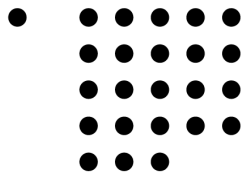

Fachhochschule Köln
Cologne University of Applied Sciences

Temporal Coherence in TD-Learning for Strategic Board Games

Case Study Report

Samineh Bagheri
Markus Thill

University of Applied Sciences Cologne
Campus Gummersbach
Faculty of Computer Science
and Engineering

In the Course of Studies
Master of Automation & IT

Supervisor: Wolfgang Konen

November 2014

Abstract

Although great progress has been made in the field of computational intelligence for games in the past decades, learning board games remained a complex task until today. Machine-learning techniques – such as Temporal Difference learning (TDL) – commonly have to deal with a variety of problems when they are applied to complex board games. Although Temporal Difference learning is known as one of the successful techniques for learning board games by self-play, the success is highly dependent on the correct selection of parameters. In [37], it was shown that even with a careful selection of the parameters for a game with moderate complexity like Connect-4, learning convergence only occurs after a several million of self-play games. In this work we investigate the impact of several online-adaptable learning rates on reducing the parameter selection dependency in order to speed up the learning process. We show, that Temporal Coherence Learning (TCL) and Incremental Delta Bar Delta (IDBD) have only a small impact on the learning speed. Later, we propose a more successful approach in speeding up the learning process, called TCL-EXP with 'geometric' learning rate changes. Additionally, we apply eligibility traces to our system for the first time. We found eligibility traces to be a very important ingredient in reducing the number of game plays required for training an almost perfect playing agent.

Keywords: Machine learning, board games, connect four, Tic Tac Toe , self-play, Reinforcement Learning, Temporal Difference Learning (TDL), temporal coherence, Q-learning, eligibility traces, Incremental Delta Bar Delta (IDBD), learning rates, self adaptation, online adaptation, n-tuple systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Learning

Board games often have simple rules but complex strategies and this is a reason which makes them a hot topic in artificial intelligence. Easy representation of the states and well defined rules in board games provide an ideal test-bed for benchmarking the decision making algorithms. Despite the broad research in machine learning, complex board games like chess are still far from being solved. However, a perfect playing agent for such a game is not found yet, current trained agents for many games are not competitive for professional human players.

We often have difficult decision making situations in board games because of a late payoff for any action. Every action will be graded based on its effect on the final game result. One of the most advanced methods in machine learning to address the mentioned issue is reinforcement learning (RL). The basic idea of RL is interaction with the environment through trial and error due to learning about the problem positive or negative rewards recieved after a sequence of actions, instead of hard-coding human knowledge about the problem. Temporal difference learning and Q-learning are known as thriving examples of reinforcement learning in various applications of artificial intelligence like robotics or board games.

In this work which is an extension of the work done by Thill in [36, 37], the successful agent is trained by temporal difference learning, which is able to win in more than 90 percent of the matches against a perfect playing Minimax agent. The search for an optimal playing agent is a sort of optimization problem. In contrast to conventional optimization tasks such as supervised learning, in board games there is no trivial objective function to assess the quality of an action. Therefore, we use self-play strategy and the agent plays many games against itself and learns from the experiences it makes. In other words, we are trying to imitate the learning procedure of a child with very constrained knowledge about the world around, which starts to learn from few interactions with the environment by trial and error.

As it is reported in [37], the temporal difference approach, used for learning the Connect-4 game, required a couple of million games and a very precise selection of parameters. This slow convergence is different from our expectation of this mimic of human learning process, because our knowledge about learning is still very limited. Humans are very efficient and fast in learning complicated tasks in new domains. Therefore, we can claim that there are missing elements in the TDL approach of [37]. As Sutton [30] has pointed out, information theoretic arguments suggest that the human learning progress is often too rapid to be justified by the data of the new domain alone. The key to success is, as it is commonly believed, that humans bring a set of correct biases from other domains into the new domain. These biases allow to learn faster, because biases direct them to prefer certain hypotheses over others or certain features over others. If machine learning algorithms can achieve a performance similar to humans, they probably need to acquire biases as well. Where do these biases come from?

Already in 1992 Sutton [30] introduced the Incremental Delta Bar Delta (IDBD) algorithm where the biases are understood as learning rates which can be different for different trainable parameters of any underlying algorithm. The key idea of IDBD is that these learning rates are not predefined by the algorithm designer but they are adapted as hyperparameters of the learning process themselves. Sutton [30] expected such adaptable learning rates to be especially useful for nonstationary tasks or sequences of related tasks and he demonstrated good results on a small synthetic nonstationary learning problem (featuring 20 weights). A similar idea was suggested as Temporal Coherence Learning (TCL) by Beal and Smith in 1999 [6, 7]. This work aimed at directly modifying the Temporal Difference Learning (TDL) algorithm to take into account self-tuning learning rates.

In this work we will present the result of the implementation of TCL and IDBD techniques on the Java framework developed by Thill in [36]. We show the impact of these techniques on the learning process. We also propose a new approach which is a variant of TCL and we call it TCL-EXP. The proposed technique shows higher impact on speeding up the learning process than the conventional TCL technique.

Although adding TCL-EXP approach yields a faster learning convergence than the results reported before, still half a million games are necessary to reach 80 percent of the max. success rate. The architecture of the problem and temporal difference learning is the one responsible for this late convergence. In many games play, updates happen only at the end of the game and many games are required to transfer the value gained in the last states to the initial states. Eligibility traces bring us a solution to overcome this problem by updating not only value of the state which has deserved a reward but all sequence of the states which yield to it. Of course, as further the

state is from the actual rewarded one the effect should be reduced. Eligibility traces implementation indicates a high impact on learning convergence.

1.2 Related Work

Several online learning rate adaptation schemes have been proposed over the years: IDBD [30] from Sutton is an extension of Jacobs' [13] earlier DBD algorithm: it allows immediate updates instead of batch updates. Sutton [31] proposed with the algorithms K1 and K2 some extensions to IDBD and compares them with the Least Mean Square (LMS) algorithm and Kalman filtering. Koop [15] used TDL methods for learning the game Go. She used the IDBD algorithm for online adaptation and investigates general aspects of temporal coherence. Almeida [2] discussed another method of step-size adaptation and applied it to the minimization of nonlinear functions. Schraudolph [23] extended on the K1 algorithm and showed that it is superior to Almeida [2].

For the game Connect-4 – although solved on the AI-level – only rather few attempts to *learn* it (whether by self-play or by learning from teachers) are found in the literature: Schneider et al. [22] tried to learn Connect-4 with a neural network, using an archive of saved games as teaching information. Sommerlund [26] applied TDL to Connect-4 but obtained rather discouraging results. Stenmark [27] compared TDL for Connect-4 against a knowledge-based approach from Automatic Programming and found TDL to be slightly better. Curran et al. [10] used a cultural learning approach for evolving populations of neural networks in self-play. All the above works gave no clear answer on the *true* playing strength of the agents, since they did not compare their agents with a perfect-playing Minimax agent.

Lucas showed that the game of Othello, having a somewhat greater complexity than Connect-4, could be learned by TDL within a few thousand training games with the n-tuple approach [17]. Krawiec et al. [16] applied the n-tuple-approach in (Co-) Evolutionary TDL and outperformed TDL in the Othello League [18]. This stirred our interest in the n-tuple approach and we applied it successfully to Connect-4 in our previous work [37]. The results against a perfect playing Minimax agent are summarized.

1.3 Report Structure

This case study is mainly focused on learning techniques to train a capable agent for the game of Connect-4. Tic Tac Toe which is a simpler game with significantly

less number of possible states, is just used as an initial illustration. We will give a brief overview about both games in 2.1 and 2.2. In Section 2.3 fundamentals of reinforcement learning will be discussed in detail. Temporal Difference learning and Q-learning will be explained in the same section. Additionally eligibility traces and its mathematical expressions are shown. Section 2.4 belongs to fundamentals of n-tuple system. Meta parameter learning techniques are explained in the following sections: 2.5 and 2.6. After explaining the fundamental of techniques and problem, we give the experimental setup details in Chapter 3, this information can be useful in order to repeat any of these experiments and gain similar results as ours which are presented in Chapter 4.

Chapter 4 contains detailed information about our test results in Connect-4. Our proposed technique (TCL-EXP) is formulated in Section 4.4 and the results are presented in the same section. In Chapter 5 we give a summary of all results and describe future work.

Chapter 2

Fundamentals

2.1 Connect Four

Connect Four is a popular board game for two players (typically *Yellow* and *Red*), played on a grid with seven columns and six rows. The aim of the both opponents is to create a row of four connected pieces with their color, horizontally, vertically or diagonally. The player who can achieve the goal first, wins the game. One main characteristic of the game is the vertical arrangement of the board, which allows both opponents to throw their pieces only into one of the seven columns (slots). Starting with *Yellow*, both players alternately drop one of their pieces into a slot. Due to gravity, the pieces fall down to the first free row of the according slot. A column containing six pieces is considered as full and reduces the number of possible moves for both players by one (initially, seven moves are possible). After 42 plys, all columns will be completely filled and, in this case, if player *Red* did not connect four own pieces with his last piece, the match ends with a draw.

Determining the state space complexity of *Connect Four* is substantially more difficult than for other games such as *Tic Tac Toe*. An initial estimation of $3^{42} \approx 10^{20}$ is an upper bound but, however, very vague. The exact number of $4531985219092 \approx 4.5 \cdot 10^{12}$ states [12] is fairly large. In comparison with games such as chess or checkers this number is still rather small. *Connect Four* was first solved by *Allis* in 1988 [1]: He found out, that the active (starting) player *Yellow* wins the game by placing his first piece in the middle column of the board.
In 1994 *Tromp* completed a database containing all positions with exactly 8 pieces [3]. The evaluation of all $67,557$ positions took him around $40,000$ hours (≈ 4.5 years) using several computers. Even nowadays, it is not trivial to develop a computer-program that is able to solve *Connect Four* in acceptable time. For instance, the popular program *Mustrum* [9] needs around 17 hours on a Pentium-4 machine and a 60 MB transposition-table to find the best move for the empty board (without the help of databases).

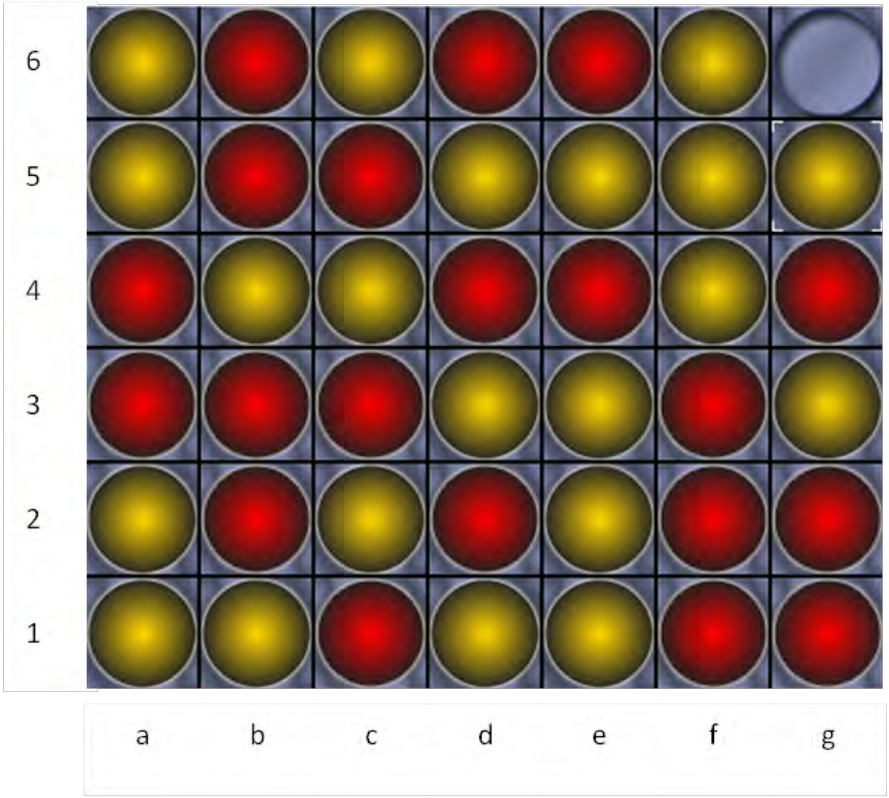

Figure 2.1: A typical position for the game *Connect Four*. The position was created by two perfect-playing agents. Player *Yellow* won the game with his last piece. However, *Red* was able to delay his defeat as far as possible.

2.2 Tic Tac Toe

Tic Tac Toe is a strategic board game for two players, which is very similar to *Connect Four*. The game is played on a board with 3×3 cells. Goal of both players (X and O) is to place three own pieces in a row, either horizontally, vertically or, diagonally. In contrast to *Connect Four* there is no gravity component in *Tic Tac Toe*, all nine spots on the initial empty board can be occupied by the players. The pieces are placed alternating by both opponents, starting with player X. The game ends after nine plys, if both sides cannot achieve a row of three pieces, with a tie. Two example positions can be found in fig. 2.2.

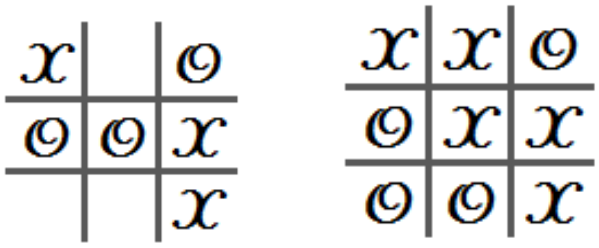

Figure 2.2: Two example positions for the game *Tic Tac Toe*. **Left:** A typical situation in *Tic Tac Toe* after six plys. **Right:** A position with a win for player X who connects three of his own peices diagonally with his last move.

It can easily be shown with a full-depth *MiniMax search* that every match of perfect players ends with a tie.
The number of states for *Tic Tac Toe* is fairly small. A first estimation is simply $3^9 = 19683$ states, which however, includes many illegal states. With a computer program we calculate an exact count of 5478 possible legal position.

2.3 Reinforcement Learning and TDL

Reinforcement Learning (*RL*) describes a field in machine learning tasks, in which agents learn how to take a sequence of actions in an unknown, dynamic environment in order to maximize a given reward. In contrast to supervised learning methods, *Reinforcement Learning* learns just by experiences (trial and error principle) and has no access to further knowledge provided by an external supervisor.

In the classical *RL*-model, the agent is able to observe his environment. This observation generally represents the *environmental state* $s \in S$, with a set of all possible states S. By interacting with his environment, the agent can transform the current state s_t into a new environmental state s_{t+1}. This is done by mapping the current state into an action a (that maximizes the cumulative predicted reward); typically, the agent selects a from a set of possible actions $A(s)$. Based on this action the environment performs a transition to a new state and returns an immediate reward to the agent. However, an action taken by the agent, does not only affect the immediate reward. In general, also future rewards are influenced. One main feature of *RL* is, that it can handle problems with delayed rewards. Strategic board

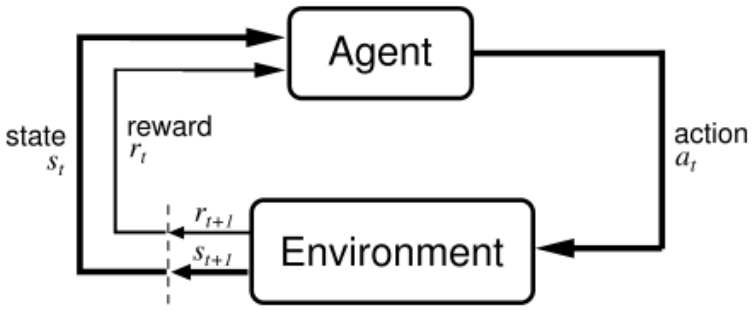

Figure 2.3: Reinforcement Learning model showing the interaction of an agent with his environment. Based on the current state s_t the agent determines and performs an action a_t which transforms the state s_t of the environment to a new state s_{t+1}. Additionally, the environment returns a reward to the agent. Maximizing the cumulative future rewards is the main aspect, that is driving the agent to learn a convenient strategy (policy) π. The figure is taken from [32].

games such as *Connect Four* belong into this category; a whole sequence of moves is necessary before a reward can be determined.
The main challenge for *RL* is performing the mapping from a state to an action in order to maximize the long-term success. A set of rules is needed, that can predict the future rewards (e.g. by value-functions) and select an action for the current state based on the estimations. The mapping $S \rightarrow A(S)$ is commonly denoted as the policy π of an agent. Often, the agents policy is also defined as a probability distribution $\pi(s, a)$, that gives the probability of selecting an action a for the state s.
However, in *Reinforcement Learning* there is always a trade-off between exploration and exploitation. A greedy agent, which just bases its decisions on old experiences and selects an already known action, that appears to be the best currently, will likely fail in finding a suitable policy. For the agent a certain degree of exploration is necessary, by simply trying unknown actions that could then yield in an even higher expected overall reward. Finding the balance between exploration and exploitation in practice is often not trivial (typically, an over time decreasing exploration-rate is chosen during the training process of the *RL*-agent).

2.3.1 Value Functions and their Approximation

As mentioned before, the policy of a *RL*-agent is often based on a value function, that estimates the future rewards for a given state or a state-action pair. In contrast to the reward function that is clearly defined in the environment, it is generally not a trivial task finding a convenient value function. Formally, the state-value function is defined as:

$$V^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} \middle| s_t = s\right\}, \tag{2.1}$$

where π is the policy followed by the agent, E_{π} is the expected reward – starting at time t with state s_t – if the agent follows its policy π.

Accordingly, the action-value function is described by:

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_t | s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right\}. \tag{2.2}$$

The discount-factor γ is commonly used to decrease future rewards slightly and is therefore typically chosen in a range of $0 \leq \gamma \leq 1$. A discount of $\gamma < 1$ is especially needed for infinite episodes with $T = \infty$ to ensure that the series converges.

The task of the *RL*-agent is to learn a value function by exploring his environment and interacting with it. The value function should be capable of mapping every possible state of the system to a real value. A naive approach for a value function could simply manage a lookup table of $|S|$ values, by assigning each environmental state to exactly one table-entry. However, this approach is not practicable in many cases, for at least two reasons: On the one hand, the amount of required memory would simply be too large. And on the other hand, it would be necessary for the agent to visit every state at least once, in order to learn the correct behavior.
Thus, for complex problems a representation is needed, that is able to generalize sufficiently and only requires a reasonable amount of memory. The generalization implies a certain degree of approximation, which introduces an error in the predictions made by the agent. This makes it challenging, to find an appropriate balance between the level of generalization and the accuracy of the approximation.

A typical approach for approximating state values are linear functions in the form

$$V(s) = f(\vec{w}, \vec{g}(s)) = \vec{w} \cdot \vec{g}(s), \tag{2.3}$$

with a parameter vector $\vec{w}$ and a feature vector $\vec{g}(s)$ (commonly simply w and $g(s)$). Finding appropriate features for $g(s)$ differs from task to task and is very often not trivial and requires expert knowledge. We will later introduce the so called *n-tuple systems* that create a mighty feature space without requiring any special knowledge of the considered task.
As all approximations, also linear functions induce a certain approximation error. Normally, when changing a single parameter (weight) in $\vec{w}$, we adjust the values of many states. This leads to the question, how to determine a parameter vector $\vec{w}$ in order to get a good approximation of all relevant states. One measure to assess the quality of the linear function could be the commonly chosen *mean squared error* (*MSE*). The *MSE* at a given time t is then:

$$MSE(\vec{w_t}) = \sum_{s \in S} P(s) \Big[V^{\pi}(s) - V_t(s) \Big]^2, \tag{2.4}$$

where $V^{\pi}(s)$ describes the perfect value-function when following policy π, $V_t(s)$ is an approximating (linear) function and, $P(s)$ is a probability distribution which is used to weight the errors of all states. This weighting has to be done, because normally there are states in a system that are more relevant than others and require a better approximation. However, by increasing the value-accuracy for one state, other states will have a loss of precision ([32], ch. 8.1).

An analytical solution for finding $\vec{w_t}$ that minimizes the MSE is not possible in most cases. Therefore, commonly *gradient descent* methods are used ([32], ch. 8.2). Instead of minimizing the error for the complete sum in Eq. 2.4, we try to minimize the error for individual states s_t that we reach during the training. In many cases this approach also indirectly addresses the problem of weighting the state-errors, as done in Eq. 2.4 (we assume, that important states are visited more often, e.g., the initial position in *Connect Four*).
We evaluate the MSE for a given example s_t. If $V_t(s_t) = f(\vec{w_t}, \vec{g}(s))$ is a differentiable function we can calculate the gradient with respect to $\vec{w_t}$ and take a small step into the opposite direction by adjusting $\vec{w_t}$. The gradient is:

$$\nabla_{\vec{w_t}} \Big[V^{\pi}(s_t) - V_t(s_t) \Big]^2 = -2 \Big[V^{\pi}(s_t) - V_t(s_t) \Big] \nabla_{\vec{w_t}} V_t(s_t) \tag{2.5}$$

Assuming a linear function

$$V(s) = \sum_{i=1}^{N} w_i \cdot g_i(s) = w_1 \cdot g_1(s) + w_2 \cdot g_2(s) + \ldots + w_N \cdot g_N(s), \tag{2.6}$$

we can rewrite $\nabla_{\vec{w}_t} V_t(s_t)$ as:

$$\nabla_{\vec{w}_t} V_t(s_t) = \begin{pmatrix} g_1(s) \\ \vdots \\ g_N(s) \end{pmatrix} = g(s), \tag{2.7}$$

which is simply the feature-vector. By selecting an appropriate step-size parameter (learning rate) α, we can now adjust the parameter vector with

$$\vec{w}_{t+1} = \vec{w}_t + \alpha \Big[V^{\pi}(s_t) - V_t(s_t) \Big] \nabla_{\vec{w}_t} V_t(s_t) \tag{2.8}$$

and for the linear case with

$$\vec{w}_{t+1} = \vec{w}_t + \alpha \Big[V^{\pi}(s_t) - V_t(s_t) \Big] g(s_t) \tag{2.9}$$

The missing factor 2 from Eq. 2.5 is considered as part of the learning-rate α.
The two above formulas can now be used in order to find the (local) minima of the *MSE*-function in Eq. 2.4. However, both, Eq. 2.8 and Eq. 2.9 have one issue: In practice the value of V^{π} is not known and can not be derived easily. One approach, to solve this problem, is presented in the following section.

2.3.2 Temporal Difference Learning

In the last section we briefly described how to minimize the error of an approximating value function for *RL*-problems by applying a *gradient descent* method on the *MSE*-function. In this way, the approximation of the real value-function converges to a (local) optimum ([32], ch. 8.1). However, the value function $V^{\pi}(s_t)$ is unknown and cannot be retrieved in many cases. Nevertheless, it can be shown, that if an *unbiased estimate* v_t is used instead, the parameter vector $\vec{w}_t$ converges to a local optimum ([32], ch 8.1).

Temporal Difference Learning is a Reinforcement Learning method that describes such an approach by estimating V^π with the value of the successor state s_{t+1}. Although the estimate is not unbiased in all cases, this method proved to be a very well suitable as a bootstrapping method. The main idea behind TD-learning is, estimating V^π in the following way:

$$V^\pi(s_t) \approx E\left\{r_t + \gamma V(s_{t+1}\right\}, \tag{2.10}$$

where γ is again the discount factor (as described in section 2.3.1) and r_t the current reward. Inserting this approximation into Eq. 2.8 leads to the following formula, which is the central element of TD-Learning ([32], ch 6.1):

$$\vec{w}_{t+1} = \vec{w}_t + \alpha\Big[r_t + \gamma V(s_{t+1}) - V_t(s_t)\Big]\nabla_{\vec{w}_t} V_t(s_t). \tag{2.11}$$

The above equation can be rewritten in the following way:

$$\vec{w}_{t+1} = \vec{w}_t + \alpha\delta_t\ e_t \tag{2.12}$$

with the expressions

$$\begin{aligned} T_{t+1} &= r_t + \gamma V(s_{t+1}) && (2.13)\\ \delta_t &= T_{t+1} - V_t(s_t) && (2.14)\\ e_t &= \nabla_{\vec{w}_t} V_t(s_t). && (2.15) \end{aligned}$$

Again, for a linear value function we get

$$e_t = g(s_t). \tag{2.16}$$

The vector e_t is commonly referred to as the eligibility traces vector. A more general definition is

$$e_{t+1} = \gamma\lambda e_t + \nabla_w f\big(w; g(s_{t+1})\big) \tag{2.17}$$

In the following section we will discuss eligibility traces in more detail.

Algorithm 1 General $TD(0)$ algorithm.

Initialize $V(s)$ arbitrarily, π to the policy to be evaluated
for Each episode **do**
  Initialize s
  for Each step of episode **do**
    $a :=$ action given by π for s
    Take action a; observe reward, and next state (s_{t+1})
    $V(s) := V(s) + \alpha[r + \gamma V(s_{t+1}) - V(s)]$
    $s_t := s_{t+1}$
  end for ▷ until s is terminal
end for

2.3.3 Eligibility Traces

Most board games such as Connect Four have in common, that rewards – given by the environment – are delayed (intermediate rewards would indicate some supervisor involved in the learning process). An action a generally does not lead to a direct reward, so that a complete sequence of actions is necessary to reach a state in which the effective reward is given. As described in the last section, Temporal Difference methods are useful techniques for solving Reinforcement Learning problems with delayed rewards. Even though simple (one-step) TD methods are able to learn value functions that predict the expected future reward, they still have to deal with some delays in the updates of their value functions: TD methods estimate the approximation error for a state-value based on the current prediction of the successive state rather than on the final outcome (reward). This means that a reward given at the end of a sequence of actions is only used to adjust the last prediction, but not the predictions before the last one.

As a simple example, assume, an inexperienced agent constantly follows a policy π, which results in an episode with in total T time steps and a final reward r_T at time step T. After completing the first episode the agent receives a reward from the environment and updates the value for $V(s_{T-1})$. During the second episode the value $V(s_{T-2})$ can be adjusted and so on. For this example, $T-1$ repeated episodes would

Algorithm 2 Incremental $TD(\lambda)$ algorithm for board games.

Assume $p_0 \in \{+1, -1\}$, the initial state s_0 and the partially trained function $f\big(w; g(s_t)\big)$

function TDLTRAIN(p_0, s_0, $f\big(w; g(s_t)\big)$)

$e_0 := \nabla_w f\Big(w; g(s_t)\Big)$ ▷ Initial eligibility traces

for $\Big(p := p_0,\ t := 0$ **;** $s_t \notin S_{Final}$ **;** $p \leftarrow (-p),\ t \leftarrow t+1\Big)$ **do**

$V_{old} := f\big(w; g(s_t)\big)$ ▷ Value for current state s_t

generate randomly $q \in [0, 1]$ ▷ Uniformly distributed

if $(q < \epsilon)$ **then** ▷ Random move with prob. ϵ

Randomly select s_{t+1} ▷ Explorative move

else

Select after-state s_{t+1}, which maximizes

$$p \cdot \begin{cases} R(s_{t+1}), & \text{if } s_{t+1} \in S_{Final} \\ f\big(w; g(s_{t+1})\big), & \text{otherwise} \end{cases}$$ ▷ Greedy move

end if

$V(s_{t+1}) := f\big(w; g(s_{t+1})\big)$ ▷ Response of linear/neural net

$r_{t+1} := R(s_{t+1})$ ▷ Reward from environment

$$T_{t+1} := \begin{cases} r_{t+1}, & \text{if } s_{t+1} \in S_{Final} \\ \gamma \cdot V(s_{t+1}), & \text{otherwise} \end{cases}$$ ▷ Target-signal

$\delta_t := T_{t+1} - V_{old}$ ▷ Error-signal

if $(q \geq \epsilon$ **or** $s_{t+1} \in S_{Final})$ **then**

$w \leftarrow w + \alpha \delta_t e_t$ ▷ Update for non-random move

end if

$e_{t+1} := \gamma \lambda e_t + \nabla_w f\big(w; g(s_{t+1})\big)$ ▷ Eligibility Traces

end for ▷ End of the self-learning algorithm

end function

be required, until a delayed reward finally affects $V(s_0)$ (assuming, that s_0 is not reached at other time steps t). Having large T, this could be thoroughly problematic and result in a slow convergence of the process. However, if the agent constantly follows his policy π then the predicted values of all states should ideally converge to the final reward that they predict $\Big(V(s_0) = V(s_1) = \cdots = V(s_T) = r_T\Big)$.

As already indicated before, classical 1-step TD methods update the value function in every time step just for the current state s_t. However, it could be convenient to assign some credit to the previous states $\{s_{t-1}, s_{t-2}, \cdots\}$ as well, since these states led to the current situation, the agent is in now.
Monte Carlo Methods follow this idea, by approaching the backups in a rather different way in comparison to simple TD methods [32]: Instead of updating the value function based on temporal differences, Monte Carlo algorithms first complete a whole episode of length T and then update the value of each state visited during this specific episode. In contrast to TD methods, the backups for s_t are not based on any predictions, only the observed rewards from time step t until T are considered. Again, for board games such as Connect Four, only the final reward r_T is provided by the environment, which is then used to equally update the values for *all* visited states of the corresponding episode. However, one main disadvantage of Monte Carlo algorithms is that the actual learning step has to be performed after the final outcome of the episode is known, so that the agent cannot directly adjust its policy and try to avoid this state, when a bad state is experienced.

In the following we show how to efficiently combine Monte Carlo algorithms and simple TD methods, creating a new class of learning algorithms. The main idea is to first break down the episodic update scheme of MC methods into temporal differences, similar to the simple TD methods (the main steps are taken from [28]). Assume all predictions $P_t = V(s_t)$ of a sequence $(s_0, \cdots, s_T)$ predict outcome r. We then get an recommended weight change (RWC), if we want to minimize the MSE for every time step with

$$\Delta w_t = \alpha \delta_t \nabla_w P_t \tag{2.18}$$

with $\delta_t = r - P_t$. The problem is, that for time step t the final outcome/reward r is unknown. This is for instance the reason that TDL is an bootstrapping process that approximates $V^*(s_t) \approx V^(s_{t+1})$. We can rewrite the error $\delta_t = r - P_t$:

$$\delta_t = r - P_t = \sum_{k=t}^{T-1} P_{k+1} - P_k \quad \text{where} \quad P_T = r \tag{2.19}$$

When evaluating the sum, it can be seen that all terms of the sum cancel out, except: $P_T - P_t = r - P_t$. Inserting (2.19) back into (2.18) leads to:

$$\Delta w_t = \alpha \sum_{k=t}^{T-1} (P_{k+1} - P_k) \nabla_w P_t. \tag{2.20}$$

If the weights shall be updated after one episode (as MC methods do), we get the following weight backup:

$$w \leftarrow w + \sum_{t=0}^{T-1} \alpha \sum_{k=t}^{T-1} (P_{k+1} - P_k) \nabla_w P_t. \tag{2.21}$$

We now rewrite (2.21) with the identity $\sum_{k=0}^{m} \sum_{j=0}^{k} a_{kj} = \sum_{j=0}^{m} \sum_{k=j}^{m} a_{kj}$:

$$w \leftarrow w + \sum_{k=0}^{T-1} \alpha \sum_{t=0}^{k} (P_{k+1} - P_k) \nabla_w P_t \tag{2.22}$$

By replacing k with t and vice versa (since t ranges from 0 to $T-1$), the sum changes to:

$$w \leftarrow w + \sum_{t=0}^{T-1} \alpha \sum_{k=0}^{t} (P_{t+1} - P_t) \nabla_w P_k. \tag{2.23}$$

From the above formula we again can extract Δw_t:

$$\Delta w_t = \alpha \sum_{k=0}^{t} (P_{t+1} - P_t) \nabla_w P_k = \alpha (P_{t+1} - P_t) \sum_{k=0}^{t} \nabla_w P_k \tag{2.24}$$

Note, that this representation is different to the former one, which was:

$$\Delta w_t = \alpha (r - P_t) \nabla_w P_t \tag{2.25}$$

Nevertheless, (2.24) and (2.25) are equivalent. The main advantage of (2.24) is, that it can be calculated incrementally, thus, step by step during an episode, similar to simple TD methods. We do not have to wait until an episode is completed. Sutton [28] introduced an exponentially decaying factor λ, which allows a weighting according to the recency of the events, which was the main step towards the TD(λ) algorithm. The TD(λ) algorithm, introduced by Sutton [28], describes an efficient, *incremental* way of combining simple TD methods and Monte Carlo algorithms,

utilizing the advantages of both approaches. The new ingredient in TD(λ) – as mentioned before – is the so called eligibility trace vector, containing a decaying trace e_i for each weight w_i [28]:

$$\begin{aligned} \vec{e}_t &= \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{\vec{w}} V(s_k) \\ &= \lambda\gamma \vec{e}_{t-1} + \nabla_{\vec{w}} V(s_t), \\ \vec{e}_0 &= \nabla_{\vec{w}} V(s_0), \end{aligned} \tag{2.26}$$

with a trace decay parameter λ and a discount factor γ (we assume $\gamma = 1$ throughout this report), which decay the individual traces e_i by the factor $\lambda\gamma$ in every time step. Thus, the effect of future events on the corresponding weights w_i exponentially decreases over time. By choosing λ in a range of $0 \leq \lambda \leq 1$, it is possible to shift seamlessly between the class of simple one-step TD methods ($\lambda = 0$) and Monte Carlo methods ($\lambda = 1$) [29].
The definition of the conventional eligibility traces in Eq. (2.26) implies, that each trace accumulates a value $\Delta e_i = \nabla_{w_i} V(s_t)$ in every time step t if the corresponding weight is activated. In certain cases, this behavior may be undesired, since specific states can be visited many times during one episode. As a result, the according traces build up to comparably large values and future TD errors give higher credit to frequently visited states, which could negatively affect the overall learning process. For instance, an agent that reaches an undesirable state several of times, but still manages to achieve a good final reward, will give higher credit to this undesirable state, which in consequence, may negatively affect the overall learning process.
Although it is unlikely or even impossible in many problems (including Connect-4) that states are revisited during an episode, the use of generalizing function approximators typically leads to reoccurring features, so that certain traces are repeatedly addressed [25].
Replacing eligibility traces, proposed by Singh & Sutton [25], depict an approach to overcome this potential problem related to conventional eligibility traces. The main idea behind replacing traces is to reset a trace to $e_i = \nabla_{w_i} V(s_t)$ each time the corresponding weight is activated, instead of accumulating its value. As before, the traces of non-active weights gradually decay over time and make the weights less sensitive to future events.
In this work, we will investigate both approaches, conventional and replacing eligibility traces. An example illustrating both types of traces is given in Fig. 2.4.

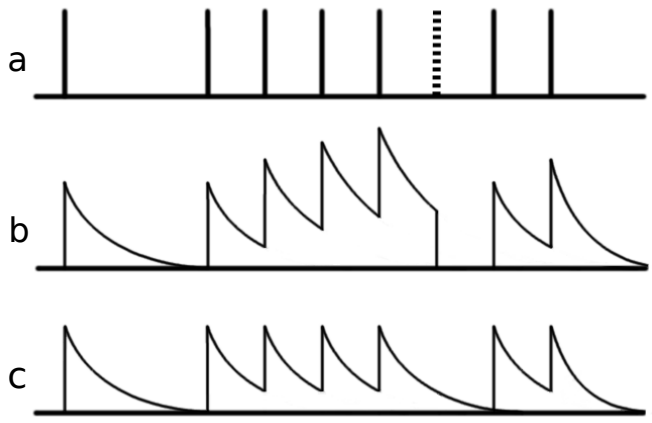

Figure 2.4: Schematic view of different eligibility trace variants: Line **a** shows the situation without elig. traces, a weight is activated only at isolated time points. The dotted vertical line represents a random move. Line **b** shows the eligibility trace with reset on random move. Line **c** shows replacing traces, this time without reset on random move.

The TD(λ) algorithm usually requires a certain degree of exploration during the learning process, thus, the execution of random moves from time to time – ignoring the current policy. When performing random moves, the value $V(s_{t+1})$ of the resulting state s_{t+1} is most likely not a good predictor for $V(s_t)$. The weight update based on $V(s_t)$ is normally skipped in this case. This also raises the question how to handle exploratory actions regarding the eligibility traces. We will consider two options: 1) Simply resetting all trace vectors and 2) leaving the traces unchanged although a random move occurred. Both options may diminish the anticipated effect of eligibility traces significantly, if higher exploration rates are used (which is not the case in our experiments).

2.3.4 Q-Learning

Q-Learning proposed by Watkins in 1989, is known as one of the most beneficial techniques in *reinforcement learning* studies. While *TD-learning* should handle the combination of prediction and control tasks simultaneously [34], *Q-Learning* is appearing superior in some specific applications to *TD-Learning* because of having an off-policy control algorithm.

The main idea which we follow in *TD-learning* is, to approximate values of every state and gradually converge to the real state values. Whereas all of this is done

in order to evaluate the policy and finally find the optimal one. In an active *TD-learning* approach, policy is not fixed and should be set to a better estimate after every update. Controlling policy by extracting the current optimal one cannot be solved completely by means of *TD-learning*. In every update, new policy should be determined by Equation (2.27). $Q(s, a)$ represents the estimated value of being in state s and performing action a.

$$\pi(s) = \arg\max_{a} [Q(s, a)] \tag{2.27}$$

$$V(s_t) \leftarrow V(s_t) + \alpha(r + \lambda V(s_{t+1}) - V(s_t)) \tag{2.28}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r + \lambda \text{max}[Q(s_{t+1}, a)] - Q(s_t, a_t)) \tag{2.29}$$

Q-learning tries to approximate Q^*, which is a function dependent on state-action pairs (Equation (2.29)). In this technique the estimation of $Q(s_t, a_t)$ is independent of the followed policy which results in converging to the optimal policy even by doing suboptimal moves. Therefore, *Q-learning* can learn even if explorative actions are performed. Here we explain the mentioned advantage of *Q-learning* by a simple example.

Assume, that we have only five possible states $s \in \{A, B, C, D, E\}$, shown in figure 2.5. We can start from any arbitrary state. In every state there are at most four actions possible, $a \in \{\textit{left}, \textit{right}, \textit{up}, \textit{down}\}$. Any action which results in exiting from the board, finishes one episode. The target is exiting from state D therefore receiving $+10$ as reward otherwise reward will be -10. Values of the states are initialized by zero. In figure 2.5, we show the *TD-learning* approach for this problem after three episodes and the same procedure will be shown in figure 2.6 for *Q-learning*. Every episode's details are listed in the Table 2.1. In this small experiment we have used $\alpha = 1/2$ and $\lambda = 1$, despite the fact that $\alpha = 1/2$ is pretty large for real examples with more states.

In episode 1 (figure 2.5) just the last move results in an update and yields in $V(D) = +5$. Therefore, whenever we go through states which were not met before update will not occur until the final state. This late update can cause a slow convergence for problems which have large number of states or weights. Assigning eligibility traces is a way to cope with this issue. In the second episode, the second move causes an update and $V(C) = 2.5$ and again $V(D)$ will be updated and gets a higher value than before. Last episode has an update in every move. First move in the last episode will change $V(E)$ from 0 to 1.25. Second move reduces value of the state C because is a wrong move. Finally, the last move of the episode three updates

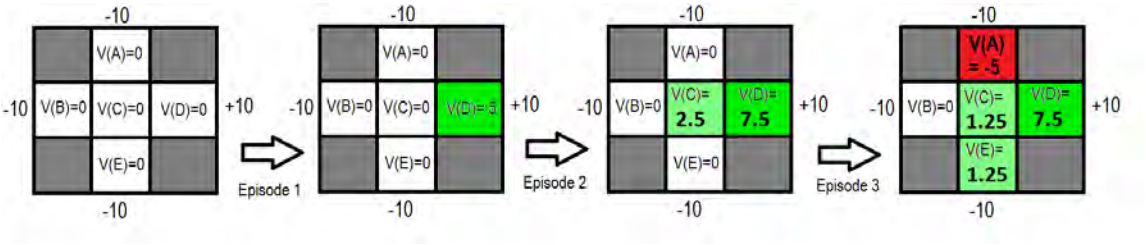

Figure 2.5: *TD-learning* approach for episodes shown in table 2.1. Every state value updates, based on the Equation (2.28).

$V(A)$ to -5. What we can see is that now state of A has a bad value while E has a good value and we know that these two states are very similar in many ways. We can observe, in a non-greedy move is possible to update some weights with wrong values.

In the figure 2.6, same episodes have occurred with *Q-learning* approach. Therefore, we update $Q(s, a)$ after every move, instead of $V(s)$. After episode 1 the only weight which is updated is $Q(D, right)$. After episode 2 there are still a lot of not touched weights. $Q(C, right) = 2.5$ and $Q(D, right)$ changes from 5 to 7.5. The main advantage that appeared in this technique in comparison with *TD-learning* is that even by moving toward a wrong state the correct weight values will not be reduced. Sub-optimal states will be set to the correct values even if we do a non-greedy wrong move. $Q(E, up) = 1.25$ and $Q(C, right)$ is still keeping its correct good value independent on the policy.

Table 2.1: Episodes. Three first episodes used in state board experiment with *TD-learning* and *Q-learning* approaches.

Episode 1				Episode 2				Episode 3			
s_t	a	s_{t+1}	r	s_t	a	s_{t+1}	r	s_t	a	s_{t+1}	r
B	$right$	C	0	E	up	C	0	E	up	C	0
C	$right$	D	0	C	$right$	D	0	C	up	A	0
D	$right$	exit	$+10$	D	$right$	exit	$+10$	A	up	exit	-10

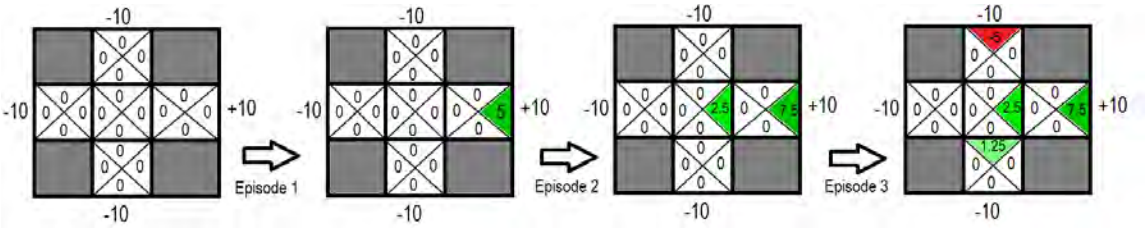

Figure 2.6: *Q-learning* approach for episodes shown in table 2.1. Every state value updates, based on the Equation (2.29).

2.4 N-Tuple Systems

Even though n-tuple systems were already introduced in 1959 for character recognition purposes [8], the application to board games is rather new. Lucas applied n-tuple systems to the strategic board game Othello and could achieve remarkable results using n-tuple systems in combination with *TDL* [17]. In this section, we will introduce n-tuple systems, based mainly on Lucas' work [17], that can be used as linear functions to approximate the real value function.

The main idea behind n-tuple systems is to map a low dimensional space (e.g., the board of a game) to a high dimensional space by sampling the low-dimensional space and applying an indexing function to the samples. In detail:
A n-tuple is defined as a sequence $T_v = (\tau_{v0}, \tau_{v1}, \ldots, \tau_{vn-1})$ with the length n. Every n-tuple is a subset of a set of sampling points P, therefore $T \subseteq P$. In *Connect Four* the sampling points would be the set of board cells (in total $|P| = 42$), *Tic Tac Toe* on the other hand would have $|P| = 9$ possible sampling points.
Every sampling point p_j can have m different states, with $p_j \in \{0, 1, \ldots, m-1\}$. In *Tic Tac Toe* we have three different states – empty, "X", "O" and, empty – and therefore choose $m = 3$. Due to the gravity component in *Connect Four* we distinguish two classes of empty board cells, so that – with the states *Yellow*, *Red*, empty and reachable with the next move and as the last state, empty but not reachable with the next move – we get $m = 4$.
Using an indexing function, every n-tuple can be mapped into m^n different states:

$$\xi(T_v, s_t) = \sum_{j=0}^{n_v-1} s_t[\tau_{vj}] \cdot m^j, \tag{2.30}$$

with $\xi \in 0, \ldots, m^n - 1$ and s_t, the current state of the overall board and $s_t[\tau_{vi}]$, the current state of a single board cell. The value computed by the indexing function ξ is used to address an element in a look-up table (in the following LUT), which basically represents a parameter-vector $\vec{w}_{v,t}$ of our linear value-function. The corresponding feature vector at a time t is defined as:

$$g_{v,i}(s_t) = \begin{cases} 1, & \text{if } i = \xi(T_v, s_t) \\ 0, & \text{otherwise} \end{cases} \tag{2.31}$$

The output of linear function with the parameter vector $\vec{w}_{v,t}$ is calculated by:

$$f_v\big(\vec{w}_{v,t}, \vec{g_v}(s_t)\big) = \sum_{i=0}^{m^{n_v}-1} w_{i,t} \cdot g_{v,i}(s_t) \tag{2.32}$$

However, n-tuple systems typically consist of many different n-tuples. A set of k n-tuples would result in a set of k parameter vectors and therefore k different linear functions. To get one overall output $V(s_t) = f\big(\vec{w}_t, g(s_t)\big)$ for the n-tuple system, we combine all parameter vectors into one single vector $\vec{w}_t$. This then results in the linear function

$$V(s_t) = f\big(\vec{w}_t, \vec{g}(s_t)\big) = \vec{w}_t \cdot \vec{g}(s_t) = \sum_v f_v\big(\vec{w}_{v,t}, \vec{g_v}(s_t)\big). \tag{2.33}$$

N-tuple systems for board games completely describe a linear function, including the determination of the feature vector. For the user it is not necessary to find appropriate features for the (linear) function, the n-tuple system already creates implicitly a huge feature space and learns during the training to ignore irrelevant features.

One enhancement, when using n-tuple systems, can be the utilization of board symmetries, which are very common in many board games. In *Tic Tac Toe* for instance, there are 8 equivalent positions for every board (based on rotation and mirroring). In *Connect Four*, only mirroring the board at the central column leads to one equivalent position. With this in mind, it is possible to learn more than one position in every time step t. The only component that has to be redefined is the feature vector. Assuming a set of equivalent positions $E(s_t)$, we now define the v-th feature vector as

$$g_{v,i}(s_t) = \begin{cases} 1, & \text{if } i \in \xi\big(T_v, E(s_t)\big) \\ 0, & \textit{otherwise} \end{cases}, \tag{2.34}$$

where ξ returns a set of indexes, one for every element of E, in total $|E|$.

2.5 Temporal Coherence in TD-Learning

In *Temporal Difference Learning* the choice of an appropriate learning rate is crucial for the success of the training. However, *Temporal Coherence in TD-Learning* (in the following *TCL*), developed by Beal and Smith [6, 7], is a technique that introduces additional adaptable learning rates for every weight of the parameter vector $\vec{w}$. The main idea behind *TCL* is to decrease the learning rate for individual weights, if they do not contribute to the learning process (or already reached their optimum values) and just add random noise to the system and, on the other hand, to increase the learning rate for relevant weights, that play an important role in the learning task. Irrelevant weights accumulate the recommended weight changes (*RWC*), but will not move into any specific direction, since the weight changes cancel out.
As a weight converges to its optimal value, the learning rate for this weight will tend towards zero, which makes the weight less sensitive to random noise. A further property of *TCL* that Beal and Smith mention, is the possibility that the individual weights can approach their final value at different times. Weights that need longer than others to reach their final value, should still have a high learning rate, even if other weights are already stable.

For adjusting the learning rates, *TCL* accumulates all the *RWC* and absolute *RWC* in two counters, N_i and A_i. In every time step t, the *RWC* $r_{i,t}$ for each weight is determined and the counters N_i and A_i adjusted in the following way:

$$N_i \leftarrow N_i + r_{i,t}, \tag{2.35}$$

$$A_i \leftarrow A_i + |r_{i,t}|, \tag{2.36}$$

$$r_i = \delta_t e_{i,t}. \tag{2.37}$$

When performing an update of the weights, next to the global learning rate also an individual learning rate for each weight is calculated by:

$$\alpha_i = \begin{cases} \frac{|N_i|}{A_i}, & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases} \tag{2.38}$$

Then the weights can be updated according to the δ-rule with

$$w_i \leftarrow w_i + \alpha\alpha_i\delta_t e_{i,t}. \tag{2.39}$$

In algorithm 3 the complete procedure is listed as pseudo-code. Note, that according to [6], first the weights are updated and after that the counters N_i and A_i. In their work, Beal and Smith describe the possibility of accumulating the RWC for every weight over a given amount of time steps, before the actual weight-update is performed. We will describe this possibility using update episodes in more detail later in our report.

Algorithm 3 General pseudo code of the proposed TCL-update algorithm. In [6], the possibility of update-episodes is described, which means, weights are only updated after a certain episode-length. Update-episodes are not considered in this pseudo code.

Initialize in the beginning $A := \vec{0}$, and $N := \vec{0}$

Define global learning rate α

for $i \in \{1, \ldots, N\}$ **do**

$$\alpha_i = \begin{cases} \frac{|N_i|}{A_i}, & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases}$$

$r_i := \delta_t e_{i,t}$ ▷ Recommended weight change

$w_i \leftarrow w_i + \alpha \alpha_i r_i$ ▷ TD-Update for the weight

$A_i \leftarrow A_i + |r_i|$ ▷ Update accumulating counter A

$N_i \leftarrow N_i + r_i$ ▷ Update accumulating counter N

end for

2.6 Incremental Delta-Bar-Delta

Similar to *TCL*, the *Incremental Delta-Bar-Delta* (*IDBD*) algorithm proposed by Sutton [30], attempts to learn individual learning rates for every weight w_i of the system. Again, the idea is to assign large learning rates to relevant weights and vice versa. The learning rates are determined by applying the exponential function to a memory parameter β_i:

$$\alpha_i = e^{\beta_i}. \tag{2.40}$$

In contrast to *TCL*, *IDBD* does not need a global learning rate, the individual rates are sufficient. The exponential function ensures that the learning rates are always larger than zero. Another desirable property is, that for a change in the

memory parameter $\Delta\beta_i$, α_i will always change by a fixed fraction of its current value [30]:

$$\alpha_{i,t+1} = e^{\beta_{i,t}+\Delta\beta_{i,t}} = \alpha_{i,t} \cdot e^{\Delta\beta_i}. \tag{2.41}$$

The pseudo-code for *IDBD* can be found in algorithm 4. In algorithm 4 the memory

Algorithm 4 General Pseudo code of the IDBD-Algorithm for linear functions [30]. Note that, when using n-tuple systems as approximating value function, then we get $x_i = g_i(s_t)$ and $x_i \in 0, 1$.

Initialize in the beginning $h := \vec{0}$, and w_i, β_i as desired

With error-signal δ_t

for $i \in \{1, \ldots, N\}$ **do**

$\beta_i \leftarrow \beta_i + \theta\delta_t x_i h_i$

$\alpha_i \leftarrow e^{\beta_i}$

$w_i \leftarrow w_i + \alpha_i\delta_t x_i$

$h_i \leftarrow h_i[1 - \alpha_i x_i^2]^+ + \alpha_i\delta_t x_i$

end for

parameters β_i are updated by

$$\beta_i \leftarrow \beta_i + \theta\delta_t x_i h_i, \tag{2.42}$$

with θ describing the meta-learning rate and h_i, a trace of recent weight changes. The parameter h_i will cause an increase of β_i, if the current error-signal δ changes the individual weights in the same direction as recent weight changes (positive correlation between $\delta_t x_i$ and h_i). A negative correlation between $\delta_t x_i$ and h_i indicates that the recent steps have been too large and the corresponding weight has to be adjusted in the opposite direction. In this case β_i has to be decreased. The parameter h_i is changed according to the following update-rule:

$$h_i \leftarrow h_i\big[1 - \alpha_i x_i^2\big]^+ + \alpha_i\delta_t x_i, \tag{2.43}$$

where $[x]^+$ returns x for $x > 0$, otherwise 0. Basically, h_i accumulates the recommended weight changes, although changes in the past are decayed by $[1-\alpha_i x_i^2]^+$. The *IDBD* algorithm described in [30] is only defined for linear functions. For nonlinear functions, e.g. a linear net with a squashing function ($\sigma = tanh$) in the output, the *IDBD* algorithm has to be adjusted.

Chapter 3

Research Questions and Experimental Setup

In this work we mainly investigate two different approaches of online-adaptable learning rates which are Incremental Delta Bar Delta (IDBD) and Temporal Coherence Learning (TCL) and whether they have the potential to speed up learning for such a complex task. Additionally we study the benefits of eligibility traces added to this system with several million weights. Different versions of eligibility traces (standard, resetting, and replacing traces) are compared. In this chapter we shortly describe the general setup of the performed experiments and state several research questions.

3.1 General Research Questions

Generally, we expect the learning-rates tuning algorithms TCL and IDBD to improve the learning-process. We believe, that both methods (TCL and IDBD) should increase the asymptotic success rate of our agents and decrease the amount of training games needed to learn Connect 4. Eligibility traces also seem to be a promising approach in order to increase the learning speed of the agents. Based on the overall goals of this report, we formulate the following research questions and try to answer them in the next chapters:

1. To our best knowledge, both TCL and IDBD have not been applied to such large systems as the n-tuple systems used for our work. Is it possible to successfully apply TCL and IDBD to a system with millions of weights?

2. How robust are online learning rate adaptation algorithms with respect to their meta-parameters? TCL requires an initial step-size α_{init} for all weights. According to [7] the parameter α_{init} in TCL *"has to be chosen, but this does not demand a choice between fast learning and eventual stability, since it can be set high initially, and the then provide automatic adjustment during the learning*

process."
IDBD requires two parameters: The initial step-size α_{init} for all weights and the meta step-size parameter θ.

3. Is TCL (IDBD) able to significantly increase the speed of learning?

4. Can TCL (IDBD) increase the strength (asymptotic success rate) of our agents?

5. Is it possible to successfully apply eligibility traces to a large scale problem with millions of weights?

6. We expect eligibility traces to speed up the learning process and increase the asymptotic success rate. Is it possible to significantly improve both aspects?

7. If one or more of the previous research questions cannot be answered positively, is it possible to find the cause of the unsatisfactory results? Can suggestions for future work be made?

3.2 Experimental Setup

There are three main components that are necessary for the learning process: The TDL-Algorithm as reinforcement learning method, an n-tuple system to approximate the real value function and (if required), an online tuning algorithm adapting the learning rates of the system. In the following we specify how the learning experiments are conducted.

The training of our TDL-n-tuple-agents is performed in an unsupervised fashion, following the pseudo-algorithm already described in Algorithm 2: Each agent is initialized with random weights uniformly drawn from $[-\chi/2, \chi/2]$ with $\chi = 0.001$.The agent plays a large number of games (10 millions) against itself as described in Sec. 2.3.2 receiving no other information from the environment than win, loss, or draw at the end of each game. Training is performed with TDL, optionally augmented by IDBD- or TCL-ingredients. To explore the state space of the game, the agent chooses with a small probability ϵ the next move at random. During training, ϵ varies like a sigmoidal function (tanh) between ϵ_{init} and ϵ_{final} with inflection point at game number ϵ_{IP}. Every 10 000 or 100 000 games the agent strength is measured by the procedure described in Sec. 3.3. We repeat the whole training run 20 times (if not stated otherwise) in order to get statistically sound results.

For this report we consider the discount-factor to be $\gamma = 1$ throughout. With $\lambda = 0$ we get the classical TD(0) algorithm without eligibility traces. With $0 < \lambda < 1$

eligibility traces are activated and we can seamlessly move between TD(0) and Monte Carlo methods. Throughout the rest of the report we use the following notation for the different eligibility trace variants: [et] for standard eligibility traces without any further options, [res] for resetting traces, [rep] for replacing traces, and [rr] for resetting and replacing traces.
As n-tuple system we mostly use the same set of n-tuples (70×8-tuples), all created by random walks on the board. Additionally, we will investigate the effect of eligibility traces on larger n-tuple systems (labeled as TCL-M).
In TDL, the global learning rate α decays exponentially from α_{init} to α_{final}. TCL instead keeps the global parameter α at a constant value, but each weight has its individual learning rate α_i. In TCL-EXP we have the additional global parameter β, while for IDBD the relevant parameters are θ and β_{init}. The precise parameter values to reproduce each of our results are given in Tab. A.1 . Since the IDBD algorithm in its standard form is only derived for linear units, we omit in this case the nonlinear sigmoid function $\sigma = \tanh$ for the TDL value function (Sec. 2.3.2). For all other algorithms we use this sigmoid function.

3.3 Evaluation

Evaluating the strength of an agent is not a trivial task. The term strength has to be defined first, which can be done in many ways. For instance, one possible definition could be:

> "The strength of an agent is defined as the ratio of correct classifications (as *win*, *draw*, *loss*) for a set with k randomly generated (legal) positions."

However, the above definition has one problem, especially concerning approximating functions: By randomly creating positions, it is very likely, to obtain positions that are considered as irrelevant by the agent; it may be, that the agent recognized during his training, that a position is not desirable and tries to avoid it. By avoiding a certain position, the whole sub-tree linked to this position will be "forgotten", so that the value-function can increase its accuracy for other relevant positions. Therefore, it is very likely, that an agent will receive a bad score, when simply evaluating a set of random positions.
For this reason, we choose the following evaluation-scheme:

> "The strength of an agent is measured by performing a tournament with k matches – all starting from the empty board – between the agent and a perfect playing *Minimax*-player. For every win of

> the agent against *Minimax* a score of 1.0 is given, a draw will be honored with a score of 0.5 and, for a loss the score will be 0.0. The overall strength (in the following *success rate*) is then defined as the sum of all scores divided by k."

However, this approach has two limitations which need some special treatment:

- Using a perfect-playing *Minimax* as opponent, makes it impossible to alternately swap the starting player in the matches; if *Minimax* starts, its opponent will loose the game in any case. For this reason, *Minimax* always moves second in our evaluation. This solution, however, makes it impossible to measure the strength of the agents when playing with the *red* stones.

- During the evaluation matches, both opponents will usually act fully deterministically, which would result in exactly the same sequence of moves for every game. We introduce a certain degree of randomness, to overcome this problem: In the opening phase of the game (the initial 12 moves), *Minimax* will be forced to randomly select a successor state, if no move can be found that at least leads to a draw. In order to prevent direct losses when performing random moves, *Minimax* only considers those successors, that delay the expected defeat for at least 12 additional moves.
  After leaving the opening phase, *Minimax* will always seek for the most distant loss. With this approach, it is very unlikely, that any of the k matches will be the same (assuming $k \approx 50$).

In order to monitor the training progress of an agent, it is necessary to evaluate the strength of the agent after certain intervals. We choose an interval length of $l = 100,000$ training games, in later experiments the number is reduced to $l = 10,000$. In every evaluation-step – if not stated otherwise – 50 matches between the agent and *Minimax* are performed. Furthermore, we repeat every experiment 10 times (later also 20 times), to make sure, that the results are reproducible. The graphs shown in the next section, are fitted lines through the 10 (20) points for every evaluation step. Details on *Minimax* can be found in [35] and [36]. It is important to note, that *Minimax* is only used for the evaluation of the agents, it does not play any role in the actual training process; the training is completely unsupervised.

We define two quality-measures that are based on the discussion of the strength of an agent. The first measure we define is the asymptotic success rate:

> "The *asymptotic success rate* is defined as the final value that an agent strengths converges to."

We will – if not stated otherwise – determine the asymptotic success rate by averaging the strengths of the last 20 evaluations (last 2 million training games). Another important measure for the quality of an agent – next to the asymptotic success rate – is the speed of learning. An agent that is able to cross a defined success rate in a shorter time is more preferable than another agent. In our report we define loosely:

> "The speed of learning for an agent is defined as the number of training games needed to cross the 80% success rate."

An optimistic interpretation of the above definition would be, to consider the first crossing of the 80% success rate, whereas a rather pessimistic interpretation would suggest to use the last crossing.

Chapter 4

Connect Four: Results and Analysis

4.1 Starting Point

In our previous work [36, 35, 37] we applied n-tuple systems – trained with TDL – to the strategic game of *Connect Four*. After the learning process for a reduced complexity (simply, by giving the rewards after 10 or 12 moves and finishing the current training game) delivered good results we tried to learn the complete game of *Connect Four*. The initial results were rather discouraging. An important observation we made was that the values for certain sampled subparts of the board are typically different, depending on which player's turn it is. Therefore, we introduced two LUTs per player, one LUT for *Yellow* and, another LUT for *RED*. This approach for the first time led to satisfying results. The TDL-agent was able to learn the game (cross the 80% success-rate) after around 4 Mio. games (as seen in Fig. 4.1, red curve). Additional tuning of the exploration rate ϵ and the step-size parameter α could finally improve the learning speed by a factor of around 2 (blue curve in Fig. 4.1).
Nevertheless, the number of training games still remains rather high, around 2 Mio. training games are needed to create an TDL-agent that is able to constantly beat a perfect playing Minimax-agent.
In this case study, the main goal is to apply the so called Temporal Coherence Learning (a technique that introduces online adaptable, individual learning rates for every weight of the n-tuple system) in order to improve the learning speed of the agent and to be less dependent on good choices of the learning rate parameter.

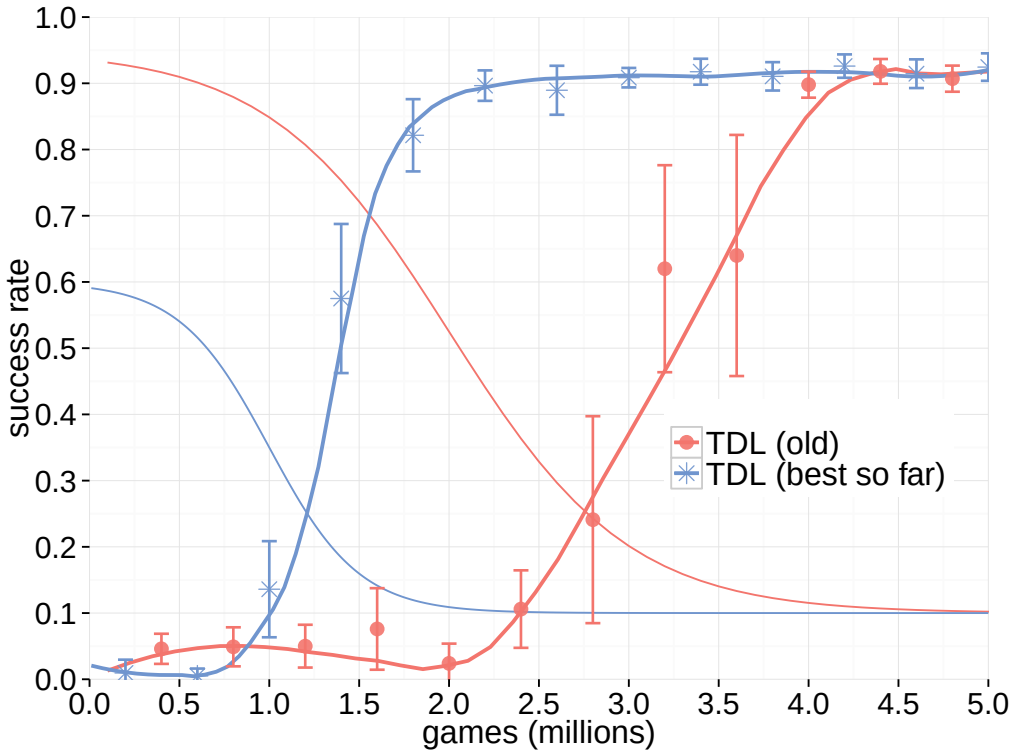

Figure 4.1: Starting-Point of the Case Study. In our previous work [36, 35, 37], we initially found a TDL-agent (red curve, TDL-old) that was able to learn the game (reach a success-rate of 80%) after around 3.7 Mio. training-games. After a careful tuning of step-size parameter and exploration rate the 80-% success-rate is now crossed after around 1.6 Mio. games (blue curve). The here shown curves are the average of 10 runs each. The agents were evaluated every 100.000 training games.

4.2 Temporal Coherence in TD-Learning

4.2.1 Initial Results using TCL

Since there were initially no suitable parameter settings known that could be applied to *TCL* we simply choose the settings for both TDL variants ([*TD1*] and [*TD2*]) described below in Sec. *Starting Point*. The results are shown in Fig. 4.2. The results seen there are rather disappointing, both *TCL* variants cannot cross the 80% success rate and are clearly outperformed by *TDL*. Further tests showed, that the global step size α was too small: For the effective step size parameter we always have $\alpha\alpha_i < \alpha$, since $\alpha_i < 1$ for all i. For this reason, the global step size has to be

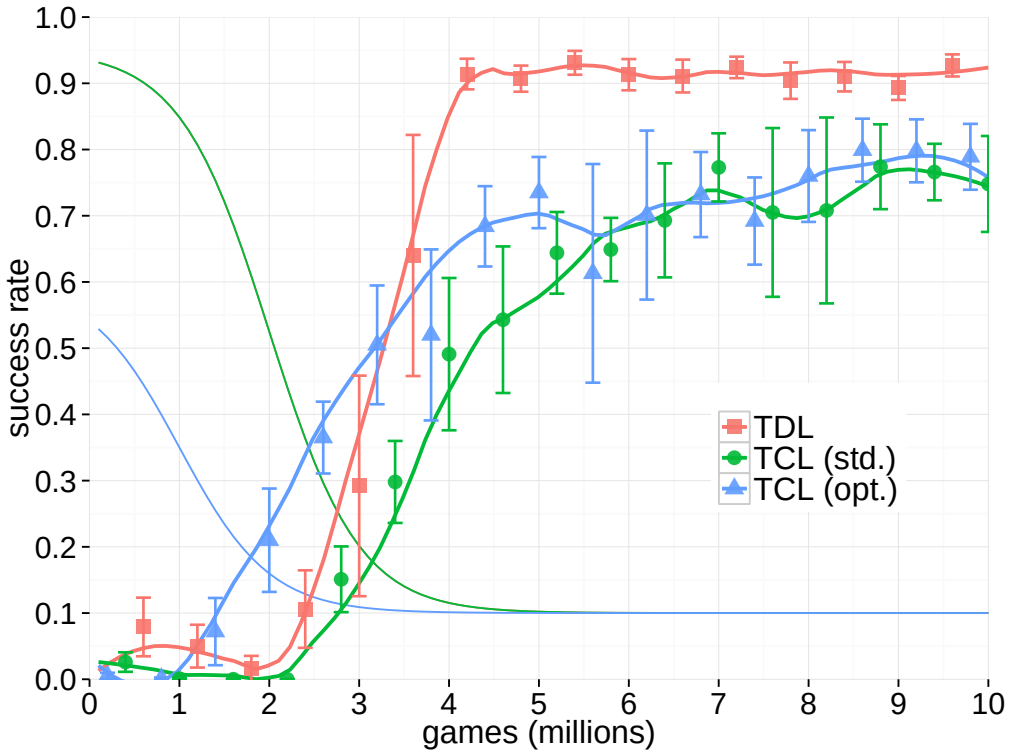

Figure 4.2: Initial Tests using *Temporal Coherence*. Because no information on the parameter-settings is available, we initialize both shown *TCL*-agents with the best two TDL-parameter settings described in the previous section (4.1). However, the results are rather disappointing (blue and green curve), both *TCL*-agents stay constantly beneath the 80% success-rate and are clearly outperformed by *TDL*. Further tests showed, that global step size α was too small and the exponential decay was interfering with the self-adaptive feature of *TCL*.

assigned with higher values. Furthermore, the exponential decay is interfering with the self-adaptive feature of *TCL*. A better choice is simply using a constant or near to constant learning rate. Note that for the initial results we updated the counters A_i and N_i in a slightly different way. Instead of adding the recommended weight change $r_{i,t} = \delta_t \nabla_{w_i} V(\vec{w}_t, s_t)$ (denoted as the [r] update rule) to the counters, we used the TD error signal δ. Also, the bias-weight is still updated according to the classical *TDL* update rule, using only the global learning rate α (which was an unnoticed bug, that we fixed later).

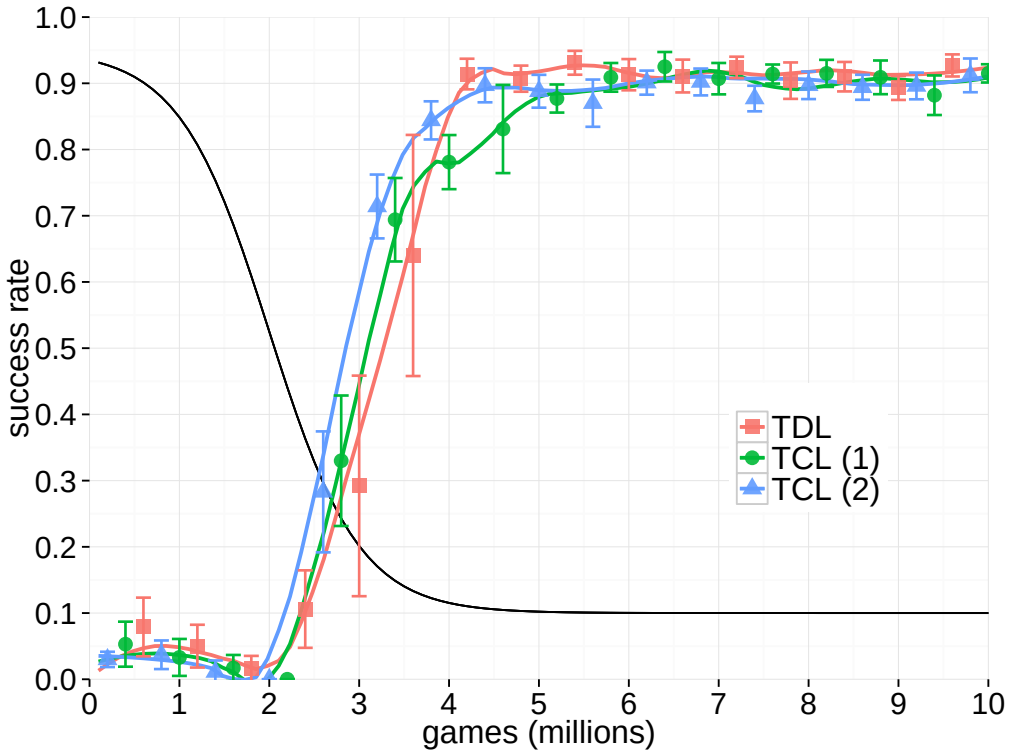

Figure 4.3: First improvements using the *TCL*-update-scheme. The main reason that *TCL* could not deliver the desired results before is, that the global step size parameter α was chosen too small. Slightly higher values finally improve the results significantly. For *TCL* (1) we choose $\alpha_{init} = \alpha_{final} = 0.02$ and for *TCL* (2) a nearly constant step-size of $(\alpha_{init} = 0.03, \alpha_{final} = 0.02)$. The remaining parameters (the black curve represents the *exploration rate*for all three agents) are chosen in the same way as for *TDL*. For the first time *TCL* is comparable to *TDL*.

4.2.2 First Improvements

First improvements – using the *TCL* update scheme – could be achieved, after choosing a larger, nearly constant global learning rate. For *TCL* (1) we choose $\alpha_{init} = \alpha_{final} = 0.02$ and for *TCL* (2) a nearly constant step-size of $(\alpha_{init} = 0.03, \alpha_{final} = 0.02)$. The remaining parameters (the black curve represents the *exploration rate* for all three agents) are chosen in the same way as for *TDL*. For the first time *TCL* is comparable to *TDL*. The results are shown in Fig. 4.3. We found, that especially near constant global learning rates lead to good results. Therefore, we choose $\alpha_{init} = \alpha_{final}$ for *TCL*.

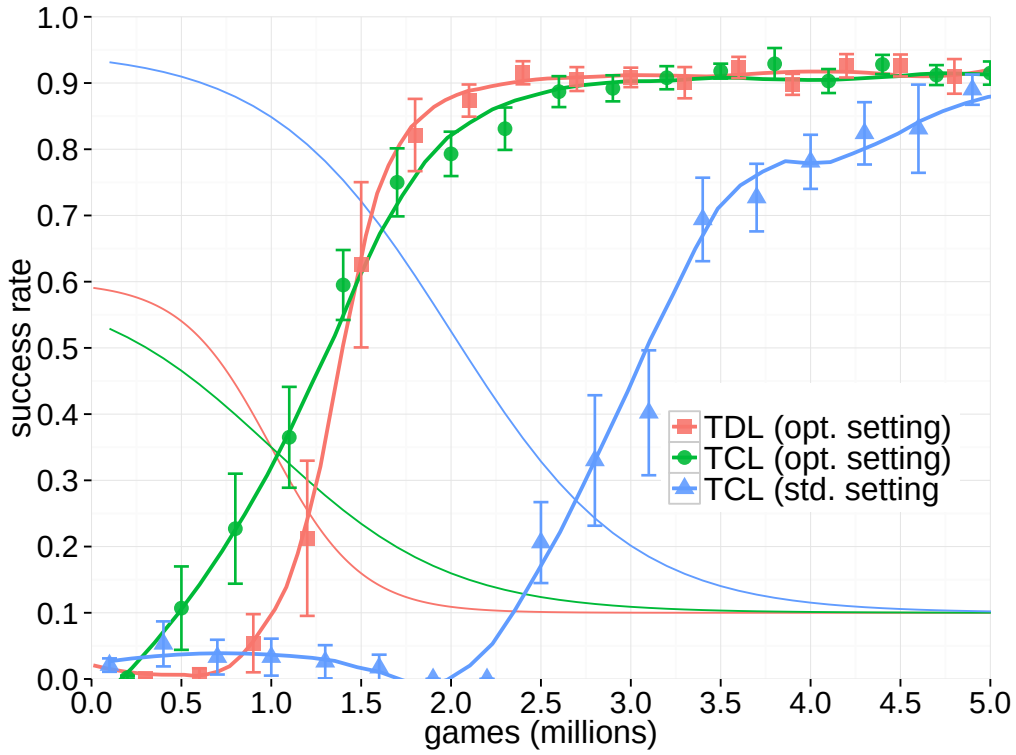

Figure 4.4: Trying a different *exploration rate* (green curve) for TCL leads to a faster training-progress, compared with the previous results (blue curve). *TDL*(red) appears to learn slightly slower in the beginning than Green, but speeds up and crosses Green after around 1.5 million training games. The reason for this appears to be the behavior of both *exploration rate*s (Green and Red), which indicates, that smaller initial values are more desirable.

Fig. 4.4 shows three training runs for different exploration rates. TDL and TCL (green curve) use similar settings, however, the slope of the *exploration rate* is chosen in a slightly different way for both experiments. TCL (green) has a higher training speed in the beginning, when the *exploration rate* is lower than for TDL. As soon as the *exploration rate*s of both curves cross, the training curve for TDL speeds up. This behavior of both curves indicates, that low *exploration rate*s somehow seem to be more desirable. For TCL again, the global learning rate is set to 0.02. However, we could not find that *TCL* could speed up the training significantly. When the *exploration rate* ϵ is chosen in the same way for TDL and TCL, the training curves are very similar. In contrast to Beal & Smith [6, 7] we cannot find TCL to speed up the training process. In the following section (Section 4.2.3) we will investigate the robustness of TCL, thus, the effect of different learning rates on the training process. We expect, that TCL should be able to handle a high α, by automatically decreasing the individual learning rates α_i.

4.2.3 Comparison of TCL and TDL for larger Step-size Range

Beal & Smith stated, that TCL would automatically self-adjust the parameters [7]:

> *"The parameter α can be set high initially, and the α_i then provide automatic adjustment during the learning process."*

In this section we will investigate this statement by screening a large range of α values and comparing the results for TDL and TCL. In Fig. 4.5 the asymptotic success rate of *TCL* for different learning rates is displayed. For every point 10 runs with 10 million training games were performed. For both, *TCL* and *TDL* α is kept constant. The asymptotic success rate is calculated by averaging the results for the last 2 Mio. games of all runs (in total 200 points). The standard deviation for the 200 values is indicated with the error-bars. Note that the X-axis is logarithmic. We analyzed two different n-tuple systems. The top plot shows the results for our standard n-tuple system with 70×8 n-tuples, for the bottom plot, we use a system with shorter n-tuples (70×7 n-tuples). In total, we found for both cases TCL not to have a larger area of high success rates than TDL, it is only shifted to larger values. We can also see, that for increasing α, the asymptotic success rate for TCL falls faster than for TDL (however, this is due to a bug in the software, where the bias-weight is still updated using the TDL update).

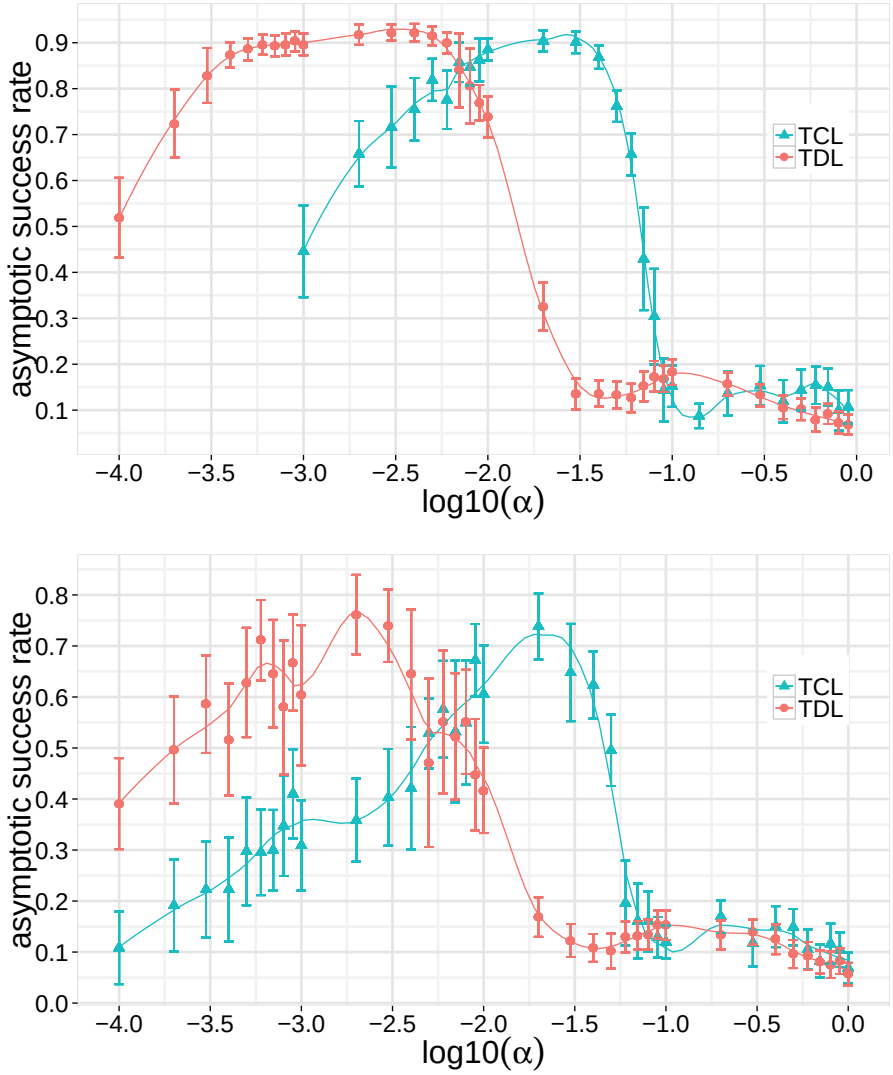

Figure 4.5: Asymptotic Success-Rate of *TDL* and *TCL* against global learning rate α. For every point 10 runs with 10 Mio. training games each are performed. For both, *TCL* and *TDL* α is kept constant. The asymptotic success rate is calculated by averaging the results for the last 2 Mio. games of all runs (in total 200 points). The standard deviation for the 200 values is indicated with the error-bars. Note that the X-axis is logarithmic.
Top: 70×8-tuple. *TCL* is – differing from our expectations – not more successful in a broader range, both curves appear to shifted.
Bottom: 70×7-tuple. Also in this case, *TCL* does not show any advantage.

4.2.4 Extended Parameter Tuning

Already the results presented in in Sec. 4.2.2 indicated, that lower exploration rates appear to be more beneficial for the learning processes of the agent and as we found in the previous section (Section 4.2.3), α cannot be chosen arbitrary. Therefore, we decided to perform an extended parameter tuning, trying to find suitable values mainly for the global learning rate α and the *exploration rate* ϵ. In total, we run around 35 experiments with different parameter combinations. Especially tuning the *exploration rate* could increase the learning of TCL significantly. From formerly around 2 Mio. training games, now only 1 Mio. games are needed, to cross the 80% success rate.

Contrary to our earlier assumptions, the learning speed can be improved, if ϵ is chosen low ($\epsilon < 0.2$) even in the initial phase of the training.

Nevertheless, final values of $\epsilon < 0.1$ decreased the asymptotic success rate of the considered agents. We suppose, that too small exploration in the final phase of the learning process may lead to an overtraining, so that the agents "forget" important *Connect Four* states in order to reduce the approximation error for other (wrongly weighted as more important) states.

The main improvement for TCL in Fig. 4.6 was achieved by tuning the *exploration rate*. Therefore, it seems natural to apply this setting to TDL as well, as done in Fig. 4.7. And indeed, the new setting for ϵ has the same effect on TDL as on TCL. The learning speed for TDL is significantly improved and seems to outperform TCL slightly (considering the crossing of the 80% and 90% success rate). We still cannot note any advantage of *TCL* over *TDL*. The tuning of the *exploration rate* improved the training progress for both, *TCL* and *TDL* equally.

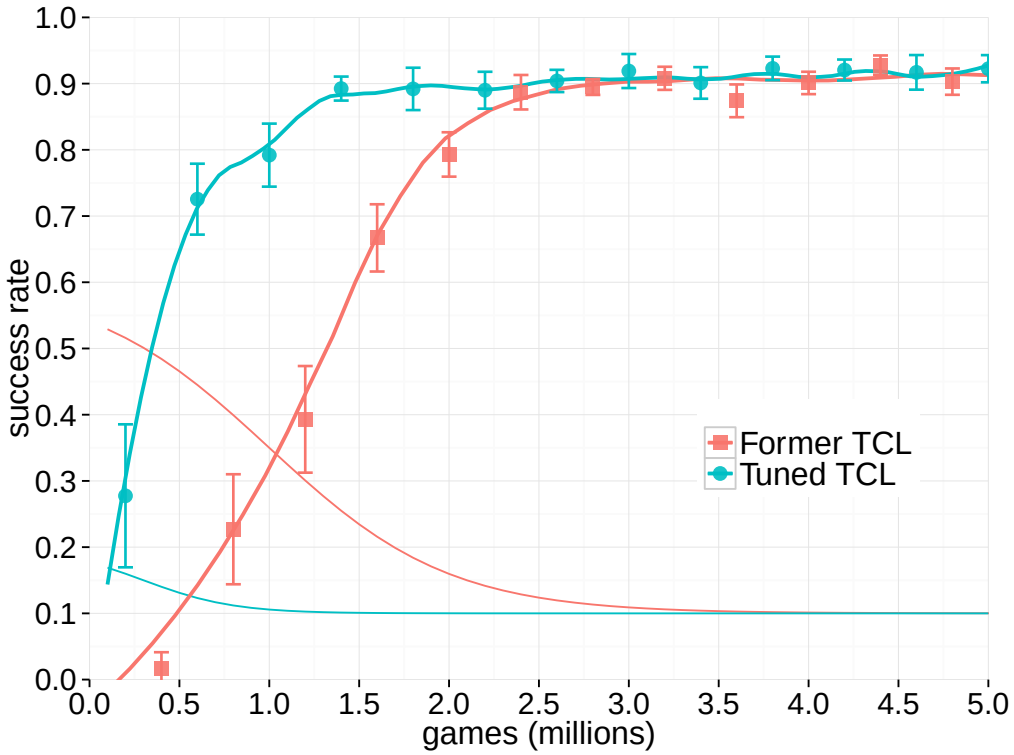

Figure 4.6: Results for an extensive tuning of *TCL*. Around 30 experiments with different parameter-settings were performed in order to tune *TCL*. The main improvement was achieved by tuning the *exploration rate*, shown in the plot by the thinner lines without points. It appears, that also in the beginning of the training, a lower *exploration rate* leads to better results. For tuned *TCL* the initial *exploration rate*is decreased to $\epsilon_{init} = 0.17$, from before $\epsilon_{init} = 0.53$ (red curve, former *TCL*). The final value in both cases is $\epsilon_{final} = 0.1$. The learning rate for tuned *TCL* (blue) was slightly increased as well, from $\alpha = 0.03$ for the former *TCL*-agent (red) to now $\alpha = 0.04$ (blue, tuned *TCL*).

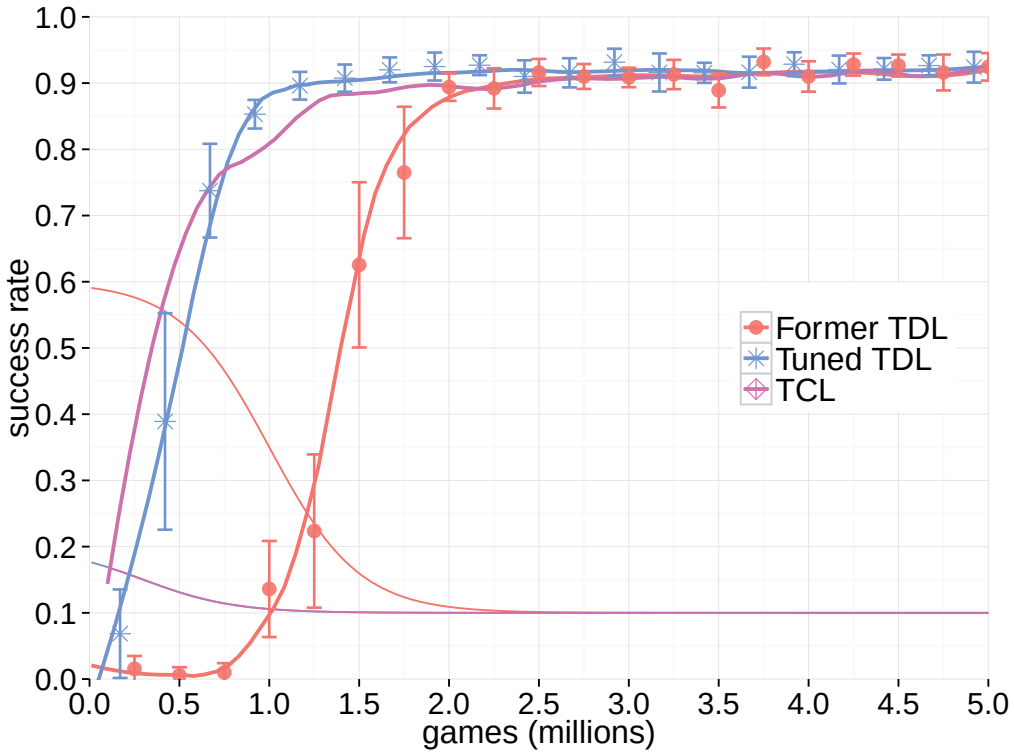

Figure 4.7: Parameter tuning for TDL. After we found the optimized parameters for *TCL*, which were mainly achieved by tuning the *exploration rate* (as seen inFig. 4.6), we apply the new ϵ-setting to *TDL* (thinner purple curve without points). The initial value is decreased to now around $\epsilon_{init} = 0.17$, from before $\epsilon_{init} = 0.53$ (red curve). In all cases the final value is set to $\epsilon_{final} = 0.1$. The performance of *TDL* increases significantly, *TDL* even slightly outperforms *TCL*. We still cannot note any advantage of *TCL* over *TDL*. The tuning of the *exploration rate* improved the training progress for both, *TCL* and *TDL* equally.

4.2.5 Implementation of the original TCL Algorithm

As already mentioned, in the previous sections, all training runs using TCL were performed by using the [δ]-rule. Instead of adding the recommended weight changes $r_{i,t} = \delta_t \nabla_{w_i} V(\vec{w}_t, s_t)$ (denoted as the [r] update rule) to the counters (as proposed by Beal & Smith in [7, 6]), we used the TD error signal δ. The relation between *TCL* [delta] and *TCL* [r] can be expressed with $r_{i,t} = \delta_t\big(1 - V^2(s_t)\big)$. In this section we will investigate the original TCL scheme ([r]-update) and compare the results to our previous results (created using the [*delta*]-rule).
Additionally, the results presented in this section (and in all following sections) use the corrected version of the program, where the bias-weight now is also updated using either TDL or TCL (before, just the TDL update was used, which is not correct). In Fig. 4.8 the results for TCL – using the [r]- and [δ]-update rule – and TDL are shown (all experiments have the same exploration). The new results for TCL [r] appear to be slightly worse than for TCL [δ].

For the original update rule using the *RWC* to update the counters A_i and N_i we again screened a larger range of α and measured the asymptotic success rate, as seen in Fig. 4.9. *TCL* (old) uses the δ-update rule to adjust the counters A_i and N_i, *TCL* [r] implements the update as originally proposed by Beal and Smith [6] and uses the *RWC* for A_i and N_i. However, the slightly better results for *TCL* [r] in this figure are due to the mentioned bug: Older program versions adjusted the bias-weight with the classical *TDL*-update rule – regardless, whether TCL or TDL were selected –, so that *TCL* (old) is more sensitive to larger α-values. Now, the range of higher success rates could be extended slightly with increasing α values. But also with the new implementation TCL [r], we still cannot observe any improvement to TDL.

4.2.6 Update Episodes in TCL

The original TCL implementation [7, 6] has the further option that the reward $r_{i,t}$ may be accumulated over a sequence of steps before a real weight update takes place. This balances a tradeoff between faster changing learning rates (short sequences) and accumulation of statistic evidence (long sequences). We did not consider this possibility until now. In Listing 5 the pseudo code for this option is given. Now, instead of updating the weights and the counters in every time step, first an episode of length L is completed before adjusting w_i, A_i and, N_i. The recommended weights changes $r_{i,t}$ and the absolute recommended weights changes $|r_{i,t}|$ for one episode have to be accumulated in separate counters for this purpose.

The results for different episode lengths (defined by a certain number of moves) can be seen in Fig. 4.10. We can observe, that already episode lengths (*EL*) slightly

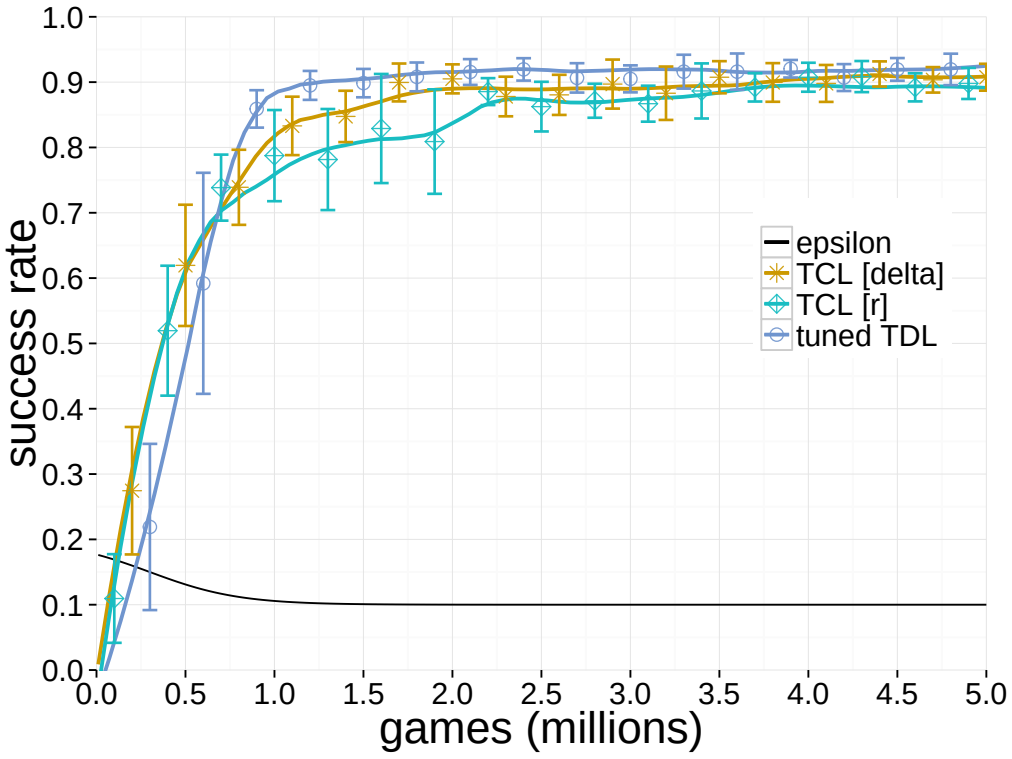

Figure 4.8: Comparison of *TCL* using the δ-update rule and using r-update rule. In the previous sections only the δ-update rule was used for *TCL*. Now, for the first time, we test *TCL* with the original approach, that uses the recommended weight change $r_{i,t}$ to adjust A_i and N_i. The relation between *TCL* [δ] and *TCL* [r] can be expressed with $r_{i,t} = \delta_t\big(1 - V^2(s_t)\big)$. The experiments show, that *TCL* [δ] is slightly better. For comparison purposes, also *TDL* is displayed. For all three experiments the *exploration rate* was chosen in the same way, as indicated by the black curve.

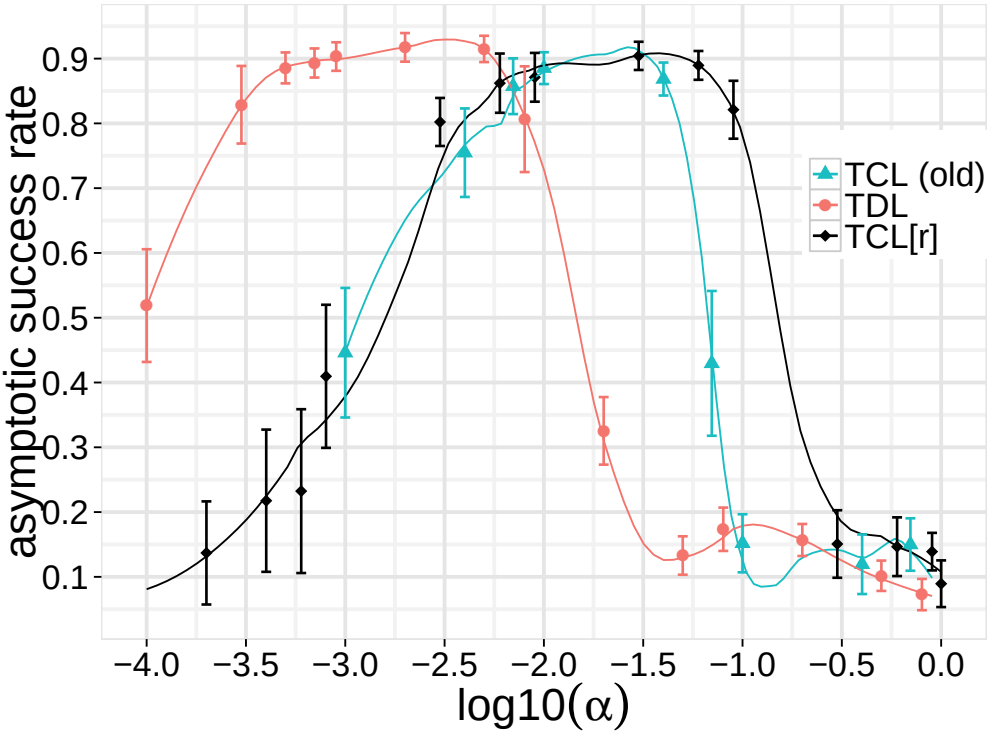

Figure 4.9: Asymptotic success rate of *TDL* and *TCL* for large α-range. *TCL* (old) uses the δ-update rule to adjust the counters A_i and N_i, *TCL* $[r]$ implements the update as originally proposed by Beal and Smith [6] and uses the *RWC* for A_i and N_i. However, the slightly better results for *TCL* $[r]$ in this figure are due to a fixed bug: Older program versions adjusted the bias-weight with the classical *TDL*-update rule, so that *TCL* (old) is more sensitive to larger α-values. For every point 10 runs with 10 Mio. training games each are performed. For both, *TCL* and *TDL* α is kept constant. The asymptotic success rate is calculated by averaging the results for the last 2 Mio. games of all runs (in total 200 points). The standard deviation for the 200 values is indicated with the error bars. Note, that the X-axis is logarithmic.

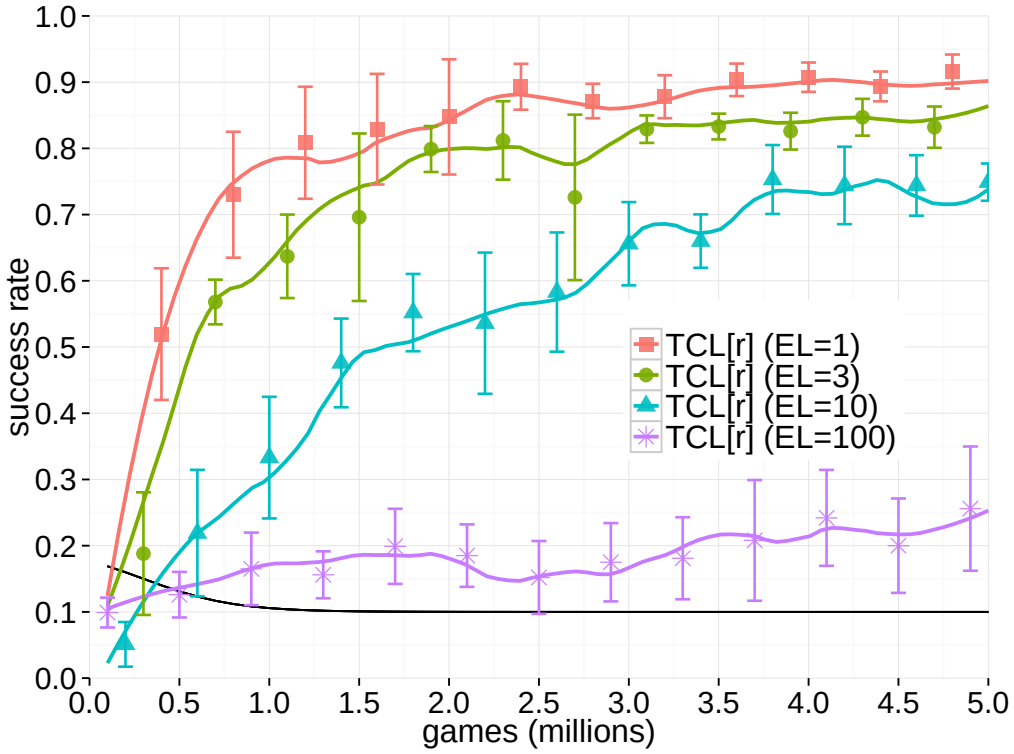

Figure 4.10: Update-Episodes in *TCL*. The original *TCL* algorithm by Beal and Smith includes the possibility of defining update episodes [6]. When using update episodes the recommended weight changes are accumulated for a given sequence of time steps. The weights w_i are only updated once after every episode. With this approach – depending on the episode length – random noise will be canceled out and the weight updates (and counter updates for N_i, A_i) are statistically more reliable. However, slower changing weights reduces the learning speed, as seen in this graph. Update episodes can not increase the performance of the *TCL*-agent. Already an episode length of $EL = 3$ significantly reduces the success of the agent.

Algorithm 5 General pseudo code of the proposed TCL-update algorithm including the possibility of update-episodes [6]. The weights w of the value function are not updated until a certain episode is completed.

Initialize in the beginning $A := \vec{0}$, and $N := \vec{0}$

Define global learning rate c

Define length of the update-episodes L

Determine in every time-step $r_{i,t} := \delta_t e_{i,t}$ ▷ Recommended weight changes

After an update-episode is completed, do:

for $i \in \{1, \ldots, N\}$ **do**

$$\alpha_i = \begin{cases} \frac{|N_i|}{A_i}, & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases}$$

$w_i \leftarrow w_i + c\alpha_i \sum_{t=0}^{L} r_{i,t}$ ▷ TD-update for the weight

$A_i \leftarrow A_i + \sum_{t=0}^{L} |r_{i,t}|$ ▷ Update accumulating counter A

$N_i \leftarrow N_i + \sum_{t=0}^{L} r_{i,t}$ ▷ Update accumulating counter N

end for

larger than 1 result in a slower learning process and lower asymptotic success rates. The main idea of this approach – to cancel out random noise and therefore get more statistically reliable weight updates (and counter updates for N_i, A_i) – could not be confirmed. The slower changing weights appear to reduce the learning speed, as seen in the graph. For our learning task in *Connect Four* , we found that update episodes cannot increase the performance of a *TCL*-agent.

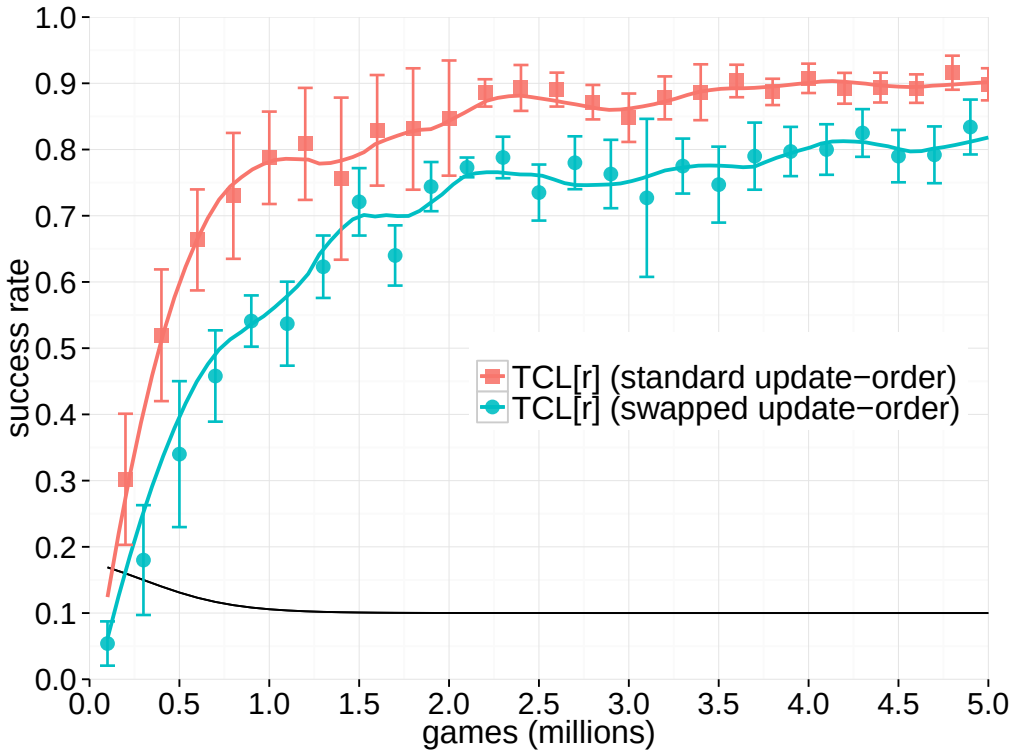

Figure 4.11: Modifying the operational order in TCL. The standard *TCL*-algorithm has the following operational order: *1.* Update weights w_i based on previous counter values A_i and N_i. *2.* Adjust counters. Using the opposite order by swapping *1.* and *2.* appeared to be sensible as well. However, the results for the swapped operational order are worse as seen in this graph. We therefore keep the classical order. The black curve describes the *exploration rate* for both experiments.

4.2.7 Modifying the Operational Order in TCL

When using TCL, two different operational orders are thinkable: (1) First, update the weights w_i and then the counters A_i and N_i or (2) First update the counters A_i, N_i and subsequently the weights w_i. In the original algorithm (1) is proposed. Nevertheless, we want to investigate the second possibility at this point as well. The results for both cases are shown in Fig. 4.11. It appears, that operational order (1) delivers better training results. In the following we therefore keep the classical update order for our experiments.

4.3 Incremental Delta-Bar-Delta

Incremental Delta-Bar-Delta (IDBD), as described in Section 2.6, is another approach to adjust the individual learning rates α_i. Fig. 4.12 allows a comparison between TCL and IDBD. Even though TCL has a slightly faster learning progress in the beginning, IDBD in the end delivers better results: IDBD crosses the 80% success rate after around 700 000 games, where TCL needs around 900 000 training games. It has to be mentioned, however, that – the original IDBD algorithm is only defined for linear functions – IDBD normally has to run TDL without the sigmoid function, which might counteract any improvements through the individual learning rates. However, as shown in Fig. 4.12, we tried the original IDBD algorithm for the non-linear case as well (*tanh* in the output of the net) but did not find any significant differences. It has to be noted however that the IDBD algorithm has to be reformulated for the nonlinear case, so that simply using the (linear) IDBD in conjunction with tanh in the output of the net is not the right procedure. Results with a correctly reformulated nonlinear IDBD are found in [4]
Furthermore, we found that there is some dependence on the right choice of parameters β_{init} and θ. If θ deviates from 0.1 no or only carefully selected values for β_{init} yield in good results. If $\theta = 0.1$ is chosen, there is a broader range of values $\beta_{init} \in [-7.5, -5.5]$ in which IDBD can deliver good results. Outside this β_{init} range the training breaks down. The results for different β_{init} values is shown in Fig. 4.13. Values $\beta > -4.6$ could not be evaluated, the linear approximation function $V(s_t)$ tends against infinite values in those cases. The highest asymptotic success rate can be found at $\beta = -5.8$. This would lead to an initial $\alpha_i = e^{-5.8} \approx 0.003$ which already appeared to be an appropriate value for the classical TDL-update scheme. The meta-learning-rate is chosen to be $\theta = 0.1$ for all experiments.

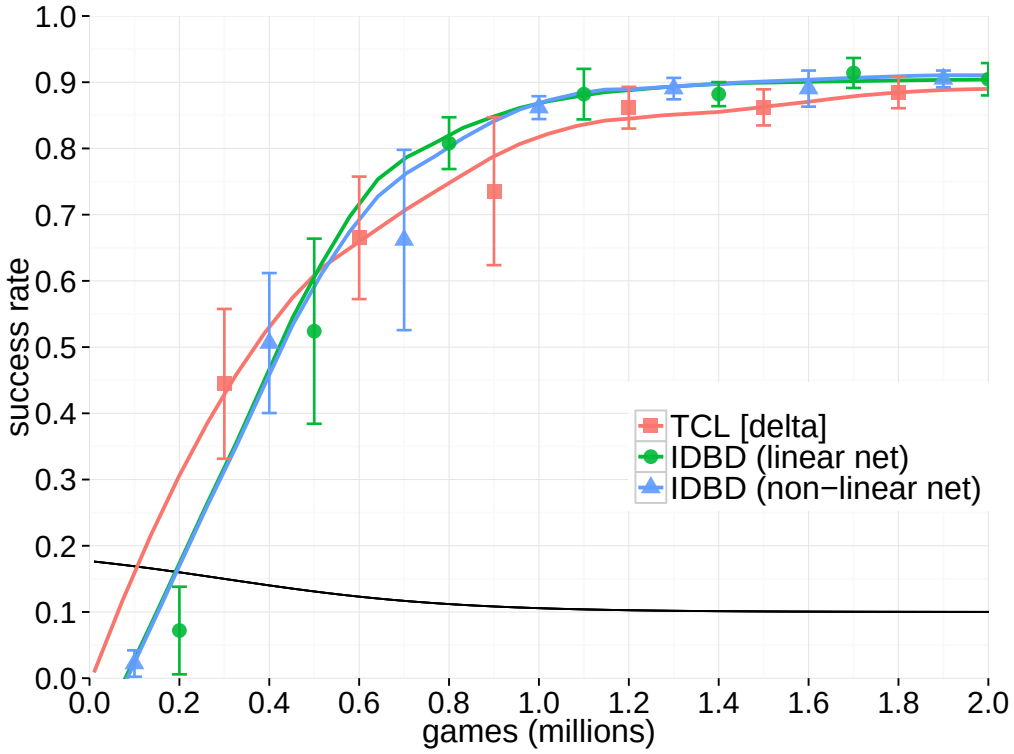

Figure 4.12: Rough Description: Results for *IDBD* with $\beta = -5.8$ and $\theta = 0.1$ which we compare with *TCL* (first curve according to legend, using the δ-update rule). The second curve describes the learning-progress for a linear value-function. We also tested the non-linear variant with the activation-function $\tanh(\cdot)$ in the output of the net. The *exploration rate* (black curve) is chosen to be the same in all three experiments. Although no extensive tuning of the *IDBD*-parameters was done, we can see that *IDBD* outperforms *TCL*. Further tuning could improve *IDBD* slightly.

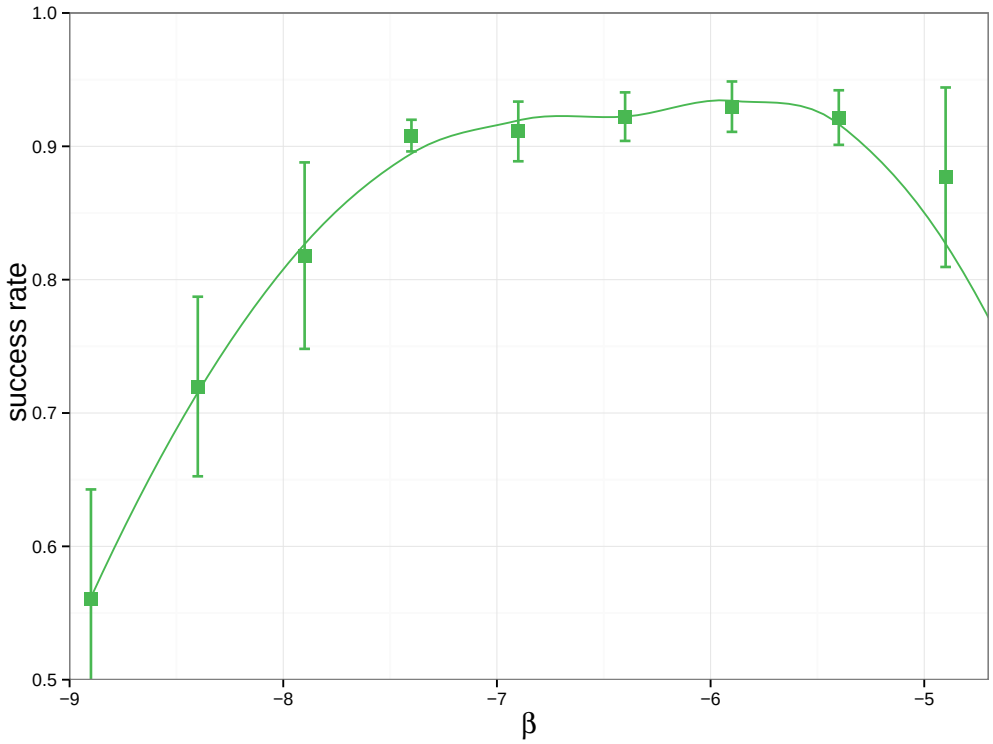

Figure 4.13: *IDBD*: Asymptotic success rate for β in the range of $[-9, -4.6]$. Values $\beta > -4.6$ could not be evaluated, the linear approximation function $V(s_t)$ tends against infinite values in those cases. The highest asymptotic success-rate can be found at $\beta = -5.8$. This would lead to an initial $\alpha_i = e^{-5.8} \approx 0.003$ which already appeared to be the appropriate value for the classical TDL-update scheme. The meta learning rate is chosen to be $\theta = 0.1$ for all experiments. For every point 10 runs with 5 Mio. training games each are performed. For both, *TCL* and *TDL* α is kept constant. The asymptotic success rate is calculated by averaging the results for the last 2 million games of all runs (in total 200 points). The standard deviation for the 200 values is indicated with the error bars.

4.4 Temporal Coherence with Exponential Scheme

As mentioned in [30], the geometric step-size in *IDBD* has the desirable property, that a fixed change in the memory parameter $\Delta\beta_i$, α_i will adjust α_i by a fixed fraction of its current value. This encouraged us to try a few adjustments in *TCL* with similar properties.
The classical *TCL* algorithm calculates α_i in the following way:

$$\alpha_i = \begin{cases} g\big(\frac{|N_i|}{A_i}\big), & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases}, \tag{4.1}$$

where the transfer function $g(x)$ is simply defined as the identity function ($g(x) = x$). However, we are not necessary restricted to the identity function. Instead, functions with other features could also be an interesting option. We tried a function with the form

$$g(x) = e^{\beta(x-1)}, \tag{4.2}$$

so that

$$\alpha_i = \begin{cases} e^{\beta(\frac{|N_i|}{A_i}-1)}, & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases} \tag{4.3}$$

In the following we will denote the above scheme as *TCL-EXP*. Another possible function could be a piecewise linear function, for instance as shown in figure 4.14 together with the classical *TCL* transfer function and *TCL-EXP*. The piecewise linear function, has in $x = 1$ the same slope as *TCL-EXP* and the value $g(0) = e^{-\beta}$ for $x = 0$. Note, that all transfer functions g(x) operate in the range $0 < x < 1$.

In total, TCL-EXP is the standard TCL algorithm with only one detail changed: the exponential transfer function for the learning rate, see Eq. (4.2). This brings a remarkable increase in speed of learning for the game *Connect Four*, as our results in Fig. 5 show: TCL-EXP reaches the 80% success rate after about 500 000 games instead of 615 000 games (IDBD) or 670 000 (Tuned TDL). At the same time it reaches asymptotically a very good success rate. We varied the parameter β systematically between 1 and 7 and found values in the range of $\beta \in [2; 3]$ to be optimal.
To test the importance of geometric step sizes, we did another experiment: We replaced the TCL-EXP transfer function by a piecewise linear function (PL) as also shown in Fig. 4.15, having the same endpoints and same slope at x = 1. The results for PL in Fig. 4.15 are worse than TCL-EXP. Therefore, it is not the slope at x = 1 but the geometric step size which is important for success.

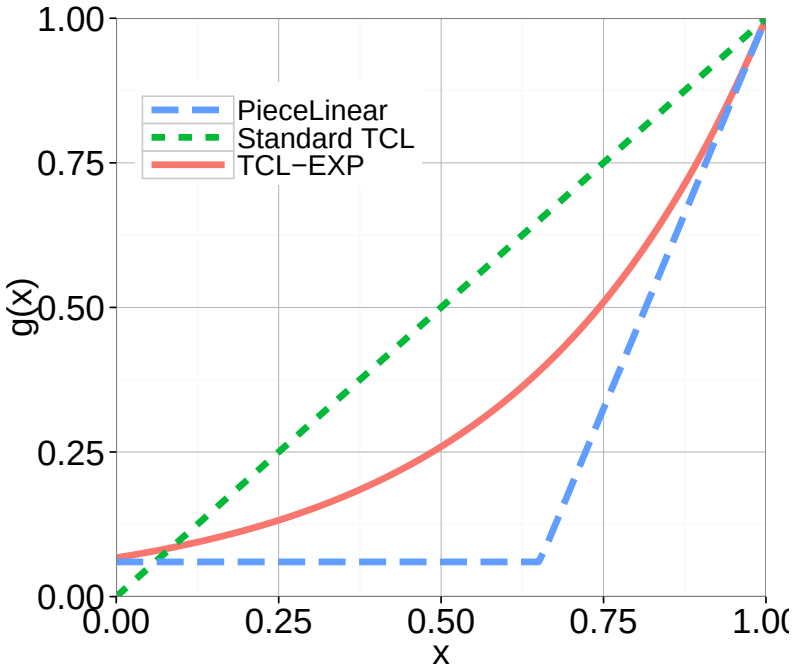

Figure 4.14: Different transfer functions $g(x)$ for *TCL*. The classical approach in *TCL* is represented by the green line. We propose a exponential transfer function (*TCL-EXP*, red curve), with $\beta = 2.7$, that has – similar to *IDBD* – a geometric step size as defined in Eq. 4.2. The here shown piecewise linear function has the same slope as TCL-EXP in $x = 1$ and the same value in $x = 0$.

4.4.1 Further Tuning of the Exploration Rate

After finding an exploration strategy with $\epsilon_{init} = 0.2$, $\epsilon_{final} = 0.1$ and $\epsilon_{ip} = 0.3 \cdot 10^6$, a next convenient step would be, to simply try $\epsilon = 0.1 = const$. A single experiment with this setting did not appear promising, so that further investigations were not performed until now. At this point, we present the results for 20 runs using this constant *exploration rate*. And indeed, $\epsilon = 0.1$ results in a slightly faster learning process. Exemplary, in Fig. 4.16, the results for TCL-EXP and standard TDL are shown. In both cases the learning speed is slightly increased.

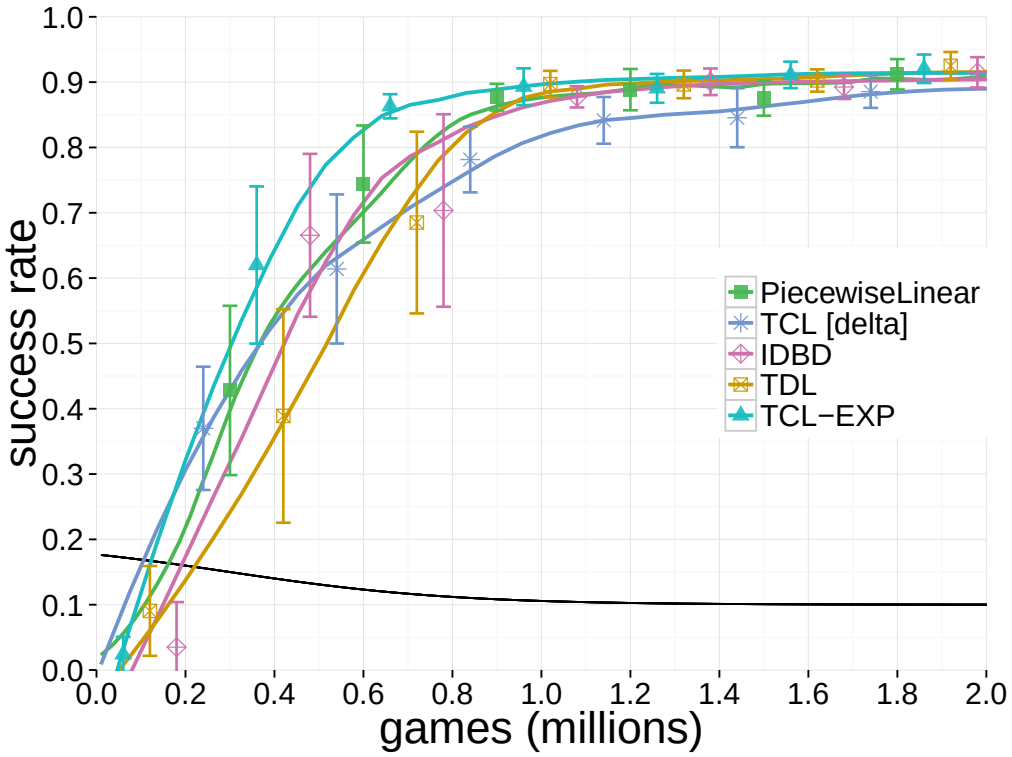

Figure 4.15: Comparison of the adjusted *TCL*-algorithm *TCL-EXP* with previous results. For the first time an algorithm significantly outperforms *TDL*. *TCL* crosses the 80% success rate after around 550 000 training games, whereas *TDL* needs around 800 000 games. *TCL-EXP* also learns slightly faster than *IDBD*, which needs 700 000 games to reach the 80% level. When using a piecewise linear function (as described in Fig. 4.14), the results are comparable to *IDBD*. The black curve represents the *exploration rate* for all experiments.

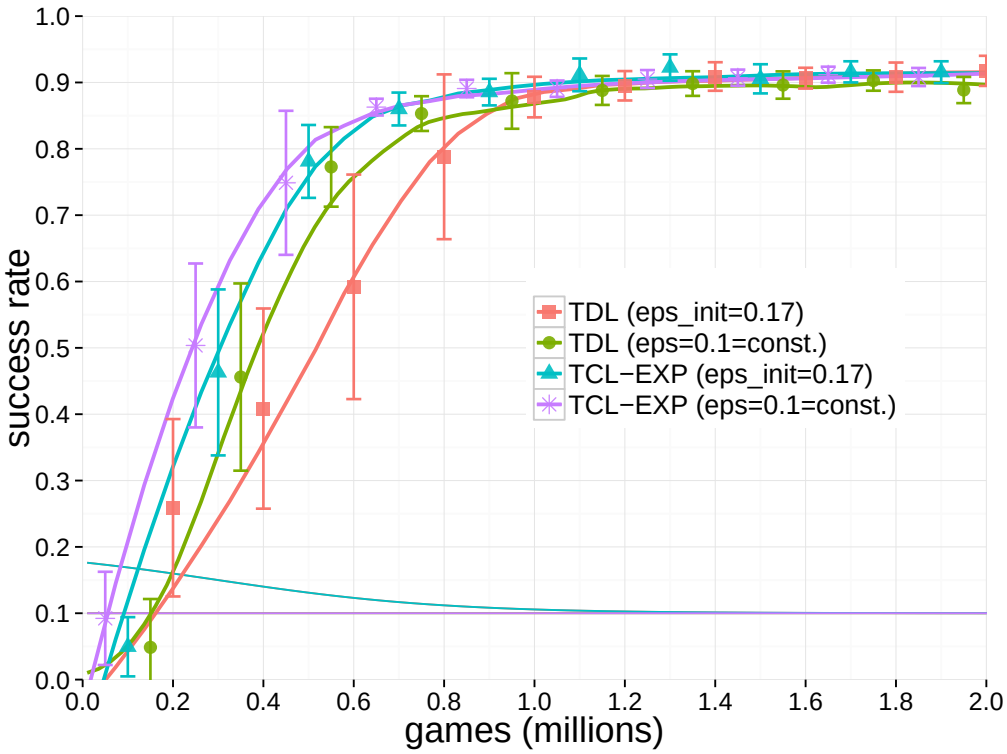

Figure 4.16: Further tuning of the *exploration rate* from $\{\epsilon_{init} = 0.17,\ \epsilon_{final} = 0.1\}$ to $\epsilon = 0.1 = const.$, exemplary for *TDL* and *TCL-EXP*. The new ϵ-value appears to speed up the training slightly more for *TDL* than *TCL-EXP*.
After finding a tuned *exploration rate* in Sec. 4.2.4, the next convenient step was trying a constant *exploration rate* with $\epsilon = 0.1 = const.$ However, a single experiment did not appear that much promising, so we delayed further tests until here. And indeed, a constant *exploration rate* increases the learning speed slightly. Exemplary, the results for TDL and TCL-EXP are shown. The graph shows the average of 20 runs each.

4.5 Q-Learning

As described in Section 2.3.4 the main difference of Q-Learning to TD-Learning is, that Q-Learning attempts to learn an action-value function rather than a state-value function. One desirable property of Q-Learning is, that the correct values for state-action pairs can be learned, even if a non-optimal policy is followed. This implies, that also explorative actions can be used to update the weights of the system. This is in contrast to the classical TD-Learning approach, where the correct state-values can be only be learned, if the optimal policy is followed. Following a non-optimal policy in TD-Learning leads to a back-propagation of wrong rewards which again results in a wrong estimation for the visited states.

In order to realize Q-Learning for our TD-Learning framework with n-tuple systems, we now create one lookup table (LUT) for every possible action. In Connect Four there are only up to 7 possible actions, so that 7 LUTs are needed for every n-tuple. The weights of the system are updated in the following way: For every n-tuple the board is sampled and the LUT-indexes are determined. In contrast to the standard TD-Learning approach, Q-Learning now selects for every n-tuple the corresponding LUT based on the action and updates the weights only in this single LUT. The six remaining LUTs for the n-tuple remain unchanged.

In Fig. 4.17 we show the results for the basic Q-Learning algorithm with different constant exploration rates. The first observation that we can make is, that the speed of learning is significantly lower than for the comparable TD-Learning approach – for Q-Learning, around 2 million training games to reach the 80% success rate, instead of 600 000 games for TD-Learning. Q-Learning is a factor of about 6-7 slower than the corresponding TD-Learning approach. One reason for this could be that Q-Learning has to maintain 7 times more weights, which are probably activated less frequently. This could lead to a slower convergence of the Q-Learning algorithm. Another observation we can make in Fig. 4.17 is that the learning process gets worse for increasing exploration rates. This observation also did not meet our first expectations, since Q-Learning should be capable of learning accurate values, regardless of which policy is followed. This may have to do with the evaluation-approach we chose for our agents. Strong agents only must be able to find a sequence correct moves against a perfect playing Minimax-agent, starting with an empty board. This means that large parts of the whole game-tree are not relevant for the decision process. So even if an agent has accurate values for a certain state, these values are not relevant if the state is not considered in the move-sequence of both agents

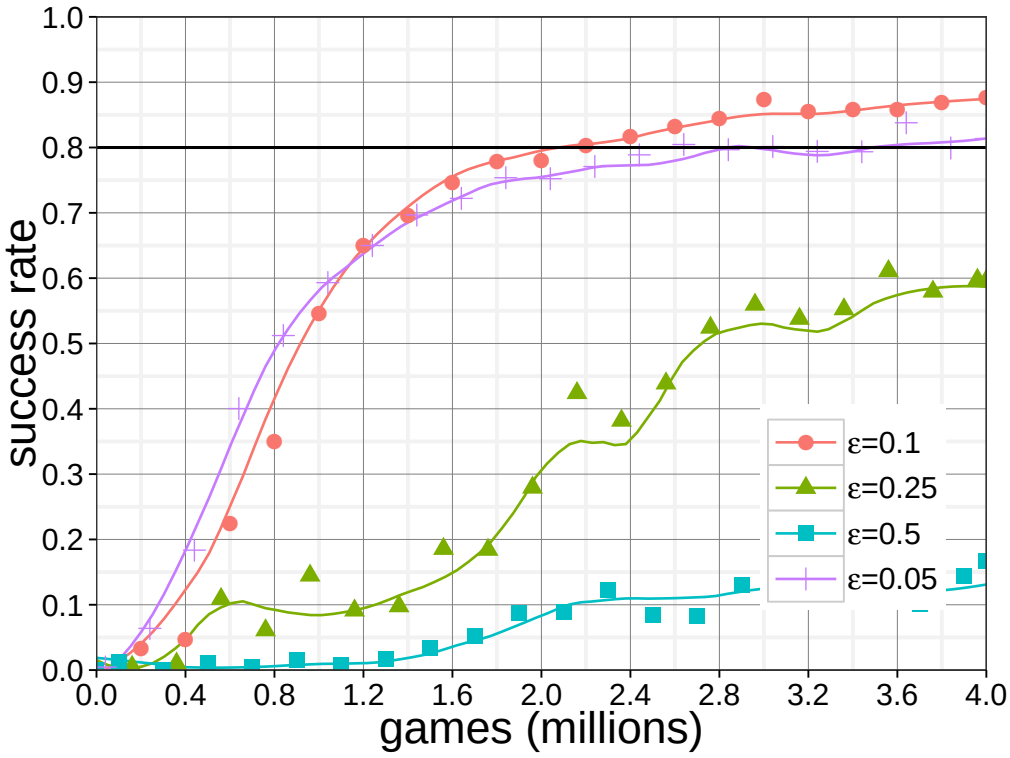

Figure 4.17: Results for basic Q-Learning with a constant exploration rate and without online adaptable learning rate algorithms. The best learning performance was achieved with a constant setting of $\epsilon = 0.1$, even though $\epsilon = 0.05$ leads to a slightly faster learning in the beginning. The global step-size parameter is chosen as before, decaying exponentially from $\alpha_{init} = 0.004$ to $\alpha_{final} = 0.002$. The number of games to learn the game is significantly larger than for standard TD-Learning (around 2 mio. training games to reach the 80% success rate, instead of 600 000 games for TD-Learning).

involved in the evaluation-match. Standard TD-Learning is somewhat biased in a way that it only attempts to learn a strong policy starting from an empty board; positions that are not considered by TD-Learning, since they are most likely not part of the desired policy will not be learned or forgotten over time. We believe that Q-Learning with higher exploration rates will explore many sub-trees of the game that are relevant for the evaluation process. In this sense, Q-Learning may lead to better results than TD-Learning, if the evaluation process would start with arbitrary boards and would not be limited to just the empty board.

Another problem when using high exploration rates in Q-Learning could be, that – since we are learning a generalizing function that has a very small number of weights in comparison to the number of game-states that are possible – many states with different real values are mapped to similar feature vectors which again would activate the same weights for many inputs. This would lead to many conflicting signals that are used to update the weights. When high exploration rates are used, the number of different states visited during the training remains high for the whole training-process, so that the weights receive more conflicting signals. On the other hand, small exploration rates would lead to near deterministic behavior and the agent visits only a very limited subset of all states which makes it much easier for the weights to converge.

4.6 Eligibility Traces

Machine-learning techniques – such as Reinforcement Learning (RL) – commonly have to deal with a variety of problems when they are applied to complex board games. One particular challenge of RL is the temporal credit assignment problem: Since no teacher signal is available, RL methods are solely dependent on rewards, which in board games are typically given at the end of a long sequence of actions. To address this problem of delayed rewards, Temporal Difference Learning (TDL) methods were developed. However, as we found in our work on the game *Connect Four*, the convergence of the learning process may be rather slow if training samples are not utilized efficiently in TDL. Even with the tuning-procedures and the other techniques (namely TCL and IDBD) we investigated in the last sections, still several hundreds of thousands of training games are needed to learn a complex game such as *Connect Four*.

In this chapter we study the benefits of eligibility traces added to this system. To the best of our knowledge, eligibility traces have not been used before for such a large system. Different versions of eligibility traces (standard, resetting, and replacing traces) are compared. We show that eligibility traces speed up the learning by a factor of two and also they increase the asymptotic playing strength.

4.6.1 Initial results

When implementing eligibility traces, it is not recommendable to work with the complete trace vector. Due to the sparseness in the vector – in every n-tuple, only a few weights are activated in each time step – it is more convenient to place the individual activated traces in a self balanced (height-balanced) binary tree (TreeMap in *JAVA*). This reduces the memory consumption of the training process and the computation time as well. The initial results for TCL-EXP using eligibility traces are shown in Fig. 4.18. As it is shown in the graph, eligibility traces significantly speed the training up, after around 300 000 games the agent crosses the 80% success rate for the first time (considering the smoothed curve through the averaged points). Furthermore, the asymptotic success rate is slightly higher than for TCL-EXP and TDL.

Generally, the $TD(\lambda)$ algorithm requires a certain degree of exploration (e.g., in completely deterministic environments) for the learning process, thus, the forced execution of random actions by the agent from time to time. When performing random moves, the value $V(s_{t+1})$ of the resulting state s_{t+1} is most likely not an appropriate approximation of $V(s_t)$. The backup of prediction $V(s_t)$ is normally skipped in this case. This also raises the question how to handle exploratory actions regarding the eligibility traces. We will consider two possibilities: 1) Simply resetting the trace vector and 2) ignoring random moves and leave the traces unchanged, which both may diminish the anticipated effect of eligibility traces significantly, when using higher exploration rates. For the initial results in Fig. 4.18, option 2 was used.
The results with eligibility traces, resetting on random moves, with $\lambda = 0.6$ can be seen in Fig. 4.19. This resetting option (TCL-EXP [res]) lets the agent learn slightly faster in the beginning, although the differences are not that significant. However, the asymptotic success rate for TCL-EXP [et] is higher than for TCL-EXP [res].

4.6.2 Replacing Eligibility Traces

Replacing eligibility traces, as described in Sec. 2.3.3, are comparable to the classical traces, but with one main difference: Although both – classical and replacing

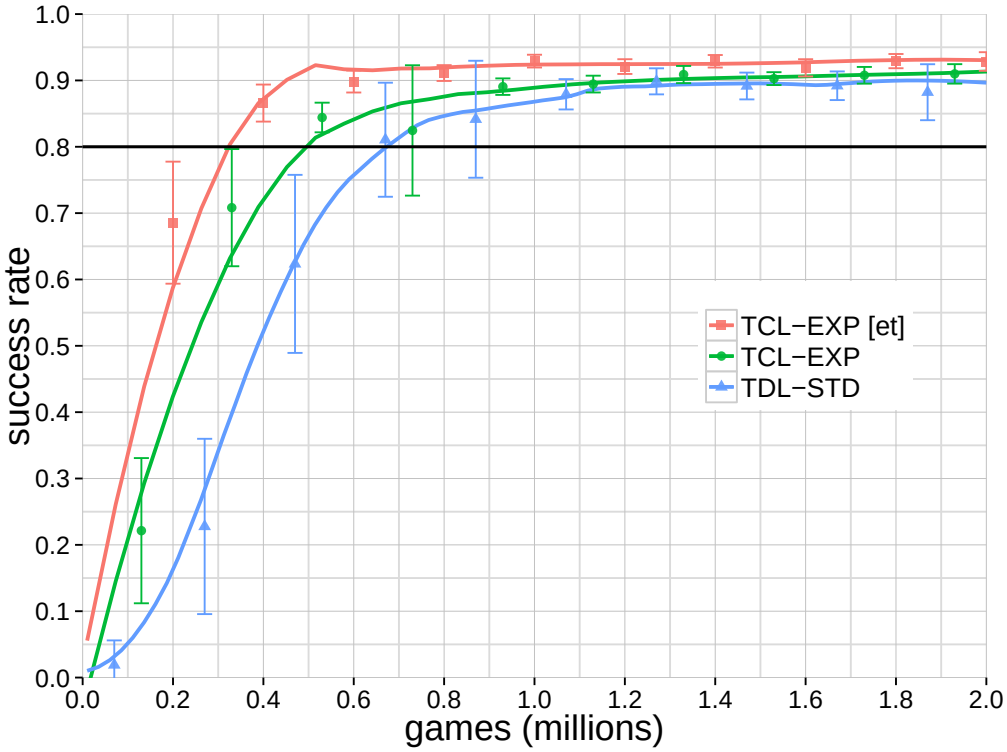

Figure 4.18: Initial results with classical eligibility traces using $\lambda = 0.5$ as described in Sec. 2.3.3. In this case, (TCL [et]) the traces are not reset on random moves. As seen in the graph, eligibility traces significantly speed the training up, after around 300 000 games the agent crosses the 80% success rate for the first time (considering the smoothed curve through the averaged points). Furthermore, the asymptotic success rate is slightly higher, than for TCL-EXP and TDL.

eligibility traces– assign credit to earlier states depending on how recently the states were visited, classical traces accumulate the credit for repeated visits of a state. In *Connect Four* , states cannot repeat during one episode. Nevertheless, since we use n-tuple systems as generalizing approximation functions, different states can still have the same features, so that individual weights are typically activated several times.

Fig. 4.20 shows the results when using replacing traces, which are reset on random moves (TCL-EXP [rr]). In comparison to ordinary traces (TCL-EXP [res]) – that are reset on random moves – no significant differences can be observed.

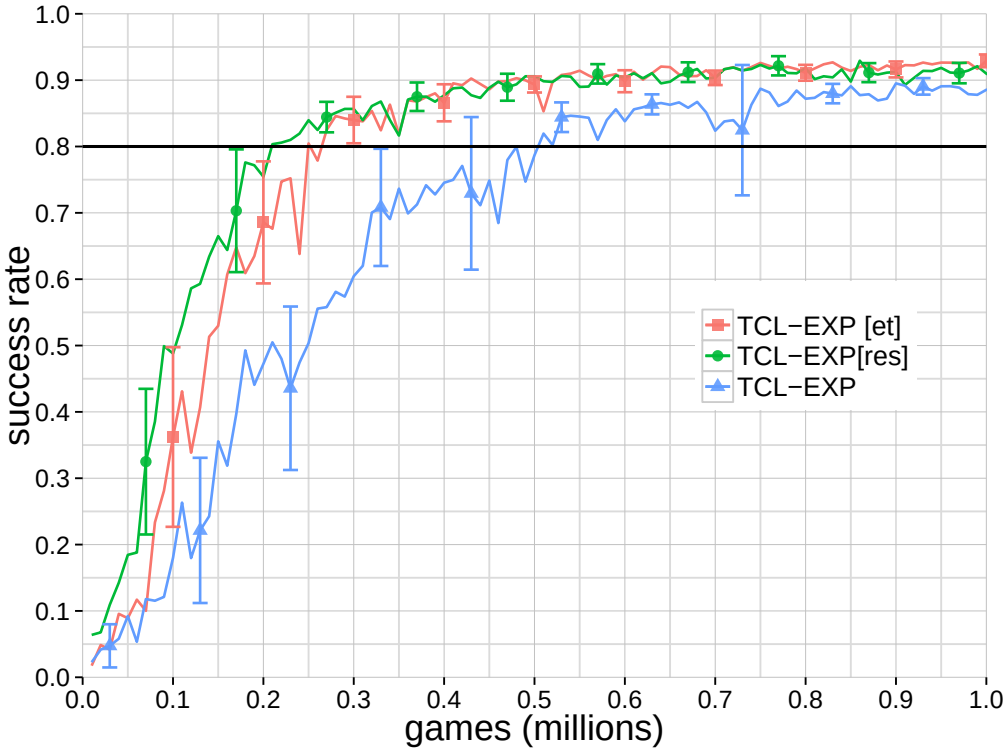

Figure 4.19: Results with eligibility traces, resetting on random moves, with $\lambda = 0.6$. Idea behind this: When performing random moves, the value $V(s_{t+1})$ of the resulting state s_{t+1} is most likely not an appropriate approximation of $V(s_t)$. Therefore, it can be sensible to reset the trace vector on random moves. As seen in the graph, this resetting option (TCL-EXP [res]) lets the agent learn slightly faster in the beginning, although the differences are not that significant. However, the asymptotic success rate for TCL-EXP [et] is higher than for TCL-EXP [res].

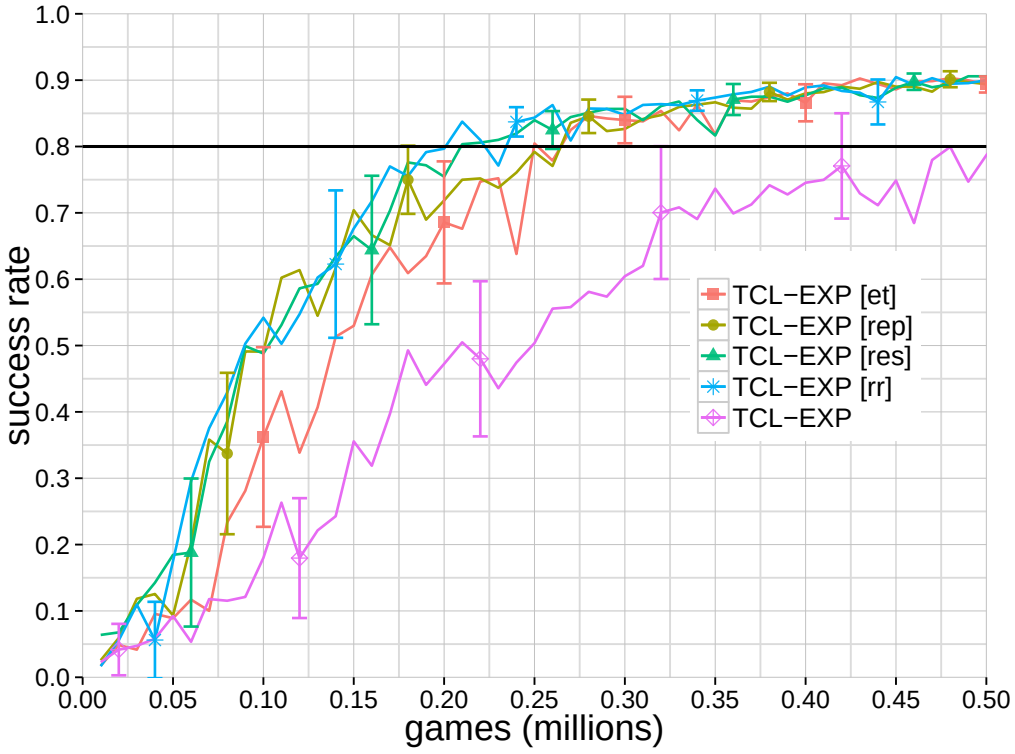

Figure 4.20: Replacing eligibility traces, resetting on random moves with $\lambda = 0.8$. In comparison to ordinary traces, that are reset on random moves (TCL-EXP [res]), no significant differences can be observed. In the beginning, ordinary traces resetting on random moves (TCL-EXP [res]), replacing traces (TCL-EXP [rep]) and replacing traces resetting on random moves (TCL-EXP [rr]) have a similar behavior. TCL-EXP [rep] slows down after a while and crosses the 80% success rate at the same point as TCL-EXP [et].

4.6.3 Detailed Comparison

For a better comparison of the results in the previous subsections, we generated a few box plots and run-time distributions, that summarize certain aspects of all results in a single plot.
in addition to the previous results, we introduce another agent TCL-EXP-M, which differs from TCL-EXP only in the number of n-tuples, which is now 150 n-tuples instead of 70. Furthermore, TCL-EXP-M has a slightly lower global learning rate ($\alpha = 0.02$ instead of $\alpha = 0.05$). The suffixes in the labels of the algorithms indicate the options used for the eligibility traces:
[*et*] – Classical implementation of the eligibility traces.
[*res*] – Classical implementation of the eligibility traces, resetting the traces on random moves.
[*rep*] – Replacing eligibility traces.
[*rr*] – Replacing eligibility traces, resetting the traces on random moves.

We created three box plots, the first one showing the asymptotic success rate of the considered algorithms but the second and third ones are indicating the time to learn, which we defined as the "amount of games to cross the 80% success rate for the *last* time" and "amount of games to cross the 90% success rate for the *first* time".

Fig. 4.21 displays the asymptotic success rate for the different algorithms we evaluated. TCL-EXP [et] and TCL-EXP-M [et] have the highest asymptotic success rates. In general, the asymptotic success rates were slightly (TCL-EXP) or notably (TDL) higher when using standard eligibility traces as compared to no traces. This enhancement was lost when adding any of the reset or replace options.

In Fig. 4.22 we can see that TCL-EXP[rr] (reset & replace) is slightly but significantly faster in reaching the target "80% success rate" than the other eligibility trace versions. However, after reaching the 80% success rate the training progress of TCL-EXP slows down so that all eligibility trace versions reach the target "90% success rate" approx. after the same amount of games.

Run-time distributions (or time to target plots) indicate the probability (on the ordinate) that an algorithm will reach a target in a given time (on the abscissa). For our purposes, we defined the targets "Crossing of the 80% success rate for the *last* time" and "Crossing of the 90% success rate for the *first* time". The run-time distributions shown in Fig. 4.23 basically show the same results as Fig. 4.22, but give a slightly different view on the results.

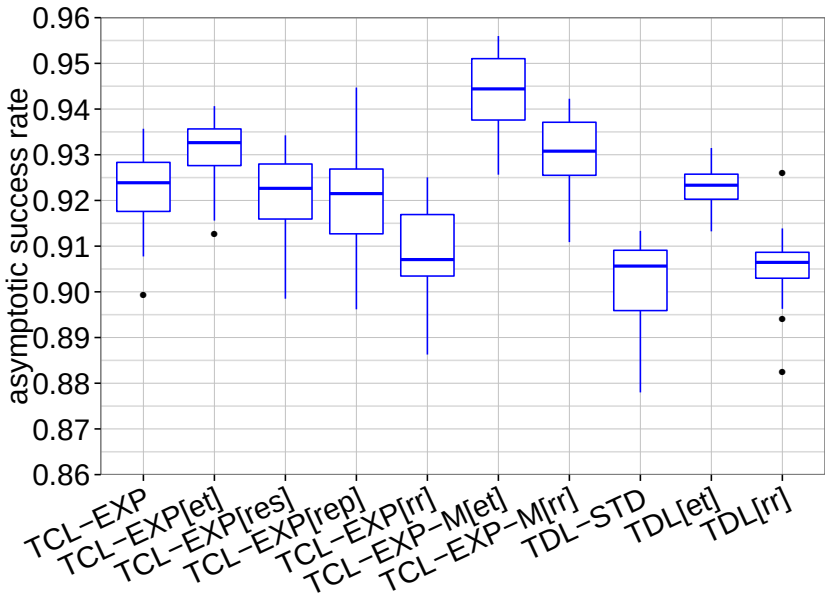

Figure 4.21: Asymptotic success rates. For each algorithm we performed 20 runs with 2 million training games in each run. The asymptotic success is the average success, measured during the last 500 000 games at 50 equidistant time points. TDL-STD contains one outlier at 0.79, which is not shown in this graph.

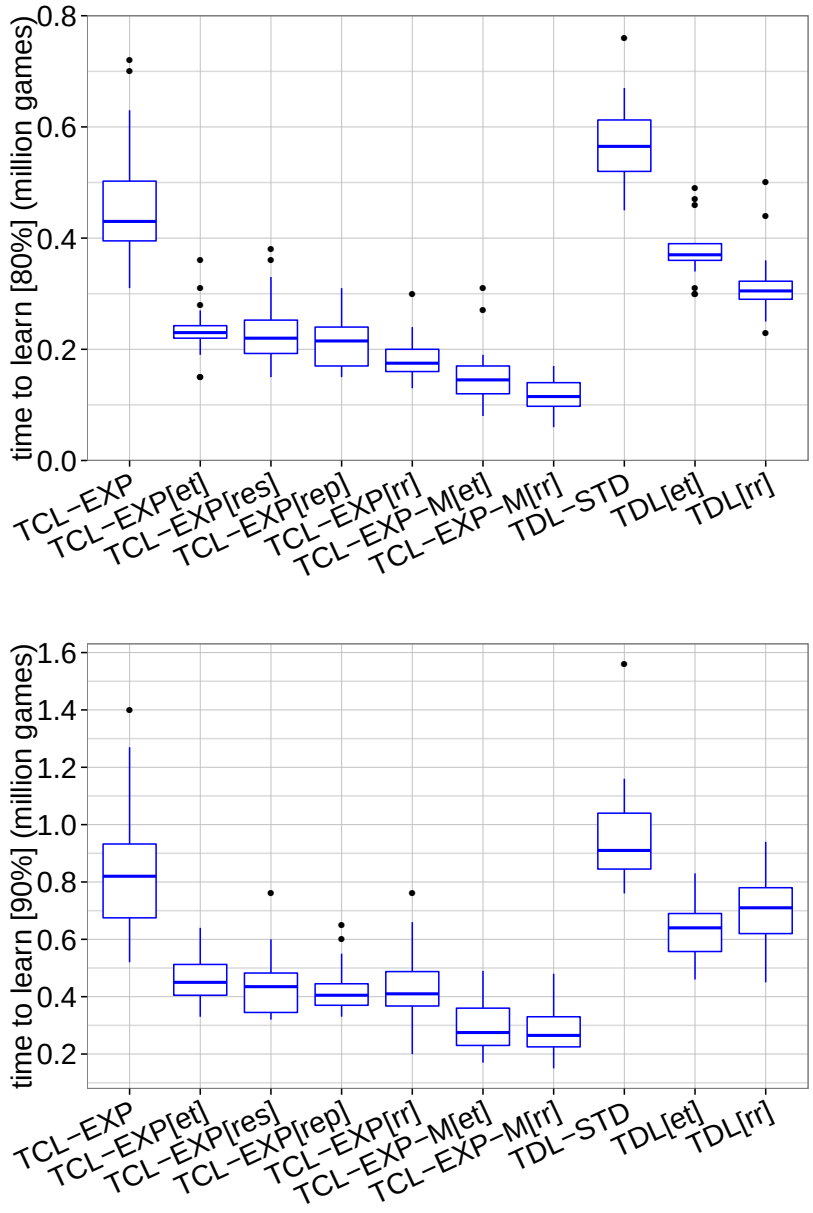

Figure 4.22: Number of training games for different algorithms to reach *80%* (*90%*) success rate. The box plots show the results of 20 runs. For each run we measure the success rate every 10 000 games and smooth this curve to dampen fluctuations. *Time to learn* is the number of games needed until this smoothed success rate crosses the 80% (90%)-line for the *last* (*first*) time. TDL-STD has one outlier at 2 million games, which is not shown in this graph. TCL-EXP-M is a bigger n-tuple system consisting of 150 (instead of 70) 8-tuples.

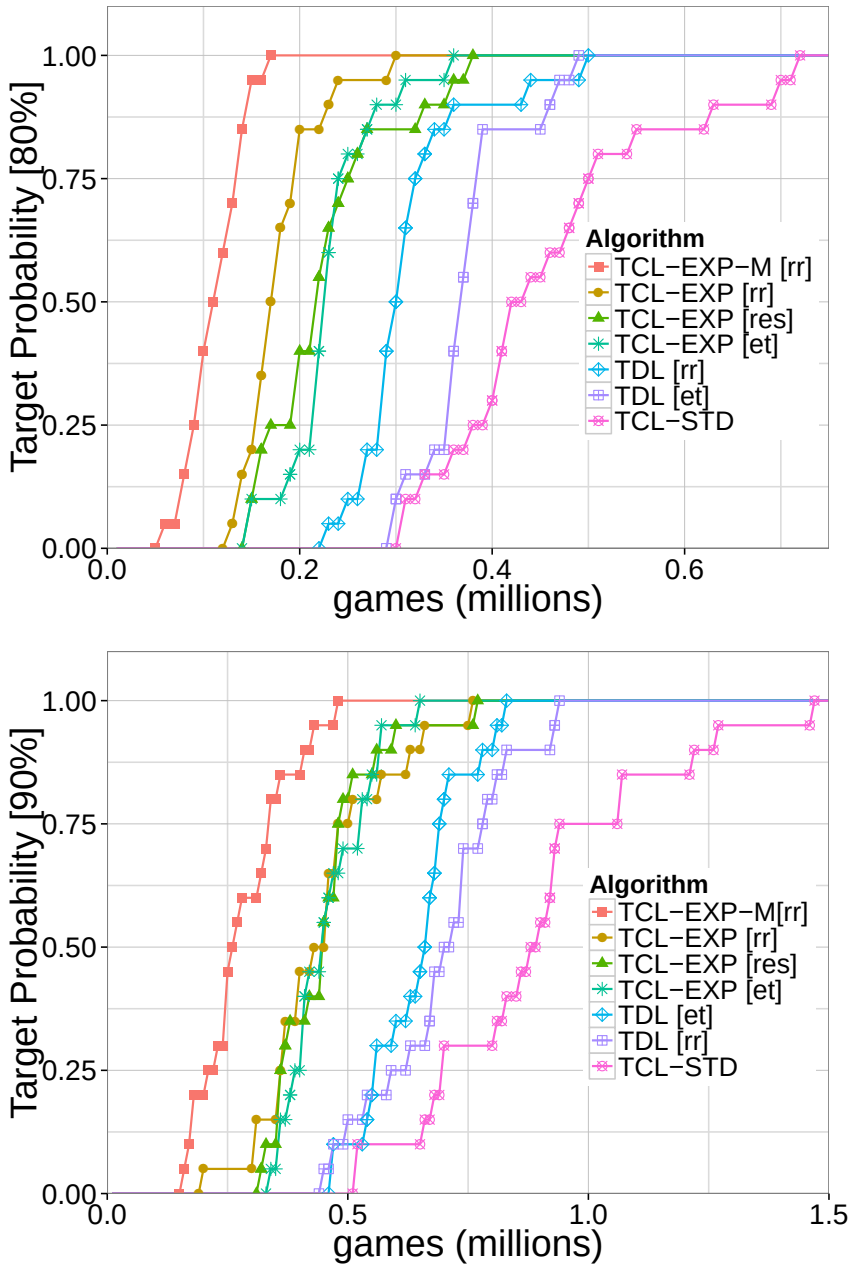

Figure 4.23: Run-time distribution for the learning speed (number of games needed to reach *80%* (*90%*) success rate) for different algorithms. The plot shows the results for 20 runs each. For each run we measured the success rate every 10 000 games and smoothed this curve to dampen fluctuations. Target Probability is the probability that the agents success rate crosses the 80% (90%)-line for the *last* (*first*) time (based on the 20 runs).

4.7 Summary

Fig. 4.24 summarizes the main findings in this chapter. We started our work with the best results found in [37] (Former TDL). A significant improvement of the learning speed of the TDL-agent could be achieved by tuning the *exploration rate* ϵ, from high initial value $\epsilon_{init} = 0.6$ to a relatively low value $\epsilon_{final} = 0.1$. The expected improvement of the TDL algorithm using the classical implementation of temporal coherence in TD-Learning stayed out, the results for TCL [r] were even worse. IDBD – another approach introducing individual learning rates for each weight – could improve the learning speed of the agent slightly. However, IDBD is defined at this stage only for linear value functions. Based on the idea of 'geometric step sizes' in IDBD, we developed TCL-EXP, which for the first time significantly increases the speed of learning compared to TDL. Finally, the implementation of eligibility traces for the n-tuple system could reduce the number of training games needed to learn the game by a factor of around two. An n-tuple system with twice the number of weights (150 × 8-tuple) can also increase the speed of learning (TCL-EXP-M [rr]). In total, we could reduce the number of training games to cross the 80% success rate from formerly around 1 800 000 to finally less than 200 000. This is a decrease by a factor of 9.

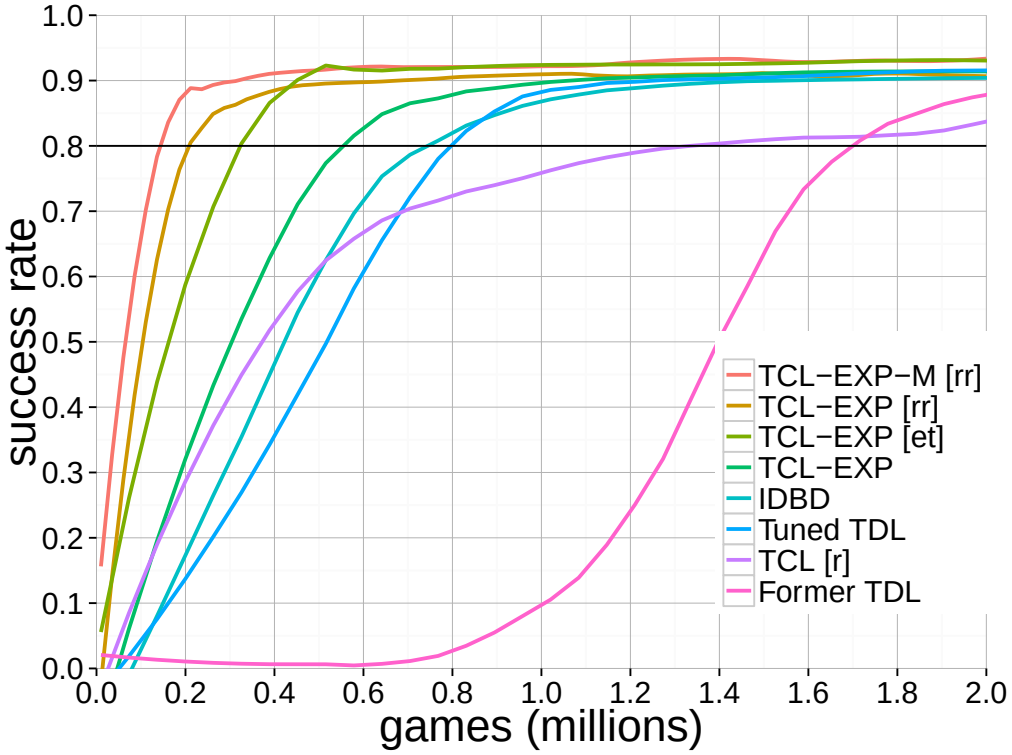

Figure 4.24: Summary of the main findings made in this chapter. Starting point was former TDL (Sec. 4.1). Tuning the *exploration rate* could significantly improve the training speed (Tuned TDL). The main task of this case study is to apply TCL to *Connect Four* . However, we found, that the original TCL algorithm cannot deliver the expected results (TCL [r]). IDBD (defined at this stage only for linear value functions) could improve the learning speed of the agent slightly. Based on the idea of 'geometric step sizes' in IDBD, we developed TCL-EXP, which for the first time significantly increases the speed of learning compared to TDL. Finally, eligibility traces could again clearly reduce the number of training games needed to learn *Connect Four* by a factor of almost 2. TCL-EXP [et], which does not reset eligibility traces on random moves, can even reach a slightly higher asymptotic success rate. An n-tuple system with 150×8-tuple (TCL-EXP-M [rr]) is the fastest learning system we found until now (but with the number of weights doubled).
Note, that the curves shown in this plot only show the fitted lines through the averaged results of 20 experiments each.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In our Case Study we investigated a complex learning task for the game Connect four. We used a temporal difference self-play algorithm in combination with an n-tuple system as position value function. We could show that a systematic parameter tuning of the TDL meta-parameters dramatically improves the performance of the learning process. It was possible to speed up the training of the TDL agent by a factor of almost 3; from formerly 1 565 000 training games to now 560 000 training games in order to reach a 80% success-rate).
The focus of this report was on the analysis of the online-adaptable learning rate algorithms TCL and IDBD, which were mainly targeted by the research questions in Chapter 3. Based on the idea of "'geometric step sizes"' in IDBD we adjusted the classical TCL algorithm by using an exponential scheme, which we named TCL-EXP. This algorithm clearly outperforms all other algorithms with respect to the learning speed of the agent. Finally, we investigated the benefits of eligibility traces, when added to our system.
We conclude this report by answering the research questions stated before:

1. *Is it possible to successfully apply TCL and IDBD to a system with millions of weights?* This question can be answered positively. We could show that both TCL and IDBD work in combination with our n-tuple system.

2. *How robust are online learning rate adaptation algorithms with respect to their meta-parameters?* This question has to be answered with a longer explanation. In contrast to the results presented in [7, 6] we could not confirm that TCL is insensitive to its meta-parameter α_{init}. Beal & Smith stated, that α_{init} *"has to be chosen, but this does not demand a choice between fast learning and eventual stability, since it can be set high initially, and the then provide automatic adjustment during the learning process"*. However, we found a clear

breakdown of our system with values $\alpha_{init} > 0.1$. Comparison of the sensitivity curves showed furthermore, that TCL has no advantage to standard TDL. The range of good training results were shifted for TCL and TDL but had about the same length.
Also IDBD does not free the user from parameter-tuning. IDBD is sensitive to its meta-parameter θ and to the initial step size $\alpha_i = e^{\beta_{init}}$.

3. *Are TCL (IDBD) able to significantly increase the speed of learning?* The original version of TCL could not improve the training process. In fact, the learning speed for TCL was even worse in comparison to the manually tuned TDL agent. Also update episodes, as suggested by Beal and Smith, do not help TCL; the performance decreases with longer episodes.
IDBD (even though we only implemented the algorithm for a linear value function) can speed up the learning process slightly, but not significantly. An agent using the IDBD algorithm needs approx. 615 000 games instead of 670 000 (Tuned TDL).
Finally, however, TCL-EXP, the modified variant of TCL, offers a significant improvement: TCL-EXP does learn substantially faster than the best tuned TDL agent (500 000 instead of 670 000 games to reach 80% success rate).

4. *Can TCL (IDBD) increase the strength (asymptotic success rate) of our agents?* This research question has to be answered negatively as well. None of the algorithms (TCL, IDBD and TCL-EXP) could reach a higher asymptotic success rate than tuned TDL. This is surprising, since TCL and IDBD promise to – next to improving the learning speed – reduce the MSE of a value function.

5. *Is it possible to successfully apply eligibility traces to a large scale problem with millions of weights?* We could show that TDL with eligibility traces works well for a large-scale problem with roughly 700 000 active weights and traces.

6. *We expect eligibility traces to speed up the learning process and increase the asymptotic success rate. Is it possible to significantly improve both aspects?* Our main result is that eligibility traces make the learning much faster: The number of training games required to learn a certain target is smaller by a factor of 2 compared with the variant without eligibility traces. Compared to our first published result on Connect-4 [37], the time-to-learn has reduced by an even larger factor 13: from the former 1 565 000 games in [37] to 115 000 games for the fastest algorithm TCL-M[rr] in this work. This reduction is however due to a combination of three factors: temporal coherence learning with exponential scheme (TCL-EXP), a larger n-tuple system with 150 instead of 70 8-tuples

and finally eligibility traces. Eligibility traces increase the asymptotic success rate slightly (but significantly) to around 93% (from formerly around 90%).

5.2 Future Work

Although we could gain many insights , there is still considerable scope for further investigations, especially in the field of online adaptable learning rates.
Sutton's IDBD was used in this Case Study for linear value functions only. In future work, the non-linear extensions of IDBD [14, 15, 33] could be applied to the learning-process. Furthermore, there are many approaches for online adaptable learning rates that –because of time limitations – we could not analyze in this work. Especially Sutton's K1 algorithm [31], Schraudolphs non-linear extension ELK1 [24], Mahmood's Autostep [20, 19] (a parameter-free extension of IDBD), Dabney and Barto's Alpha-Bounds algorithm [11] and Riedmiller's RPROP [21] deserve further attention. However, many of these mentioned algorithms are only described for linear value functions, so that the extension to non-linear functions could be an interesting research task. Most of the above algorithms also do not define, how eligibility traces in TDL should be treated. Since eligibility traces are essential for a fast learning – as we showed in this report – we believe that the combination of eligibility traces with learning rate algorithms is an important step for further improvement of the learning process.
The analysis of different activation functions and loss functions – as done in [15] – may reveal some more interesting insights. Since many above algorithms and TDL itself are dependent on certain meta-parameters a systematic tuning – for instance using Sequential Parameter Optimization [5]– of these could slightly improve the results.

Investigations in future could also concentrate on other topics, such as the n-tuple generation. Until now simply randomly created n-tuples are used for our system. A more systematic approach to create relevant n-tuples could improve the training of the agents. It would have to be analyzed which measures are able to separate good n-tuples from bad ones. For instance, irrelevant features should lead to weights with absolute values close to zero (assuming that the random noise accumulates to zero in long term). This may be a starting point to determine the relevance of an n-tuple.

A more technical improvement could be the implementation of the n-tuple system using sparse representations. Until now all weights (and corresponding learning rates) of an n-tuple system are maintained in arrays. During the training most weights will never be addressed due to the non-realizable states in an n-tuple. A more memory

efficient approach could use tree-structures instead, containing only those weights, which are necessary. However, this might increase the computation time of the learning process.

There is still some room to analyze Q-Learning results more deeply. We believe that Q-Learning agent can appear stronger than TD-Learning agent if we start from a random board position. Currently, all evaluations are done with an empty starting board. There was not enough time to test and prove such a hypothesis. This can be done as a future work.

In future we are also planning to learn completely different board games. Especially the simple game dots-and-boxes could be an interesting test-bed for many of the techniques described in this report and other techniques we did not investigate yet. Dots-and-boxes is a strategic board-game for two players with a very simple rule-set. The board-size can be varied, which allows to scale the complexity of the game in a simple way. Similar to *Connect Four* every game ends after a predefined number of moves (no position can be repeated). In contrast to *Connect Four* , the decision complexity is much larger: E.g., from the initial board all free lines can be chosen (Connect Four: Maximum of 7 possible moves). Also the state-space complexity dramatically increases with increasing board sizes. Similar to *Connect Four* in Dots-and-Boxes zugzwang plays an important role during the game: A certain position can have totally different values, depending on the player to move.
A Java-framework including several agents and a GUI for dots-and-boxes in Java is already available, which could save a lot of development time.

Bibliography

[1] Allis, V.: A knowledge-based approach to connect-four. the game is solved: White wins. In: Masters thesis, Vrije Universiteit. p. 4 (1988)

[2] Almeida, L., Langlois, T., Amaral, J.D.: On-line step size adaptation. Tech. Rep. RT07/97, INESC, 1000, Lisboa, Portugal (1997)

[3] Bache, K., Lichman, M.: UCI machine learning repository. Tromp's 8-ply database (2013), `http://archive.ics.uci.edu/ml`

[4] Bagheri, S., Thill, M., Koch, P., Konen, W.: Online adaptable learning rates for the game Connect-4. Transactions on Computational Intelligence and AI in Games (under review), 1 – 8 (2014)

[5] Bartz-Beielstein, T., Lasarczyk, C., Preuss, M.: Sequential parameter optimization. In: Congress on Evolutionary Computation. pp. 773–780 (2005)

[6] Beal, D.F., Smith, M.C.: Temporal coherence and prediction decay in TD learning. In: Dean, T. (ed.) IJCAI. pp. 564–569. Morgan Kaufmann (1999)

[7] Beal, D.F., Smith, M.C.: Temporal difference learning for heuristic search and game playing. Information Sciences 122(1), 3–21 (2000)

[8] Bledsoe, W.W., Browning, I.: Pattern recognition and reading by machine. In: Eastern Joint Computer Conference. pp. 225–232. New York (1959)

[9] Bremer, L.: Denkspiele und mehr - Mustrum (2010), `http://mustrum.de/mustrum.html`

[10] Curran, D., O'Riordan, C.: Evolving Connect-4 playing neural networks using cultural learning. NUIG-IT-081204, National University of Ireland, Galway (2004)

[11] Dabney, W., Barto, A.G.: Adaptive step-size for online temporal difference learning. In: 26th AAAI Conference on Artificial Intelligence (2012)

[12] Edelkamp, S., Kissmann, P.: Symbolic classification of general two-player games. In: Dengel, A., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T. (eds.) KI. Lecture Notes in Computer Science, vol. 5243, pp. 185–192. Springer (2008)

[13] Jacobs, R.A.: Increased rates of convergence through learning rate adaptation. Neural networks 1(4), 295–307 (1988)

[14] Konen, W., Koch, P.: Adaptation in nonlinear learning models for nonstationary tasks. In: Filipic, B. (ed.) PPSN'2014: 13th International Conference on Parallel Problem Solving From Nature, Ljubljana. Springer, Heidelberg (2014)

[15] Koop, A.: Investigating Experience: Temporal Coherence and Empirical Knowledge Representation. Canadian theses, Univ. of Alberta (Canada) (2008)

[16] Krawiec, K., Szubert, M.G.: Learning n-tuple networks for Othello by coevolutionary gradient search. In: GECCO'2011: Genetic and Evolutionary Computation Conference. pp. 355–362. ACM, New York (2011)

[17] Lucas, S.M.: Learning to play Othello with n-tuple systems. Australian Journal of Intelligent Information Processing 4, 1–20 (2008)

[18] Lucas, S., Runarsson, T.P.: Othello competition. `http://algoval.essex.ac.uk:8080/othello/League.jsp` (2011), cited 20.4.2011

[19] Mahmood, A., Sutton, R., Degris, T., Pilarski, P.: Tuning-free step-size adaptation. In: Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on. pp. 2121–2124 (2012)

[20] Mahmood, A.: Automatic step-size adaptation in incremental supervised learning. Ph.D. thesis, University of Alberta (2010)

[21] Riedmiller, M., Braun, H.: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: IEEE Int. Conf. on Neural Networks. pp. 586–591 (1993)

[22] Schneider, M., Garcia Rosa, J.: Neural Connect-4 - a connectionist approach. In: Brazilian Symposium on Neural Networks. pp. 236–241 (2002)

[23] Schraudolph, N.N.: Online local gain adaptation for multi-layer perceptrons. Tech. Rep. IDSIA-09-98, IDSIA, Lugano, Switzerland (1998)

[24] Schraudolph, N.N.: Online learning with adaptive local step sizes. In: Neural Nets WIRN Vietri-99, pp. 151–156. Springer (1999)

[25] Singh, S.P., Sutton, R.S.: Reinforcement learning with replacing eligibility traces. Machine Learning 22(1-3), 123–158 (1996)

[26] Sommerlund, P.: Artificial neural nets applied to strategic games. Unpublished, last access: 05.06.12. (1996), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4690

[27] Stenmark, M.: Synthesizing board evaluation functions for Connect-4 using machine learning techniques. Master's thesis, Østfold University College, Norway (2005)

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

[29] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)

[30] Sutton, R.S.: Adapting bias by gradient descent: An incremental version of delta-bar-delta. In: Swartout, W.R. (ed.) Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992. pp. 171–176. AAAI Press / The MIT Press (1992)

[31] Sutton, R.S.: Gain adaptation beats least squares. In: Proc. Yale Workshop on Adaptive and Learning Systems. pp. 161–166 (1992)

[32] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press (March 1998)

[33] Sutton, R.S., Koop, A., Silver, D.: On the role of tracking in stationary environments. In: 24th Int. Conf. on Machine Learning. pp. 871–878. ACM (2007)

[34] Tesauro, G.: Practical issues in temporal difference learning. Machine Learning 8, 257–277 (1992)

[35] Thill, M.: Using n-tuple systems with TD learning for strategic board games (in German). CIOP Report 01/12, Cologne University of Applied Science (2012)

[36] Thill, M.: Reinforcement learning with n-tuple systems for connect four (in german). In: Bachelors thesis, Cologne University of Applied Sciences (2012)

[37] Thill, M., Koch, P., Konen, W.: Reinforcement learning with n-tuples on the game Connect-4. In: Coello Coello, C., Cutello, V., et al. (eds.) PPSN'2012: 12th International Conference on Parallel Problem Solving From Nature, Taormina. pp. 184–194. Springer, Heidelberg (2012)

Appendix A

Figures and corresponding Experiments

Table A.1: List of Settings for the figures. A detailed list of parameters for each setting (e.g., [*TD1*]) can be found in Appendix B

Figure	Curve Label	Setting
4.1	TDL (old)	[*TD1*]
	TDL (best so far)	[*TD2*]
4.2	TDL	[*TD1*]
	TCL (std.)	[*TC1*]
	TCL (opt.)	[*TC2*]
4.3	TDL	[*TD1*]
	TCL(1)	[*TC3*]
	TCL(2)	[*TC4*]
4.4	TDL (opt. setting)	[*TD2*]
	TCL (opt. setting)	[*TC5*]
	TCL (std. setting)	[*TC1*]
4.5	TCL	[*TC6*], [*TC7*]
	TDL	[*TD3*]
4.6	Former TCL	[*TC5*]
	Tuned TCL	[*TC8*]
4.7	Former TDL	[*TD2*]
	Tuned TDL	[*TD5*]
	TCL	[*TD8*]
4.8	TCL [delta]	[*TC8*]
	TCL [r]	[*TC9*]

(Continued on next page)

Table A.1: (continued)

Figure	Curve Label	Setting
	tuned TDL	[*TD5*]
4.9	TCL (old)	[*TC6*]
	TDL	[*TD3*]
	TCL [r]	[*TC10*]
4.10	TCL[r] (EL=1)	[*TC11*]
	TCL[r] (EL=3)	[*TC11*]
	TCL[r] (EL=10)	[*TC11*]
	TCL[r] (EL=100)	[*TC11*]
4.11	TCL[r] (standard update-order)	[*TC9*]
	TCL[r] (swapped update-order)	[*TC12*]
4.12	TCL [delta]	[*TC9*]
	IDBD (linear net)	[*IDBD1*]
	IDBD (non-linear net)	[*IDBD2*]
4.13	no label	[*IDBD3*]
4.15	Piecewise Linear	[*TC13*]
	TCL [delta]	[*TC8*]
	IDBD	[*IDBD1*]
	TDL	[*TD5*]
	TCL-EXP	[*TC14*]
4.16	TDL ($eps_{init} = 0.17$)	[*TD5*]
	TDL ($eps = 0.1 = const.$)	[*TD6*]
	TCL-EXP ($eps_{init} = 0.17$)	[*TC14*]
	TCL-EXP ($eps = 0.1 = const.$)	[*TC15*]
4.18	TCL-EXP [et]	[*TC16*]
	TCL-EXP	[*TC15*]
	TDL-STD	[*TD6*]
4.19	TCL-EXP [et]	[*TC16*]
	TCL-EXP [res]	[*TC17*]
	TCL-EXP	[*TC15*]
4.20	TCL-EXP [et]	[*TC16*]
	TCL-EXP [rep]	[*TC18*]
	TCL-EXP [res]	[*TC17*]
	TCL-EXP [rr]	[*TC19*]

(Continued on next page)

Table A.1: (continued)

Figure	Curve Label	Setting
	TCL-EXP	[*TC15*]
4.22	TCL-EXP	[*TC15*]
	TCL-EXP [et]	[*TC16*]
	TCL-EXP [res]	[*TC17*]
	TCL-EXP [rep]	[*TC18*]
	TCL-EXP [rr]	[*TC19*]
	TCL-EXP-M [rr]	[*TC21*]
	TDL-STD	[*TD6*]
	TDL [et]	[*TD7*]
	TDL [rr]	[*TD8*]
4.23	TCL-EXP	[*TC15*]
	TCL-EXP [et]	[*TC16*]
	TCL-EXP [res]	[*TC17*]
	TCL-EXP [rep]	[*TC18*]
	TCL-EXP [rr]	[*TC19*]
	TCL-EXP-M [rr]	[*TC21*]
	TDL-STD	[*TD6*]
	TDL [et]	[*TD7*]
	TDL [rr]	[*TD8*]

Appendix B

Experiments

B.1 TDL-Experiments

Table B.1: Parameter settings for TDL experiments.

Setting	α_{init}	α_{final}	ϵ_{init}	ϵ_{final}	$\epsilon_{ip}/10^6$	λ	λ_{mode}	Remark
TD1	0.01	0.001	0.95	0.1	2.0	0.0	–	–
TD2	0.004	0.002	0.6	0.1	1.0	0.0	–	–
TD3	–	–	0.6	0.1	1.0	0.0	–	–
TD4	–	–	0.6	0.1	1.0	0.0	–	$70 \times 7 - tuple$
TD5	0.004	0.002	0.2	0.1	0.3	0.0	–	–
TD6	0.004	0.002	0.1	0.1	–	0.0	–	–
TD7	0.004	0.002	0.1	0.1	–	0.5	et	TDL [et]
TD8	0.004	0.002	0.1	0.1	–	0.5	rr	TDL [rr]

B.2 TCL-Experiments

Table B.2: Parameter settings for TCL experiments.

Setting	Alg	α_{init}	α_{final}	ϵ_{init}	ϵ_{final}	ϵ_{ip}	λ	λ_m	β	Remark
TC1	$[\delta]$	0.01	0.01	0.95	0.1	2.0	0.0	–	–	–
TC2	$[\delta]$	0.004	0.002	0.6	0.1	1.0	0.0	–	–	–
TC3	$[\delta]$	0.02	0.02	0.95	0.1	2.0	0.0	–	–	–
TC4	$[\delta]$	0.03	0.02	0.95	0.1	2.0	0.0	–	–	–
TC5	$[\delta]$	0.02	0.02	0.6	0.1	1.0	0.0	–	–	different slope for ϵ
TC6	$[\delta]$	–	–	0.6	0.1	1.0	0.0	–	–	–
TC7	$[\delta]$	–	–	0.6	0.1	1.0	0.0	–	–	$70 \times 7 - tuple$
TC8	$[\delta]$	0.04	0.04	0.2	0.1	0.3	0.0	–	–	–
TC9	$[r]$	0.04	0.04	0.2	0.1	0.3	0.0	–	–	–
TC10	$[r]$	–	–	0.6	0.1	1.0	0.0	–	–	–
TC11	$[r]$	0.04	0.04	0.2	0.1	0.3	0.0	–	–	$EL \in \{1, 3, 10, 100\}$
TC12	$[r]$	0.04	0.04	0.2	0.1	0.3	0.0	–	–	inverted operational order
TC13	$[r]$	0.05	0.05	0.2	0.1	0.3	0.0	–	2.7	piecewise linear
TC14	$[r]$	0.05	0.05	0.2	0.1	0.3	0.0	–	2.7	TCL-EXP
TC15	$[r]$	0.05	0.05	0.1	0.1	–	0.0	–	2.7	TCL-EXP
TC16	$[r]$	0.05	0.05	0.1	0.1	–	0.5	et	2.7	TCL-EXP [et]
TC17	$[r]$	0.05	0.05	0.1	0.1	–	0.6	res	2.7	TCL-EXP [res]
TC18	$[r]$	0.05	0.05	0.1	0.1	–	0.6	rep	2.7	TCL-EXP [rep]
TC19	$[r]$	0.05	0.05	0.1	0.1	–	0.6	rr	2.7	TCL-EXP [rr]
TC20	$[r]$	0.05	0.05	0.1	0.1	–	0.6	et	2.7	TCL-EXP-M [et] $(150 \times 8 - tuple)$
TC21	$[r]$	0.05	0.05	0.1	0.1	–	0.6	rr	2.7	TCL-EXP-M [rr] $(150 \times 8 - tuple)$

B.3 IDBD-Experiments

Table B.3: Parameter settings for IDBD experiments.

Setting	β_{init}	θ	ϵ_{init}	ϵ_{final}	$\epsilon_{ip}/10^6$	Remark
IDBD1	-5.8	0.01	0.2	0.1	0.3	linear net
IDBD2	-5.8	0.01	0.2	0.1	0.3	non-linear net
IDBD3	–	0.01	0.2	0.1	0.3	linear net