java software has been improved and extended continuously since 2008 and is now part of the GBG framework [Konen, 2022] which is available as open source from Github. The extensions include n-tuple features, temporal coherence and an MCTS agent for comparison.

The first application of n-tuple features for TD(λ)-based game learning was described in Lucas [2008a,b] für the complex game **Othello** (in German: Reversi).

The application of TD(λ) to the likewise complex game **Connect-4** is covered in the papers Thill et al. [2012], Thill [2012], Thill et al. [2014], Konen and Koch [2014], Bagheri et al. [2015], Thill [2015]. These papers use N-tuple features as well and this leads to large networks with several millions of weights. The authors show that such big networks nevertheless can be trained efficiently using TD(λ) and eligibility traces.

The associated Java software is available as open source from Github [Thill and Konen, 2014] and is now also part of the GBG framework [Konen, 2022]. This software offers an efficient implementation of eligibility traces for millions of weights. Furthermore, it allows the comparison of numerous online-adaptive learning methods (IDBD, Autostep, Temporal Coherence, K1 and others), see Bagheri et al. [2015], Konen and Koch [2014] for further details.

7 Conclusion

In practice, the most relevant algorithm for board games is the variant „Self-Play: Incremental TD(λ)“ (Sec. 4.3). It is formulated in such a way that the pseudo code from Sutton and Bonde [1992], which implements a simple neural network (Backprop) as function approximator, can be relatively easy incorporated.

Likewise, a function approximation with a linear function over a feature vector can be included. The gradient is in this case nothing else than the feature vector itself (see Appendix B).

As the numerous applications in Sec. 6 show, the TD(λ) algorithm is a powerful tool in the area of game learning.

Appendix

A Eligibility Traces

This appendix tries to give a short, relavitvely understandable explanation for the method of eligibility traces.

A problem in temporal difference learning is that a good final reward $r(\boldsymbol{s}_N)$ has first to be transferred by learning to $V(\boldsymbol{s}_{N-1})$. Only if this is done (or at least partially done), learning for $V(\boldsymbol{s}_{N-2})$ can proceed and so forth. This makes learning slowly and cumbersome.

There are other approaches. One is described in Sutton and Barto [1998] as the so-called *Monte Carlo method* (see also Thill et al. [2014]). In this method one waits first for the end of the game, collects the final reward $r(\boldsymbol{s}_N)$ and defines an error signal $\delta_t = r(\boldsymbol{s}_N) - V(\boldsymbol{s}_t)$ for all states $\boldsymbol{s}_t$ visited during this game. Only when these final error signals δ_t are available, the learnings steps for all $\boldsymbol{s}_t$ are carried out in analogy to Eq. (6). This method is good in principle, but unwieldy in practice, since all states and their gradients have to be memorized until the end of the game. Furthermore, at the end of the game N updates are undertaken in a row. Possible impacts of the 1st, 2nd, ... update on successive updates are not taken into account.

Let us look again at TD(0) (learning without eligibility traces) and analyze why this methods learns usually only „slowly and cumbersome“. As an example consider a game with N moves which finally ends with a reward $r(\boldsymbol{s}_N)$. Let us assume that a certain weight w_i is involved in every move

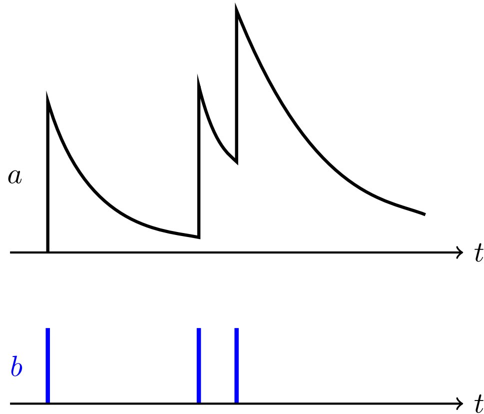

Figure 2: Exemplary time course of an eligibility trace (a), if the associated weight gets activations at certain points in time (b). In the case $\lambda < 1$ shown here we have exponentially decaying curves. In the case $\lambda = 1$ we would have an increasing step function.

$\boldsymbol{s}_1, \ldots, \boldsymbol{s}_N$.[2] Thus the weight ideally should get N times an update $u_t = \alpha\delta_t \nabla_w V$. But if the value function is not yet trained[3], then we get the answers $V(\boldsymbol{s}_0) = \ldots = V(\boldsymbol{s}_{N-1}) \approx 0$ from the value function evaluation. Consequently, the error signal $\delta_t = V(\boldsymbol{s}_{t+1}) - V(\boldsymbol{s}_t)$ and the update u_t will be zero as well. Only the last step will see a non-zero error signal $\delta_N = r(\boldsymbol{s}_N) - V(\boldsymbol{s}_{N-1}) \approx r(\boldsymbol{s}_N)$ and there is *once* an update $u_N = \alpha\delta_N \nabla_w V$ with $\delta_N \approx r(\boldsymbol{s}_N)$. This will slow down learning tremendously.

Eligibility traces are a way to cure this. Note at first that the dimension of the eligibility trace vector $\boldsymbol{e}_t$ is equal to the dimension of the weight vector $\boldsymbol{w}_t$. Every weight has its own eligibility trace. The name *eligibility trace* says that e_{ti} codes how *eligible* (for changes) a weight w_{ti} is due to previous activations of that same weight. e_{ti} contains a *trace* in time of all such earlier activations.

To explain how eligibility traces work, we look at the special case $\lambda = \gamma = 1$. This is most closely to the Monte-Carlo methods mentioned above. Since our weight w_i is involved in every move, it will receive acc. to Eq. (7) in every step an additive contribution $\nabla_w V$ to *its* eligibility trace:

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

If we assume for simplicity that the gradient has the same value in every step,[4] we get finally $e_{N-1} = N\nabla_{\boldsymbol{w}} V$. If we enter the final learning step Eq. (6) with this e_{N-1} and take $\delta_N \approx r(\boldsymbol{s}_N)$ into account we get:

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

That is, at the end of the complete episode we get a cumulative update N-times $u_t = \alpha r_N \nabla_w V$.

[2] With 'involved' we mean that this weight is substantial activated in every move. If we take for example a binary feature $x_i = \boldsymbol{g}_i(\boldsymbol{s}_t) \in \{0, 1\}$, then the associated weight w_i is in case of a linear function approximater either fully involved in a move $\boldsymbol{s}_t$ ($\nabla_w V = x_i = 1$) or fully uninvolved ($\nabla_w V = x_i = 0$).

[3] This means that the value function has small random weights and any weight sum is close to 0 with high probability.

[4] This is exactly the case for binary features $x_i = \boldsymbol{g}_i(\boldsymbol{s}_t) \in \{0, 1\}$ and a linear function approximator.

This is what we ideally wanted to have (see above) and what the Monte-Carlo method would deliver as well.

Now we have an understanding for the TD(1) method. The general TD(λ) method offers with parameter $\lambda \in [0,1]$ the possibility to steer continuously between classical TD methods ($\lambda = 0$) and Monte-Carlo methods ($\lambda = 1$). The TD(1) method has compared to the original Monte-Carlo method the big advantage that no complicated bookkeeping of states and gradients is necessary. TD(1) collects the individual weight change elements in the eligibilty trace and applies them finally in one step, when the reward becomes available.

Is TD(1) always the best among all possible TD(λ) methods? – This is *not* the case since the TD(1) method sums up – similar to the Monte-Carlo method – all changes for a weight and applies them in one step. Possible cross-relation effects which the 1st, 2nd, ... learning step could have on the successive learning steps remain unconsidered. In practice it is often the case that a TD(λ) method with $\lambda < 1$ is better. Finding the right λ is often a matter of experiments. In many applications, best results are achieved with $\lambda \in [0.7, 0.9]$.

When using TD(λ) with $\lambda < 1$ the activations more distant in the past will be „forgotten“ more and more. Every activation in the past creates an exponentially decaying trace in the correspoding eligibility trace (Fig. 2).

Further explanations on eligibility traces and their application to a complex learning problem (Connect-4) are presented in Thill et al. [2014]. This work explains in addition other eligibility trace variants (*replacing traces*, *resetting traces*).

B Typical Function Approximators

Typical function approximators for the value function in Sec. 4.1.2 are:

1. **Linear function without output sigmoid**:

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

   - The gradient is in this case $\nabla_{\boldsymbol{w}} f(\boldsymbol{w};\boldsymbol{s}_t) = \boldsymbol{g}(\boldsymbol{s}_t)$.
   - $\boldsymbol{g}(\boldsymbol{s}_t)$ is either a feature vector of arbitrary length formed out of the state $\boldsymbol{s}_t$, or it is simply $\boldsymbol{s}_t$ itself.

2. **Linear function with output sigmoid**:

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

   - The output sigmoid is for example the Fermi function

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

     which has the derivative $\sigma'(y) = \sigma(y)(1-\sigma(y))$.
   - The gradient is in this case $\nabla_{\boldsymbol{w}} f(\boldsymbol{w};\boldsymbol{s}_t) = f(1-f)\boldsymbol{g}(\boldsymbol{s}_t)$.

3. **Neural network with output sigmoid**:

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

- The neural network contains H hidden neurons, one output neuron, hidden-to-output weights w_j and input-to-hidden weights v_{ji}.
- The output sigmoid is for example the Fermi function

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

  which has the derivative $\sigma'(y) = \sigma(y)(1-\sigma(y))$.
- The gradient is in this case

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

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

C Q-Self-Play

Q-learning and SARSA [Sutton and Barto, 1998, Chapter 6.4 and 6.5] are the usual forms of reinforcement learning for control problems („Find an optimal policy or control for a certain task“). We show in this appendix that our „Self-Play“ algorithm is pretty close to Q-learning or SARSA. We give a concrete formulation of „Self-Play“ in Q-learning form in Algorithm 3, function QselfPlay.

Let $Q(s,a)$ be the value of the state-action pair (s,a). In many board games, a state and an action uniquely determine the after-state $s' = A(s,a)$.[5] In such cases we may write $Q(s,a) = V(A(s,a)) = f(w; A(s,a))$ and use the same approximating function as before. Whenever we have to calculate $Q(s,a)$ in Algorithm 3 we actually calculate $f(w; A(s,a))$.

The trick of Q-learning: Whatever we take as next action a, a greedy move or a random explorative move, we always set the learning target according to the greedy strategy („How would one have to change the weights, if Q-learning would follow from state $s' = A(s,a)$ the greedy strategy?“). This allows the algorithm to learn during explorative moves as well.

Algorithm 3 lists the complete function QselfPlay.

Remark: Sutton & Barto write in Chapter 7.6, that some care has to be taken with Q-learning and eligibility traces across exploratory moves. In principle one should stop learning for an episode once the first random move occurs. But in the end they say that the naive implementation used in Algorithm 3 is probably not so bad in practice anyhow.

References

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*, 8(1):33–42,

[5] This is for example the case in TicTacToe or Connect-4, but it is not the case in Backgammon, where rolling the dices introduces a source of randomness.

Algorithm 3 „Q-Self-Play“: Incremental Q-learning algorithm for 1- and 2-player board games. Input: Starting player p [$= +1$ (White) or -1 (Black)], R: number of players, $\boldsymbol{s}_0$: initial position. Furthermore, (partially trained) weights $\boldsymbol{w}_0$ for function $f(\boldsymbol{w}; \boldsymbol{s}_t)$ approximating the value function $V(\boldsymbol{s}_t)$. Reward function $r(\boldsymbol{s}_t)$ comes from the game-playing environment. Output: Improved weights $\boldsymbol{w}_{t+1}$ at end of episode.

function QSELFPLAY(p, R, $\boldsymbol{s}_0$, $\boldsymbol{w}_0$)
  $\boldsymbol{e}_{-1} \leftarrow 0$ ▷ eligibility traces are 0 before episode starts
  for $\big(t \leftarrow 0$ **;** $\boldsymbol{s}_t \notin S_{Final}$ **;** $t \leftarrow t+1\big)$ **do** ▷ S_{Final}: set of all final states
    Choose randomly $q \in [0,1]$
    if $(q < \epsilon)$ **then**
      Choose a random action $\boldsymbol{a}_t$ ▷ Explorative move (random move)
    else
      Choose a greedy action $\boldsymbol{a}_t$ maximizing $p \cdot Q(\boldsymbol{s}_t, \boldsymbol{a}_t)$ ▷ Greedy move
      (Skip illegal actions $\boldsymbol{a}_t$.)
    end if
    $Q_t \leftarrow Q(\boldsymbol{s}_t, \boldsymbol{a}_t) = f(\boldsymbol{w}_t, A(\boldsymbol{s}_t, \boldsymbol{a}_t))$ ▷ Q-value current state-action pair
    Take action $\boldsymbol{a}_t$, observe reward r_{t+1} and next state $\boldsymbol{s}_{t+1}$
    $Q_{t+1} \leftarrow \begin{cases} 0, & \text{if } \boldsymbol{s}_{t+1} \in S_{Final} \\ p \cdot \max_{a'} p \cdot Q(\boldsymbol{s}_{t+1}, a'), & \text{otherwise} \end{cases}$ ▷ Q-value next state-action pair
    $\delta_t \leftarrow r_{t+1} + \gamma Q_{t+1} - Q_t$ ▷ Error signal
    $\boldsymbol{e}_t \leftarrow \gamma\lambda \boldsymbol{e}_{t-1} + \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_t; A(\boldsymbol{s}_t, \boldsymbol{a}_t))$ ▷ Eligibility traces
    $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{e}_t$ ▷ Learning step
    $p \leftarrow (-p)^R$
  end for
  return $\boldsymbol{w}_{t+1}$
end function

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. GBG: The General Board Game playing and learning framework, May 2022. URL http://github.com/WolfgangKonen/GBG. 8, 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. 8

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. 8

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. 8

S. M. Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008b. 8

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998. 1, 4, 6, 9, 12, 15

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, 9, 15

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. 15

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, 11

Sutton and Bonde [1992] is a 'nearly' complete TD(λ) implementation, astonishingly short (!). Only I/O and random number generator are missing. The theory behind is explained in Sutton and Barto [1998], Sec. 6.1-6.4 (TD), (6.8: after states), 7.1-7.3 (eligibility traces), 8.1-8.2 (function approximation, gradient descent).

Tesauro [1994] is the famous TD-Gammon paper in which Tesauro shows that an RL agent with astonishingly little prior knowledge can learn to play the game backgammon on a world-class master level.