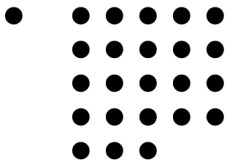

Fachhochschule Köln
Cologne University of Applied Sciences

Temporal Difference Learning Methods with Automatic Step-Size Adaption for Strategic Board Games: Connect-4 and Dots-and-Boxes

Markus Thill

Master Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Engineering

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

In the Course of Studies

Master of Automation & IT

First Supervisor: Prof. Dr. Wolfgang Konen
Cologne University of Applied Sciences

Second Supervisor: Prof. Dr. Heinrich Klocke
Cologne University of Applied Sciences

June 2015

Abstract

Machine learning tasks for board games which rely solely on self-play methods remain rather challenging up till today. The perhaps most impressive breakthrough in this field was achieved by Tesauro's TD-Gammon, which was able to learn the game backgammon at expert level with a self-play variant of the temporal difference learning (TDL) algorithm. Since then, many studies attempted to replicate some of TD-Gammon's success by applying TDL to other board games, however, mostly with mixed results.
We found in our earlier work on the board game Connect-4 that a rich feature set is required to successfully learn a near-perfect strategy. Nonetheless, several millions of self-play training games were necessary in order to generate strong Connect-4 agents.

In this thesis we will mainly focus on two topics, namely online-adaptable learning rate methods and eligibility traces, and investigate whether these approaches have the potential to speed up learning.
For the Connect-4 learning task we show that algorithms with geometric step-size changes have the best performance, in some cases reducing the required number of training games to learn the game by more than 40%.
In a case study, we compare several state-of-the-art step-size adaptation algorithms with respect to their sensitivity towards certain meta parameters.
In the further course of this thesis, we investigate the benefits of different eligibility trace variants. Additionally, we extend several learning rate algorithms to eligibility traces and examine their performance. We could observe that eligibility traces improve the speed of learning by a factor of two for our Connect-4 task.
Overall, with several additional enhancements, we could reduce the number of training games to learn Connect-4 to slightly more than 100 000, which is an improvement by a factor of 13, compared to previously published results.
In the last sections of this work, we apply the learning framework that we developed for Connect-4 – with several adjustments – to the strategic board game Dots-and-Boxes and discuss the main problems that we observed for our initial experiments.

Keywords: Machine Learning, Reinforcement Learning, Online adaptable learning Rates, Connect-4, Dots-and-Boxes, Temporal Difference Learning, Eligibility Traces, Self-Play

Contents

List of Tables

List of Figures

List of Symbols

AI	Artificial Intelligence
CSV	Comma Separated Values
CTDL	Coevolutionary Temporal Difference Learning
DP	Dynamic Programming
GGP	General Game Playing
GUI	Graphical User Interface
IDBD	Incremental Delta Bar Delta
LMS	Least Mean Square
LUT	Lookup Table
MCTS	Monte Carlo Tree Search
MSE	Mean Squared Error
NLMS	Normalized Least Mean Square
TCL	Temporal Coherence Learning
TD	Temporal Difference
TDL	Temporal Difference Learning
RL	Reinforcement Learning
RWC	Recommended Weight Change
SGD	Stochastic Gradient Descent
SMD	Stochastic Meta Descent

Chapter 1

Introduction

Since the early days of computers, the field of artificial intelligence (AI) has been of particular interest for the research community. Strategic board games, arguably one of the oldest areas in AI, especially attracted the attention of many researchers. Over the last decades numerous publications introduced various approaches to create computer programs that are able to play chess, checkers, Go and many other games.
There are several reasons why board games are so popular among researchers: first of all, strategic board games can be described by a limited, uncomplicated set of rules and the goals are normally clearly defined. Since the states of board games have a simple representation and the actions are well-defined, a game-playing environment can be implemented in software without great effort and in the environment different agents can easily interact with each other or with human players and matches between the players can be repeated as often as desired. Furthermore, in contrast to intelligent agents for other domains (which typically are equipped with various sensors and actuators), game-playing agents can be connected to the environment without requiring complex perceptual or actuating components. This allows the developer to focus on the development of intelligent learning and/or playing strategies. But although the rules of most board games are easy to understand and can be implemented without any great effort in a game-playing environment, the enormous complexity of many games makes a solution with elementary methods (such as a rudimentary tree search) infeasible. Instead, sophisticated algorithms have to be developed that require a high degree of creativity. Overall, board games depict an ideal testbed to study new methods and ideas which later might be transferred to other domains as well.

For many years computer chess was considered as a supreme discipline in AI and a vast amount of effort has been put into the development of algorithms that play chess at master level. Especially in computer chess, sophisticated Minimax search trees have proven to be successful. The main idea behind search trees is to try all possible moves for a board position in a recursive manner until a certain depth and then estimating the values at the leaf-nodes of the tree with some evaluation function. The evaluation function typically weights many different game-specific features. The strongest programs available often have an evaluation function that is fine-tuned by world-class players. But despite all the advanced techniques that have been developed to enhance tree-based algorithms, one has to note, that the recent success of the strongest chess programs is mainly the result of extensive

tree-search, combined with expert knowledge and massive computation power. Comparably few attempts have been made to really learn the game with machine learning techniques.

Learning evaluation (predicting) functions for complex board games remains a difficult task up till today. While humans have the perceptual and cognitive abilities to identify and evaluate certain patterns in board games and to learn important strategies of the game in a short time, machine learning techniques can often not sufficiently generalize from already experienced situations and transfer the acquired knowledge to new situations. Typically, a large problem for many learning algorithms is to find suitable features autonomously, so that commonly hand-crafted features are used that were designed with the help of expert-knowledge. In recent work [69, 67], we found that remarkable results can be achieved for the strategic board game Connect-4, if so called n-tuple systems [36] are used to generate a large feature space; no game-specific knowledge was required to create the features. In this work we will rely on n-tuple systems as well, in order to learn the games Connect-4 and Dots-and-Boxes, although the focus will be on other topics, discussed in the following.

Another particular challenge for many machine learning tasks is that it is often unclear at which rate new experience (training examples) should be acquired by the learning algorithm in order to update its current information: if the rate is chosen too high, then valuable information based on previous experiences may be overridden and if chosen too low, the learning speed might significantly decrease. Especially gradient-descent based methods – such as the temporal difference learning (TDL), used in this work – are heavily dependent on the learning rate (in this sense often also referred to as step-size parameter). However, there is no straight-forward approach to find suitable values for the learning rate, so that often extensive tuning runs are required (if feasible), in which different settings are tried. Values that have been found during a tuning process are typically constants and generally also only suboptimal: for most learning tasks, high initial step sizes ensure fast convergence towards the optimal solution; during the training the step size is then gradually decreased in order to avoid overshoots and to reduce the asymptotic error. For these reasons, many step-size adaptation algorithms have been proposed over the years, which promise to automatically adjust the step size during the training and free the user from tuning it.

In this work we study the performance of several of the state-of-the-art step-size adaptation algorithms, when applied to a complex learning task, in which the strategic board game Connect-4 is learnt by temporal difference Learning (TDL) with an n-tuple system as function approximation. The agents are trained solely by self-play, without involving an external teacher in the learning process.
Additionally, we will investigate the benefits of eligibility traces, which can be seen as an enhancement of TDL. Eligibility traces introduce a whole family of algorithms that bridge the gap between plain temporal difference and Monte Carlo methods. Typically, eligibility traces help TDL to utilize training samples more efficiently and increase the overall learning

speed of a system.
The learning framework for Connect-4 (TDL, n-tuple systems, step-size adaptation algorithms) is sufficiently general to be applied to other board games as well. In the final steps of our work, we will analyze if and how it is possible to transfer the framework to the non-trivial game Dots-and-Boxes.

Two recent publications [8, 68] emerged from the work on this thesis, which already partially cover the results presented in this study. This thesis extends the work of [8, 68] in several aspects: Two additional step-size adaptation algorithms, namely RProp and SMD (Section 3.5.3) are examined, the step-size adaptation algorithms nl-IDBD (Section 3.5.2) and SMD (Section 3.5.3) are extended for usage in conjunction with the TD(λ) algorithm and applied to our Connect-4 task and, different views on the results presented in [8, 68] are given. Furthermore, in addition to Connect-4, we also investigate the general applicability of the learning framework (TDL, n-tuple systems, step-size adaptation) to the strategic board game Dots-and-Boxes (Sections 3.1.2 & 5.2).

The rest of the thesis is structured as follows: In Section 1.1 we briefly review the related work and the recent advances in the field machine learning, in particular for board games, temporal difference methods and step-size adaptation algorithms. In Chapter 2 we pose several research questions that will steer the later experiments. Chapter 3 introduces the methods, theoretical aspects and background knowledge for this thesis. Chapter 4 briefly describes the software framework developed for Connect-4 and Dots-and-Boxes, the general setup of our experiments and the evaluation process for the agents. In Chapter 5 we present and discuss the results of the experiments. Finally, Chapter 6 concludes this thesis and gives an outlook on future work.

1.1 Related Work and State of the Art

Already in the year 1959, Samuel described an approach to learning the game of checkers solely by self-play [43]. Although the trained agents did not reach master level, they were able to compete with strong amateur players, such as the author himself. More important, Samuel laid the foundations for temporal difference (TD) methods in this early work, which were then formalized later in 1984 and 1988 by Sutton [56, 57].
Perhaps the most remarkable success of temporal difference learning (TDL) is Tesauro's TD-Gammon that plays backgammon at the same level as the best computer programs and human players [65, 66]. TD-Gammon learnt the game by training a neural network with 80 hidden units using TDL and required 1 500 000 self-play training games to achieve master level.
Despite the remarkable results of TD-Gammon, most attempts to apply TD methods to complex board games, such as Go [50], checkers [44] or chess [70, 12], had only limited success or could only succeed in combination with an extensive game tree-search.

More recently, Lucas showed that it is possible to create strong agents for the game Othello with so called n-tuple systems that can be trained by TDL [36]. The main advantage of n-tuple systems is that they induce a mighty feature space, without requiring any game-specific expert knowledge. Also other successful attempts to learn Othello, either by coevolutionary learning [30] or coevolutionary temporal difference learning (CTDL) [63, 35], use n-tuple systems to approximate a state-value function. In [64], a strategy for the popular puzzle game *2048* is learnt – by employing n-tuple systems and TDL – that allows around 97% of all games to be won.

Although Connect-4 was already weakly solved in 1984 independently by Allen & Allis [2, 3] and strongly solved 10 years later in 1994 by Tromp [7], the game remains difficult to learn with self-playing agents. Several attempts to learn Connect-4 (whether by self-play or by learning from teachers) are found in the literature: Schneider et al. [46] tried to learn Connect-4 with a neural network, using an archive of saved games as teaching information. Sommerlund [53] applied TDL to Connect-4 but obtained rather discouraging results. Stenmark [54] compared TDL to a knowledge-based approach from Automatic Programming and found TDL to be slightly better. Curran et al. [21] used a cultural learning approach for evolving populations of neural networks in self-play to play Connect-4. However, all the above attempts fail to provide a common reference point for the strength of their agents, which makes it difficult to compare their actual strength.
In our previous work [69, 67], we established a TDL agent with an n-tuple system for the game Connect-4 solely trained by self-play. The agent reached near perfect play – winning around 90% of the games when it played first against a perfect opponent. However, 1 500 000 training games were required to learn Connect-4. In this thesis, we will explore several enhancements that allow us to increase the training speed as well as the strength of our earlier TDL agent. These enhancements (mainly online adaptable learning rates & eligibility traces) are already partly discussed in recent publications [8, 68].
To the best of our knowledge there is no published work that attempts to learn the game Dots-and-Boxes by RL methods. There are several solvers available, based either on classical tree-search techniques [27, 11, 10] or on retrograde analysis [73]. The largest Dots-and-Boxes board to be solved up till today has had the size of 4×5 boxes [10, 11].

One particular problem that many TD learning tasks are confronted with is the temporally delayed credit assignment, that is, an agent may have to learn from rewards that are given after a long sequence of actions. In one-step TD methods for example, the rewards are only propagated one step back to earlier states.This can result in slow learning and high computing times. A possible solution to this problem are eligibility traces introduced in the TD(λ) algorithm [56, 57] and further developed by Singh and Sutton [52]. TD(λ) combined with self-play has been shown to improve the results over one-step methods for the card game hearts [55], Go [51], give-away checkers [40], and Tesauro's TD-Gammon [66]. To the best of our knowledge, no results have been reported for the game Connect-4, although Stenmark briefly describes eligibility traces in [54]. Larger systems with eligibility traces are

not found very often in the literature. A remarkable exception is the work of Geramifard et al. [26] extending a special TD algorithm (iLSTD) with eligibility traces for a system with n=10 000 traces. To the best of our knowledge, no results were reported so far for systems with more than a million traces, as they occur in our task here.

In recent years another RL method, namely Monte Carlo Tree Search (MCTS) has received a lot of attention by the research community. MCTS attempts to solve the RL problem by building a tree-structure, where each node represents an individual state. Similarly to other RL techniques, MCTS is sample based (like TDL & Monte Carlo methods) and learns by averaging sample returns (like Monte Carlo methods). Although early work in 1990 [1] and 1992 [18] showed the efficiency of Monte Carlo simulations, a major breakthrough was not achieved until 2006, when Coulom [19] introduced MCTS – which combined classical tree search techniques with Monte Carlo evaluations; MCTS techniques turned out to be especially successful for the board game Go [25], for which no strong players were available until then. A survey summarizes the recent advances in this field [17].

Step-size (learning rate) adaptation is an important aspect for many machine learning tasks. For this reason, a large number of learning rate adaptation schemes have been introduced in various publications over the years: Jacob's Delta-Bar-Delta (DBD) algorithm [29] was initially designed for batch learning tasks and later extended to incremental tasks by Sutton's Incremental Delta-Bar-Delta (IDBD) [58] algorithm. Additionally, Sutton eliminated two meta-parameters that were required in the original DBD algorithm. In a later work [61], Sutton introduced the K1 algorithm – which is similar to IDBD, but derived from normalized Least Mean Square (NLMS) instead of LMS – and the K2 algorithm. Recently, Mahmood and Sutton [39, 38] proposed with Autostep another extension to IDBD which has much less dependence on the meta step-size parameter than IDBD. IDBD, K1, K2 and Autostep are defined only for linear units. Sutton & Koop extended IDBD to a non-linear variant nl-IDBD and Schraudolph introduced ELK1 [47, 48], a non-linear extension of the K1 algorithm. Konen & Koch [33] introduced another interesting non-linear version of the IDBD algorithm, which successfully deals with undesired saturation effects when non-linear activation functions are employed and which showed better performance than other algorithms on a non-stationary benchmark task. Stochastic Meta Descent (SMD) [48, 49] can be seen as a more general extension of IDBD that does not approximate the instantaneous Hessian matrix and that can also be used in conjunction with non-linear function approximations.
Almeida et al. [5, 6] discussed another method of step-size adaptation and applied it to the minimization of nonlinear functions. Dabney & Barto proposed α-bounds, a step-size adaptation algorithm for online temporal difference learning (TDL) that finds the optimal upper and lower bounds of the learning rate and does not require additional meta-parameters. With RPROP, Riedmiller & Braun [42] introduced a very popular algorithm for batch learning tasks. Schaul et al. [45] proposed a tuning-free step-size adaptation algorithm that appears to perform especially well on non-stationary problems. Temporal coherence learn-

ing (TCL) by Beal & Smith [13, 14] is especially designed for TD Learning and can be used for linear or non-linear function approximations. In this work we will investigate the capabilities of a selected subset of the mentioned step-size adaptation schemes.

Chapter 2

Research Questions

The general goal of this work is to develop stronger machine learning agents for board games, which can learn just by letting them play against themselves. Building upon existing machine learning algorithms, the aim is to increase the learning speed of the agents, their game-playing strength or both. Furthermore, we want to take a step towards general game-playing architectures, which are not specialized to a certain game but which are sufficiently general and can be applied to new games without any or only slight modifications and without the necessity of extensive parameter tuning.

Steered by these general goals, this study is mainly concerned with three different areas that will be explored in the further course of the thesis: initially, we want to investigate several well-known online-adaptable learning rate algorithms and analyze whether they have the potential to speed up learning, reduce the overall error and increase the robustness of the learning process for our Connect-4 task. Subsequently, we study the benefits of eligibility traces added to the Connect-4 learning process. Different versions of eligibility traces (standard, resetting, and replacing traces) will be compared. Finally, we transfer our learning framework (TD-Learning with n-tuple systems and step-size adaptation algorithms) to a new domain, namely the game Dots-and-Boxes.

The learning rate (step-size parameter) is a central ingredient of many learning algorithms, such as the TDL algorithm used in this work. A wrong choice of the step size may result in slow learning or in a high asymptotic error. In many cases, an extensive tuning of the step size is not feasible, since a single training-run may take many hours and can therefore not often be arbitrarily repeated. Step-size adaptation algorithms attempt to learn suitable step sizes based on the training data, mostly in an incremental manner. Temporal difference learning of complex board games might be a challenging task for step-size adaptation algorithms, since typically the error function and learning signals are rather noisy, and TDL constitutes a non-stationary problem, due to its bootstrapping nature and due to adjustments in the policy. In this work, we will investigate how well the individual learning rate algorithms perform in such a setup, guided by the following research questions:

1. *Can online learning rate adaptation increase the performance of a complex learning task with millions of weights?*

Initially, it has to be investigated if the individual algorithms can be at all applied to large problems such as our Connect-4 task with around 9 million weights. The learning rate algorithms with the highest memory requirements have a complexity of $4n$, where n is the number of inputs. Hence, for each weight three additional memory traces are required. Based on our rough estimate, this should be realizable, although several hundreds of MB of memory will be necessary for the training process.
Another important aspect is the computational complexity. Even algorithms for which the computation time scales linearly with the number of weights might be infeasible, if millions of weights are involved, since several million traces might have to be touched in every time step. It has to be examined if the individual learning rates algorithms compensate the additional computation effort.
Then, in the next steps the performance of the individual algorithms should be observed. The performance can be assessed mainly by two measures: The number of training games required to learn (denoted as 'time-to-learn' in this work) and the final strength of the agent ('asymptotic success rate' against a perfect playing opponent).

2. *How sensitive are the step-size adaptation algorithms towards their own meta parameters?*

   Many of the proposed step-size adaptation algorithms analyzed in this work promise to reduce or even completely eliminate the dependency on the step-size parameter, by providing an adaptation scheme that adjusts the step size (or a vector of step sizes) automatically during the training process. However, most step-size adaptation algorithms require additional meta parameters as well, such as a meta learning rate or an initial step size. An interesting starting point could be to investigate the dependency of the individual algorithms on their meta-parameters and analyze if the algorithms can provide a self-tunable learning rate adaptation or if they in effect simply shift the learning problem to another layer.

The second aspect that we want to investigate is the extension of the classical TDL algorithm by eligibility traces. This is a well-known approach in TDL which allows more weights to participate in learning during a specific episode and could be especially beneficial for systems with sparse input vectors, such as n-tuple systems as used in this study. In contrast to one-step TD methods, where the prediction of a state is only updated based on the prediction (and reward) of its successor state, eligibility traces propagate rewards back to all earlier visited states, discounted by a trace decay parameter for each step back. In principal, we expect eligibility traces to significantly speed up the training of the agents by utilizing the training samples more efficiently. Accordingly, our experiments with TDL & eligibility traces will try to give answers to the following main research question:

3. *Can TDL, augmented with eligibility traces, utilize training samples more efficiently and improve the speed of learning for our complex Connect-4 task?*

   Again, the initial tests have to show if eligibility traces are applicable to problem such

as our Connect-4 task, with several million weights involved in the training process. Usually, each weight of the system requires a corresponding eligibility trace which would significantly increase the memory requirements and computation time, since all traces have to be adjusted in every learning step. Therefore, an efficient data structure is required to maintain the traces.
Several versions of eligibility traces can be implemented (standard, resetting, and replacing traces) which may perform differently. One task will be, to evaluate and compare these different variants in order to see whether or not certain variants outperform others with respect to the learning speed or asymptotic error.
If an efficient implementation for the eligibility traces can be found for our current setup [69], it is natural to ask whether an even larger feature space would increase either learning speed, final strength, or both.
Many of the step-size algorithms that will be examined in this work are not defined for eligibility traces. It could be interesting to investigate, how well selected online adaptable learning rate algorithms perform, when their definitions are extended to eligibility traces. In this work, we will extend some of the algorithms to eligibility traces for the first time and analyze the results for them.

The basics concepts (temporal difference learning, eligibility traces, n-tuple systems, online adaptable learning rate methods) used for our Connect-4 task are sufficiently general and can be easily transferred to other board games as well. Although there are a few examples, where n-tuple systems were successfully applied to board games (Othello [36, 30, 35] or 2048 [64]), it is an open question how well especially n-tuple systems perform for most other complex board games. Consequently, in the last sections of this work we want to focus on the strategy game Dots-and-Boxes, for which we plan to perform a few initial experiments. Since the board size of Dots-and-Boxes is scalable, we can gradually increase the complexity, beginning with small boards, which is a great advantage in the early test phase. In summary, we can state one last research question for this work:

4. *Is it possible to successfully transfer the developed learning framework (TDL with eligibility traces, n-tuple systems, online adaptable learning rates) without significant adjustments to the strategic board game Dots-and-Boxes?*

   We consider the attempt to transfer the frame to Dots-and-Boxes as successful if it is possible to train near-perfect agents under reasonable conditions (considering the number of training games, the size of the n-tuple system and other aspects), hence, agents that are capable of finding optimal move sequences in more than 80% of all matches against a perfect-playing opponent.
   Small boards (e.g., 1×2 boxes) might be suitable for early tests; later, we plan to move to larger boards, although we do not expect to produce strong agents for boards larger than 3×3 boxes in the early stage. During the experiments potential problems have to be identified and possible solutions for future work should be suggested. If

Dots-and-Boxes cannot be learnt with the above concepts, it might at least be possible to find the reasons for failure.

Chapter 3

Methods

3.1 Board Games

Board games depict an ideal testbed for machine learning tasks. On the one hand, they are described by a fixed set of rules, which are typically easy to implement. On the other hand, even games with a simple set of rules can have an enormous complexity, which often does not allow a solution in reasonable time. These properties make board games an interesting for scientists, who can use the games in order to test and assess the quality of various algorithms. In this work, we mainly want to conduct our experiments with the game Connect-4 a two-player game with a medium complexity which is ideal to test different algorithmic variants. Later, we move on to the game Dots-and-Boxes, which has a scalable complexity. Dots-and-Boxes will be used for some initial tests in order investigate if and how concepts described in the preceding chapter (TDL, n-tuple systems and adaptable learning rates) can be transferred to another board game.

3.1.1 Connect-4

Connect-4 is a popular board game for two players (typically *Yellow* and *Red*) played on a grid with seven columns and six rows. Initially, the board is completely empty so that 42 free positions are available. One main characteristic of the game is the vertical arrangement of the board, which restricts both players to only dropping their pieces into one of the seven columns (slots). Starting with Yellow, both players in turn drop one of their pieces into a slot of their choice. Under the force of gravity, the pieces then fall down to the lowest free position of the respective slot. If all six positions of a slot are occupied, it is not possible to place further pieces into this slot and the number of possible moves for both players is reduced by one. Both opponents attempt to create a continuous line of four adjacent pieces with their color, either horizontally, vertically or, diagonally. The player who can achieve this first, wins the game. If none of the opponents is able to create a line of four with his own pieces during the 42 moves match, all columns will be completely filled and – according to the rules – the match ends with a tie. An example position for Connect-4 is shown in Fig. 3.1.

Although Connect-4 has a medium state space complexity of around $4.5 \cdot 10^{12}$ different positions [24] and a game-tree complexity of approximately 10^{21} [4], solving the game

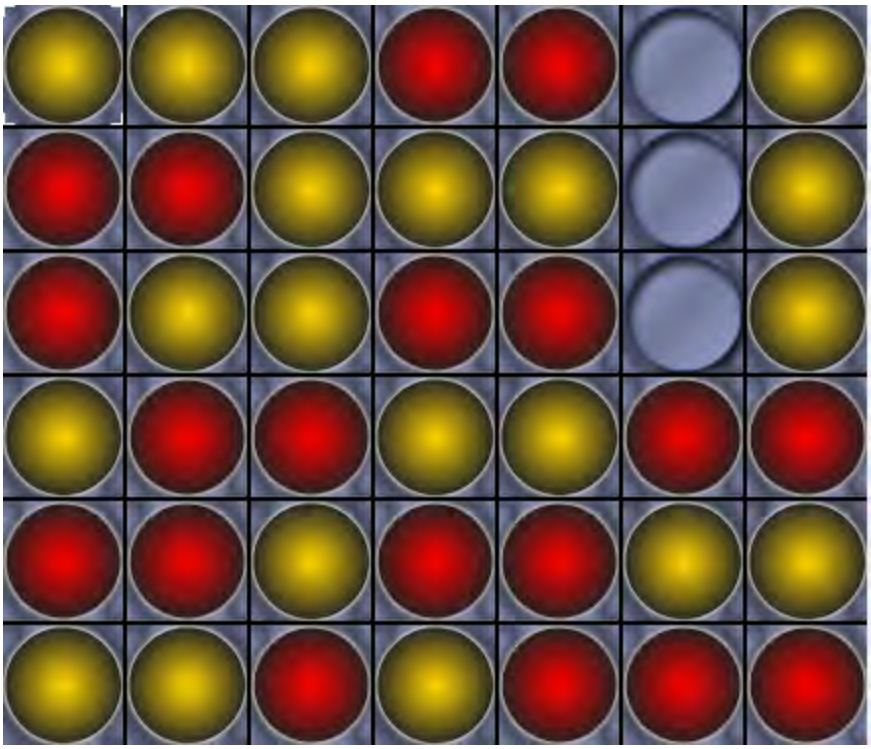

Figure 3.1: Typical Connect-4 position, created during a match of a temporal difference learning agent (Yellow) against a perfect playing Minimax agent (Red), with Minimax as the next player to move. Both types of agents will be described in later sections of this thesis. For the given position, Minimax is under zugzwang and will eventually loose the game, however, the defeat could be delayed as far as possible.

remains a non-trivial task until today. The first solution was independently found by Allen [2] and Allis [3] in 1988. In their work the game was weakly solved; both could show – assuming perfect play of both opponents – that Yellow wins the game if she places her first piece in the center slot. Red on the other hand, can delay her defeat as far as possible to the end and force Yellow to use all her pieces.
In 1994, Tromp [7] completed a database containing the game-theoretic values (win/loss/draw) of all non-trivial[1] positions with exactly 8 pieces, which strongly solved Connect-4. The evaluation of all 67, 557 positions took him around 40, 000 hours (≈ 4.5 years).

[1]Tromp considered all positions with an immediate threat as trivial. This may be true for positions with an immediate threat for Yellow, since she can win with her next move. However, immediate threats for Red can be neutralized by Yellow and the match would continue normally. This definition of triviality is therefore not completely correct.

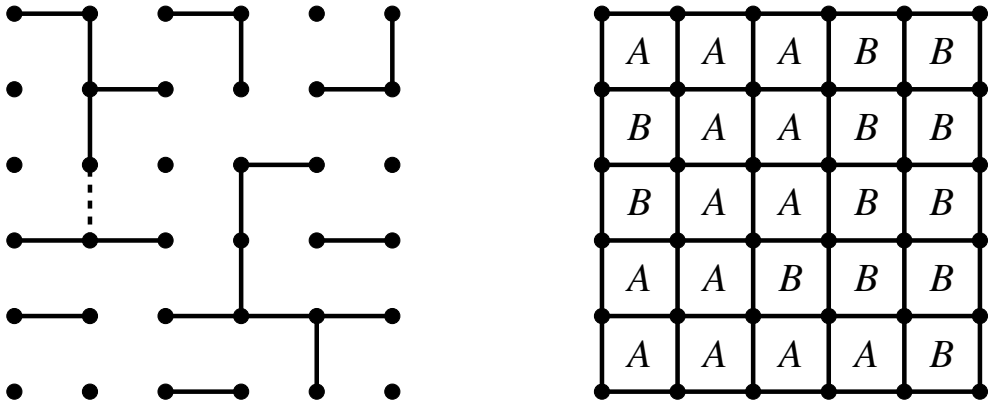

Figure 3.2: Left: Example position on a 5×5 grid for Dots-and-Boxes, taken from Berlekamp's Dots-and-Boxes book ([15], problem 12.15), with player A to move. There is only one optimal move that allows A to win, indicated here by the dashed line. All other moves lead to a win for player B.
Right: One possible outcome for the given position on the left, assuming perfect play of both opponents. Player A wins by a narrow margin of one box. The solution to this problem was found with the help of Wilson's solver [73].

3.1.2 Dots-and-Boxes

Dots-and-Boxes is a two-player (typically the players are denoted as "A" and "B") strategy game with simple rules which can be played with just paper and pencil. The game starts with an empty grid of points. Both players in turn connect two adjacent points either by vertical or horizontal lines. It is not allowed to fill in a line more than once. If a player manages to complete the fourth line of a box she takes the ownership of this box and must perform an additional move, hence, draw another line. A player may but is not forced to capture a box if other moves are still available. The game ends after all lines on the grid are drawn. The player who owns the most boxes in the end wins the game. If both players captured the same amount of boxes, the match is considered as a tie. A tie is only possible for those boards which consist of an even number of boxes. An example for a typical Dots-and-Boxes position on a 5×5 grid is given in Fig. 3.2.

One important characteristic of this game is the possibility to vary the size of the grid as desired. Theoretically, Dots-and-Boxes can be played on all grids with $M \times N$ boxes (with $M, N \geq 1$), which corresponds to a grid with $(M + 1) \times (N + 1)$ points. M and N may be varied in order to steer the complexity of the game, although not all combinations of M and N may be reasonable, since many are trivially solvable or induce an overly complex problem. Since the state-space complexity and the game tree size rapidly grow with increasing grid size, the largest board to be solved until today has the size of 4×5 boxes, which took about one month on a 3.33GHz Xeon machine [10]. In order to solve the 4×5 board, Barker & Korf performed a classical *alpha-beta search* with several game-specific enhancements for

pruning the search-tree. Additionally, the search was supported by a 24GB transposition table.
In general, the complexity for different board sizes can be estimated as follows: for a $M \times N$ board, there are

$$\begin{aligned} p &= 2MN + M + N \\ &= M(N+1) + N(M+1) \end{aligned} \tag{3.1}$$

possible edges that can be filled in. The game-tree size, hence, the total number of possible games that can be played from the empty board, is then $p!$. For example, a 4×4 board with $p = 40$ has a game tree size of $40! \approx 8 \cdot 10^{47}$, which makes it impossible for a naive Minimax-search to solve.
Since many permutations of a move sequence lead to the same game state, the state-space complexity is several orders of magnitude smaller[2]. Moreover, the estimation of the state-space complexity can be simplified if only the edge configuration of a board is considered; for the decision-making process the information about the number, the owners and the position of captured boxes is not relevant, since both players attempt to maximize the number of boxes with their remaining moves [10]. Additionally, Dots-and-Boxes is an impartial game: the player to move is irrelevant for decision-making, the optimal move in a certain situation is the same for both opponents [10]. With these observations, only two possible states (empty and filled) per edge have to be considered, so that the state-space complexity is simply given by 2^p. Again, for a 4×4 board, the state-space complexity would then be $2^{40} \approx 1.1 \cdot 10^{12}$. In comparison with Connect-4 (see previous section), the game-tree complexity for the 4×4 board is larger by approximately 27 orders of magnitude due to the large branching factor of Dots-and-Boxes. The state-space complexity for both games is comparable.
In [15], Berlekamp approaches Dots-and-Boxes in far more depth with combinatorial game theory and discusses many theoretical and strategical aspects, such as the relation between Dots-and-Boxes and the game Strings-and-Coins, Nimber values, double-crosses, chain rules, half- and hard-hearted handouts, loony moves and more.

3.2 Reinforcement Learning

Reinforcement learning (RL) describes an area in *machine learning*, in which agents learn how to take a sequence of actions in an unknown, dynamic environment in order to maximize a cumulative long-term reinforcement signal, the so called reward. Unlike many other machine learning techniques, such as supervised learning, a RL-agent learns solely through experience (trial and error principle) and has no access to further knowledge provided by an

[2]This is beneficial for tree-search algorithms with transposition tables, since recurring nodes do not have to be expanded again, one can simply retrieve the game-theoretic value from the transposition-table

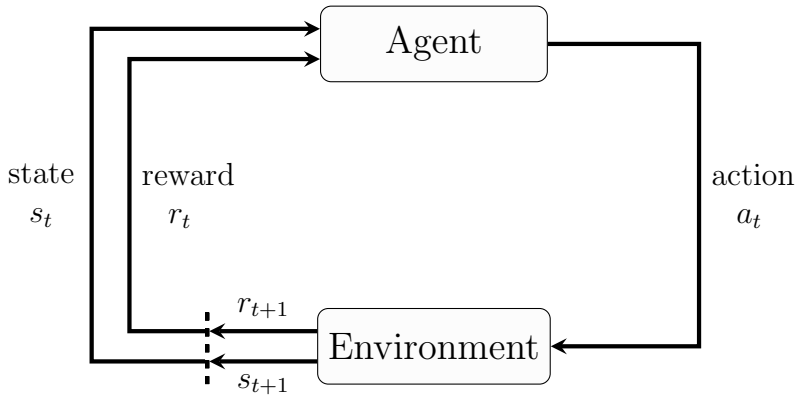

Figure 3.3: General Reinforcement Learning model illustrating the interaction of an agent with his environment. Based on the current state s_t the agent determines and performs an action a_t that transforms the state s_t of the environment into a new state s_{t+1}. Additionally, the environment returns a reward to the agent. Figure taken from [59, Sec. 3.1].

external supervisor [31, p. 239]. A general model of the interaction between an agent and its environment is shown in Figure 3.3: At each discrete time step t (with $t = 0, 1, 2, \ldots$), the agent observes its environment and receives some information on the environment's state, coded in s_t, where $s_t \in S$ is an element in a set of all possible staes S. Then, the agent selects and performs an action a_t from a set of possible actions $A(s_t)$ that are available for the current state s_t. The selected action causes a transition of the environment from the current state s_t into a new state s_{t+1}. Additionally, the environment generates an immediate reward $r_{t+1} = R(s_{t+1})$, $R(s) \in \mathbb{R}$ that is passed to the agent. However, the reward r_{t+1} is only an indicator for the immediately following state s_{t+1} and does not reflect how desirable s_{t+1} is in the long run; it might happen that a certain action results in a high immediate reward, but the long term reward for another action may have been higher. In fact, in many applications (such as most board games) it is not even possible to provide intermediate rewards, since it cannot be foreseen to which outcome (win/loss/draw) an action leads; typically, rewards are delayed towards the end of the episode and in extreme cases the reward is only given when a terminal state is reached. This is also known as the temporal credit assignment problem [31, p.251], which might be the major challenge for RL-agents. The particular problem for the agent is to find a suitable mapping from a state to an action in order to maximize the reinforcement in the long run. This mapping is typically referred to as the policy of the agent [59, Sec. 3.1] and is generally defined as a probability distribution $\pi : S \times A \to [0, 1]$ which specifies the probability that an action a_t is selected if the agent is in state s_t [59, Sec. 3.1]. Ideally, a RL-agent aims at learning an optimal policy π^* [59, Sec. 3.8] that ensures the highest long term reward for all states S. Although this does not necessarily imply that the agent should predict the expected future reward accurately for any situation, the agent typically still should be able to separate good from

bad actions with some suitable estimate. A common approach to solve the RL problem is to learn a value function that predicts the expected future reward for a state or a state-action pair [59, Sec. 3.7]. A state value function is formally defined as [59, Sec. 3.7]:

$$V^{\pi}(s) = E_{\pi}\left\{R_t \,|\, s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{T-t} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\}, \tag{3.2}$$

where π is the policy followed by the agent, $E_{\pi}\{\cdot\}$ is the expected return R_t for state s_t if the agent follows its policy π. The discount-factor γ is commonly used to slightly decrease future rewards and is chosen in the 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. Per definition, the value of a state is always zero for $t \geq T$ and the reward is zero for $t > T$. Equation (3.2) can be written in a recursive form, which is denoted as the Bellman equation for V^{π} [59, Sec. 6.1]:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left\{\sum_{k=0}^{T-t} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+2} \,\middle|\, s_t = s\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \,\middle|\, s_t = s\right\}. \end{aligned} \tag{3.3}$$

If the agent follows an optimal policy π^*, then the corresponding value function is called optimal and can be written as:

$$V^*(s) = \max_{\pi} E_{\pi}\left\{\sum_{k=0}^{T-t} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\}. \tag{3.4}$$

In order to find an optimal policy one can therefore seek for an optimal value function $V^*(s)$. For a given state s, the optimal value function can also be expressed in a recursive manner, since the value of a state is the expected reward for an optimal action performed from this state. This relation is commonly referred to as the Bellman optimality equation, which is defined as [59, Sec. 3.8]:

$$V^*(s) = \max_{a} E_{\pi^*}\left\{r_{t+1} + \gamma V^*(s_{t+1}) \,\middle|\, s_t = s, a_t = a\right\}. \tag{3.5}$$

Since the optimal value function is typically unknown, a RL agent will have to predict V^* in above equation and gradually improve its current policy towards π^*. One particular challenge that arises for RL agents from this is known as the trade-off between *exploitation*

and *exploration* [59, Sec 1.1]. On the one hand the agent should use its past experience to select an action that promises the highest reward (exploitation), based on the current prediction. But on the other hand, the agent cannot find alternatives that lead to higher rewards without exploring unknown regions of the state space. A suitable balance of both exploration and exploitation is required for the RL agent to succeed in his learning task. A common approach is using an ϵ-greedy policy: With a probability of $(1-\epsilon)$, $\epsilon \in [0,1]$, an action is selected for which the agent expects the highest reward. With a probability of ϵ the agent is forced to perform an action at random. Finding a suitable choice for ϵ is not trivial for many problems, since its value could be a constant, follow a function or could be tuned on-line by an appropriate algorithm.

3.2.1 Temporal Difference Methods

Typically, one wants to find an optimal policy π^*, which maximizes the cumulative long-term reward of an agent. Several methods from *Dynamic Programming* (DP), such as policy iteration and value iteration, are guaranteed to converge to obtain an optimal value function, from which an optimal policy then can be derived [31, pp.248]. The general idea behind DP methods is to utilize the recursive relationship described by the bellman equation and to base the prediction of a state on the prediction of its successor state. However, the main drawback of DP methods is that they have to sweep through the whole state space of the system, which becomes infeasible for environments with more than several million states [59, Sec. 4.7]. Additionally, DP methods require detailed knowledge of the environment (e.g., of the state transition probability function for stochastic environments), which cannot be provided in every case. To overcome these obstacles, typically *generalizing value functions* (such as n-tuple systems, as described in Section 3.3) are learnt in a *sample based* manner. Ideally, with a suitable learning algorithm, only a subset of all states (the samples) have to be visited by the agent and the value function learns how to transfer the acquired knowledge to unknown situations. *Temporal difference learning* (TDL) is one of the most popular methods for solving the Reinforcement Learning problem and can be used for learning such a value function. The central idea behind TDL is to approximate the Bellman equation in a way that a prediction $V_t(s_t)$ is updated based on the prediction $V_t(s_{t+1})$ of its successor state. This bootstrapping process is somewhat similar to the one performed in DP. But in contrast to DP methods, TDL is a sample based technique and does not have to iterate through the whole state space. The main ingredient for TD methods is the so called TD error signal δ_t, which indicates the error between the value of the current state s_t and the value of the successor-state s_{t+1} (the difference of both sides of the bellman equations in (3.3) and (3.5)) [59, Sec. 6.6]:

$$\begin{aligned} \delta_t &= r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \\ &= T_t - V_t(s_t), \end{aligned} \tag{3.6}$$

where $T_t = r_{t+1} + \gamma V_t(s_{t+1})$ is known as the target signal. Ideally, the TD error should converge towards zero during the training process. The TD algorithm attempts to reduce this error by adjusting $V_t(s_t)$ to match $r_{t+1} + V_t(s_{t+1})$ more closely. But since V_t is typically a generalizing, parametrized value function that only approximates the real function, it is not possible to reduce the error for all states to zero. Therefore, a suitable minimization approach has to be found that can trade off (balance) the accuracy of the individual states. One possibility is to weight the individual errors and minimize the mean squared error (MSE) loss function [59, Sec. 7.8]:

$$L_{mse}(\vec{w}_k) = \sum_{s \in S} \Big\{ P(s_t) \big[r_{t+1} + \gamma V_k(s_{t+1}) - V_k(s_t) \big]^2 \,\Big|\, s_t = s \Big\}, \tag{3.7}$$

where $P(s)$ is a probability distribution which is used to weight the errors of all states. As already mentioned, this weighting has to be done, because there are normally states in a system which are more relevant than others and require a better approximation, so that their error should be taken into account more.
Finding an analytic solution for $\vec{w}$ that minimizes above loss function is normally impossible and also a gradient descent approach will mostly fail, since the training set S is too large for most problems. Furthermore, the distribution $P(s)$ is typically unknown, since it depends on several factors such as transition probabilities, the policy of the agent and others.
TD methods can elegantly solve these problems: The MSE loss function in (3.7) can be seen as a sum of individual objectives (loss functions), where each objective represents the squared error of a certain state. Instead of evaluating all objectives, TD methods just sample a subset and attempt to minimize the objective for these samples. This approach is known as *Stochastic Gradient Descent* (SGD). The nice side-effect of this procedure is that also no knowledge of the weighting distribution $P(s)$ is required any longer, since a TDL agent can mimic the effect of $P(s)$ naturally in the long-run, simply by following its current policy. As a result, the prediction accuracy of commonly visited states will increase at the expense of less frequently visited states.
The gradient of the sample loss function, which is simply the squared TD error signal, is computed as follows:

$$\begin{aligned} g_t &= \frac{1}{2} \nabla_{\vec{w}_t} \big[\delta_t \big]^2 \\ &= \delta_t \, \nabla_{\vec{w}_t} \Big[T_t - V_t(s_t) \Big]. \end{aligned} \tag{3.8}$$

For the simplest TD method, known as TD(0) [3], the target signal in the gradient of Equation (3.8) is omitted [59, sec. 8.2] and the following weight update rule can be derived:

$$\vec{w}_{t+1} = \vec{w}_t + \alpha\delta_t \nabla_{\vec{w}_t} V_t(s_t), \tag{3.9}$$

where α is a positive step-size parameter. Note, that since the gradient of the target signal T_t is omitted, which contains the weight-dependent function V_t, $TD(0)$ is not a real gradient descent method any longer. In certain situations, this may lead to convergence problems; only for tabular value function (mapping each state to exactly one weight), $TD(0)$ is guaranteed to converge [9, 72]. Nevertheless, TDL has proven to be a robust learning technique in practice and was successfully applied to many problems.

In the following section we describe how to utilize samples more efficiently when classical TD methods are extended by eligibility traces.

3.2.2 The TD(λ) Algorithm and Eligibility Traces

The Temporal Credit Assignment Problem

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 many board games are typically given at the end of a long sequence of actions. This may lead to a rather slow learning progress when one-step temporal difference (TD) methods are used, due to the bootstrapping involved in the update process: the prediction of a state is updated based only on the estimation (and reward, if available) of its successor state, without waiting for the final outcome of the current episode, as expressed by the TD error signal $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$. This approach has the advantage that the value function can be updated in every time step, without the need to know the final outcome of the episode. However, it also has the major drawback that training samples are not utilized efficiently, since only the prediction of a single state is adjusted in each time step.
As a simple example, assume that an initially inexperienced agent constantly follows a policy π in an environment that provides a reward only at the end of each episode. The policy π may result in an episode with T time steps and a final reward r_T. After completing the first episode, the agent receives a reward from the environment and updates the value for the estimate $V(s_{T-1})$. During the second episode, the predictions $V(s_{T-1})$ and $V(s_{T-2})$ will be adjusted and so forth, assuming that each state s_t is only visited once during an episode. In this example, $T-1$ repeated episodes would be required, until the delayed

[3] $TD(0)$ is one member in a whole family of TD methods that span from one-step TD methods TD(0) on the one end to Monte Carlo methods TD(1) on the other. In the following sections we will describe these methods.

reward finally affects $V(s_0)$. In the case of large T, this can be problematic and may result in a slow convergence of the learning process. But if the agent constantly follows his policy π anyhow, then the predicted values of all states can directly be set to the final reward, i. e. $V(s_0) = V(s_1) = \cdots = V(s_T) = r_T$.
In order to use training samples more efficiently, it could be convenient to assign some credit to the preceding states $\{s_{t-1}, s_{t-2}, \ldots\}$ as well, when updating the value function for the current state s_t, since these preceding states led to the agent's current situation.
Monte-Carlo methods follow this idea, albeit in an extreme manner, in that they use the overall return of an episode as target for the backup of each visited state. For this purpose, a whole episode has to be completed first, before $V(s_t)$ can be updated [57]. In the same way as before in Equation 3.9, one can update the value function $V(s_t)$ by minimizing the squared error on the observed examples with a stochastic gradient descent approach:

$$\vec{w}_{t+1} = \vec{w}_t + \alpha[R_t - V(s_t)]\nabla_{\vec{w}_t} V_t(s_t), \tag{3.10}$$

where the target is now – instead of $r_{t+1} + V(s_{t+1})$ – the exact long-term return R_t for state s_t [59, Sec. 7.1], as already described in Section 3.2:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T. \tag{3.11}$$

Unifying Temporal Difference and Monte-Carlo Methods

Both, one-step TD methods and Monte-Carlo methods may have their advantages on different problems and both converge reliably under certain assumptions, although TD methods generally appear to perform slightly better in practice [57] [59, Sec. 6.2 & 6.3], [62, p. 22]. However, there is a way to synthesize one-step TD and Monte-Carlo methods in order to combine the advantages of both approaches. Sutton showed that both methods can be located at two ends of a whole family of methods, described by the so called TD(λ) algorithm [57, 59]. With a new trace decay parameter $\lambda \in [0, 1]$ it is possible to seamlessly shift between one-step TD-methods ($\lambda = 0$) on the one end and Monte-Carlo methods ($\lambda = 1$) on the other end. The main ingredient to achieve this is the *eligibility trace vector*, which contains a decaying trace for each weight w_i of the system. In [59, Sec. 7.2 & 7.3], Sutton & Barto describe two possible views on the TD(λ) algorithm, the *forward* and the *backward* view. The forward view is more theoretical and not directly implementable for reasons that will be resolved shortly. The backward view is more mechanistic and allows the TD(λ) algorithm to be implemented in an incremental manner by utilizing the already mentioned eligibility traces [59, Sec 7.3].
Although the backward view is sufficient for the general understanding of the effects and the advantages of eligibility traces, the derivations beginning with the forward view will be helpful in later sections of this thesis. As a first step towards eligibility traces, one could consider n-step returns instead of a one-step return as the target value of the prediction $V(s_t)$. In this case, the rewards of the next n time steps and the estimation for the remaining

rewards are taken into account, resulting in the n-step return [59, Sec. 7.1]:

$$\begin{aligned} T^{(n)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}) \\ &= \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n V(s_{t+n}) \\ &= R_t^{(n)} + P_t^{(n)}, \end{aligned} \tag{3.12}$$

with:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} \tag{3.13}$$

$$P_t^{(n)} = \gamma^n V(s_{t+n}). \tag{3.14}$$

Any value $n \geq 1$ can be chosen, whereby $n = 1$ corresponds to the one-step return and a value $n > T - t$ results in the exact long-term return R_t, as used by Monte-Carlo backup in Equation (3.10). One could now go even one step further: Instead of a single n-step return, a weighted average of a set of different n-step returns could be used as target signal, with the condition that all weights sum up to one [59, Sec. 7.2]. One particular approach to average the n-step returns is the λ-return [59, Sec. 7.2], which is the main component in the *forward view* of TD(λ) and depicts the target of $V(s_t)$:

$$\begin{aligned} T_t^{\lambda} &= (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} T^{(n)} \\ &= (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \big(R_t^{(n)} + P_t^{(n)} \big) \end{aligned} \tag{3.15}$$

This sum contains *all* n-step returns which are weighted by the factor λ^{n-1}, where $\lambda \in [0, 1]$. With the properties of the geometric series, it can be trivially shown that all weight factors sum up to $(1-\lambda)^{-1}$. For this reason a leading normalization factor $(1-\lambda)$ is required. Note that the sum depicts an infinite series, which can however for episodic tasks easily be transformed into a finite series and an additional term , since $T^{(n)} = R_t$ for $n \geq T - t$.
The squared error loss function is given by:

$$L_{mse}^{\lambda}(\vec{w}_t) = \frac{1}{2} \big[T_t^{\lambda} - V_t(s_t) \big]^2. \tag{3.16}$$

In much the same way as before (Equation (3.9) and (3.10)), the backup step, using the λ-return as target value can then be written as:

$$\vec{w}_{t+1} = \vec{w}_t - \alpha \vec{g}_t = \vec{w}_t + \alpha \big[T_t^{\lambda} - V_t(s_t) \big] \nabla_{\vec{w}_t} V_t(s_t), \tag{3.17}$$

where the negative sign in $\vec{w}_t - \alpha\vec{g}_t$ indicates that we are solving a minimization problem (of the squared error) by gradient descent. The gradient of the loss-function, is given by:

$$\vec{g}_t = \nabla_{\vec{w}_t} L^{\lambda}_{mse}(\vec{w}_t) = \frac{1}{2}\nabla_{\vec{w}_t}\big[T^{\lambda}_t - V_t(s_t)\big]^2 = -\big[T^{\lambda}_t - V_t(s_t)\big]\nabla_{\vec{w}_t} V_t(s_t). \tag{3.18}$$

The problem with this formulation is that it is acausal, since future rewards are required in order to compute the λ-return in each time step t. One possibility for overcoming this problem is to wait until the episode is completed and then perform a full backup for the whole episode (off-line backup), which can be expressed as:

$$\vec{w}_{T+1} = \vec{w}_T - \alpha\vec{g}_T = \vec{w}_T - \alpha\sum_{t=0}^{\infty}\vec{g}_t, \tag{3.19}$$

where $\vec{g}_T$ is the (negated) recommended weight change (RWC) for the overall episode. Obviously, the above equation cannot be implemented for infinite episodes. But with a few simple transformations we can obtain a causal representation, that also allows infinite episode lengths and, more important, that can be implemented in an on-line incremental manner. This representation is referred to as the backward view of TD(λ) [59, Sec. 7.3]. Although Sutton & Barto did not describe how to arrive at the backward view when the starting point is Equation (3.17) (forward view), they did show that both views are only equivalent in the case of off-line backups (the prediction is updated only episode-wise). Nevertheless, the on-line implementation of the backward view will typically be approximately the same as for the off-line case, if the step-size parameter α is small enough, since the changes in V_t will be sufficiently small then. Additionally, on-line backups have the advantage that an agent may already learn to avoid or to seek for certain states during its interaction with the environment and adjust it's behavior instantaneously.
The derivation of the backward view of the TD(λ) algorithm can be found in Appendix A.1, where we mainly derive a new formulation of the gradient in (3.18):

$$\vec{g}_t = \nabla_{\vec{w}_t} L^{\lambda}_{mse}(\vec{w}_t) = -\delta_t\vec{e}_t. \tag{3.20}$$

Inserting (3.20) back into (3.17) finally delivers the weight update rule of the TD(λ) algorithm:

$$\vec{w}_{t+1} = \vec{w}_t + \alpha\delta_t\vec{e}_t. \tag{3.21}$$

The new ingredient in TD(λ) is the so called eligibility trace vector, containing a decaying trace $\vec{e}_t = (e_{i,t})^T$ for each weight w_i [57]:

$$\begin{aligned} \vec{e}_t &= \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{\vec{w}} V(s_k) \\ &= \lambda\gamma \vec{e}_{t-1} + \nabla_{\vec{w}} V(s_t), \\ \vec{e}_0 &= \nabla_{\vec{w}} V(s_0), \end{aligned} \tag{3.22}$$

with the trace decay parameter λ and the discount factor γ, which decay (or discount) 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 ($\lambda = 1$) [57].

Replacing Traces

One problem with the definition of the conventional eligibility traces in Equation (A.19) may arise due to the fact, 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. This behavior may be undesirable if certain states are visited many times during one episode, since the corresponding traces can build up to large values. As a result, the TD(λ) algorithm might give higher credit for future rewards to frequently visited states than to more recent states, which could negatively affect the overall learning process. For example, consider an agent that selects the same wrong state s_w many times during an episode, but nevertheless manages to receive a good final reward by selecting a correct state s_c in a later phase of the episode. Most likely a training process with accumulating eligibility traces will then give higher credit to s_w than to s_c, which could impair the policy of the agent and slow down the learning.
Although many RL problems do not allow states to revisit a state in an episode (e.g., many board games such as Connect-4 and Dots-and-Boxes), a certain feature set might be active in many time steps when generalizing function approximators – such as n-tuple systems, as introduced in Section 3.3– are used.
Singh & Sutton [52] proposed *replacing eligibility traces*, in order to overcome this potential problem related to conventional accumulating eligibility traces. The main idea behind replacing traces is to replace a trace with $e_i = \nabla_{w_i} V(s_t)$ 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 for both types of traces is given in Figure 3.4.

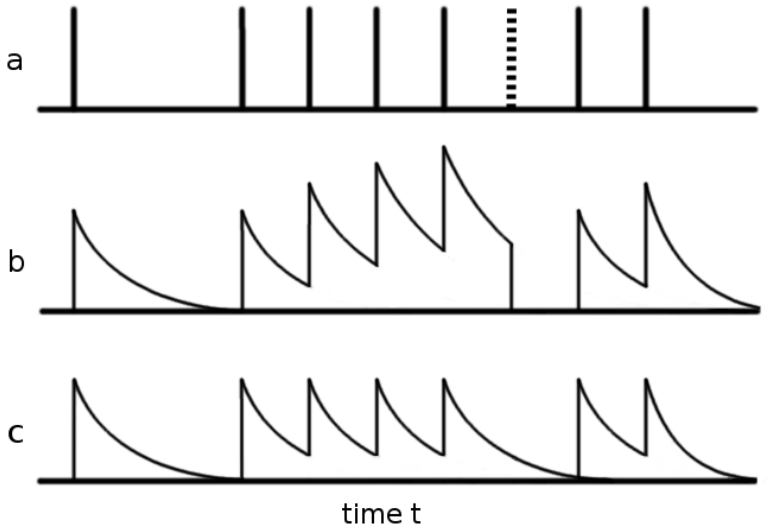

Figure 3.4: Schematic view of different eligibility trace variants: Line **a** illustrates the situation without eligibility traces ($\lambda = 0$), where a weight is activated only at isolated time points. The dotted vertical line represents a random move. Line **b** shows the eligibility trace with reset on random move. Line **c** shows replacing traces, this time without reset on random move.

Resetting Traces

The TD(λ) algorithm usually requires a certain degree of exploration (see Section 3.2) 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 prediction for $V(s_t)$. The weight update based on $V(s_t)$ is normally skipped in this case. This also raises the question of 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. Resetting traces are also illustrated in Figure 3.4.

3.3 Function Approximation with N-Tuple Systems

One central challenge for many RL tasks is the design of a suitable function approximation that can map states or state-action pairs to real values. Combined with a appropriate learning algorithm, the approximating value function should ideally converge towards the true (yet unknown) value function during the training phase. For small problems with a small state-space complexity one could simply use a tabular value function that directly maps a state (state-action pair) to an entry in the table. However, for most real-world problems, tabular value functions are impractical for three reasons: 1) The table cannot

be realized due to the constrained memory when the state-space is large. 2) Visiting all states of the system (typically several times) in order to learn the state-values is not feasible since time is a limiting factor. 3) It is not possible to transfer already gained experience to new situations. The generalization capability is one of the most important criteria when a value function is designed, since an agent typically has to learn from a very limited set of training samples and then has to be able to make the right decisions in new situations. For these reasons, the developer is normally forced to use approximations. However, there are several advantages that an approximation can offer: For many states – e.g., those that are only visited with a very low probability or that are irrelevant for other reasons – no exact values are required so that the function approximation can increase the accuracy of values for other relevant states. For example, agents learning board games normally only have to be able to find strong moves in a match against an expert player; large parts of the state-space are typically irrelevant, since strong players will attempt to avoid many states (and the sub-trees linked to these states) during a match. If the system states are mapped into a suitable feature space, an approximation may also have the advantage that certain patterns in the training samples can be identified, which allows us to accurately predict the values of states with similar patterns.
In this section we introduce n-tuple systems, which can be used to approximate the true value function in many applications, for instance for board games. In this work, n-tuple systems are the only type of function approximation that we will use.

Even though n-tuple systems were already introduced by Bledsoe & Browning in 1959 for character recognition purposes [16] and later for other tasks such as face recognition [37], the application to board games is rather new. More recently, Lucas proposed employing the n-tuple architecture for game-playing purposes: Lucas applied n-tuple systems to the strategic board game Othello and could achieve remarkable results using n-tuple systems in combination with TDL [36]. This n-tuple approach works in a way similar to the kernel trick used in support vector machines (SVM): The low dimensional board is projected into a high-dimensional feature space by the n-tuple indexing process. In the following we describe more detailed how n-tuple indexing process works and how n-tuple systems may be used to create a value function for the board games such as Connect-4 and Dots-and-Boxes.

N-tuple Systems for Board Games An *n-tuple* is defined as an (ordered) sequence of n so-called sampling points $T_\nu = (\tau_{\nu 0}, \tau_{\nu 1}, \ldots, \tau_{\nu n-1})$, where $\nu \in \{1, \ldots, m\}$ addresses individual tuples in a set of m n-tuples. Each sample point typically represents an individual cell on a board. For example, a chess board has 64 possible cells that can be sampled. Connect-4 consists of $7 \times 6 = 42$ cells in total, a sample point could therefore be coded as $\tau_{\nu j} \in \{0, \ldots, 41\}$. For Dots-and-Boxes it is sensible to sample the edges of the board, so that a sampling point can take any value $\tau_{\nu j} \in \{0, \ldots, M(N+1) + N(M+1)\}$. Depending on the game, every sampling point can be in P different states.

Depending on the current occupation of cell $\tau_{\nu j}$, its current state can assume a value $s_t[\tau_{\nu j}] \in \{0, \ldots, P-1\}$, where s_t represents the current state of the overall board. We

distinguish $P = 4$ states for Connect-4:

0 = empty and not reachable, 1 = Yellow,

2 = Red, 3 = empty and reachable.

By *reachable* we mean an empty cell that can be occupied in the next move. The reason behind this is that it makes a difference whether e.g. three yellow pieces in a row have a reachable empty cell adjacent to them (a direct threat for Red) or a non-reachable cell (indirect threat) [69].

A sampling point in Dots-and-Boxes can only be in one of $P = 2$ possible states (it does not play a role which player set the edge):

0 = empty,

1 = set.

With P possible values per sampling point, every n-tuple (of length n) can have P^n different states. Each n-tuple state is mapped to an unique index value $k_\nu \in \{0, \ldots, P^n - 1\}$ with

$$i = k_\nu = \sum_{j=0}^{n-1} s_t[\tau_{\nu j}] P^j. \tag{3.23}$$

For every n-tuple state, the generated index can be used to address a weight $w_{i,\nu,t}$ in a look-up table LUT_ν associated with n-tuple T_ν. Generally speaking, the individual LUTs can be considered as parameter vectors $\vec{w}_{\nu,t}$ that will be adjusted in every time step t during the training. Depending on the current occupation of cell $\tau_{\nu j}$ and the cell beneath, its current state can assume a value $s_t[\tau_{\nu j}] \in \{0, \ldots, P - 1\}$. One improvement when using n-tuple systems is the utilization of board symmetries, which are very common in many board games. In Connect-4, the mirroring of the board along the central column leads again to an equivalent position and in Dots-and-Boxes rotation, mirror and corner symmetries can be used (refer to Section 3.4 for details). With this in mind, we can calculate a set of index values K_ν for all symmetric equivalents of the current position. The individual weights of $\vec{w}_{v,t}$ are then activated according to a state-dependent input vector $\vec{x}_\nu(s_t)$, generated as follows:

$$x_{i,\nu}(s_t) = \begin{cases} 1, & \text{if } i \in K_\nu \\ 0, & \text{otherwise} \end{cases}. \tag{3.24}$$

Figure 3.5 shows an example board position with a 4-tuple and its mirrored equivalent. For a given board position at time step t, the output for each n-tuple is generated by calculating the product $\vec{w}_{\nu,t}^T \vec{x}_\nu(s_t)$. Since we typically have a system of m n-tuples T_ν with $\nu = 1, \ldots, m$,

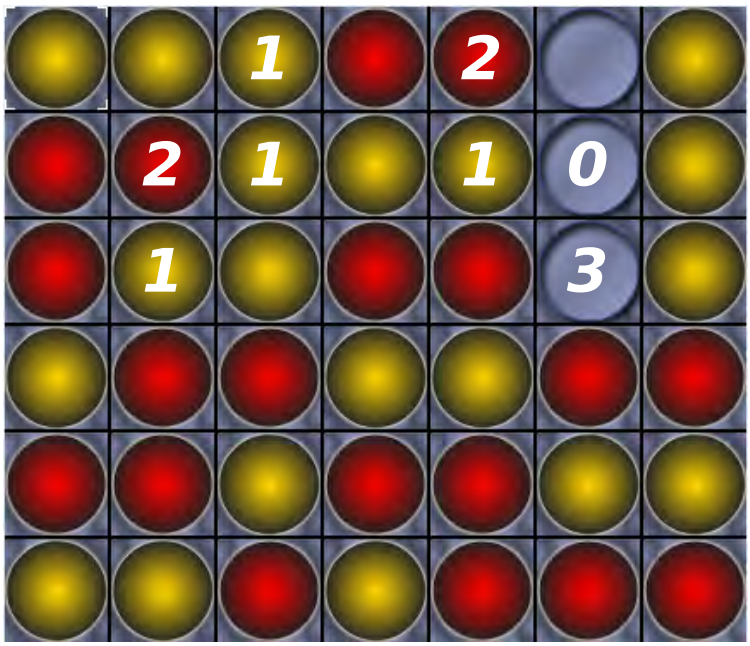

Figure 3.5: Example for an n-tuple (left-hand-side) and its mirrored equivalent (right-hand-side) on a Connect-4 board. The n-tuple is of length 4 and was created by a random walk on the board. Assuming that we follow the n-tuple on the left hand side from its lowest sampling point to its highest, by sampling the cells of the n-tuple we get the sequence 1–2–1–1 and for the mirrored n-tuple 3–0–1–2. The corresponding index values are calculated as $k = 1 \cdot 4^0 + 2 \cdot 4^1 + 1 \cdot 4^2 + 1 \cdot 4^3 = 89$. Likewise, the mirrored n-tuple has $\overline{k} = 147$.

the overall output of the n-tuple network can be formulated as a linear function

$$\begin{aligned} f\left(\vec{w}_t, \vec{x}(s_t)\right) &= \vec{w}_t^T \vec{x}(s_t) \\ &= \sum_{\nu=1}^{m} \vec{w}_{\nu,t}^T \vec{x}_\nu(s_t) \\ &= \sum_{\nu=1}^{m} \sum_{i=0}^{P^n-1} w_{i,\nu,t} \cdot x_{i,\nu}(s_t), \end{aligned} \tag{3.25}$$

where $\vec{w}_{\nu,t}$ and $\vec{x}_\nu(s_t)$ are considered here as sub-vectors, contained in two large vectors $\vec{w}_t$ and $\vec{x}(s_t)$, respectively. The output of the n-tuple network may be put through an activation function such as tanh to force the values into a certain range. All weights in $\vec{w}_t$ are updated according to the TDL update rule (Section 3.2.2).

Overall, n-tuple networks induce a mighty feature space. The main advantage is that it is not necessary to design certain features, which would require detailed knowledge about the game, but the agent learns to select the right ones itself.

The memory consumption of an n-tuple system scales linearly with the number of n-tuples and exponentially with the n-tuple length. The run-time for computing the value function and updating the weights is nearly independent of the LUT-size (n-tuple length) and scales linearly with the number of n-tuples.

3.4 Learning Board Games with TD Methods and N-Tuple Systems

At this point we have everything together to define an incremental TD(λ) algorithm for board games, in which an agent learns the weights of an n-tuple system entirely by performing *self-play* matches. In this section we describe how to combine the TD(λ) algorithm with n-tuples and discuss several related aspects regarding the board games Connect-4 and Dots-and-Boxes. Again we want to emphasize that no external knowledge (in form of databases, tree-search algorithms or other teachers) will be used during the training process.

3.4.1 The incremental TD(λ) Algorithm for Board Games

The pseudo-code of the incremental TD(λ) algorithm for board games is listed in Algorithm 1 and describes the procedure for a single training game. Typically, this procedure is repeated many times (up to several million of repeats). The main idea behind the algorithm is to learn a state-value function based on the definition of the Bellman optimality equation in (3.5). Since the optimal value function $V^*(s)$ is unknown, the Bellman equation is approximated by inserting the current estimate V_t (which does not exactly satisfy the Bellman equation). The sample error δ_t (typically mean-squared and weighted proportional to state frequency) between both sides the Bellman equation is then minimized by the TD(λ) weight-update rule (line 22 in Algorithm 1).
Since the agent selects the actions for both players, it attempts to maximize the expected reward for the first player and minimize the expected reward of the second player (greedy policy, indicated by line 16 in the pseudo-code). However, in order to explore the state-space of the game, the agent does not always perform greedy actions. Instead, an ϵ-greedy policy is followed which forces the agent to randomly select an action with a probability of ϵ. Random actions are generally not optimal and consequently, the weight update is skipped for these actions, with exception of situations that result in a terminal state [4].

For most board games it is sensible to provide rewards only at the end of each training-game, when the actual outcome of the match is known. Providing intermediate rewards may prevent the agent from learning a strong policy. Typically, the reward function will

[4]We assume that an action that leads to a terminal state is optimal for the player that performed this action, since this situation mostly represents a win-situation. For games, where a player can loose by her own move, this assumption does not hold any longer.

Algorithm 1 Incremental $TD(\lambda)$ algorithm for board games, based on [32]. Prior to the first game, the weight vector $\vec{w}$ is initialized with random values. Then the following algorithm is executed for each complete board game. During self-play the player is toggled between +1 (first player) and -1 (second player).

Set $REP = true$, if using replacing traces
Set $RES = true$, if resetting elig. traces on random moves
Set the initial state s_0 (usually the empty board) and $p = 1$
Use partially trained weights $\vec{w}$ from previous games
function TDLTRAIN(s_0, $\vec{w}$)
    $\vec{e}_0 \leftarrow \nabla_{\vec{w}}\, y(\vec{w}, \vec{x}(s_0))$ ▷ Initial eligibility traces
    for $\big(t \leftarrow 0$ **;** $s_t \notin S_{Final}$ **;** $t \leftarrow t+1, p \leftarrow (-p)\big)$ **do**
        $V_{old} \leftarrow y(\vec{w}_t, \vec{x}(s_t))$ ▷ Value for s_t
        generate randomly $q \in [0, 1]$
        if $(q < \epsilon)$ **then**
            Randomly select s_{t+1} ▷ Explorative move
            if (RES) **then**
                $\vec{e}_t \leftarrow 0$
            end if
        else
            Select after-state s_{t+1}, which maximizes ▷ Greedy move
            $$p \cdot \begin{cases} R(s_{t+1}), & \text{if } s_{t+1} \in S_{Final} \\ y(\vec{w}_t, \vec{x}(s_{t+1})) + R(s_{t+1}), & \text{otherwise} \end{cases}$$
        end if
        $V(s_{t+1}) \leftarrow y(\vec{w}_t, \vec{x}(s_{t+1}))$
        $\delta_t \leftarrow R(s_{t+1}) + \gamma V(s_{t+1}) - V_{old}$ ▷ TD error-signal
        if $(q \geq \epsilon$ **or** $s_{t+1} \in S_{Final})$ **then**
            $\vec{w}_{t+1} \leftarrow \vec{w}_t + \alpha \delta_t \vec{e}_t$ ▷ Weight-update
        end if
        for (every weight index i) **do** ▷ Update elig. traces
            $\Delta e_i \leftarrow \nabla_{w_i}\, y(\vec{w}_{t+1}, \vec{x}(s_{t+1}))$
            $$e_{i,t+1} \leftarrow \begin{cases} \Delta e_i, & \text{if } x_i(s_{t+1}) \neq 0 \wedge REP \\ \gamma \lambda e_{i,t} + \Delta e_i, & \text{otherwise} \end{cases}$$
        end for
    end for
end function ▷ End of TD(λ) self-play algorithm

therefore return $R(s_{t+1}) = 0$, as long as $s_{t+1} \notin S_{Final}$. For our Connect-4 task rewards are only given in terminal states. We will later discuss one option for Dots-and-Boxes, where it is reasonable to also provide intermediate rewards.

Eligibility traces are reset at the beginning of each episode and initialized with the gradient of the state-value function, evaluated for the initial state s_0. If the option 'RES' is selected, then the trace vector is reset (line 13) when a random action is performed and if the option 'REP' is selected, then for all active inputs x_i the individual traces e_i are replaced with the new gradient information $\nabla_{w_i} y(\vec{w}_{t+1}, \vec{x}(s_{t+1}))$ (line 26).

As already described in previous sections, the response of the n-tuple system for a board position can be denoted as:

$$f\big(\vec{w}_t, \vec{x}(s_t)\big) = \vec{w}_t^T \vec{x}(s_t) \tag{3.26}$$

We put this response through an (activation) function σ, which leads to the value function:

$$V_t(s_t) = y(\vec{w}_t, \vec{x}(s_t)) = \sigma\Big(f\big(\vec{w}_t, \vec{x}(s_t)\big)\Big), \tag{3.27}$$

where σ could be a logistic sigmoid function, the $tanh(\cdot)$ function or simply an identity function. In our experiments, the choice of σ will depend on the step-size adaptation algorithm (introduced later in Section 3.5). The gradient of $V_t(s_t)$ in vector-element form is therefore:

$$\nabla_{w_i} y(\vec{w}_t, \vec{x}(s_t)) = \sigma'(a_t) x_i(s_t), \tag{3.28}$$

where $a_t = f\big(\vec{w}_t, \vec{x}(s_t)\big)$ is the response of the n-tuple system. For example, with $\sigma = \tanh$, we retrieve:

$$\nabla_{w_i} y(\vec{w}_t, \vec{x}(s_t)) = \big[1 - f\big(\vec{w}_t, \vec{x}(s_t)\big)^2\big] x_i(s_t). \tag{3.29}$$

Implementing Eligibility Traces for N-Tuple Systems Since the TD(λ) algorithm requires one eligibility trace for each weight of a system, the number of traces for an n-tuple system would typically be in the range of several millions. Handling such a large number of traces – considering the computation effort (theoretically, each trace has to be touched in every time step) and memory requirements – seems to be nearly impossible at first sight. We realized however that the degree of activation is rather sparse in an n-tuple system (for both tasks, Connect-4 and Dots-and-Boxes). Consequently, the trace vector $\vec{e}_t$ is sparse as well. For example, with the standard n-tuple system – consisting of 70 n-tuple of length 8 – used for our Connect-4 task an upper bound for the number of inputs that can be activated is: the episode length (42) times the number of n-tuples (70) times 2 (mirror states), hence, in total 5 880 inputs. Therefore, in maximum 5 880 traces will be non-zero during an episode. In practice, this number is even smaller, since not all active inputs change in each time step, mirrored states of an n-tuple activate the same input and

the episode is often completed before all 42 pieces are placed on the board. For the typical setup of our Connect-4 training, we measured that on average only around 1 800 individual traces are non-zero for our standard n-tuple system (70 × 8-tuple). In Dots-and-Boxes this number is typically slightly larger, but still in the range of only several thousands. Since the trace vector is reset again after each episode, the additional resources needed to implement eligibility traces can be reduced to a reasonable amount: the individual activated traces are placed in a self-balancing (height-balanced) binary search tree (TreeMap in Java). This approach significantly reduces the memory consumption of the training process and the computation time as well. Instead of millions of addressable traces we realize in each episode only the active traces which are only a few thousand in average.

3.4.2 N-tuple Creation Process and Symmetries

N-tuple Generation There are many different approaches to create the individual n-tuples of a system: One could use a set of hand-designed n-tuples, randomly sample the board, perform random walks on the board or use a combination of these approaches. For our Connect-4 task, we found the following creation process to be the best (as already described in [69]): Similarly to Lucas' snake method [36], we also create our n-tuples by performing random walks on the board, moving from one board-cell to another adjacent cell. However, our method ensures that no cell is visited twice so that each n-tuple meets its required length. To create an n-tuple of length K, we start at a random board position, which will the first cell of the n-tuple. Subsequently, we visit one of its 8 neighbors and add it to the n-tuple (if not already in there). We continue to one of its neighbors, and so on, until we have K cells in the n-tuple. An example for a random walk on a Connect-4 board was already given in Figure 3.5.

Similarly, one could perform a 'random line walk' in Dots-and-Boxes, where we create the n-tuples, by moving along the edges of the board. Although we did not have enough time to test different n-tuple generation approaches in detail, we found the following method to deliver the best results: Instead of performing a 'random line walk', we generate an n-tuple of length K by a 'random box walk'. Hence, starting from one randomly selected box, we move from each box to one of its 8 neighbors and add all edges adjacent to the box to our n-tuple. If the number of edges for the n-tuple exceeds K (since every additional box adds three new edges), we randomly remove one or two of the last edges. Since chains play an important role in Dots-and-Boxes, we believe that this method is superior to other approaches. An example is given in Figure 3.6.

Exploiting Symmetries Board games often have several symmetries in the board position. In case of Connect-4, there exists only one symmetry, the mirror reflection of the board along the middle column. For Dots-and-Boxes, many symmetric positions can be created under rotation and mirroring of the board: for rectangular boards 4 and for squared boards 8 such symmetric positions exist. It can be shown that another symmetry exists [11]

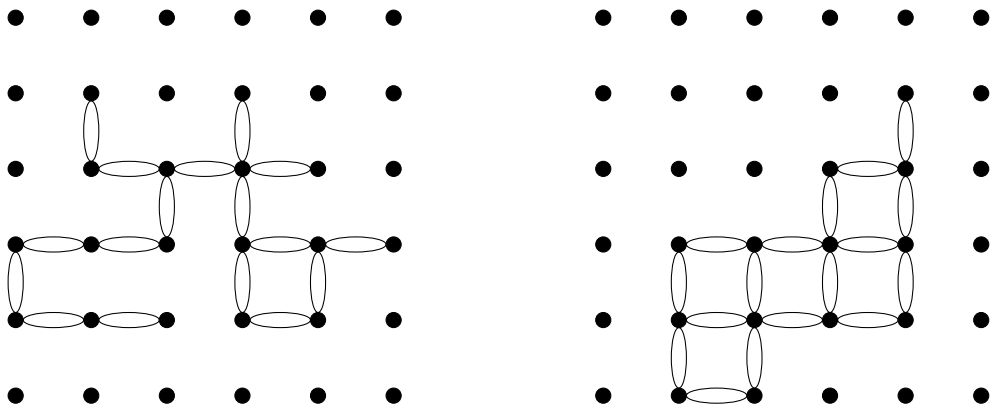

Figure 3.6: Example n-tuple creation process for Dots-and-Boxes, following the 'random line walk' approach (left) or the 'random box walk' (right). Here, both approaches create a single 17-tuple on a board with 5×5 boxes. Each ellipse represents an individual sampling point.

that may not be as obvious as others: For any corner of the board, represented by a pair of edges each, with one edge empty and the other edge being filled-in, it can be shown that an equivalent position (with the identical game-theoretic value) can be created by simply interchanging the empty edge with the filled-in edge. If $m \leq 4$ corners of the board have this constellation of one edge being empty while the other edge is filled-in, then the following number of 'corner-symmetric' positions for the specified board exist:

$$\sum_{i=0}^{m} \binom{m}{i} = 2^m. \tag{3.30}$$

A square board with $m = 4$ would therefore have in total $8 \cdot 2^4 = 128$ equivalent positions (including the original board).

As already discussed before in Section 3.3, symmetric positions can be utilized in the indexing process of the n-tuple system and improve the overall learning speed of the agent. We will use all of the here described symmetries for our later experiments. For Connect-4, we will create the n-tuples by the 'random walk' method while the n-tuples for our Dots-and-Boxes task will be created by 'random box-walks'.

3.4.3 Learning Value Functions for both Opponents

We observed for our learning tasks Connect-4 [69, 67] & Dots-and-Boxes that the values of sampled sub-parts of the board can have rather different values, depending on the player to move. If only one LUT is trained for each n-tuple and the information about the player to

move is not considered, then these different values will result in conflicting weight update signals and hamper the overall training process. In the following we describe two possible approaches in order to overcome this problem.

Separating the Value Functions The most obvious solution to above problem is to simply introduce two LUTs per n-tuple, with one LUT for each player. The learning process can then access the LUTs, depending on the player to move. This approach can be used for any board game, but it doubles the memory requirements and can slow down the training-progress for certain games, if both sides have to learn the same or very similar strategies, since both sides then have to learn an individual strategy separately. For our Connect-4 task, this was the only approach that led to good results. It appears that the strategies for both sides (Yellow and Red) are rather different.

Board Inversion If the optimal strategy of both opponents is similar or even the same, one could consider an approach in which the agent learns the value function for only one of the two players. For a game with the players A and B this would mean that only the value function for states with player A to move is learnt. If the value of a state with player B to move has to be determined, we simply invert the current board configuration (interchanging all pieces and other player-related information of the opponents) and consult the value function for the inverted state; the real value is then the negated response of the value function. This approach is especially beneficial for (but not necessarily limited to) impartial games such as Dots-and-Boxes, where the possible moves only depend on the current position and not on the player to move (Connect-4 is not impartial, since the opponents play with pieces of different color). Since the impartiality property also implies that the optimal moves from a certain position are always the same – no matter if the first or second player is to move – it is sufficient to learn a strategy for one of both players.

Intermediate Rewards in Dots-and-Boxes Similar to the problem of the missing player information in an n-tuple system – which is solved by introducing a LUT per player or using board inversion – for Dots-and-Boxes another problem arises: the n-tuple system only samples the edges of the board and does not code any information about the current occupation of the boxes, hence, the number of boxes owned by A and B, which can be expressed as a box-margin (number of boxes for A minus number of boxes for B). However, to predict the outcome (win/loss/tie) of a game for a certain position, the current box-margin is a crucial information. If this information is not considered by the n-tuple system, then the individual n-tuple states will receive many conflicting signals, since they are typically part of winning and losing situations. In other words: it cannot be generally assumed that a particular n-tuple state tends to result in a win, loss or a tie, if the box-margin is not known. Even though a certain pattern – sampled by an n-tuple – may be desirable, since it allows a player to capture many boxes in future, the speci-

fied player might still loose the match, because she did not capture enough boxes in the end.

Instead of predicting the final outcome of a match, a sensible and simple approach to remove the dependency on the box-margin is to predict the number of boxes that can be captured in future, starting from the current board position. In order to predict this, the information about the current box-margin is not required any longer. This approach also allows us to find the best moves for a position, since a player always aims at – totally independent from the current box-margin – maximizing the number of captured boxes with her remaining moves (minimizing at the same time the boxes of the opponent). In order to achieve this, intermediate rewards are necessary. Consider a state-value function $V(s_t)$ that predicts the number of boxes that both players will capture in the remaining episode, indicated again by a box-margin. For instance, a value of $V(s_t) = -2$ signalizes that player B will capture two boxes more than A until the end of the match. The estimate is updated with the standard TD-update rule:

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\Big]. \tag{3.31}$$

If the environment now provides intermediate rewards for every captured box ($r_{t+1} = +1$, if player A captures a box for the transition of s_t to s_{t+1} or, accordingly, $r_{t+1} = -1$, if player B captures a box) then the estimate $V(s_t)$ will converge towards the real values during the training process, which then also allows us to find the optimal moves for any board position.

3.5 Online Adaptable Learning Rate Methods

Many minimization problems are solved by gradient-descent approaches, where an algorithm iteratively takes steps in the direction of the steepest descent of an objective function. The direction of steepest descent at a certain point can be determined by computing the negated gradient of the function at the specified point. Many algorithms in the field of machine learning rely on this basic concept.
Also TD-Learning (although it cannot be seen as a real (stochastic) gradient descent method, as discussed in Section 3.2.1) minimizes the squared (sample) temporal difference error δ_t^2 by using the gradient information to gradually improve a weights-vector $\vec{w}$ of a parametrized approximation function. Recall that the TD weight-update rule is:

$$\vec{w}_{t+1} = \vec{w}_t + \alpha\delta_t\vec{e}_t, \tag{3.32}$$

where α is a positive step-size (learning rate) parameter. TD-Learning as well as most gradient descent methods have in common that they require suitable values for their step sizes in order to reliably converge towards the minimum. If the step sizes are chosen too small then convergence may be rather slow and if chosen too high then the process may not reach the optimum or could even diverge. Typically, every optimization step has a different

optimal step size.
Another aspect that has to be considered for many tasks is that gradient descent methods might have to deal with non-stationary problems, where the target values change over time, e.g. in dynamic environments. The TDL algorithm can be seen as a non-stationary learning task, since the involved bootstrapping process causes the individual targets to drift during the training. Furthermore, the policy of the agent is typically adjusted over time which also results in changing target values. In order to track these drifting targets, sufficiently high step sizes are required. Once the system is stationary, the step sizes should then decrease again in order to reduce the asymptotic error.
Finding suitable step sizes is a non-trivial task and many publications in recent years have been concerned with this problem, proposing many step-size learning algorithms for various learning tasks. In this section we will investigate several of the suggested online adaptable learning rate methods. Here, 'online' indicates that the step size(s) of the step-size adaptation method are adjusted directly for each observed training sample. In contrast to this, offline (batch) methods would adjust the step sizes after a whole 'batch' of training samples were processed.

The simplest choice for a step-size parameter might be a fixed scalar value, which in many cases only leads to sub-optimal results. In a next step, one could define a deterministic step-size adjustment scheme that adjusts the step size according to a predefined sequence of values. In our previous work [69, 67], we used such a deterministic scheme, which exponentially decayed the step size during the training from an initial value α_{init} to a final value α_{final}.
Many algorithms used in this work extend the concept of scalar step sizes to vector-valued step-size algorithms, thus, these algorithms introduce an individual step size for each weight w_i of a parametrized approximation function. Note that with such an approach the direction of the gradient is changed, since every dimension of the gradient is scaled individually. Nevertheless, this class of algorithms often perform better than adaptation schemes with scalar step size.

In the following, we will discuss in detail the TCL & IDBD algorithm and subsequently briefly mention the remaining algorithms that we will examine in this work.

3.5.1 Temporal Coherence Learning

Temporal coherence learning (in the following TCL), developed by Beal & Smith [13, 14], is a technique designed especially for Sutton's TD(λ) algorithm, which maintains additional adjustable learning rates α_i for each weight of the parameter vector $\vec{w}$. The basic idea behind TCL – and many of the other approaches discussed in later sections – is rather simple: in order to achieve a faster convergence of the weights, individual step sizes are increased (although in TCL no step size can exceed an upper-bound) if the corresponding weights w_i are relevant for the learning process. Once a weight approaches its optimum

value and no significant learning can be observed any longer, the step size for this weight is reduced again, to prevent the weight fluctuating around its optimum value and adding random noise to the system. Also, the step sizes for those weights should be decreased, which do not contribute to the learning process. Such irrelevant weights will not tend to any specific direction and also only induce unnecessary random noise in the system, although they accumulate the recommended weight changes (RWC) over a longer period of time.
Using individual step-sizes in TCL has the desirable effect, that the weights can approach their final values at different instances of time and the step sizes adapt based on the current learning progress of the weights. Weights that need longer than others to reach their final value should still have a higher learning rate, even if other weights are already stable.
In order to adapt the learning rates, two counters N_i and A_i for each weight accumulate the RWC and the absolute RWC respectively. In each time step, the counters are adjusted in the following manner:

$$r_{i,t} = \delta_t e_{i,t}, \tag{3.33}$$

$$N_i \leftarrow N_i + r_{i,t}, \tag{3.34}$$

$$A_i \leftarrow A_i + |r_{i,t}|, \tag{3.35}$$

where r_i denotes the recommended weight change (RWC) for weight i and $e_{i,t}$ the i-th element of the eligibility trace vector; since eligibility traces are already included in the formulation of TCL, there are no additional adjustments necessary.
If all weight changes r_i have the same sign, then $|N_i|/A_i = 1$ and the corresponding step size will remain at an upper bound. On the other hand, if the sequence of RWC is noisy and does not carry (or no longer carries) any useful information for the learning of weight i, then the counter N_i will tend towards zero and $|N_i|/A_i \to 0$. With the two counters N_i and A_i, the individual learning rate α_i can be expressed as:

$$\alpha_i = \begin{cases} g(\frac{|N_i|}{A_i}), & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases}, \tag{3.36}$$

where $g(x)$ is a transfer function that may be selected as desired, taking into consideration two conditions: 1. The function operates in the range $0 \leq x \leq 1$ and must not have any discontinuities in this range. 2. The function should be (strictly monotonic increasing). 3. The function values are positive. In its original form, as described in [13, 14], TCL simply uses the identity transfer function $g(x) = x$, so that the individual step size is limited to the range $\alpha_i \in [0, 1]$. More recently, we introduced an exponential scheme [8] for the transfer function $g(x)$ in TCL, with:

$$g(x) = \mathrm{e}^{\beta(x-1)} \tag{3.37}$$

This variant (in the following denoted as TCL-EXP) was inspired by Sutton's Incremental Delta-Bar-Delta (IDBD) algorithm [58], which computes the individual step sizes with an exponential function (discussed later in Section 3.5.2). An exponential function e^x has the

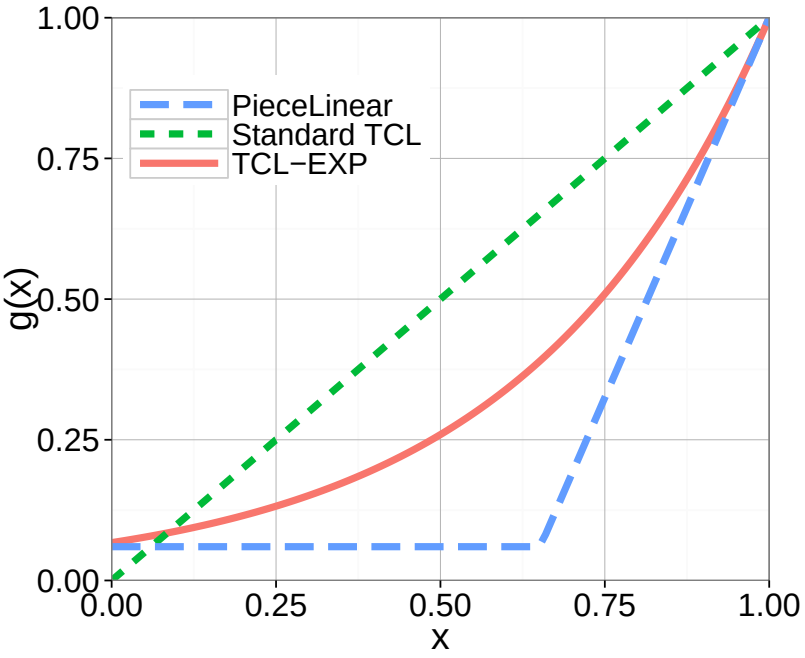

Figure 3.7: Different options for the transfer function $g(x)$ in TCL. The original TCL implementation simply uses the identity transfer function $g(x) = x$ (green dashed curve). The variant TCL-EXP uses an exponential scheme, with $g(x) = \exp(\beta(x-1))$ and $\beta = 2.7$, illustrated here by the red curve. Another possible transfer function is a piecewise linear (PL) function, represented by the blue dashed curve, which has the same endpoints as TCL-EXP and the same slope in $x = 1$.

characteristic that a fixed change in x will change e^x by a fixed fraction of its current value, i.e. $e^{x+\Delta x} = e^x \cdot e^{\Delta x}$, which allows geometric step-size changes in TCL-EXP.

After determining the individual step sizes with Equation (3.36), the weights of the system are updated according to the δ-rule with

$$w_i \leftarrow w_i + \alpha\alpha_i\delta_t e_{i,t}, \tag{3.38}$$

where α is the global step-size parameter. Since the effective individual step size $\alpha\alpha_i$ is limited to an upper bound, the value for α should be chosen sufficiently large and – according to [13, 14] – α_i will then provide an automatic adjustment during the learning process. For the update process, the following operational order should be obeyed: First, the weight update is performed using the previous counter-values and then, subsequently, N_i and A_i are adjusted according to (3.35). For details, refer to the pseudo code provided in Algorithm 2. This code replaces the weight update step in the original TD(λ) algorithm (Algorithm 1, line 22). The original TCL algorithm also has the further possibility to

Algorithm 2 General TCL algorithm in pseudo code. The counters A_i and N_i are initialized *once* at the beginning of the training. In the original algorithm, the individual learning rates α_i are calculated with the identity transfer function $g(x) = x$, but also other transfer functions may be selected, taking into consideration certain conditions.

Initialize counters $A_i = 0$ and $N_i = 0 \quad \forall i$.
Set global learning rate α.
for (every weight index i) **do**
    $\alpha_i = \begin{cases} g\left(\frac{|N_i|}{A_i}\right), & \text{if } A_i \neq 0 \\ 1, & \text{otherwise} \end{cases}$ ▷ Individual step-size
    $r_{i,t} = \delta_t e_{i,t}$ ▷ recommended weight change
    $w_i \leftarrow w_i + \alpha \alpha_i r_i$ ▷ TD update for each weight
    $A_i \leftarrow A_i + |r_{i,t}|$ ▷ update accumulating counter A
    $N_i \leftarrow N_i + r_{i,t}$ ▷ update accumulating counter N
end for
with a transfer function $g(x)$,
the approximation error δ_t and,
the eligibility traces $e_{i,t}$.

perform episodic updates: The *RWC* $r_{i,t}$ may be accumulated over a sequence of time steps before the actual weight update takes place. This option balances a tradeoff between faster changing learning rates (short sequences) and statistically more reliable weight updates (long sequences).[5] Since we found short episodes, ideally of minimum length, to provide the best results for our application, we do not consider update episodes for our experiments.

3.5.2 Incremental Delta-Bar-Delta

Similarly to TCL and many other step-size adaptation methods, Sutton's Incremental Delta-Bar-Delta algorithm [58] (in the following IDBD) is a meta-learning algorithm that introduces an individual learning rate α_i for every weight w_i of a *linear unit*.[6] Sutton assessed IDBD's capabilities on a small artificial non-stationary task and showed that IDBD significantly outperforms an ordinary LMS algorithm with an optimal learning rate in terms of the asymptotic error. Furthermore, he demonstrated that IDBD could detect irrelevant inputs

[5] The usage of update episodes may have the disadvantage that additional temporary counters are needed, which would significantly increase the memory requirements for systems with many weights.

[6] Non-linear units require a re-computation of IDBD's update rules. This can be done analogous to the derivation for linear units in [58]

and adjust their step sizes accordingly to small values and that the step sizes of relevant inputs converged to their optimal values.
Although Sutton only tested IDBD on supervised learning tasks with a teacher providing the target y^*, the algorithm should also be applicable to TDL. TDL does not have a teaching signal, instead, the target value is based on a future prediction (refer to Section 3.2 for details). In this sense, TDL is similar to a non-stationary supervised learning problem, due to its bootstrapping process that induces a drifting target that has to be tracked. Also the adjustments in the policy of a TDL agent cause the target values to change during the training.
In the IDBD algorithm, the individual learning rates are all determined by applying the exponential function to a memory parameter β_i:

$$\alpha_i = \mathrm{e}^{\beta_i}. \tag{3.39}$$

A global learning rate – such as in TCL – is not required in IDBD. Using the exponential function has the advantage that α_i will never be negative. Additionally, another desirable property is, that for a fixed change $\Delta\beta_i$ in the memory parameter β_i, α_i is adjusted by a fixed fraction of its current value [58]:

$$\alpha_{i,t+1} = \mathrm{e}^{\beta_{i,t}+\Delta\beta_{i,t}} = \alpha_{i,t}\mathrm{e}^{\Delta\beta_{i,t}}. \tag{3.40}$$

This allows geometric steps in α_i, so that the individual step sizes can converge fast towards their optimal values. For instance, a change in β_i by a value of $\Delta\beta_i = 0.1$ will increase α_i by approximately 10%.
The whole update process for IDBD is listed in Algorithm 3 and can be interpreted as follows[7]: Before the actual learning process starts, the memory parameters of the IDBD algorithm have to be initialized and a suitable meta-learning rate θ has to be chosen (line 2 to 3). The parameter β_i may be initialized as desired with β_{init}. Alternatively, in [61], Sutton proposes to set $\beta_i = \log(N^{-1})$ in the beginning.
In addition to β_i, IDBD maintains a memory trace h_i for every weight of the system and initializes $h_i = 0$. According to the update rule in line 9, h_i can be understood as a decaying trace, accumulating past weight changes. In every time step the current weight change $\Delta w_i = \alpha_i \delta_t x_i$ is added to h_i. If a sequence of weight changes moves the corresponding weight w_i into a certain direction, then also h_i will shift into the same direction. If Δw_i is just plain noise, then h_i will tend towards zero. The sum of previous weight changes in h_i is decayed by the factor $[1 - \alpha_i x_i^2]^+$. The expression $[d]^+$ – where $[d]^+$ is d if $d > 0$, otherwise 0 – ensures that the decaying factor is always positive.
Since h_i is a trace for recent weight changes, the correlation with the current recommended weight change $\delta_t x_i$ indicates whether the last steps in w_i were either too small or too large.

[7] Sutton's derivation of the IDBD algorithm following a gradient descent approach – in much the same way as the $\delta-$update rule for the weights w_i, but in the space of individual step size parameters – can be found in [58].

Algorithm 3 Pseudo code of the IDBD algorithm for linear function approximators [58].

1 Initialize for all i: $\beta_i = \beta_{init}$ **or** $\beta_i = \log\frac{1}{N}$ (proposed in [61])
2 Initialize for all i: memory trace $h_i = 0$
3 Set the meta-learning rate θ
4 **for** (every weight index i) **do**
5 $\quad \beta_i \leftarrow \beta_i + \theta\delta_t x_i h_i$ ▷ Step in β_i
6 $\quad \alpha_i = \mathrm{e}^{\beta_i}$ ▷ Determine individual Step-size
7 $\quad \Delta w_i = \alpha_i \delta_t x_i$ ▷ Calculate weight change
8 $\quad w_i \leftarrow w_i + \Delta w_i$ ▷ Update weight
9 $\quad h_i \leftarrow h_i[1 - \alpha_i x_i^2]^+ + \Delta w_i$ ▷ Decaying trace of recent weight changes
10 **end for**
11 with:
12 $\quad [d]^+ = d$ **if** $d > 0$, 0 **else**,
13 $\quad$ the real-valued inputs x_i,
14 $\quad$ the number of inputs N and,
15 $\quad$ the approximation error δ_t

Accordingly, the step size should be increased or decreased, respectively. This is expressed by line 5 in the pseudo code: If the *RWC* $\delta_t x_i$ is positively correlated with h_i, then β_i is increased (since the recent update steps could have been larger). A negative correlation indicates an overshoot and the step size is reduced.
Note that β_i and h_i do not change, if the input signal x_i is zero; the time complexity of the algorithm scales linearly with the degree of activation in the system. This property is especially beneficial for learning tasks with sparse activation, such as n-tuple systems.

Non-Linear IDBD: nl-IDBD

The original IDBD algorithm is only defined for linear units. For non-linear function approximators IDBD's update rules have to be re-computed. In [34, pp. 31] and [60], Koop presents a non-linear version of the IDBD algorithm (in the following referred to as nl-IDBD) which uses a logistic sigmoid squashing function σ in the output:

$$V_t(s_t) = \sigma\big(a(s_t)\big), \tag{3.41}$$

$$a(s_t) = \vec{w}_t^T \vec{x}(s_t), \tag{3.42}$$

$$\sigma(\nu) = \frac{1}{1 + \mathrm{e}^{-\nu}}. \tag{3.43}$$

Furthermore, instead of the squared error, the loss is measured based on the cross-entropy between the target signal $T_t = T_t(s_t)$ and the current prediction $V_t = V_t(s_t)$:

$$L_{CE}(\vec{w}_t) = -T_t \log\big(V_t\big) - (1 - T_t) \log\big(1 - V_t\big), \tag{3.44}$$

The update rules for nl-IDBD (for details regarding the derivation, refer to Appendix A.2) can be summarized as follows:

$$\begin{aligned}
\beta_{i,t+1} &= \beta_{i,t} + \theta \delta_t x_i h_i \\
\alpha_{i,t+1} &= \mathrm{e}^{\beta_{i,t+1}} \\
w_{i,t+1} &= w_{i,t} + \alpha_{i,t+1} \delta_t x_{i,t} \\
h_{i,t+1} &= h_{i,t} \big[1 - \alpha_{i,t} V_t \big(1 - V_t\big) x_{i,t}^2\big]^+ + \alpha_{i,t} \delta_t x_{i,t}.
\end{aligned} \tag{3.45}$$

The above update rules are nearly the same as before for the linear case, only the decaying trace h_i (line 9 in Algorithm 3) has to be adjusted slightly; the general procedure and operational order of nl-IDBD remains the same as in Algorithm 3.

Non-Linear IDBD for the TD(λ) Algorithm: nl-IDBD(λ)

Going a step further, a natural question to ask would be whether it is possible to derive backup rules for nl-IDBD, when the TD(λ) algorithm is used for learning a value function. In a naive approach, one could simply replace the input vector $\vec{x}_t$ in the defined update rules by the eligibility trace vector $\vec{e}_t$. This is, however, not correct. The key idea in order to derive correct update rules is to use the λ-return T_t^λ (Section 3.2.2) as target signal instead of the standard TD target $T_t = r_{t+1} + \gamma V_t(s_{t+1})$. At first glance, this may appear not straightforward, since the provided target signal T_t^λ in the forward view of TD(λ) is acausal, as already discussed in Section 3.2.2. For this reason, the forward view is not directly implementable. To overcome this issue, eligibility traces were introduced in a backward view of TD(λ), yielding a causal, implementable formulation of the algorithm. Similar to the explanations in Section 3.2.2, we can also re-formulate the cross-entropy loss of nl-IDBD with the λ-return as target value and then subsequently derive (in Appendix A.3) new update rules for $\vec{w}_t$, $\vec{\beta}_t$ and $\vec{h}_t$:

$$\begin{aligned}
\beta_{i,t+1} &= \beta_{i,t} + \theta \delta_t e_{i,t} h_{i,t} \\
\alpha_{i,t+1} &= \mathrm{e}^{\beta_{i,t+1}} \\
w_{i,t+1} &= w_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} \\
h_{i,t+1} &= [1 - \alpha_{i,t+1} V_t (1 - V_t) x_{i,t} e_{i,t}]^+ h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} \\
e_{i,t+1} &= \lambda \gamma e_{i,t} + x_{i,t+1}.
\end{aligned} \tag{3.46}$$

The operational order should be kept as specified above. In Algorithm 4 it is described how nl-IDBD(λ) can be used in conjunction with the incremetnal TD(λ) algorithm for board

games (Algorithm 1).
As before, $h_{i,t}$ is a decaying trace, although the factor $[1 - \alpha_{i,t+1} V_t(1 - V_t) x_{i,t} e_{i,t}]$ only then causes a decay, when the corresponding input x_i is active. This could prevent a too large decrease of h_i in a short time, since h_i is adjusted in each time step of the remaining episode, once the trace e_i is triggered. Note that we again retrieve Koop's & Sutton's original update rules if we set $\lambda = 0$. In the following, this variant of nl-IDBD algorithm with eligibility traces will be denoted as nl-IDBD(λ).

Algorithm 4 Adjustment of the incremental TD(λ) algorithm for board games in Algorithm 1 when used in conjunction with nl-IDBD(λ). In the original algorithm, the lines 21 – 27 have to be replaced by the following pseudo-code:

```
1  for (every weight index i) do
2      if (q ≥ ε  or  s_{t+1} ∈ S_Final) then
3          β_{i,t+1} ← β_{i,t} + θ δ_t e_{i,t} h_{i,t}
4          α_{i,t+1} ← e^{β_{i,t+1}}
5          w_{i,t+1} ← w_{i,t} + α_{i,t+1} δ_t e_{i,t}
6          h_{i,t+1} ← [1 − α_{i,t+1} V_t(1 − V_t) x_i(s_t) e_{i,t}]^+ h_{i,t} + α_{i,t+1} δ_t e_{i,t}
7      end if
8      Δe_i ← x_i(s_{t+1})
9      e_{i,t+1} ← { Δe_i,              if x_i(s_{t+1}) ≠ 0 ∧ REP
                   { γλ e_{i,t} + Δe_i,  otherwise
10 end for
```

Note that in the derivation of the update rules (in Appendix A.3), we omit the gradient of the target signal, similarly as already done in Section 3.2.1, where the TD weight-update rule was derived. Formally, this is not correct for TDL tasks, since the target signal is not an independent teacher signal, but also based on the estimate V_t. For some small artificial problems this might cause divergence of the learning process [9]. However, in practice this typically does not lead to any problems.

3.5.3 Other Algorithms

Besides the already mentioned algorithms (Beal & Smith's TCL[14, 13], our extension TCL-EXP [8], Sutton's IDBD [58], Koop's nl-IDBD [60, 34]), in this study we will also use several other step-size adaptation algorithms: Sutton's K1 [61], Schraudolph's ELK1 [48], Mahmood's Autostep [39, 38], and Dabney's α-bounds [22]. Although only defined for batch learning tasks, also Riedmiller's RProp [42] algorithm will be tested for the Connect-4 task.

Stochastic Meta Descent (SMD) Additionally, we derived from Schraudolph's Stochastic Meta Descent (SMD) [48, 49] an algorithm – similar to Koop's nl-IDBD – that is based on the cross-entropy loss and that can also be used in conjunction with eligibility traces.

The SMD algorithm as described in [48, 49] can be seen as a general extension of Sutton's IDBD algorithm [58]. In contrast to IDBD, SMD does not approximate the instantaneous Hessian matrix (a matrix containing all second-order partial derivatives of the objective function) and is not limited to linear parametrized approximation functions. Furthermore, SMD can be used for any arbitrary scalar objective function (provided that the function is twice differentiable with respect to $\vec{w}$) [49], for example, the sample squared error or the cross-entropy loss. Once the parametrized approximation function and the objective function are defined, one has to compute the gradient and the Hessian of the objective function in order to retrieve the exact update rules. For this study we choose the same setup as for nl-IDBD(λ), described by Equations (3.41) – (3.44). In order to derive the update rules including eligibility traces, the target signal T_t is chosen to be the λ-return T_t^λ. For details, refer to Appendix A.4.

Classification of all Algorithms The algorithms can be classified according to different criteria: For example, several of the algorithms are based on meta-optimization approaches, hence, they optimize meta-parameters (step sizes) by minimizing some loss-function. Some algorithms are defined for linear units only while others employ a nonlinear activation function, adapt only a scalar step size or maintain vector-valued step sizes, can or cannot be used in conjunction with eligibility traces and more. All algorithms are summarized in Table 3.1.

Table 3.1: Summary of all step-size adaptation algorithms compared in this work. The column 'Activation' shows which activation function (tanh, logistic sigmoid or none) are used for our experiments. The column 'Memory' depicts the memory requirements of the individual algorithms. The requirements are at least n, since n weights w_i have to be maintained. Column 'Traces' indicates if the specified algorithm can be used in conjunction with eligibility traces. The fourth column indicates, which for which system the algorithm was derived: Least Mean Square (LMS), normalized Least Mean Square (NLMS) or based on the cross-entropy loss (CE).

Algorithm	Type	Activation	System	Memory	Meta-Param.	Traces	Reference
α-bounds	scalar	linear	–	n	–	$\surd$	[22]
Autostep	vector	linear	NLMS	$4n$	α_{init}, μ, τ	–	[39, 38]
IDBD	vector	linear	LMS	$3n$	β_{init}, θ	–	[58]
nl-IDBD	vector	log. sig.	CE	$3n$	β_{init}, θ	–	[60, 34]
nl-IDBD(λ)	vector	log. sig.	CE	$3n$	β_{init}, θ	–	A.3
K1	vector	linear	NLMS	$3n$	β_{init}, θ	–	[61]
ELK1	vector	tanh	NLMS	$3n$	β_{init}, θ	–	[48]
TCL	vector	tanh	–	$3n$	α_{init}	$\surd$	[14, 13]
TCL-EXP	vector	tanh	–	$3n$	α_{init}, β	$\surd$	[8]
SMD	vector	log. sig.	CE	$3n$	α_{init}, θ, μ	–	[48, 49]
SMD(λ)	vector	log. sig.	CE	$3n$	α_{init}, θ, μ	$\surd$	A.4
RProp	vector	tanh	LMS	$4n$	η^-, η^+,	–	[42]

Chapter 4

Experimental Setup

4.1 Software Framework

We established a software framework for the games Connect-4 and Dots-and-Boxes in Java, which allows to conduct and perform learning experiments, based on the techniques described in the previous chapter: TD-Learning with eligibility traces, n-tuple systems for the value function approximation and online-adaptable step-size adaptation algorithms. For Connect-4 a GUI is available, which provides visual inspections capabilities, evaluation tools, import/export functionalities, direct interaction between user and various agents (random, TDL, *Minimax*, Monte Carlo tree search) and more[1].
Although only a simple console-based environment can be provided for Dots-and-Boxes at this stage, the software already contains several powerful tools, that are especially useful for designing and executing experiments: The user simply defines the general setup of the experiments (e.g., a grid-search based on n parameters) and the software then creates all necessary parameter files (XML-based), executes the experiments (in parallel, using multithreading, if desired) and writes the results into CSV-files.

4.1.1 Minimax Agents for Evaluation Purposes

For both games, Connect-4 and Dots-and-Boxes, perfect playing *Minimax* agents are available (up to 4×4 boxes for Dots-and-Boxes) that will be used for evaluation purposes during the experiments. These agents are based on advanced tree-search algorithms and are supported by extensive databases to master the opening phase of the game. For Connect-4 we make use of Tromp's 8-ply opening-database[2] [7] and additionally we generated a Huffman-encoded database containing all positions with 12 pieces [67, pp. 42–44], the expected outcome and the exact distance to the end of the game, assuming perfect play of both opponents. The agent in Dots-and-Boxes has access to databases created by Wilson's Dots-and-Boxes retrograde analysis tool [73].

[1] The Java framework for Connect-4 is available as open source from GitHub and can be downloaded from: `http://github.com/MarkusThill/Connect-Four`.

[2] We extended Tromp's database by all 8-ply positions with an immediate threat for yellow, since all positions with immediate threats were missing.

The tree-search (*Minimax* algorithm) in Connect-4 is enhanced by alpha-beta pruning, move-ordering mechanisms [67, pp. 30–34], symmetries, two-stage transposition tables in order to identify identical positions created by permutations of a certain move sequence [67, pp. 38–39], Zobrist Hashing [74],[67, pp. 36–37], Enhanced Transposition Cutoffs [41] and more. In addition to the already mentioned techniques, the *Minimax* algorithm for Dots-and-Boxes employs several techniques discussed in [10].

As the main underlying data-structure for the board-representation of Connect-4 and Dots-and-Boxes we use so called bit-boards. In this representation, each board cell (in Connect-4) or edge (in Dots-and-Boxes) can be represented by one or two bits in a bit-field[3]. In total, 2×42 bit are required to represent a Connect-4 position (42 bit per player) and 2×20 bit are required for a 4×4 Dots-and-Boxes field (20 bit for the vertical and additional 20 bit for the horizontal edges). If a suitable arrangement of all board cells/edges in the bit-field is chosen, then many operations can be performed efficiently, since a modern CPU already provides many fast bit-wise instructions (AND, OR, XOR, bit- and byte-wise inversion, etc.). Frequently recurring operations on a board typically are: placing and removing stones/edges, mirroring and rotation, copying, identifying win/loss/draw situations and others. Many of these mentioned operations can be executed faster with bit-boards than with conventional data-structures such as arrays. For example, in Dots-and-Boxes we found a representation that allows to rotate or mirror a complete board with very few CPU-cycles. In Connect-4, the recognition of terminal states (4 pieces in a row) is a time-critical operation, for which we could find an efficient solution by using bit-boards [67, pp. 27–29]. Another advantage of bit-boards is the memory-efficiency, which is especially beneficial when many board positions have to be stored in transposition tables.
Overall, our *Minimax* agent for Connect-4 is one of the fastest perfect-playing agents available: the empty board can be analyzed in less than 4 minutes on a Pentium-4 computer, without the support of the opening-database and using only a 24 MB transposition table. In the typical setup – with the 12-ply opening-database and a 24 MB transposition table – *Minimax* can analyze each board position in fractions of a second.

4.2 Experimental Setup for Connect-4

All agents described in this thesis are trained solely by self-play, no external teacher is involved in the training process and no other information than the training samples are used. Each training sample consists out of a sequence of moves (starting from the empty board) and the outcome (final reward) of the match. The general procedure of all conducted experiments is as follows: In the beginning, the components of the TDL-algorithm (Algorithm 1), the n-tuple system (Section 3.3) and the (optional) step-size adaptation

[3]For Dots-and-Boxes also a data-structure containing the information about captured boxes is required. In most cases, however, it is sufficient to maintain a simple counter that expresses the current box-difference (boxes player A minus boxes player B).

algorithm have to be initialized:
The TDL-algorithm requires the three parameters ϵ, γ and λ. The parameter ϵ steers the exploratory behavior of the agent, which generates the training samples by following an ϵ-greedy policy: mostly the agent selects actions that are expected to maximize the long-term reward, but with a small probability ϵ random actions are performed, in order to explore the state space of the game. This exploration rate is set in all experiments (with only one exception) to the constant value $\epsilon = 0.1$, forcing the agent to perform 10% of all moves at random.
The discount factor γ is chosen to be $\gamma = 1.0$ throughout the thesis. In Chapter 5, eligibility traces are not used before Section 5.1.4 ($\lambda = 0$). When activating eligibility traces ($0 < \lambda \leq 1$), the options replacing and resetting traces can be combined in several ways. Throughout the rest of the thesis we use the following notation for these 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.

We experimented with several n-tuple architectures and found a system with 70 n-tuples – all of length 8 and generated by random walks on the board – to be a convenient choice. If not stated otherwise, the same set of 70×8-tuple is used in all experiments. All agents learn two value functions, one for yellow and one for red, so that two sets of LUTs are necessary (doubling the memory requirements). Furthermore, only the option $P = 4$ is considered in all experiments, resulting in 4 possible states for each n-tuple sample point. In total, for a system with 70×8-tuple, $2 \cdot 70 \cdot 4^8 \approx 6 \cdot 10^6$ weights are created. Each weight is then initialized with a random value uniformly drawn from $[-\frac{\xi}{2}, \frac{\xi}{2}]$, with $\xi = 0.001$.

In the plain TDL algorithm, the global learning rate α decays exponentially from $\alpha_{init} = 0.004$ to $\alpha_{final} = 0.002$. TCL instead keeps the global parameter α at a constant value, but each weight has its individual learning rate α_i. TCL-EXP requires an additional parameter β, which is chosen to be $\beta = 2.7$, if not stated otherwise. The α-bounds algorithm adjusts a non-deterministic scalar step-size parameter. All remaining step-size adaptation algorithms maintain an individual learning rate for each weight of the system and require a meta-step-size parameter θ. For those algorithms, which are only defined for linear units (IDBD, K1, Autostep), the squashing function (either $tanh$ or a logistic function) is omitted.

After initialization, the agent plays a large number of training games against itself – typically 2 million, in a few cases also 10 million games – and attempts to learn a state-value function. Each training is repeated 20 times, in order to get statistically sound results. For every run the random number generator is initialized with a new seed value (current system time in milliseconds).
Every 10 000 games, the current strength of the agent is measured and recorded by a suitable evaluation method. At the end of each training other measures such as time-to-learn and asymptotic success rate are determined. Details regarding the agent evaluation are discussed in the coming Section.

4.2.1 Agent Evaluation

A fair agent evaluation for board games is not trivial. There is no closed-form objective function to determine the agent's strength. The evaluation of all board positions is infeasible. Although it might be possible possible to evaluate a limited set of k randomly generated (legal) positions and express the agent's strength as the ratio of correct classifications (win, loss, draw), it is not clear, which relevance has to be assigned to each position: the accuracy of predictions for certain states (e.g., the initial board) are typically more important than for others. In our setup the agent aims to learn a perfect sequence of moves, starting from an empty board; due to the generalization process, the accuracy of irrelevant states – and sub-trees linked to them – will decrease during the training, in order to increase the prediction accuracy of relevant states. This is somewhat similar to the principle idea behind the alpha-beta algorithm, where irrelevant sub-trees are simply pruned. All these points make a fair evaluation rather difficult. One common evaluation method for assessing the strength of an agent is the direct interaction with other agents, which allows a direct comparison of the involved agents, but often still fails to provide an indication of the real strength which could be used as an objective, common reference point.

As the main benchmark for our evaluation process in Connect-4, we developed a perfect playing *Minimax* agent – based on sophisticated techniques such as alpha-beta pruning, transposition tables, and a 12-ply opening-database – in order to get comparable and replicable results. The evaluation-process is as follows: To estimate the strength of an agent, a tournament of 200 matches against *Minimax* is performed. Since *Minimax* would win every match as starting player (Yellow), we consider only games where *Minimax* moves second.
During the evaluation matches, both opponents will usually act fully deterministically, which would result in exactly the same sequence of moves for every game. To overcome this problem, we introduce a certain degree of randomness: 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. This approach increases the level of difficulty for the evaluated agent, which has to prove that it can play perfectly over a longer sequence.[4]
For each win against *Minimax* an agent receives a score of 1.0, for a draw 0.5 and for a loss 0. The overall score (success rate) is determined by averaging all 200 results. An ideal agent is therefore expected to reach a total score of 1.0 against *Minimax*.
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 based on self-play of the individual agent.

[4]We observed, that a game that ends with a defeat of *Minimax* lasts in average 34 moves.

For later analysis of the experimental results, we define two criteria to assess the strength of an agent: (a) The *asymptotic success rate* which is calculated as the average success during the last 200 000 or 500 000 training games of a training run and (b) the *time-to-learn*, in our case, the number of training games needed to cross the 80% or 90% success rate for the first time.

4.2.2 Representation and Visualization of the Training Results

In Chapter 5 most results will be provided as visual representations, such as box-plots, sensitivity plots, scatter-plots and others. In this section we shortly mention how the different types of figures are generated, in order to avoid multiple descriptions in the captions of the figures.

Typically, the results will be compared in a standard X-Y line chart, where the success rate of the agent is plotted against the number of training games. Each point in the graph is the mean of all 20 runs and the error bars attached to each point indicate the standard deviation within 20 runs. Although the agent strength is measured every 10 000 games we only display a subset of all points to avoid visual clutter in the plot. Yet, the lines are a fit to the noisy observations based on *all* measured points. For this reason, the points do not necessarily lie on the corresponding curves. Note, that in some cases the visual crossing of the 80% or 90% line (which we defined as the time-to-learn) may be somewhat different to the exact values given in the text or in the tables.

After smoothing the curves for each run in order to dampen fluctuations, we get the time-to-learn value for every run by determining the number of games needed for the smoothed curve to cross the 80% or 90% success rate. We compare the learning speed of selected agents usually in a box-plot. For every box-plot it is specified, if the 80% or 90% line is used to measure the time-to-learn. If not stated otherwise, the box-plots also show the results for 20 repetitions each. The median of 20 runs is then used to describe the overall learning speed of an agent.
The asymptotic success rate of a run is determined by averaging the measured points of the last 200 000 or 500 000 training games (we specify for every box-plot how many points are used). The overall time-to-learn and asymptotic success rate of an agent is the median of 20 runs.

4.3 Experimental Setup for Dots-and-Boxes

The general setup for Dots-and-Boxes will be similar to Connect-4. However, there are a few differences that should be noted:

- The exploration rate is chosen slightly higher for most experiments, typically $\epsilon = 0.2 = const.$ if not stated otherwise.
- The number and the length of the n-tuples for the experiments will be described in the corresponding sections.
- The number of possible states per sampling point is always $P = 2$ (empty or set edge).
- As discussed earlier (Section 3.4.3), the TDL agent does not receive rewards at the end of each training game. Instead, intermediate rewards are provided for every captured box.
- For Dots-and-Boxes also a perfect playing opponent is available for the evaluation process (based on Wilson's solver [73] and a tree-search). Similarly to Connect-4, this agent also randomly selects an action from a set of optimal actions. In order to test the correctness of our tree-search algorithm, we compared the game-theoretic values for around 20 000 randomly generated positions with the corresponding values of Wilson's solver.
- The outcome of the game, assuming perfect play of both opponents, depends on the board size. Therefore, we decided to – independently from the board size – to constantly swap the sides after each tournament match. For example, if 100 evaluation games are chosen, the TDL agent will actually play 100 matches, 50 matches as player A and 50 matches as player B. The provided scores are as follows: The TDL agent receives a score of $+1$ for a win, -1 for a loss and 0 for a tie. Independently from the board size and the expected outcome of the match, the optimal score is then always 0, indicating perfect play. The lowest possible score of -1 indicates that the TDL agent lost all tournament matches against it's perfect playing opponent.
- Similarly to Connect-4, also the 'time-to-learn' will be measured for the Dots-and-Boxes task. If not stated otherwise, this will be the number of training games needed to cross the score of '-0.2' for the first time.

Chapter 5

Results and Analysis

5.1 Results for Connect-4

5.1.1 Starting Point

In our earlier work [69] we showed that TDL with n-tuple systems produces strong agents. Although the agent reached asymptotic success rates of approx. 90% against *Minimax*, in total around 1 565 000 training games were required to cross the success rate of 80%. We made the observation that lower exploration rates can significantly improve the learning speed of the agent. After performing a more systematic parameter tuning (latin hypercube sampling), we found that the tuned TDL-agent reaches the 80%-success-rate after 670 000 games (Figure 5.1), which is faster by more than a factor of 2. This tuning result emerged as the optimal result from testing about 60 different parameter configurations. As already mentioned, the key difference to the former TDL result is a suitably reduced exploration rate ϵ. In the following, we only consider the result of tuned TDL for the comparison with other algorithms, which can be seen as the starting point of this work. In the remaining sections of this thesis we want to investigate the benefits of online adaptable learning rates and eligibility traces added to the plain TDL system, guided by the research questions formulated in Section 2.

5.1.2 Temporal Coherence Learning

We started our experiments by performing runs for the standard TCL algorithm without eligibility traces ($\lambda = 0$). We systematically tested many parameter configurations, with the main goal to find an appropriate global step-size parameter α. However, we could not find a setting that was able to produce comparable or better results than the plain TDL algorithm. This was somewhat surprising, since Beal & Smith stated, that TCL would automatically self-adjust the parameters: *"The parameter α can be set high initially, and the α_i then provide automatic adjustment during the learning process."* [13]. We found that TCL is as sensitive towards the global step size α as TDL, the best settings are only shifted towards slightly larger values. We will discuss the sensitivity of both algorithms and several others on the step size in more detail in Section 5.1.3.

The results for the best found setting of the original TCL algorithm (labeled as TCL[r])

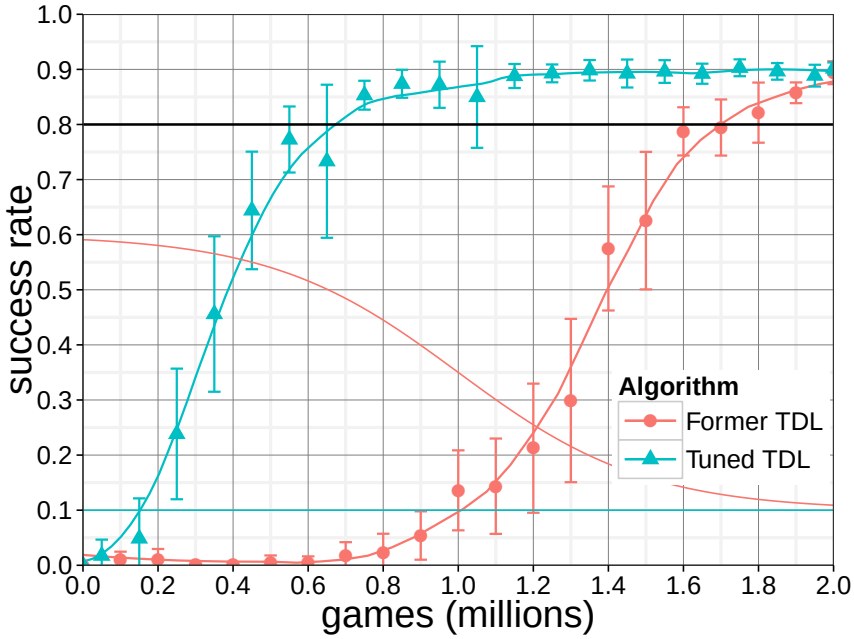

Figure 5.1: Starting point: Tuning of the exploration rate. Formerly, we used a sigmoid function to adjust the exploration rate from initially $\epsilon_{init} = 0.6$ to a final value of $\epsilon_{final} = 0.1$. In our experiments, we found lower exploration rates to be more beneficial. Tuned TDL therefore keeps the exploration rate at the constant value $\epsilon_{init} = \epsilon_{final} = 0.1$. The exploration rates are indicated by the lines without points. In the following, we will only use tuned TDL for comparisons.

in comparison to TDL is shown in Figure 5.2. In the further course of this work we label the original TCL algorithm as TCL[r], indicating that the recommended weight change $r_{i,t} = \delta_t e_{i,t}$ is used to adjust the counters A_i and N_i in Algorithm 2. We also tested a slightly different variant of TCL, namely TCL[δ], where we only use the TD error signal δ_t to adjust A_i and N_i. In Algorithm 2, $r_{i,t}$ is therefore simply replaced by δ_t in lines 7 and 8. TCL[δ] improves the result slightly but not significantly. The best setting for TCL[δ] is shown in Figure 5.2.

Inspired by the geometric step-size property in Sutton's IDBD [58] algorithm, which allows adjustments in α_i by a fixed fraction of its current value for fixed changes $\Delta\beta_i$ in the memory parameter β_i, we also experimented with a similar exponential scheme in TCL. This led to the new TCL variant, described in Section 3.5.1, which uses a modified exponential transfer function (Figure 3.7). This new algorithm is labeled TCL-EXP. The exponential scheme brings a remarkable increase in speed of learning for the game Connect-4, as our results in Figure 5.2 demonstrate. TCL-EXP reaches the 80% success rate after about 385 000 games instead of 560 000 (Tuned TDL). At the same time it also reaches asymptotically a very good success rate. The variant TCL-PL, with a piecewise linear

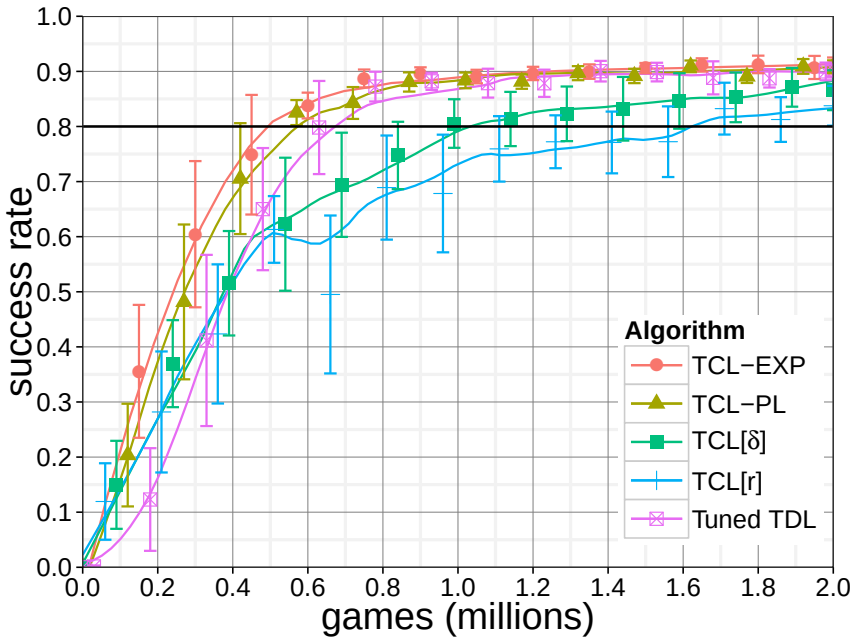

Figure 5.2: Results for the individual TCL variants described in Section 3.5.1. The original TCL algorithm TCL[r] and the slightly modified variant TCL[δ] perform significantly worse than TDL. Inspired by the geometric step-size property in Sutton's IDBD [58] algorithm, we also experimented with a similar exponential scheme in TCL, which led to the TCL-EXP algorithm. TCL-EXP brings a remarkable increase of the learning speed for the game Connect-4. TCL-PL uses a piecewise linear transfer function, that has the same slope as TCL-EXP in the endpoints $x = 0$ and $x = 1$. Note, that the visual crossing of the 80% line may be somewhat different from the exact values given in the text and in Table 5.1.

transfer function (introduced in Section 3.5.1 and illustrated in Figure 3.7) produces similar – but slightly worse – results. Therefore, it is not the slope of the piecewise linear function, but the geometric step size which is important for success.
However, TCL-EXP introduces a new parameter β that may require some additional tuning. We varied the parameter β systematically between 1 and 7 and found values in the range $\beta = [2, 3]$ to be optimal.

5.1.3 Survey of Step-Size Adaptation Algorithms

Meta-learning algorithms methods attempt to improve the learning-algorithm by modifying certain meta-parameters based on the experience made. In this section we especially want to focus on various step-size adaptation algorithms and investigate in which ways these algorithms can improve the actual learning process. In the context of temporal difference learning and board games there are several measures that could be used to assess the performance of a step-size adaptation algorithm: For instance, one could measure the improvement in the learning speed or in the final strength of the agent, once suitable

meta-learning parameters are found. Depending on the meta-learning algorithm, the tuning process for the meta-learning parameters may require more or less effort. Therefore, one important quality measure is the sensitivity of an algorithm towards its meta-learning parameters. The sensitivity can be understood as a measure that describes how changes in a meta-learning parameter impact another performance indicator, such as 'time to learn' or 'asymptotic success rate'.

In the following, we will compare plain TDL to, in total, 8 different step-size adaptation-algorithms: Beal & Smith's TCL[r] [14, 13], TCL-EXP, Sutton's IDBD [58], Koop's nl-IDBD [60, 34], Sutton's K1 [61], Schraudolph's ELK1 [47], Mahmood's Autostep [39, 38], and Dabney's α-bounds [22]. All algorithms are implemented exactly as described in their original papers (if not stated otherwise) and then applied to our Connect-4 task. Some algorithms (IDBD, K1, Autostep) are only derived for linear units. We omit in this case the nonlinear sigmoid function $\sigma = \tanh$ for the TDL value function. For all other algorithms we use this sigmoid function. An exception is Koop's nl-IDBD being derived for the logistic sigmoid function, which we use in this case instead of tanh.
Dabney and Barto's α-bounds algorithm [22] performs poorly in its original linear version. Out of 20 runs 6 never pass the 80%-line and 'time-to-learn' has a median of 1.9 million games, not shown in the figures). A purely linear TDL would perform similarly. We extended α-bounds to the nonlinear case by making a linear expansion of the sigmoid function and deriving a slightly modified rule

$$\alpha_t = min\left(\alpha_{t-1}, |\sigma'(f)\,\vec{e}_t \cdot (\gamma\vec{x}_{t+1} - \vec{x}_t|^{-1})\right) \tag{5.1}$$

which has an additional $\sigma'(f)$ as compared to Eq. (15) in [22]. As Dabney and Barto already suggested in their conclusion [22], an extension of the α-bounds algorithm from the current version with just a scalar learning rate to a vector of individual learning rates for each weight might increase the performance of the algorithm.

Sensitivity Analysis

All adaptive learning rate algorithms have a parameter α_{init} (or equivalently $\beta_{init} = \ln(\alpha_{init})$) and some of them have a meta step-size parameter θ. For each algorithm we estimated the best pair (α_{init}, θ) by grid search. In the case of TDL and TCL, $\alpha = \alpha_{init}$ is kept at this constant value. Then we assessed the sensitivity by fixing one parameter at its best value and varying the other over a broad range. The results for the meta-learning parameter α_{init} are displayed in Figure 5.3. In order to avoid overplotting, the results are divided into several plots. The left column of the grid shows how 'time to learn' is affected by different values of α_{init}, whereas the right column shows the impact on the asymptotic success rate. It can be seen, that the algorithms TDL and TCL[r] only have a very narrow region of α_{init} values that produce good results, for both measures, time to learn and asymptotic success rate. For TCL-EXP and nl-IDBD only a slightly larger range in α_{init} leads to good results, but

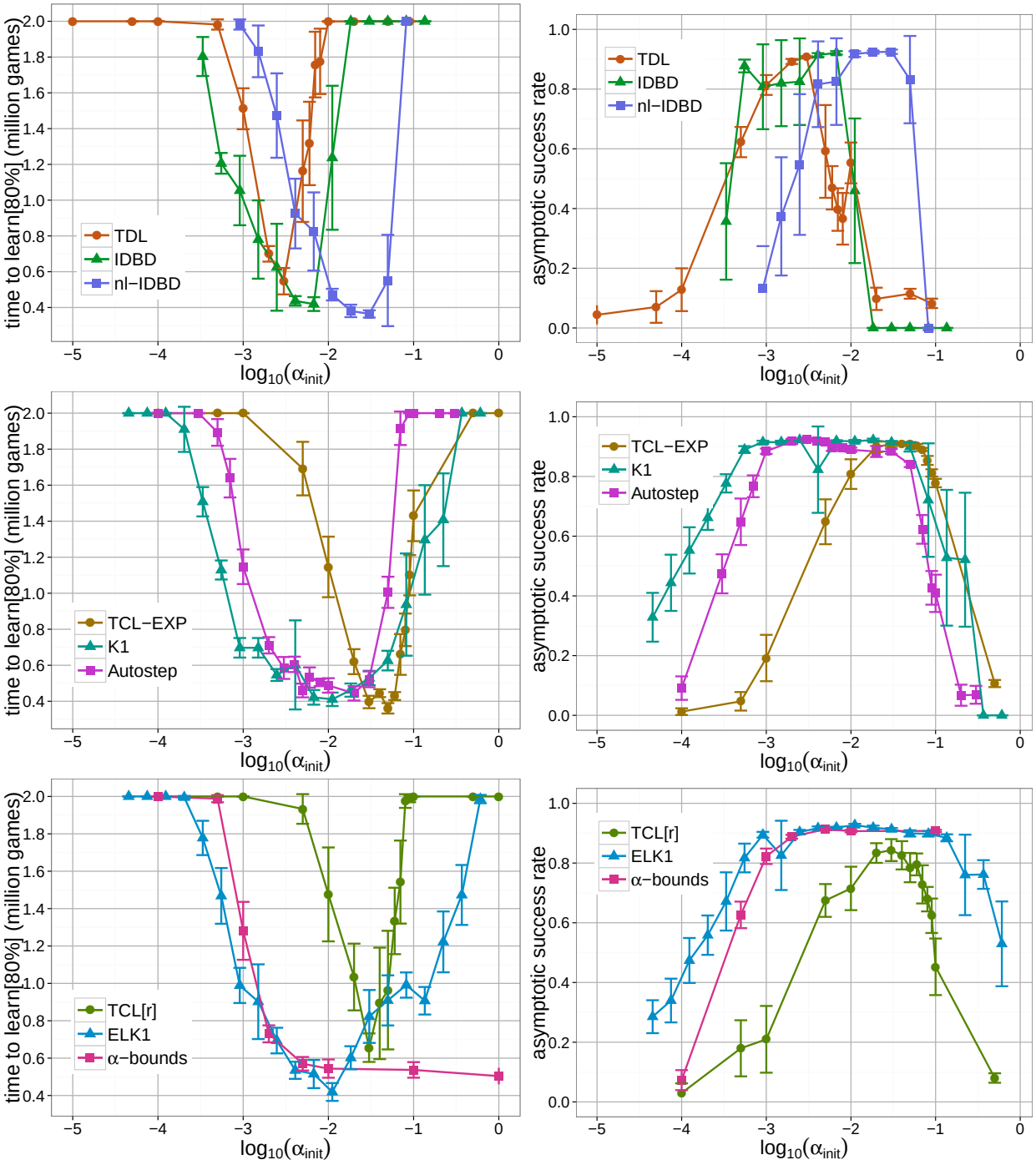

Figure 5.3: Sensitivity on the initial learning rate α_{init} for all algorithms. For algorithms specifying β_{init} (like IDBD), we use the transformation $\alpha_{init} = e^{\beta_{init}}$. Parameter α_{init} is screened over a large range. Each point is the *mean* of 10 runs with 2 million training games. In the case of TDL and TCL, $\alpha = \alpha_{init}$ is kept at this constant value. The left column shows the 'time to learn' and the right column the asymptotic success rate of the different algorithms. If a run never crosses the 80% success rate line, 'time to learn' is set to 2 million. The asymptotic success rate of a single run is the mean success rate of the last 200 000 games. The error-bars indicate the standard-deviation within 10 runs.

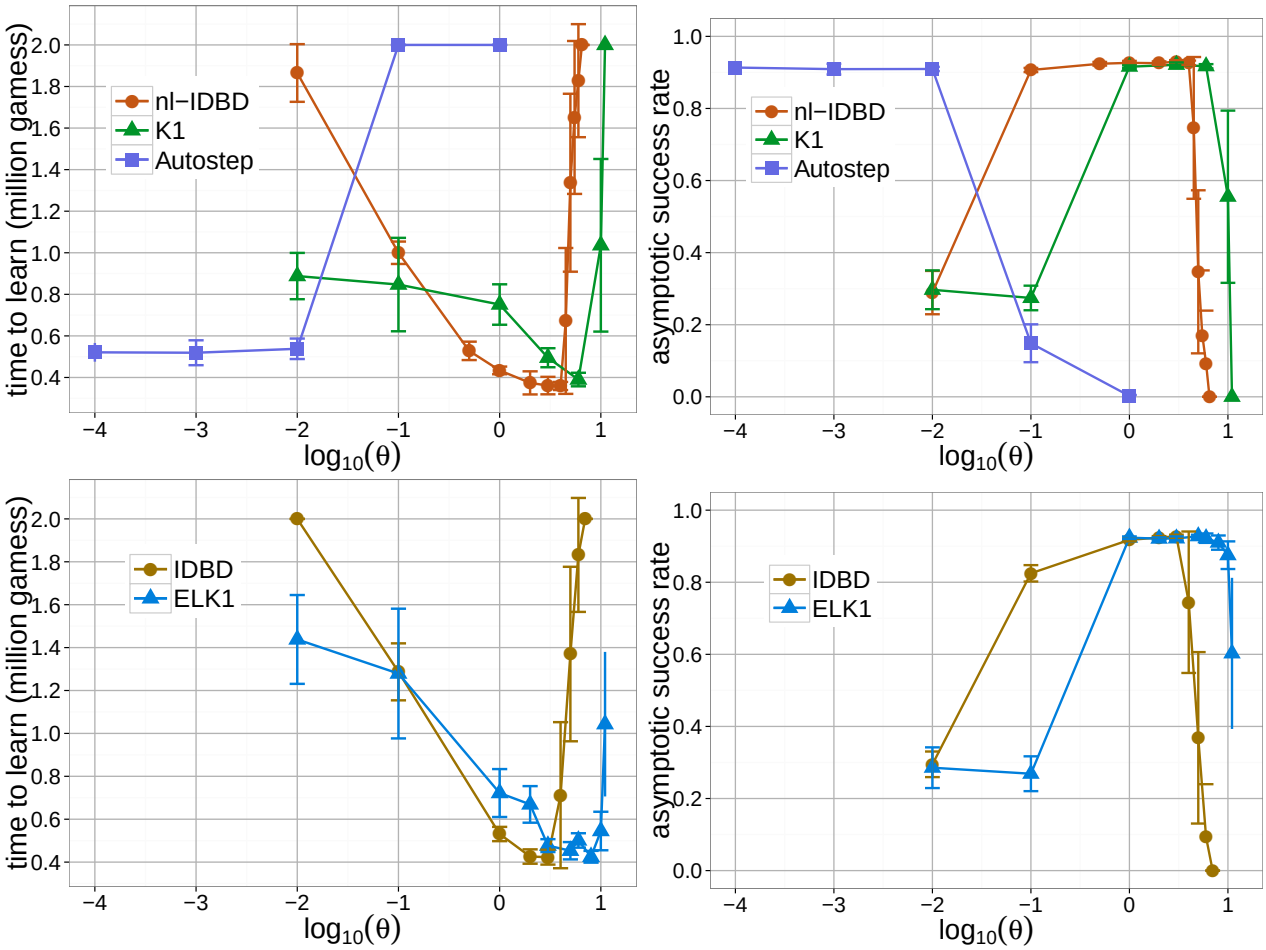

Figure 5.4: Sensitivity on the meta-learning rate θ for all relevant algorithms having this parameter (μ in the case of Autostep). As before, each point is the mean of 10 runs with 2 million training games. The left column of the grid shows the 'time to learn' and the right column the asymptotic success rate of the different algorithms. If a run never crosses the 80% success rate line, 'time to learn' is set to 2 million. The asymptotic success rate of a single run is the mean success rate of the last 200 000 games.

yet both algorithms – as the only two – learn Connect-4 in less than 400 000 training games for their best settings. Autostep and K1 work in larger regions and α-bounds is very stable for values larger than $\alpha_{init} > 10^{-1}$, although it can never learn in less than 500 000 games. From all algorithms, ELK1 has the largest range, in which the asymptotic success-rate exceeds 80%. However, none of the surveyed algorithms is robust enough to perform well over the complete inspected value range. Figure 5.4 shows the sensitivity on the meta-learning rate θ (μ in the case of Autostep) for all algorithms that require this parameter. We could observe, that Autostep performs remarkably well in a broad range of small θ values, even for smaller values than shown in this graph; we inspected values until $\theta = 10^{-7}$ and could

find nearly the same results as for the range $[10^{-4}, 10^{-2}]$. All algorithms have in common that for movements into the region of larger values of θ, the learning performance suddenly breaks down. However, the best results regarding the speed of learning are achieved with values very close to the breakdown, as seen clearly for the K1 algorithm.
Also for the second meta-learning parameter θ, all algorithms show some degree of sensitivity, whereby Autostep is effectively tuning-free in this respect, since all inspected values smaller than $\theta = 10^{-2}$ produce good results.

Final Comparison

For several other step-size adaptation algorithms we performed some initial tests, but we could not explore them in detail due to reasons of time. We used RProp to learn the opening phase (after 8 plys[1] a reward was provided by the database) of Connect-4. However, RProp performed worse than plain TDL in terms of learning speed, although RProp could eventually learn the opening phase perfectly. The training of the whole game failed completely, the agent augmented with RProp (we tested the variants Rprop$^+$, RProp$^-$, iRProp$^+$ and iRProp$^-$) could not reach a stable strength; although success rates of 60% could be measured in between, the value almost always fell back to 0% again at the next measurement. We could observe that most of the learning rates approached their lower bound (10^{-5}) after a short time and did not recover from this low value. Most likely, the error function $E_t = \delta_t^2$ is too noisy for this task. This noise is mainly induced by the exploration component (given by ϵ) and some other factors. As a consequence, the signs of a gradient-element constantly switch and a sequence of gradient-signs carries no useful information in order to adjust the step-size parameters adequately. Instead, the switching signs simply lead to decreasing step sizes, since the algorithm "believes" that the previous steps have been to large. It appears that RProp is not suitable for stochastic gradient descent algorithms such as TD-learning and requires the gradient information from a larger training-set. One solution to the above problem could be to define epochs of a certain length (e.g., one complete match or even several matches) and sum up the gradient information during the epoch. The disadvantage of such an approach could be that, unlike an online learning approach, the agent cannot adjust its policy during an epoch and might suffer from slower learning progress.

We also performed some initial tests with Schraudolph's SMD (Stochastic Meta-Descent) algorithm. In contrast to Sutton's IDBD or Koop's nl-IDBD algorithm, SMD does not diagonalize the instantaneous Hessian matrix (for details, refer to Appendix A.4). For this reason, SMD requires some additional computation time in order to determine the Hessian, although the computation time scales linearly with the number of inputs (since the Hessian is symmetric) and can be significantly reduced for systems with sparse activation, such as n-tuple systems. The additional computation time therefore only increases moderately for

[1] A ply is a single move by either of the players.

our Connect-4 learning task. In this work we choose the setup of the SMD algorithm to be similar to Koop's nl-IDBD: SMD attempts to minimize the cross-entropy loss instead of the MSE. Furthermore, we choose the same logistic sigmoid function in the output of the network and use the same meta-learning rate θ and initial step size α_{init} as for nl-IDBD. We found that SMD can increase the learning speed slightly in comparison to nl-IDBD; the asymptotic success rate is slightly lower (Figures 5.5–5.7). It could be interesting to evaluate the sensitivity towards the meta-learning rate and initial step size and compare the results to nl-IDBD, which we could not do until now due to time restrictions.

In the Figures 5.5–5.7 we compare the different step-size adaptation algorithms, by providing different views on the results. For each algorithm the best found setting is displayed. The general settings can be found in Table 5.1. Figure 5.5 shows the box-plots for the measures 'time to learn' and 'asymptotic success rate'. It can be seen that SMD is the fastest learning algorithm. The learning speed could be increased from formerly 1 565 000 training games for our initial results with plain TDL to now 320 000 for SMD, which is nearly a factor of 5 faster. Koop's nl-IDBD (345 000 training games) can improve the former result by a factor of about 4.5. TCL-EXP, the third-fastest algorithm with 385 000 training games, improves the learning speed by a factor of 4. From all step-size adaptation algorithms, the original TCL algorithm (TCL[r]) has the worst performance considering the learning speed as well as the asymptotic success rate and also the modified TCL[δ] produces only slightly better results. Surprisingly, both TCL variants are even inferior to TDL, which only uses one scalar step-size parameter (exponentially decaying). Since TCL-EXP only differs from the original TCL[r] algorithm by an exponential transfer function instead of a linear one, geometric step sizes appear to be an important factor for faster learning. In fact, all remaining step-size adaptation algorithms with geometric step-sizes have a higher learning speed than those algorithms which do not.
Figure 5.6 shows the learning progress of all algorithms over time, until 2 million training games. Note that each curve is a smoothed fit through the average of 20 runs. Due to this, the visual crossing of the 80% line may differ somewhat from the exact values given in Table 5.1.
In Figure 5.7 the two measures, 'time-to-learn' and 'asymptotic success rate' for the algorithms are plotted against each other, allowing a direct comparison based on both metrics. We can see that SMD ranks first for 'time-to-learn' and nl-IDBD ranks first for 'asymptotic success rate'. Although TCL-EXP is the third fastest algorithm, it has a lower asymptotic success rate after 2 million games than ELK1, but a slightly larger one than the SMD algorithm.

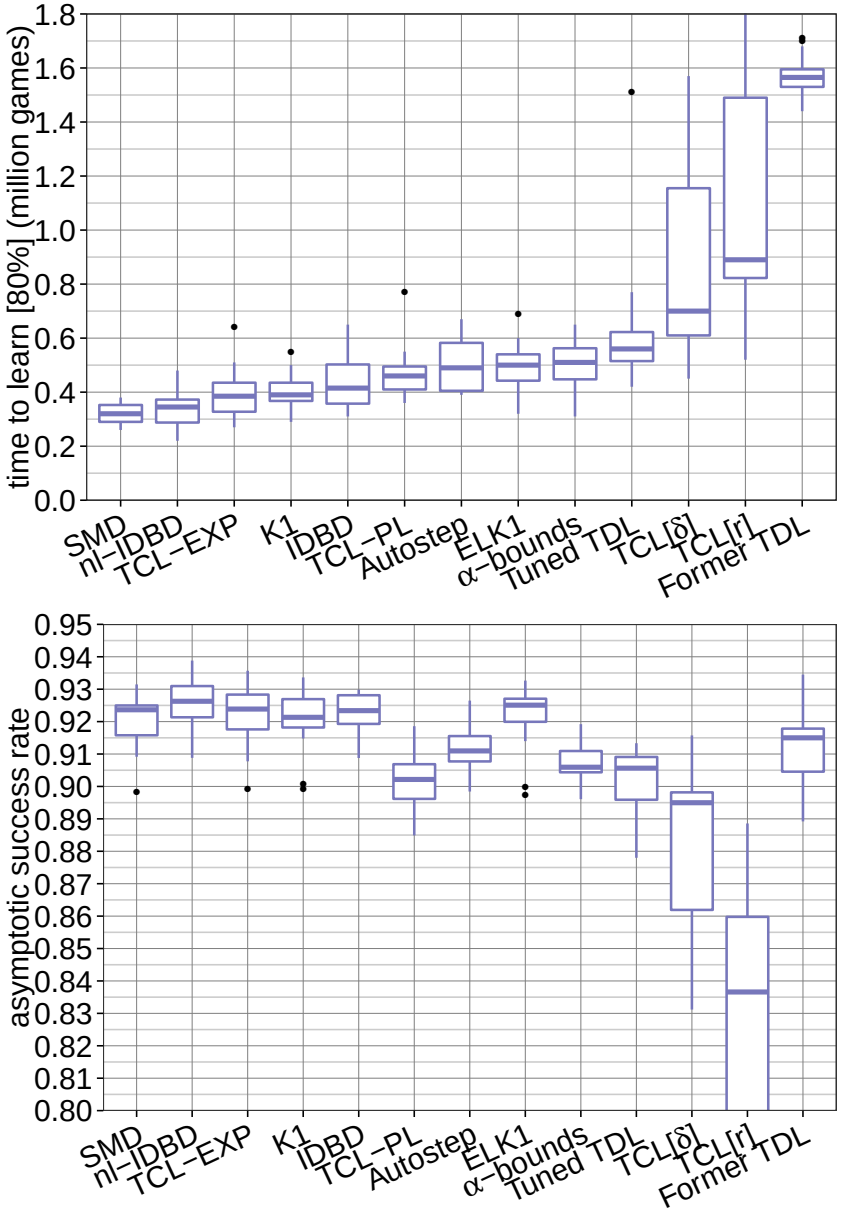

Figure 5.5: Learning speed and asymptotic success rate after 2 million games for the best settings of the individual algorithms. In both box plots the results of 20 runs each are shown, sorted in ascending order according to the 'time to learn'. The asymptotic success rate of a run is determined by averaging the measured points of the last 200 000 training games (20 values). The asymptotic success rate for TCL[r], the original TCL algorithm [14, 13], is not completely shown in the graph, in order to have a better view of the remaining algorithms. Note that the box plot for the asymptotic success rate has one outlier for tuned TDL at 0.79 and three outliers for TCL[δ] at 0.61, 0.70 and 0.74. The exact values can be found in Table 5.1, which also contains the asymptotic success rate after 10 million training games.

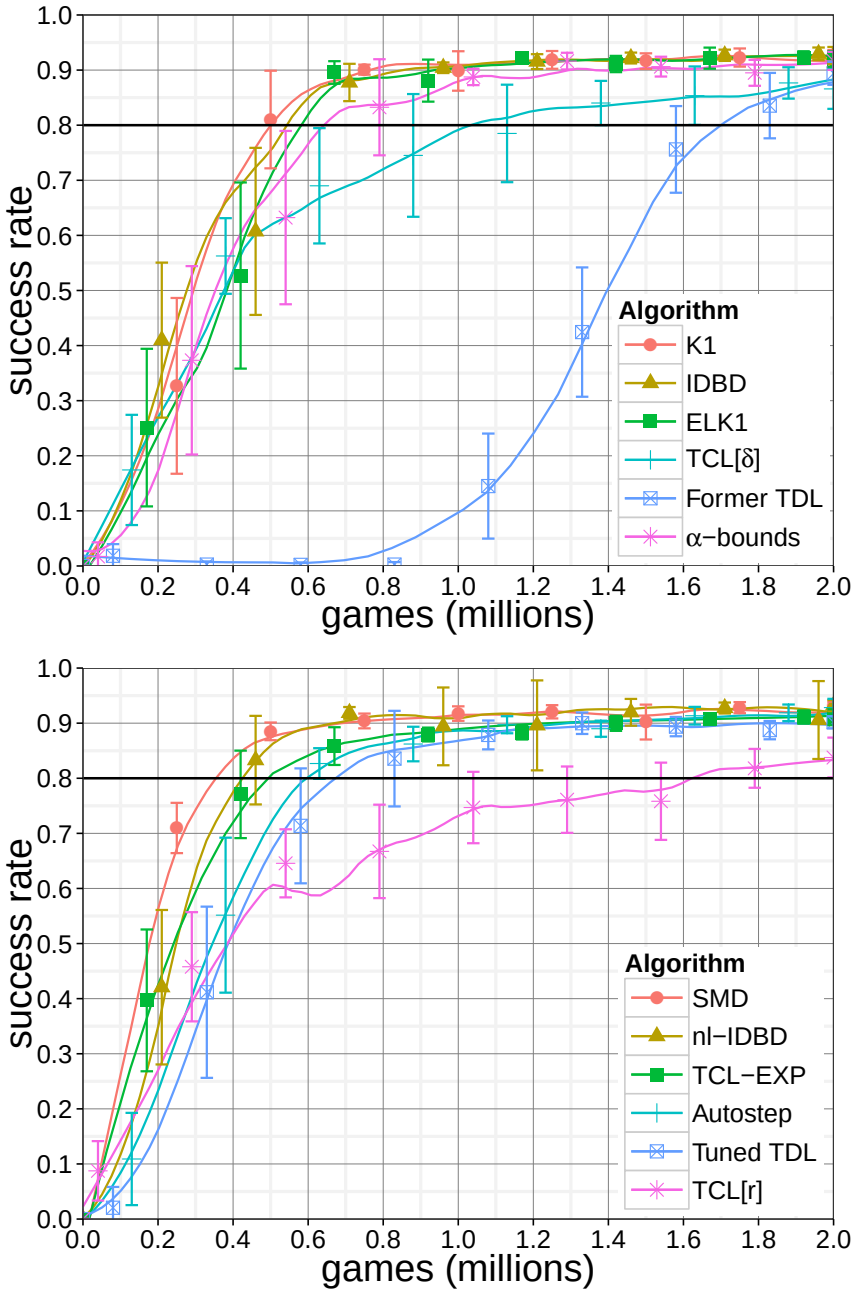

Figure 5.6: Final comparison of different step-size adaptation algorithms for Connect-4. Every setting is repeated 20 times with different random initialization and random exploration. The exact values are given in Table 5.1.

Table 5.1: Summary of the parameter settings, training times and asymptotic success rates for the different step-size adaptation algorithms. The settings shown in this table represent the best found (with exception of Former TDL, that follows a suboptimal exploration strategy) for all algorithms, shown in Figures 5.5–5.7. Column *TCL* denotes, whether we used for TCL the recommended-weight update [r] or the δ-update [δ]. For all algorithms specifying β_{init} the relation $\alpha_{init} = e^{\beta_{init}}$ holds. Therefore, we only list α_{init} in this table. All listed algorithms use a constant exploration rate $\epsilon = 0.1$, except Former TDL, which has an exploration rate decaying from initially $\epsilon_{init} = 0.6$ to $\epsilon_{final} = 0.1$ following a sigmoid function (illustrated in Figure 5.1). Every setting is repeated 20 times with different random initialization and random exploration. The agent strength was measured every 10 000 games. The median of 'time-to-learn' (Figure 5.5) is given in the last but one column. The last column depicts the computation time in minutes (including evaluation every 10 000 games). Time is measured on a standard PC (single core of an Intel Core i7-3632QM, 2.20 GHz, 8 GB RAM). The columns seven and eight show the asymptotic success rate ASR (median of 20 runs), after 2 and 10 million games, respectively.

						ASR		time to learn [80%]	
						2m	10m	80% line	
Algorithm	α_{init}	α_{final}	β	θ	TCL	[%]	[%]	[games]	[minutes]
Former TDL	0.004	0.002	–	–	–	89.7	91.6	1 565 000	102.0
Tuned TDL	0.004	0.002	–	–	–	90.7	92.5	560 000	39.7
TCL[δ]	0.04	–	–	–	[δ]	89.4	91.4	670 000	50.3
TCL[r]	0.04	–	–	–	[r]	83.9	90.4	890 000	67.2
TCL-EXP	0.05	–	2.7	–	[r]	92.5	93.1	385 000	30.1
IDBD	0.0067	–	–	3.0	–	92.5	92.7	415 000	33.1
nl-IDBD	0.0302	–	–	3.1	–	92.8	93.5	350 000	27.7
SMD	0.0302	–	–	3.1	–	92.4	N.A.	320 000	30.7
Autostep	0.005	–	–	10^{-4}	–	91.5	92.6	490 000	38.7
K1	0.009	–	–	6.0	–	92.0	92.7	390 000	30.8
ELK1	0.0111	–	–	5.0	–	92.5	92.8	500 000	39.5
α-bounds	1.0	–	–	–	–	90.8	92.4	560 000	39.2

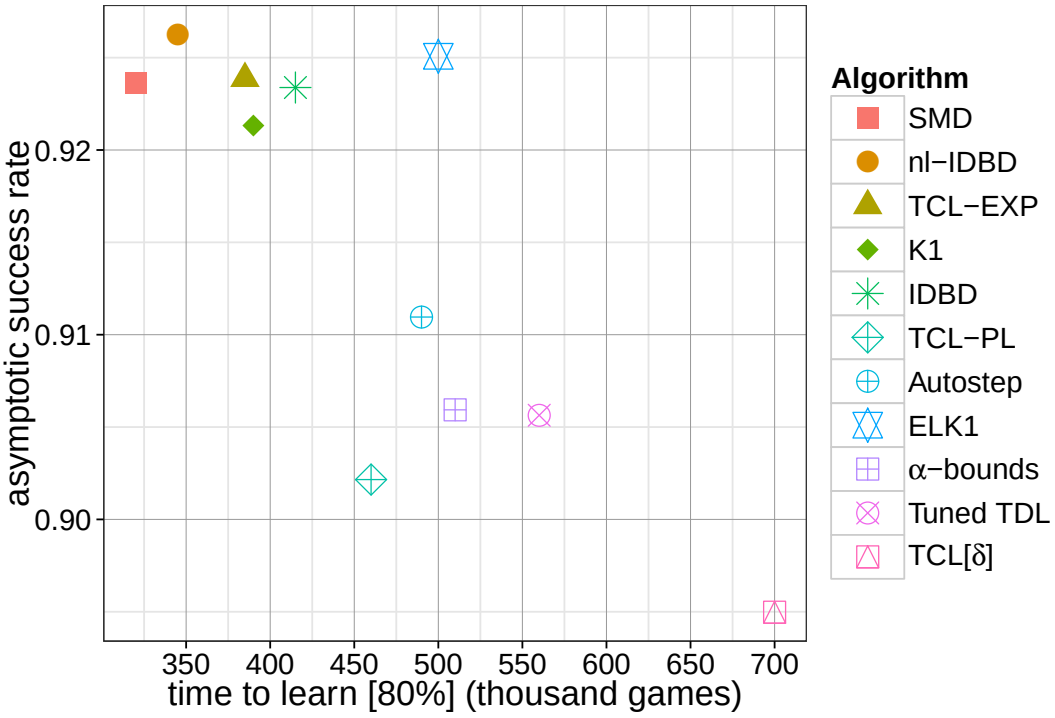

Figure 5.7: Final comparison of the metrics 'time to learn' and 'asymptotic success rate' (after 2 million games) for different step-size adaptation algorithms. For every algorithm the best setting is shown. In order to get a better view of the results, we removed the algorithms 'Former TDL' and TCL[r] from this plot. All results are generated without eligibility traces. Details regarding the experiments can be found in Table 5.1.

5.1.4 TD(λ) and Eligibility Traces

Although n-tuple systems have millions of weights, the degree of activation is rather small. For a system with 70 tuples of length 8, as used for most of our experiments in Connect-4, in every time step only 140 weights (70 n-tuple states and the mirrored equivalents) are activated. During one episode (training game with max. 42 moves), in total not more than 0.06% of all weights of the system will be trained[2]; the actual number will be even smaller, since not every move activates a new state in each n-tuple. Not surprisingly, learning required several million games initially. As discussed in the last sections, with a more systematic tuning and online adaptable learning rates, the training time could be decreased to less than 500 000 games. The main motivation for us to implement eligibility traces was to increase the degree of activation in the system in the hope of achieving even faster learning and a slight increase in the final strength of the Connect-4 agent.
In our previous work we did not yet consider eligibility traces, since a naive implementation would require millions of additional trace parameters and would significantly increase memory consumption and computation time. We realized later that it is possible to exploit the sparseness in the system's activation and to implement the eligibility trace vector using a suitable sparse representation. We decided to use a self-balanced binary tree, in which the individual traces e_i are inserted and sorted according to their index i. Most operations on the tree – such as insertion and searching – can be realized in logarithmic time $O\big(\log(n)\big)$, with a small overhead caused by the re-balancing transformations of the tree. The tree can be traversed in linear time.
We measured for our standard setting (70 n-tuples of length 8) that on average only around 1 800 (3 600 in the case of TCL-M – a variant of TCL-EXP with 150 8-tuples instead of 70) traces are active during one game (episode), for a system without resetting and replacing traces. Therefore, a binary tree reduces the memory requirements by a factor of 5 000, since it maintains on average only 1 800 (3 600 for TCL-M) traces instead of around 9 million (20 million) addressable traces, which would be required in a primitive implementation.

Initial Results for TCL-EXP

For our first experiments we trained a TCL-EXP agent using the basic variant of eligibility traces (no resetting and replacing traces). We varied the trace decay parameter λ systematically in the range of 0 to 1 in steps of 0.1. The results for several values of λ are illustrated in Figure 5.8. For $\lambda \in [0.0, 0.4]$ we could observe a slight increase in the learning speed and we found values $\lambda \in [0.4, 0.7]$ to be optimal. It can be clearly seen that eligibility traces significantly increase the learning speed: the number of required training games for the first crossing of the 80% line reduces from formerly 385 000 (TCL-EXP) to now 230 000 (TCL-EXP[et], $\lambda = 0.5$). Also a slight increase of around 1% in the asymptotic success rate

[2] As described in Section 3.4.1, in maximum 5 880 weights will be activated during one episode. The n-tuple system consists of in total 4^8 (4 possible states per board cell and an n-tuple length of 8) times 70 (number of n-tuples) times 2 (one LUT for each player) = 9 175 040 weights.

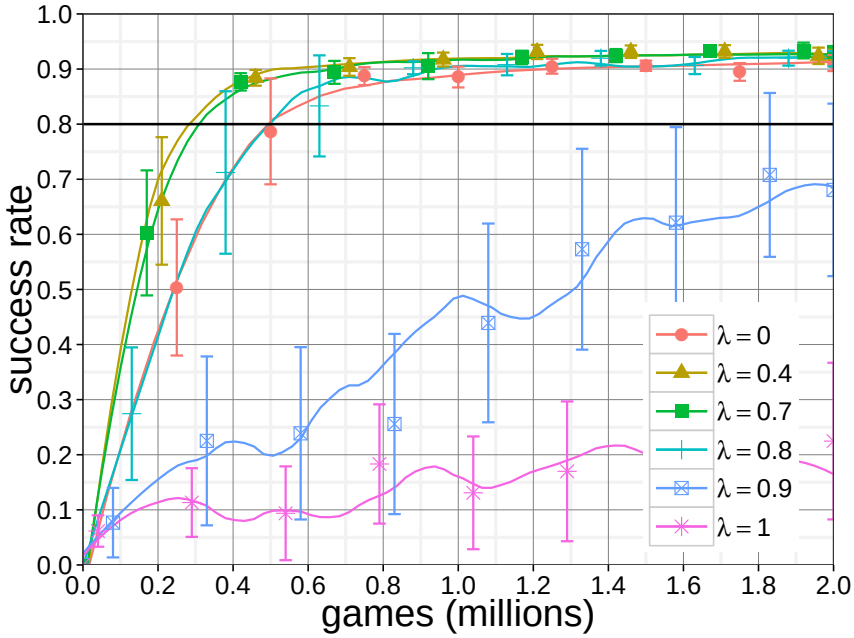

Figure 5.8: Initial results for TCL-EXP with eligibility traces, the options replacing traces and resetting traces turned off (TCL-EXP[et]). We varied the trace decay parameter λ in steps of 0.1 over the whole range from 0 to 1 and found values $\lambda \in [0.4, 0.7]$ to be optimal. For values $\lambda \geq 0.9$ the system breaks down.

can be observed. For values of $\lambda \geq 0.9$ the system however breaks down completely. The reason for this most likely is that exploratory moves create non-optimal move sequences so that wrong rewards are projected too far back to earlier states. We confirmed this by performing an experiment with TCL-EXP[res] and $\lambda = 0.9$: when the trace vector is reset after a random move, the result is similar to the curve for TCL-EXP[et] with $\lambda = 0.8$.
In Figure 5.9, we compare several agents that are trained with and without the eligibility trace option [et] (resetting and replacing traces turned off). Also for a standard TDL agent, eligibility traces are found to be beneficial: time-to-learn decreases by 200 000 games and the asymptotic success rate increases by almost 2%.
Furthermore, we investigated the effect of eligibility traces on agents with larger n-tuple systems. TCL-M consists of 150 n-tuples instead of 70, but is otherwise the same as TCL-EXP. For TCL-M with eligibility traces (TCL-M[et]) the time-to-learn value falls clearly below 200 000 games; overall, only 145 000 training games are required to cross the 80% success rate for the first time. At the same time, the percentage of lost games reduces from 7.0% (asymptotic success rate for TCL[et]) to 5.5% (TCL-M[et]), which is an improvement of 25%.

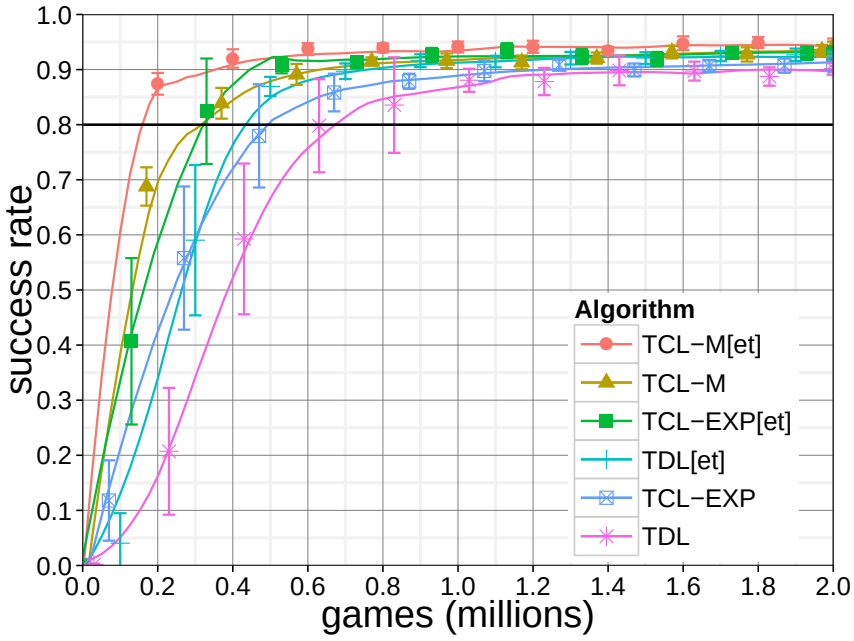

Figure 5.9: Comparison of different agents with and without eligibility traces. The notation [et] indicates that eligibility traces without the options replacing traces and resetting traces are used. TCL-M has the same settings as TCL-EXP with the exception, that a bigger n-tuple system consisting of 150 (instead of 70) 8-tuples is used. Both, TDL and TCL-EXP significantly increase their learning speed, if eligibility traces are employed and also the asymptotic success rate slightly increases in both cases. Also agents with larger n-tuple systems (TCL-M) benefit from eligibility traces.

Resetting and Replacing Traces for TDL and TCL-EXP

Considering the binary options *RES* (resetting traces) and *REP* (replacing traces) in Algorithm 1, in total 5 different eligibility trace variants can be used for the training process of an agent, displayed in Figure 5.10 for TCL-EXP. The averaged curves for TCL-EXP[res] and TCL[rr] are very similar for the first 1 million games of the training, the visual crossing of the 80% success rate for both variants is at around 200 000 games, followed by TCL-EXP[rep] (260 000 training games) and TCL-EXP[et] (320 000 training games). Note that the visual results maybe distorted by outliers among the 20 runs of an experiment. The exact values and the settings can be found in Table 5.2. Figure 5.11 summarizes the results for TDL and TCL-EXP with eligibility traces with run-time distributions for the targets at 80% and 90% success rates.

Results for nl-IDBD(λ) and SMD(λ)

The original nl-IDBD and SMD algorithm cannot be directly applied to the TD(λ) algorithm, since their definition does not include eligibility traces. In the Appendices A.3 & A.4 we derive slightly modified update rules and call the resulting algorithms nl-IDBD(λ) and

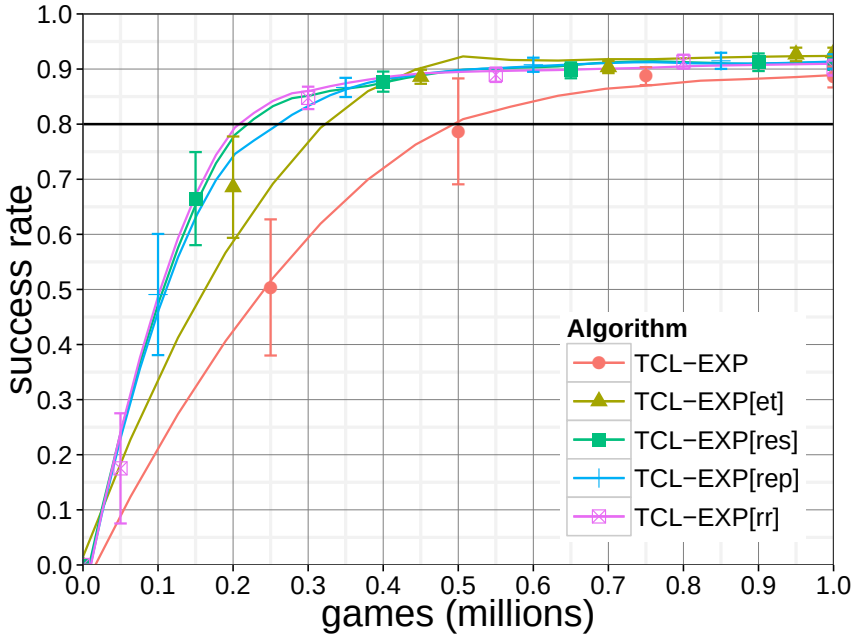

Figure 5.10: Comparison of all eligibility traces variants for TCL-EXP. The options resetting [res] and replacing traces [rep] allow to create 5 different variants of a TCL-EXP agent, all shown in this graph; TCL-EXP: agent w/o eligibility traces (λ= 0); [et]: standard implementation of eligibility traces; [res]: resetting the traces on exploratory moves; [rep]: replacing traces; [rr]: resetting & replacing traces.

SMD(λ). For $\lambda = 0$ we again receive the original versions of both algorithms.
For both algorithms we systematically varied λ in the range of $\lambda \in [0, 1]$ for the option [et] and found the results to be similar as before: the optimal value of the trace decay parameter lies in the range $\lambda \in [0.4, 0.6]$ for both, nl-IDBD(λ) and SMD(λ); for TCL-EXP the range was slightly larger with $\lambda \in [0.4, 0.7]$.
The option of replacing & resetting traces [rr] produced the best results when choosing λ slightly smaller as for TCL and TDL: For nl-IDBD we found $\lambda = 0.5$ to be optimal and for SMD a value of $\lambda = 0.7$.
In total, the differences of nl-IDBD and SMD to TCL-EXP are smaller than expected, all algorithms are more or less comparable with respect to the learning speed and final strength, although nl-IDBD appears to cross the 90% line slightly faster than the other algorithms.
We used the same meta-parameter settings (meta learning rate θ and initial step size α_i) for nl-IDBD(λ) and SMD(λ) as before. It may be possible, that the meta-parameters require some adjustments when eligibility traces are employed in order to gain optimal results. However, we did not investigate this aspect due to time restrictions.
A detailed comparison of all algorithms which we tested with eligibility traces is provided in the Figures 5.12–5.14 and in Table 5.2.

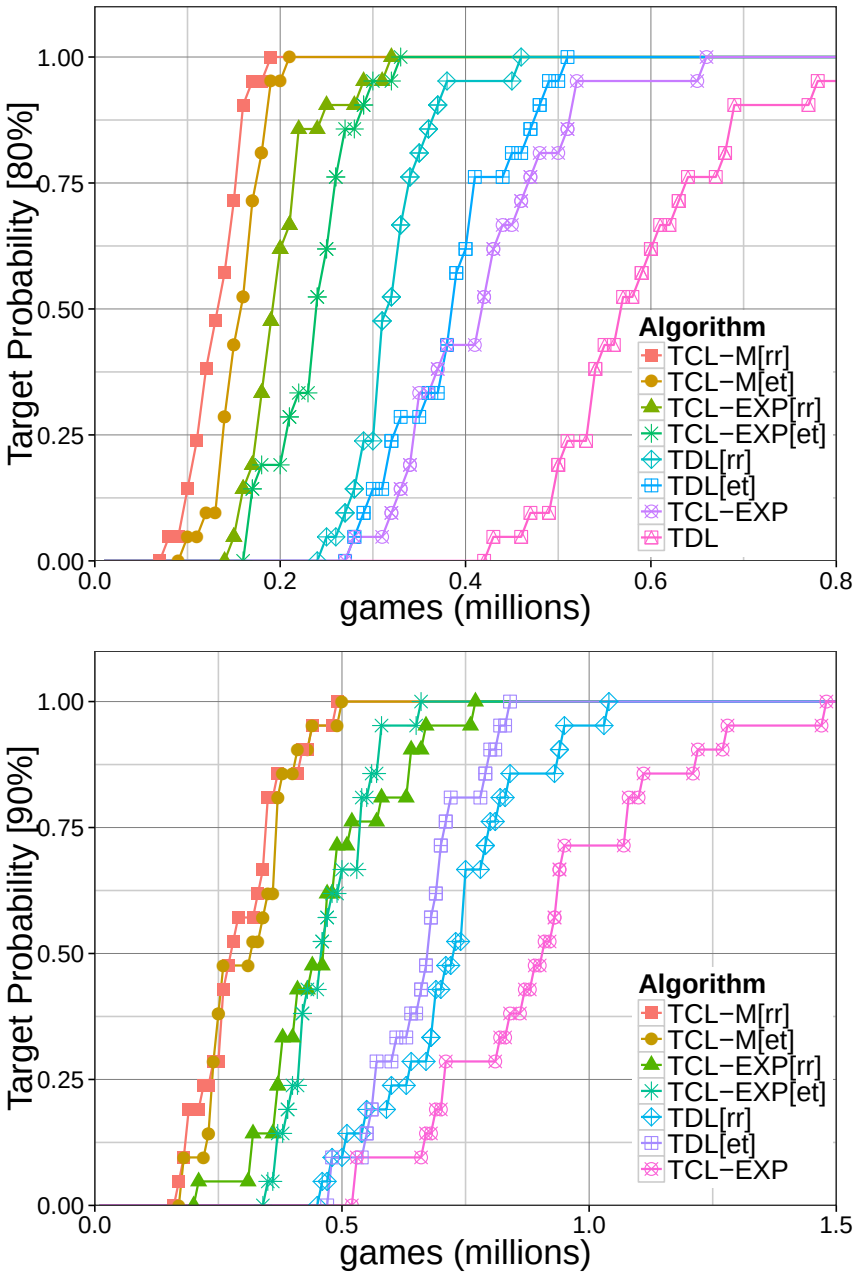

Figure 5.11: Run-time distribution for TDL and TCL-EXP with different eligibility trace variants. The graphs display the percentage of runs (on the ordinate axis) for each algorithm that has reached the specified target (time-to-learn for 80% and 90% success rate in the upper and lower graph, respectively) within a given amount of training games (on the abscissa axis). The run-time distribution of each algorithm is based on 20 runs. TDL is not shown in the lower graph. Details to each algorithm can be found in Table 5.2.

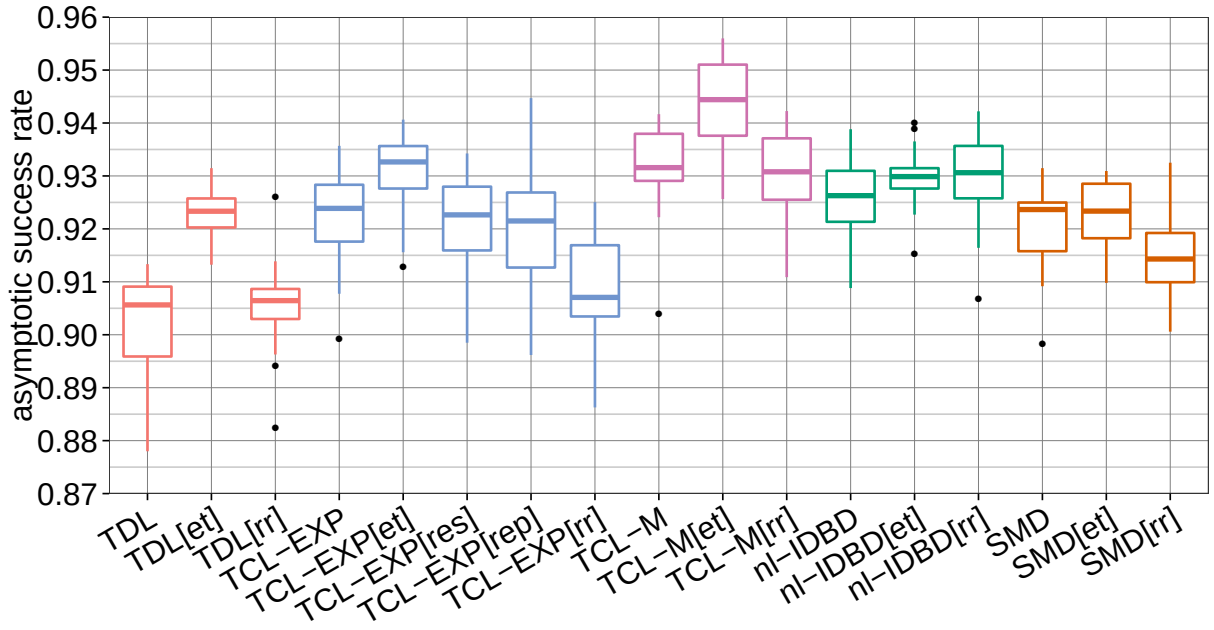

Figure 5.12: Asymptotic success rate for different eligibility trace variants. The asymptotic success is the average success, measured during the last 500 000 games at 50 equidistant time points. For each algorithm we selected the trace decay parameter λ that delivers the best results. TDL contains one outlier at 0.79, which is not shown in this graph. Details to each algorithm can be found in Table 5.2.

Summary for all Algorithms with Eligibility Traces

Figure 5.12 shows the asymptotic success rate of all relevant algorithms in a box-plot. The asymptotic success rates are slightly (TCL-EXP) or notably (TDL) higher when using standard eligibility traces as compared to no traces. This improvement is lost when adding any of the options "reset" or "replace". For nl-IDBD and SMD the option [et] results in the same asymptotic success rate as the corresponding algorithm without any traces. The asymptotic success for the variant "reset" & "replace" [rr] is either equal to or even less (with exception of nl-IDBD) than for the algorithm without any eligibility traces. TCL-M[et] reaches an asymptotic success rate of 94.4%, which is the highest among all algorithms.

The reasons why the algorithms with the option [et] mostly produce agents with a higher final strength are not definitely clear yet. We assume that this option may prevent an over-training of the generalizing value function, when some (discounted) credit is assigned to all earlier states as well, after a random move is performed. Especially if the random move is only one of several optimal moves, which is typically not considered by the greedy-policy of the agent, this approach could help the agent to learn about other variants. As an example, one could consider a typical mid-game situation with Red to move, where Yellow has control over the game and will win, provided that she continues her perfect

play. If Red would play a greedy move, then Yellow would most likely win the match, since she has seen this deterministic move sequence many times during the training process and knows the correct response. But if Red now performs a random move and manages to win/tie the game in the following because Yellow was not prepared for this random variant, then the states preceding the random move should receive a signal, which indicates that the win for Yellow may not be as clear as suggested by the current state-values. Eligibility traces could help to carry this information back to earlier states. However, it appears to be important that the credit given to earlier states is sufficiently discounted in this case. If λ is chosen too high ($\lambda \geq 0.8$) – as seen in Figure 5.8 and as discussed before – then wrong rewards could be projected too far back to previous states and the training breaks down.

In Figure 5.13, time-to-learn for different eligibility trace variants is shown in a box-plot for the 80% and 90% success rate target. It can be clearly seen that all eligibility trace variants speed up the learning of the agents considering both the 80% and 90% targets. The eligibility trace variant replace & reset [rr] produces the fastest learning agents for all algorithms, considering the first crossing of the 80% line. For the 90% target the results are more diverse: While [et] is the fastest option for TDL, for TCL-EXP the best options [rep] and [rr] have the same median. The remaining algorithms TCL-M, nl-IDBD and SMD cross the 90% line in the fastest way with the option [rr].
Figure 5.14 allows us to directly compare both metrics 'time-to-learn' and 'asymptotic success rate' for all different eligibility trace variants we investigated for our Connect-4 task.
Our main result is that eligibility traces make the learning much faster: The number of training games required to learn a certain target (whether it is the 80% success rate or the 90% success rate in Figure 5.13) is smaller by a factor of 1.5–2 compared with the variant without eligibility traces. The exact values and settings for all analyzed algorithms and eligibility trace variants can be found in Table 5.2. Compared to the results for our starting point on Connect-4 ('Former TDL' in Figure 5.1.1), the time-to-learn has decreased by an even larger factor 13: from the 1 565 000 games for 'Former TDL' to 115 000 games for the fastest algorithm TCL-M[rr] in this work. This decrease is however due to a combination of four factors: tuning of the exploration rate, step-size adaptation, a larger n-tuple system (TCL-M), and, last but not least, eligibility traces.

There are many good explanations in the standard TD literature – and also partly discussed in this thesis – why eligibility traces improve learning, but we want to emphasize one point in particular, which we already described in a recent publication [68]:

> *"here and in other n-tuple applications, the activation is usually very sparse: each board position will only activate 0.02% ($\approx 2 \cdot 70/650\,000$) of all active weights. Eligibility traces improve learning for such sparse activations, because more weights can learn in parallel in response to a given reward from the environment. Or, to put it in other words: Eligibility traces allow a certain weight to learn during a greater percentage of all time steps. If the individual weight updates are correlated, learning can*

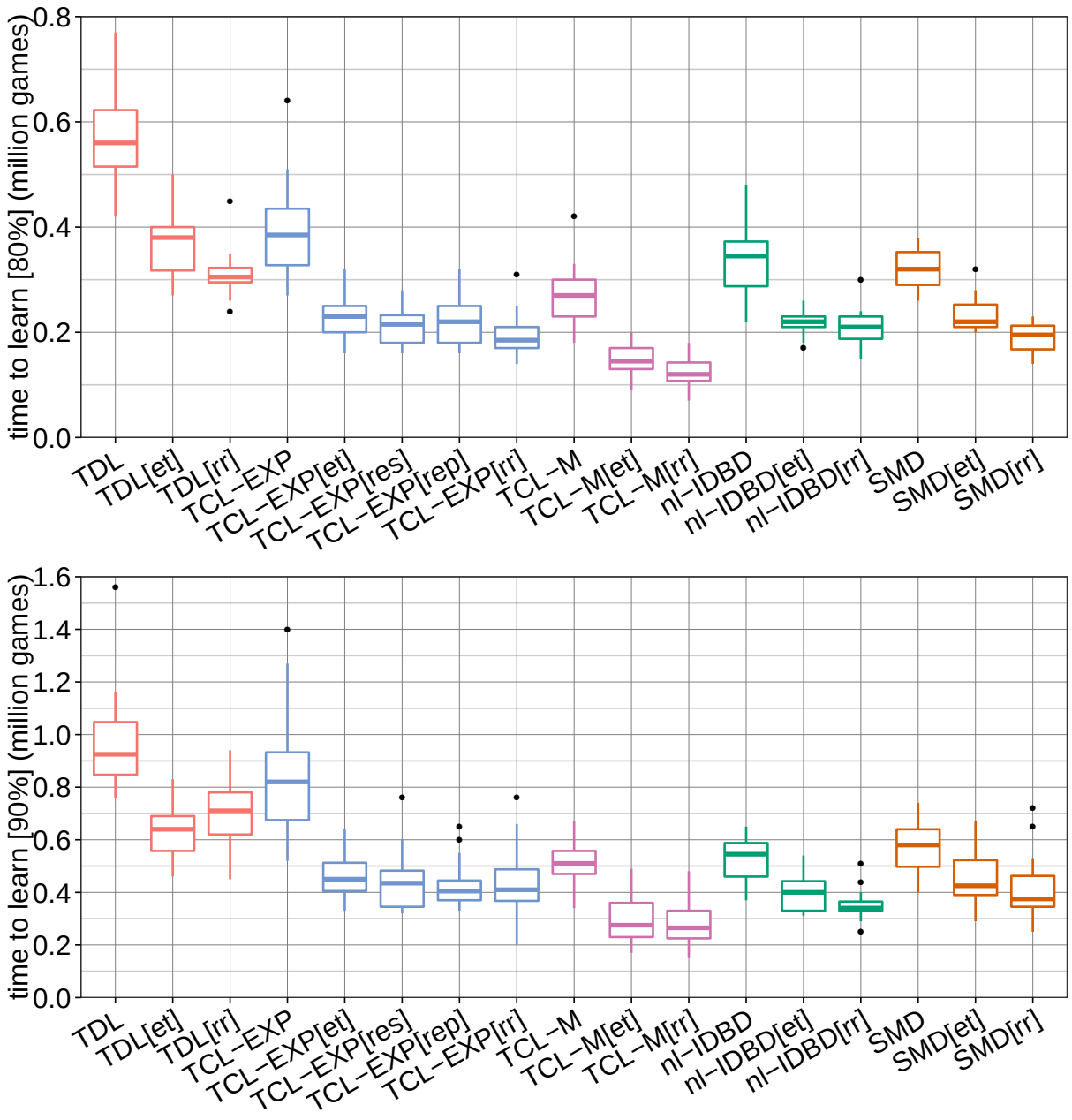

Figure 5.13: Time-to-learn for TDL and TCL-EXP with different eligibility trace variants. The upper graph shows the required training games for an algorithm to cross the 80% success rate for the first time and similarly, the results for the first crossing of the 90% line are displayed in the lower graph. Details to each algorithm can be found in Table 5.2.

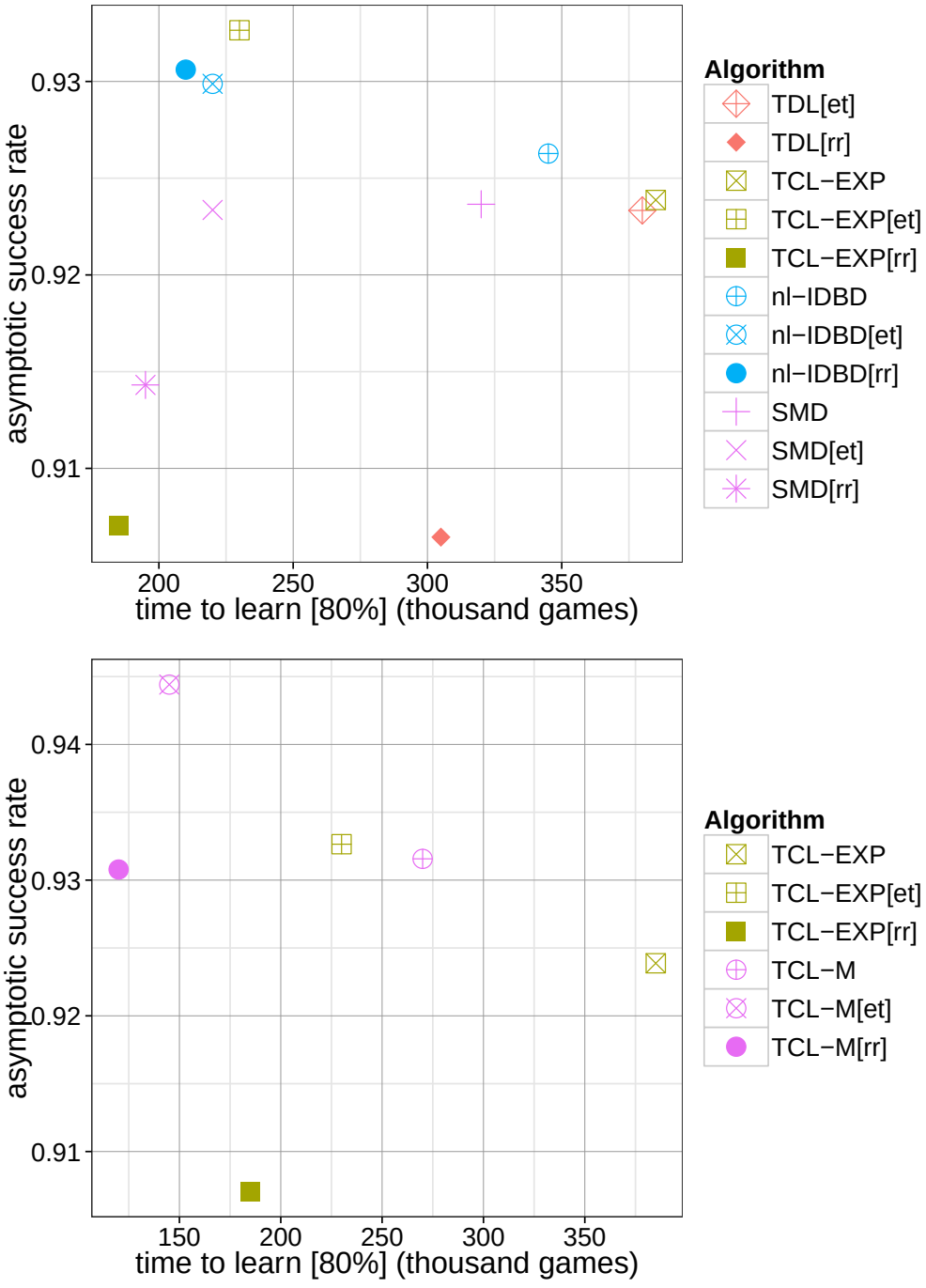

Figure 5.14: Final comparison of the metrics 'time to learn' and 'asymptotic success rate' (after 2 million games) for the different eligibility trace variants of the investigated algorithms. Lower values for 'time to learn' and higher values for 'asymptotic success rate' are better. In order to get a better view on the results, we removed the algorithm 'TDL' from this plot. The lower graph compares TCL-EXP for different n-tuple configurations (70×8-tuple for TCL-EXP and 150×8-tuple for TCL-M). Details regarding the experiments can be found in Table 5.2.

Table 5.2: Settings and performance of the different eligibility trace variants shown in Figure 5.12 – Figure 5.14. The columns *RES* and *REP* depict the options resetting and replacing traces, respectively. The two rightmost columns show the training times for the algorithms presented in this work. In the second column from the right ([games]) the number of training games needed to cross the 80%-success-line for the first time is given, by measuring the median (Figure 5.13) from the 20 runs. Accordingly, the last column shows the values for the 90%-success-line. The algorithm TCL-M differs from TCL-EXP only in the respect that an n-tuple system with 150 instead of 70 tuples of length 8 are used and that the global step size is slightly decreased. For all listed experiments the exploration-rate is chosen to be constantly $\epsilon = 0.1$.

						time to learn	
					ASR 2m	[80%]	[90%]
Algorithm	n-tuple	λ	*RES*	*REP*	[%]	[games]	[games]
TDL	70×8	0.0	–	–	90.6	565 000	925 000
TDL[et]	70×8	0.5	–	–	92.3	370 000	640 000
TDL[rr]	70×8	0.8	$\surd$	$\surd$	90.6	305 000	710 000
TCL-EXP	70×8	0.0	–	–	92.4	385 000	820 000
TCL-EXP[et]	70×8	0.5	–	–	93.3	230 000	450 000
TCL-EXP[res]	70×8	0.6	$\surd$	–	92.3	220 000	435 000
TCL-EXP[rep]	70×8	0.6	–	$\surd$	92.1	215 000	405 000
TCL-EXP[rr]	70×8	0.8	$\surd$	$\surd$	90.7	175 000	410 000
TCL-M	150×8	0.0	–	–	93.2	250 000	510 000
TCL-M[et]	150×8	0.5	–	–	94.4	145 000	275 000
TCL-M[rr]	150×8	0.8	$\surd$	$\surd$	93.1	115 000	265 000
nl-IDBD	70×8	0.0	–	–	92.8	350 000	545 000
nl-IDBD[et]	70×8	0.5	–	–	93.0	220 000	400 000
nl-IDBD[rr]	70×8	0.5	$\surd$	$\surd$	93.1	210 000	340 000
SMD	70×8	0.0	–	–	92.4	320 000	580 000
SMD[et]	70×8	0.5	–	–	92.3	220 000	425 000
SMD[rr]	70×8	0.7	$\surd$	$\surd$	91.4	195 000	375 000

proceed faster. If they are not correlated, the net change of the weight will be small. The effect is in a sense similar to TCL. In contrast to TCL, it is concentrated on a game episode and it allows a cross talk between states occurring later in a specific game and weights set to an eligible state earlier in that game."

5.2 Results for Dots-and-Boxes

In this section we briefly present the initial results when our learning framework (TDL, eligibility traces and online adaptable learning rate methods) is applied to the board game Dots-and-Boxes in order to investigate the general feasibility for future work. The scalability of the board size allows us to control the complexity of the game in a simple manner.

5.2.1 Initial Results for 2×2 Boards

We started our experiments with a board consisting of 2×2 boxes. Initially, only TDL without any step-size algorithm was used. All possible symmetries (including corner-symmetries, described in Section 3.4) were exploited during the learning process. The exploration rate was set to $\epsilon = 0.2 = \text{const.}$ which appeared to result in a higher asymptotic strength than $\epsilon = 0.1$. We observed that comparably long n-tuples were required to achieve perfect play against an equally perfect opponent: Overall, 40 7-tuple were created by the 'random line-walk' method. Since the 2×2 board consists of 12 edges, each n-tuple covers nearly 2/3 of the board. The number of weights created for the n-tuple system is $40 \cdot 2^7 = 5\,120$; all n-tuple states are realizable so that all weights can be trained during the learning process. Note that, in contrast to Connect-4, for each run a new set of n-tuples is generated. This was done in order to demonstrate different randomly generated sets of n-tuples produce comparable results for a certain setting.

Learning the outcome of the game (win/draw/loss) did not lead to satisfactory results in the beginning. This changed when we used the approach 'board inversion' combined with 'intermediate rewards', as described in Section 3.4.3, which makes it possible to learn a single value function for only one player (player A). Positions with player B to move are evaluated by inverting the current state-information before consulting the value function.
For the above setup, we found that near perfect play (crossing of the success-rate -0.1) can be learnt in several hundred games, typically after approximately 600–700 games, if a suitable step size is selected. Constant step sizes in the range of $\alpha = [4 \cdot 10^{-4}, 10^{-3}] = \text{const.}$ delivered the best results (exponential adjustment-schemes may slightly improve the results). For these values, perfect play (success-rate of 0.0) is achieved latest after 1 700 training games.
Subsequently, we also performed several tests with Sutton's IDBD as the step-size adaptation method and found mostly similar results: For suitable settings of β_{init} and θ the game can be learnt after 500–700 games. For $\beta_{init} = -7,4185809$ and $\beta_{init} = 1.0$ the 'time-to-learn' (crossing of the success-rate -0.1) was 400 games.
Finally, we performed several experiments for Autostep. As for our Connect-4 learning task, Autostep was rather insensitive to its meta-learning rate μ. To our surprise, Autostep

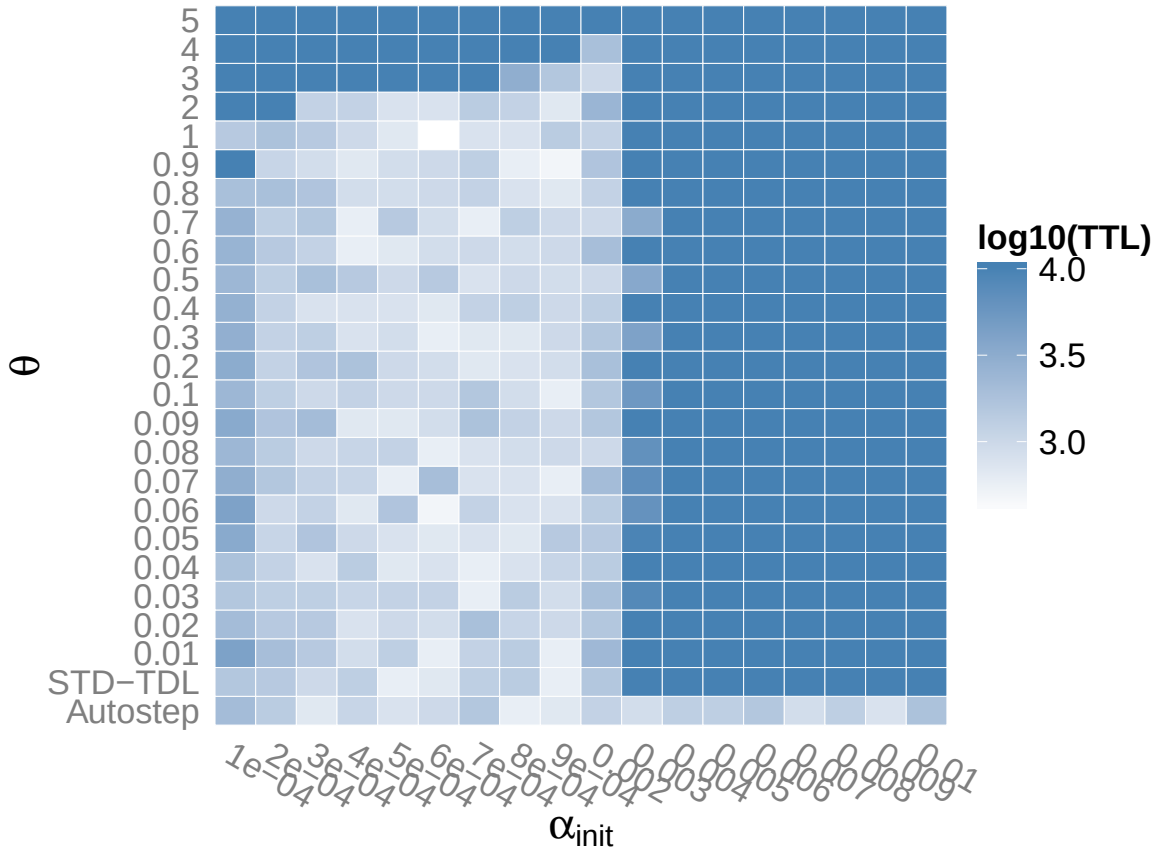

Figure 5.15: Qualitative Comparison of the 'time-to-learn' (TTL) for TDL, IDBD and Autostep for a 2×2 board in Dots-and-Boxes. The colors indicate the logarithmic (log10) 'time-to-learn' (first crossing of the -0.1 line), increasing gradually from white to dark blue. On the y-axis the values for the meta-learning rate θ for IDBD are shown. The last two rows show the results for standard TDL and Autostep. Since Autostep showed no significant sensitivity towards its meta-learning rate μ, all experiments for Autostep are performed with $\mu = 0.001$. The initial step size α_{init} is plotted on the abscissa (β_{init} for IDBD, which can be transformed with the relation $\alpha_{init} = e^{\beta_{init}}$).

showed nearly no sensitivity to the initial step size α_{init}, for the range we observed. Although the 'time-to-learn' value exceeded 1 000 games for too small or too large values of α_{init}, the training never broke down completely, as for standard TDL or IDBD. The qualitative results, comparing Autostep to standard TDL and Suttons's IDBD are displayed in Figure 5.15. The figure shows, that while IDBD and TDL break down for values of $\alpha_{init} = 0.002$, Autostep performs well in the whole evaluated range $\alpha_{init} = [10^{-4}, 10^{-1}]$.

5.2.2 Results for 3×3 and 4×4 Boards

Although the state-space complexity for the 3×3 board is – with 2^{24} states (see Section 3.1.2) – fairly moderate, the game-tree complexity is $24! \approx 6.2 \cdot 10^{23}$, which is already larger than for Connect-4 (approximately 10^{21} [4]). In comparison to a 2×2 board, the state-space of the 3×3 board is larger by a factor of 4096 and the game-tree complexity by a factor of around $1.3 \cdot 10^{15}$, which makes the learning task significantly more complex.

Since Autostep performed well over a large range of initial step sizes for the 2×2 board, we decided to use Autostep as well for the initial experiments on the 3×3 board, with $\alpha_{init} = \exp(-6)$. Other initial step sizes were not tested for Autostep. Again, we found an exploration rate of $\epsilon = 0.2 = \text{const.}$ to lead to the best results. Therefore, we used this exploration scheme for all other experiments as well. However, we realized later that such a high exploration could negatively affect the algorithms with eligibility traces (as described below).
We experimented with different n-tuple systems and again we observed that comparably long n-tuples are required in order to learn the game: For all experiments described in the following the same set of $80\times$ 16-tuple is used. As before, this means that each n-tuple covers two-thirds of the board.
In addition to Autostep, we also tested TCL-EXP and nl-IDBD(λ) with and without eligibility traces and standard TDL. For standard TDL the step size was systematically varied and we found $\alpha = 7 \cdot 10^{-4}$ to deliver the best results (the step size is kept constant during the training). Due to time restrictions, we did not perform extensive tuning runs for TCL-EXP and nl-IDBD(λ).
The results are displayed in Figure 5.16 and in Table 5.3.
To reach the target success value of -0.2, TCL-EXP[rr] – the fastest learning algorithm – requires 13 000 training games, followed by nl-IDBD[rr] (16 000 games) and Autostep (21 000 games). Surprisingly, neither TCL-EXP, nor nl-IDBD(λ) can clearly outperform standard TDL. In fact, TCL-EXP performs even worse than TDL. However, as already mentioned, this might be due to the fact that no systematic tuning was performed for TCL-EXP and nl-IDBD(λ), as we did for our Connect-4 task. Although no tuning process was performed for Autostep and no eligibility traces were used, the algorithm performs remarkably well.
Again, eligibility traces have proven to increase the learning speed and the asymptotic success of the agents: Both, TCL-EXP and nl-IDBD learn faster by a factor of approximately 1.8 and 1.6, respectively, when augmented with eligibility traces. A few additional runs indicated that the algorithms with eligibility traces can learn slightly faster if the exploration rate is reduced to $\epsilon = 0.1 = \text{const.}$ The reason for this is that higher exploration rates diminish the positive effect of eligibility traces, since the trace vector is reset more often for the variants with resetting traces and for the variants without resetting traces, more wrong rewards – a consequence of random moves – are propagated back to earlier states and negatively affect the corresponding state values.

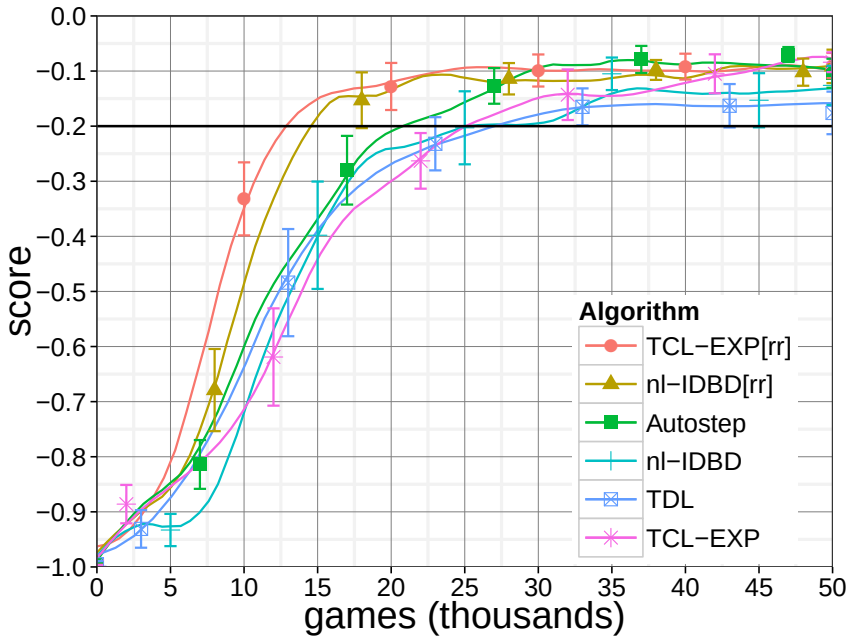

Figure 5.16: Results for different algorithms on a 3×3 Dots-and-Boxes board. For each setting 20 separate runs were performed. The agent's strength was assessed every 1 000 games based on 50 evaluation matches. Each point in the graph is the mean of all 20 runs. For better display of the error bars (standard deviation within 20 runs), only a subset of all points is shown in the plot. The lines are a smoothed fit to the noisy observations with the fit being based on all measured points (including the points not shown in this graph). Note that due to this procedure (averaging the 20 runs and smoothing the curve) the visual crossing of the '−0.2' line and the asymptotic success may differ somewhat from the values given in the text and in Table 5.3, which are based on the median of 20 runs.

We also performed a few training runs for 4×4 boards (only using Autostep), but did not receive good results. After 500 000 training games the average success rate was typically around '−0.8', hence, the trained TDL-agent could only achieve a tie in 20% of all evaluation matches[3]. We presume two reasons for these poor results: 1) The number of training games might have been insufficient. It could be possible that the agent reaches higher strengths if the number of training games is increased (e.g., to several millions).
2) The length of the n-tuples might be insufficient. We used an n-tuple system consisting of 80 16-tuple. Each n-tuple therefore covers 40% of the board. For smaller board sizes we observed that good results can be only achieved if the individual n-tuples cover around two-thirds of the board. For 4×4 boards, this would imply that n-tuples of length 27 are required, which is technically infeasible since the memory requirements would exceed the available memory of most modern PCs. Furthermore, such a large system most likely cannot be trained in reasonable time and puts the generalization capabilities of the state-value function into question.

[3]For a 4×4 Dots-and-Boxes board, assuming perfect play of both opponents, a match always results in a tie.

Table 5.3: Summary of the parameter settings, training times and asymptotic success rates for different algorithms on the Dots-and-Boxes task. For all experiments the same n-tuples were used (80×16-tuple). Column *TCL* denotes, whether we used for TCL the recommended-weight update [r] or the δ-update [δ]. For all algorithms specifying β_{init} the relation $\alpha_{init} = e^{\beta_{init}}$ holds. Therefore, we only list α_{init} in this table. For TDL, the step size is constant, hence, $\alpha = \alpha_{init} = \alpha_{final} = const.$ All listed algorithms use a constant exploration rate $\epsilon = 0.2$. Every setting is repeated 20 times with different random initialization and random exploration. The agent strength was measured every 1 000 games. The median of 'time-to-learn' is given in the column 'TTL'. The last but one column shows the asymptotic success rate ASR (median of 20 runs, the ASR of each run is based on the last 15 measured points) after 50 000 games.

								ASR	TTL [−0.2]
Algorithm	α_{init}	λ	REP	RES	β	θ	TCL	[%]	[games]
TDL	$7 \cdot 10^{-4}$	0.0	–	–	–	–	–	-0.14	27 000
TCL-EXP	0.001	0.0	–	–	1.3	–	[r]	-0.1	24 000
TCL-EXP[rr]	0.001	0.8	$\surd$	$\surd$	1.3	–	[r]	-0.1	13 000
Autostep	exp(−6)	0.0	–	–	–	10^{-4}	–	-0.09	21 000
nl-IDBD	exp(−7.6)	0.0	–	–	–	1.0	–	-0.13	25 000
nl-IDBD[rr]	exp(−7.6)	0.6	$\surd$	$\surd$	–	1.0	–	-0.11	16 000

Overall, n-tuple systems might not be suitable to generate useful features for a problem such as Dots-and-Boxes, where typically many long chains in various forms are created. Too short n-tuples – that only partially cover most chains – cannot describe the current chain configuration accurately and will result in a wrong estimation of the state value. One major difficulty that we had to deal with during our experiments was the long training durations, which allowed us to only test a few settings. In order to complete 500 000 training games, initially 4–5 days were required on a standard PC (Intel i7-3520M CPU with 8 GB RAM). We identified the evaluation process (based on Wilson's solver [73]) as the main reason for this: In order to complete 50 evaluation matches around 2–3 hours were needed[4]. After we combined Wilson's solver with our own tree search[5] the evaluation time could be reduced to approximately 25 minutes for 50 evaluation matches.
Another reason for the long training times is the high branching factor for the 4×4 board, which makes it necessary for the TDL agent to access the n-tuple system many times in each time step. For example, in order to perform a greedy move for the empty board, the TDL agent has to consult the n-tuple system for each of the 40 possible after-states. This

[4]Wilson's solver relies on a huge database of 41 GB for the 4×4 board, which was generated with a retrograde analysis approach. In order to evaluate a position, the hard-drive has to be accessed many times. We assume that the long access times for the hard-drive are the bottleneck of the solver.

[5]Wilson's solver is used to find the next move for positions with less than 15 edges. For positions with more edges, our tree search can find the next move significantly faster.

involves sampling the after-state position and its symmetric equivalents for every n-tuple, computing the weight indexes for the LUTs and summing up all activated weights in order to determine the state value.
Since the weight indexes are always the same for a certain position, we implemented a hash-table that records the set of weight indexes for the position[6]. With this approach it is possible to avoid the costly index computation for recurring board positions and reduce the computation time.
Overall, with the modified agent evaluation and the additional hash-table, the computation time for 500 000 training games could be reduced to around 15 hours instead of the former 4–5 days.

[6]In case of a collision, older entries are automatically overwritten by the newer ones

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we studied complex learning tasks for the strategic board games Connect-4 & Dots-and-Boxes. We could already show in our earlier work [69, 67] that it is possible to learn the game Connect-4 solely by self-play. Especially n-tuple networks, which generate a large number of features, were a crucial ingredient for success. However, the number of required training games (more than 1.5 million) remained rather high.
In this work, we investigated the benefits of step-size adaptation algorithms and eligibility traces, when added to the existing framework. Besides several already existing state-of-the-art algorithms such as SMD and Autostep, we also introduced and tested a modified version of the TCL algorithm, which we labeled TCL-EXP. Similarly to other methods, TCL-EXP adjusts the learning rates in geometric steps and we could show that it performs significantly better on our learning tasks than the original algorithm. Furthermore, we extended the definitions of several learning rate methods in such a way that they can be used in conjunction with eligibility traces.
In order to be able to implement eligibility traces in a memory- and time-efficient way, we exploited the sparseness in n-tuple networks and found self-balanced binary trees to be suitable for representing the eligibility trace vector. Overall, eligibility traces could significantly increase the training speed and slightly increase the asymptotic strength of the agents.

We conclude our work by answering the research questions stated in Chapter 2:

1. *Can online learning rate adaptation increase the performance of a complex learning task with millions of weights?*

   This research question can be answered positively for most of the examined learning rate adaptation algorithms. All algorithms that we tested, could be applied to the Connect-4 learning task with more than half a million weights. Since the algorithms only updated the step sizes of active weights in each time step, the increase in computation time was only moderate, due to the sparse activation in the n-tuple system. Effectively, most algorithms could even decrease the computation time because of the improved learning speed of the agents. Only the original TCL algorithm performs

worse than the original TDL algorithm. Especially algorithms with geometric step sizes and nonlinear output units (nl-IDBD, SMD) could significantly increase the learning speed. Inspired by the geometric step-size property of these algorithms we developed a modified version of TCL, which uses an exponential transfer function. This new TCL-EXP algorithm performs similarly, yet slightly worse than nl-IDBD and SMD on the Connect-4 task. The fastest algorithm, Schraudolph's Stochastic Meta Descent (SMD) [49], could improve the learning speed in comparison to the original TDL algorithm by a factor of almost 2. For most algorithms also a slight increase of 1–2% in the agent's strength could be observed (after 2 million games), although TDL without learning rate adaptation approaches the same strength after 10 million games. Overall, geometric step sizes and non-linear units appear to be a suitable combination to significantly reduce the training time of an agent.

2. *How sensitive are the step-size adaptation algorithms towards their own meta parameters?*

   We found that none of the examined algorithms can free the user from tuning the meta parameters for our Connect-4 task. All step-size adaptation methods were sensitive towards their initial step size (defined either by α_{init} or β_{init}): wrong choices effectively lead to a breakdown of the system or extremely slow learning. Those algorithms performing meta-descent (which tune the step sizes by minimizing a certain loss function with a gradient-descent approach), also mostly show a large sensitivity towards their meta-learning rate θ. A noteworthy exception is Autostep, which basically does not require any tuning for its meta-learning rate μ.

3. *Can TDL, augmented with eligibility traces, utilize training samples more efficiently and improve the speed of learning for our complex Connect-4 task?*

   This research question can be answered positively as well: Although some differences regarding the learning speed and asymptotic strength could be observed between the different eligibility trace variants that we compared, all variants significantly increase the training speed of an agent for our Connect-4 task. The time to reach a certain target (whether it is the 80% success rate or the 90%) can be decreased by a factor of 1.5–2. This also applies to the algorithms nl-IDBD(λ) and SMD(λ), which we derived from Koop's nl-IDBD [60, 34] and Schraudolph's SMD.

4. *Is it possible to successfully transfer the developed learning framework (TDL with eligibility traces, n-tuple systems, online adaptable learning rates) without significant adjustments to the strategic board game Dots-and-Boxes?*

   Based on the initial results that we received for small Dots-and-Boxes boards with up to 3×3 boxes, this research question has to be answered negatively: First of all, it was

not possible to apply the existing learning infrastructure to Dots-and-Boxes without adjustments. Initially, even for a small 2×2 board the agent failed to learn the game. Only after introducing the concept of intermediate rewards (described in Section 3.4.3) it was possible to train near perfect-playing agents for boards with 2×2 and 3×3 boxes. However, we observed that – in contrast to Connect-4 or Othello [36] – rather long n-tuples are required for successful learning; each n-tuple had to cover around two-thirds of the board. Since the LUT sizes of an n-tuple system scale exponentially with the length of the n-tuples, for larger boards we quickly run into a problem of combinatorial explosion, if each n-tuple has to sample two-thirds of the board in order to train strong agents. Already for a 4×4 board the memory requirements would exceed the available memory of most modern computer systems.
In our experiments with the 4×4 board we employed shorter n-tuples of length 16 – which only cover around 40% of the board – and received poor results; the trained agents could only compete with a perfect-playing opponent in around 20% of all evaluation matches, all other matches were lost.
Altogether, n-tuple systems might not be the correct approach to tackle a learning task such as Dots-and-Boxes, where large segments of the board have to be sampled in order to generate meaningful features.

Overall, it is remarkable that an agent can learn near perfect play for a complex board game like Connect-4, simply by interacting with itself (self-play) and without access to any teacher or other external game-specific knowledge and without the use of hand-crafted features. With the enhancements investigated in this thesis, we showed that it is possible to learn the non-trivial game Connect-4 in less than 120 000 games for the best found setting. Compared with the starting point of this work, where around 1 565 000 training games were required to learn Connect-4, this is an increase in the learning speed by a factor of more than 13. We identified 4 main ingredients that were necessary to achieve this improvement: 1) The tuned exploration rate reduced the number of training games by a factor of approximately 2.8. 2) Step-size adaptation algorithms improve the learning speed by a factor of about 1.5, whereby algorithms with non-linear units learn faster by a factor of 1.2 (in comparison to the variants which are restricted to linear functions). 3) Larger n-tuple systems with twice the number of weights speed-up the training by a factor of 1.5. 4) Last but not least, eligibility traces reduce the number of required training games by another factor of 2.

6.2 Future Work

Although many ideas could be realized during the work on this thesis, there is considerable scope for further investigation.
Especially in the field of online adaptable learning rates we still see room for improvement: In this work we could not identify any step-size adaptation algorithm that can be considered

as tuning-free, when applied to our RL tasks, although Autostep already made a step into this direction. All algorithms were more or less sensitive towards their meta parameters and required some tuning process in order to be applicable to the Connect-4 learning task. For many other – even more complex – learning tasks such tuning would be very time consuming or even infeasible. We have the impression that more research is necessary on step-size adaptation algorithms – especially in the field of reinforcement learning – in order to reduce the dependency on the meta parameters and to move towards domain-independent, tuning-free methods. Many of the algorithms we examined in our work are defined for supervised learning (e.g., Autostep) and not for RL tasks. Although it is possible to apply these methods to online temporal difference learning (as we did), some adjustments might be required, taking into account some specific characteristics of TDL. For example, TDL typically imposes – due to the bootstrapping process involved and the policy improvement process of the agent – a highly non-stationary problem, especially during the initial training phase. This should be considered in the design of the step-size adaptation algorithms. Another aspect are eligibility traces, defined in the TD(λ) algorithm: the definition of most algorithms does not incorporate eligibility traces yet. Although we derived update rules for nl-IDBD and SMD that can be used in conjunction with the TD(λ) algorithm, it has to be investigated if we can also similarly derive new rules for the remaining algorithms.
Furthermore, several of the analyzed step-size adaptation algorithms are only defined for linear units. Extending these algorithms to the more general case, with non-linear function approximations, might allow improvement of learning in many cases.

Next to the step-size parameter, the exploration rate is another important ingredient for TDL. Typically, it is not a trivial task to balance the amount of exploration and exploitation during the training and the overall learning performance is typically highly dependent on the correct choice of the exploration strategy. Although we could achieve good results by following a simple ϵ-greedy policy, other exploration strategies might be more efficient. Several approaches, such as'Value-Difference Based Exploration' [71] which automatically adapts the exploration-rate for an ϵ-greedy policy, Model based Bayesian Exploration [23] and others [20, 28], could also improve learning and help to decrease the training time. Especially for board games with a high branching factor, such as Dots-and-Boxes, where the agent has many actions at its disposal, more efficient exploration strategies might be particularly beneficial.

Investigations in future could also concentrate on other topics, such as the n-tuple generation process for board games. In this and in previous work we simply used randomly generated n-tuples for the n-tuple network. A more systematic approach for creating relevant n-tuples could improve the training of the agents. It would have to be analyzed as to which objective functions are able to separate good n-tuples from bad ones. Also hand-crafted n-tuples or regular n-tuples like they were used in the game 2048 [64] might be an option to improve learning.

In future, we are planning to continue our work on the game Dots-and-Boxes, for which we have already discussed initial results in this thesis. Although some open questions and several problems (such as the high branching factor) remain for Dots-and-Boxes, we are optimistic that with our learning framework (TDL and n-tuple networks) in future also strong players could be trained for boards with 4×4 boxes or larger. Also other strategic board games, such as Nines Mens Morris, Checkers, Abalone, Hex or Paletto could be interesting choices for such a learning framework.

Our long-term goal is to gain a deeper understanding of machine learning approaches, especially of autonomously learning agents and to advance the state-of-the-art in general game playing (GGP).

Bibliography

[1] Abramson, B.: Expected-Outcome: A General Model of Static Evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI) 12(2), 182–193 (1990)

[2] Allen, J.D.: A Note on the Computer Solution of Connect-4. In: Levy, D., Beal, D. (eds.) Heuristic Programming in AI 1: The First Computer Olympiad, pp. 134–135. Ellis Horwood, London (1989)

[3] Allis, V.L.: A knowledge-based approach of Connect-4. The game is solved: White wins. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1988)

[4] Allis, V.L.: Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis, University of Limburg, p. 163 (1994)

[5] Almeida, L., Langlois, T., Amaral, J.D.: On-Line Step Size Adaptation. Tech. Rep. RT07/97, INESC, 1000, Lisboa, Portugal (1997)

[6] Almeida, L.B., Langlois, T., Amaral, J., Plakhov, A.: Parameter adaptation in stochastic optimization. On-Line Learning in Neural Networks, Publications of the Newton Institute pp. 111–134 (1998)

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

[8] Bagheri, S., Thill, M., Koch, P., Konen, W.: Online Adaptable Learning Rates for the Game Connect-4. IEEE Transactions on Computational Intelligence and AI in Games (T-CIAIG) (accepted 11/2014), 1 (2015)

[9] Baird, L.: Residual algorithms: Reinforcement learning with function approximation. In: Prieditis, A., et al. (eds.) Proceedings of the Twelfth International Conference on Machine Learning (ICML 1995). pp. 30–37. Morgan Kauffman, San Francisco, CA, USA (1995)

[10] Barker, J.K., Korf, R.E.: Solving 4x5 Dots-And-Boxes. In: Burgard, W., et al. (eds.) Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. AAAI Press, Palo Alto, California (2011)

[11] Barker, J.K., Korf, R.E.: Solving Dots-And-Boxes. In: DieterFox, et al. (eds.) Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence. AAAI Press, Palo Alto, California (2012)

[12] Baxter, J., Tridgell, A., Weaver, L.: Learning to Play Chess Using Temporal Differences. Machine Learning 40(3), 243–263 (2000)

[13] Beal, D.F., Smith, M.C.: Temporal Coherence and Prediction Decay in TD Learning. In: Dean, T. (ed.) International Joint Conferences on Artificial Intelligence (IJCAI). pp. 564–569. Morgan Kaufmann, San Francisco, CA (1999)

[14] Beal, D.F., Smith, M.C.: Temporal Difference Learning for Heuristic Search and Game Playing. Information Sciences 122(1), 3–21 (2000)

[15] Berlekamp, E.R.: The dots-and-boxes game – sophisticated child's play. A K Peters Series, Taylor & Francis (2000)

[16] Bledsoe, W.W., Browning, I.: Pattern Recognition and Reading by Machine. In: Bloch, E. (ed.) 1959 Proceedings of the Eastern Joint Computer Conference, pp. 225–232. Academic, New York (1959)

[17] Browne, C., et al.: A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games 4(1), 1–43 (2012)

[18] Brügmann, B.: Monte Carlo Go. Tech. rep., Max-Planck-Institute of Physics, München (1993)

[19] Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., et al. (eds.) Computers and Games (CG). Lecture Notes in Computer Science, vol. 4630, pp. 72–83. Springer, New York (2006)

[20] Şimşek, O., Barto, A.G.: An intrinsic reward mechanism for efficient exploration. In: Proceedings of the 23rd International Conference on Machine Learning. pp. 833–840. ICML '06, ACM, New York, NY, USA (2006)

[21] Curran, D., O'Riordan, C.: Evolving Connect-4 Playing Neural Networks Using Cultural Learning. NUIG-IT-081204, National University of Ireland, Galway (2004)

[22] Dabney, W., Barto, A.G.: Adaptive Step-Size for Online Temporal Difference Learning. In: 26th AAAI Conference on Artificial Intelligence. AAAI Press, Palo Alto, California (2012)

[23] Dearden, R., Friedman, N., Andre, D.: Model based Bayesian Exploration. In: Laskey, K.B., et al. (eds.) Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI). pp. 150–159. Morgan Kaufmann, San Francisco, CA (1999)

[24] Edelkamp, S., Kissmann, P.: Symbolic classification of general two-player games. In: KI 2008: Advances in Artificial Intelligence, pp. 185–192. Springer (2008)

[25] Gelly, S., Silver, D.: Monte-Carlo tree search and rapid action value estimation in computer Go. Artif. Intell. 175(11), 1856–1875 (2011)

[26] Geramifard, A., Bowling, M., Zinkevich, M., Sutton, R.S.: iLSTD: Eligibility traces and convergence analysis. In: Schölkopf, B.o. (ed.) Advances in Neural Information Processing Systems 19 (NIPS'06). pp. 440–448. MIT Press (2007)

[27] Grossman, J.: Dabble. `http://wilson.engr.wisc.edu/boxes/program.shtml` (2002), last Access: 20.05.2015

[28] Hester, T., Lopes, M., Stone, P.: Learning Exploration Strategies in Model-Based Reinforcement Learning. In: Proceedings of the twelfth International Conference on Autonomous Agents and Multiagent Systems (AAMAS). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2013)

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

[30] Jaskowski, W.: Systematic N-tuple Networks for Position Evaluation: Exceeding 90% in the Othello League. CoRR abs/1406.1509 (2014)

[31] Kaelbling, L.P., Littman, M.L., Moore, A.P.: Reinforcement learning: A survey. Journal of Artificial Intelligence Research 4, 237–285 (1996)

[32] Konen, W., Bartz-Beielstein, T.: Reinforcement Learning: Insights from Interesting Failures in Parameter Selection. In: Rudolph, G. (ed.) Proceedings of the 10th International Conference on Parallel Problem Solving From Nature (PPSN). pp. 478–487. Springer, Berlin (2008)

[33] Konen, W., Koch, P.: Adaptation in Nonlinear Learning Models for Nonstationary Tasks. In: Filipic, B. (ed.) Proceedings of the 13th International Conference on Parallel Problem Solving From Nature (PPSN). Springer, Heidelberg (2014)

[34] Koop, A.: Investigating Experience: Temporal Coherence and Empirical Knowledge Representation. Master Thesis, University of Alberta, Canada (2008)

[35] Krawiec, K., Szubert, M.G.: Learning n-tuple networks for Othello by coevolutionary gradient search . In: Krasnogor, N. (ed.) Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO). pp. 355–362. ACM, New York (2011)

[36] Lucas, S.M.: Learning to Play Othello with N-Tuple Systems. Australian Journal of Intelligent Information Processing 4, 1–20 (2008)

[37] Lucas, S.M.: Face recognition with the continuous n-tuple classifier. In: Clark, A.F. (ed.) Proceedings of the British Machine Vision Conference. British Machine Vision Association (1997)

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

[39] Mahmood, A.: Automatic step-size adaptation in incremental supervised learning. Master Thesis, University of Alberta, Canada (2010)

[40] Mańdziuk, J., Osman, D.: Temporal difference approach to playing give-away checkers. In: The 14th International Conference on Artificial Intelligence and Soft Computing (ICAISC), pp. 909–914. Springer (2004)

[41] Plaat, A., Schaeffer, J., Pijls, W., Bruin, A.D.: Nearly Optimal Minimax Tree Search? Tech. rep., Department of Computing Science, University of Alberta, Edmonton, Alberta (1994)

[42] Riedmiller, M., Braun, H.: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: Proceedings of 1993 IEEE International Conference on Neural Networks (ICNN). pp. 586–591 (1993)

[43] Samuel, A.: Some studies in machine learning using the game of checkers. IBM Journal of Research and Development 3(3), 210–229 (1959)

[44] Schaeffer, J., Hlynka, M., Jussila, V.: Temporal Difference Learning Applied to a High-Performance Game-Playing Program. In: Nebel, B. (ed.) Proceedings of the 17th international joint conference on Artificial intelligence (IJCAI). pp. 529–534. Morgan Kaufmann (2001)

[45] Schaul, T., Zhang, S., LeCun, Y.: No More Pesky Learning Rates. In: Bach, F., et al. (eds.) Proceedings of the International Conference on Machine Learning (ICML) (2013)

[46] Schneider, M., Garcia Rosa, J.: Neural Connect-4 - a connectionist approach. In: Proceedings 7th Brazilian Symposium on Neural Networks. pp. 236–241 (2002)

[47] Schraudolph, N.N.: Local Gain Adaptation in Stochastic Gradient Descent. In: Proceedings of the ninth International Conference on Artificial Neural Networks (ICANN). pp. 569–574. IEEE, London (1999)

[48] Schraudolph, N.N.: Online learning with adaptive local step sizes. In: Proceedings of the 11th Italian Workshop on Neural Nets (WIRN), pp. 151–156. Springer (1999)

[49] Schraudolph, N.N., Aberdeen, D., Yu, J.: Fast Online Policy Gradient Learning with SMD Gain Vector Adaptation. In: Proceedings of the twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS) (2005)

[50] Schraudolph, N., Dayan, P., Sejnowski, T.: Using the TD(λ) algorithm to learn an evaluation function for the game of Go. Advances in Neural Information Processing Systems 6 (1994)

[51] Silver, D., Sutton, R.S., Müller, M.: Temporal-Difference Search in Computer Go. In: Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS) (2013)

[52] Singh, S.P., Sutton, R.S.: Reinforcement Learning with Replacing Eligibility Traces. Machine Learning 22(1-3), 123–158 (1996)

[53] Sommerlund, P.: Artificial Neural Nets Applied to Strategic Games. Unpublished, last access: 05.06.2015. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4690 (1996)

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

[55] Sturtevant, N., White, A.: Feature construction for reinforcement learning in hearts. In: Computers and Games, pp. 122–134. Springer (2007)

[56] Sutton, R.S.: Temporal Credit Assignment in Reinforcement Learning. Ph.D. thesis, University of Massachusetts, Amherst, MA (1984)

[57] Sutton, R.S.: Learning to Predict by the Methods of Temporal Differences. Machine Learning 3, 9–44 (1988)

[58] 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. pp. 171–176. AAAI Press, Palo Alto, California (1992)

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

[60] Sutton, R.S., Koop, A., Silver, D.: On the role of tracking in stationary environments. In: Ghahramani, Z. (ed.) Proceedings of the 24th international conference on Machine learning (ICML). pp. 871–878. ACM (2007)

[61] Sutton, Richard S.: Gain adaptation beats least squares? In: Narendra, K.S. (ed.) Proceedings of the 7th Yale Workshop on Adaptive and Learning Systems. pp. 161–166. Center for Systems Science, Yale University (1992)

[62] Szepesvari, C.: Algorithms for Reinforcement Learning. Synthesis lectures on artificial intelligence and machine learning, Morgan & Claypool (2010)

[63] Szubert, M., Jaskowski, W., Krawiec, K.: Coevolutionary temporal difference learning for Othello. In: Proceedings of the 5th international conference on Computational Intelligence and Games. pp. 104–111. IEEE Press, Piscataway, NJ (2009)

[64] Szubert, M.G., Jaskowski, W.: Temporal difference learning of N-tuple networks for the game 2048. In: Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games (CIG). pp. 1–8 (2014)

[65] Tesauro, G.: Temporal difference learning of backgammon strategy. In: Sleeman, D., et al. (eds.) Proceedings of the 9th international workshop on Machine learning (ML). pp. 451–457. Morgan Kaufmann Publishers Inc. (1992)

[66] Tesauro, G.: TD-Gammon, a self-teaching backgammon program, achieves master-level play. Neural Computation 6(2), 215–219 (1994)

[67] 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)

[68] Thill, M., Bagheri, S., Koch, P., Konen, W.: Temporal Difference Learning with Eligibility Traces for the Game Connect Four. In: Proceedings of the 2014 IEEE International Conference on Computational Intelligence and Games (CIG). pp. 586–591. IEEE (2014)

[69] Thill, M., Koch, P., Konen, W.: Reinforcement learning with n-tuples on the game Connect-4. In: Coello Coello, C., et al. (eds.) Proceedings of the 12th International Conference on Parallel Problem Solving from Nature (PPSN). pp. 184–194. Springer, Heidlberg (2012)

[70] Thrun, S.: Learning to Play the Game of Chess. In: Tesauro, G., et al. (eds.) NIPS. pp. 1069–1076. MIT Press (1994)

[71] Tokic, M.: Adaptive epsilon-Greedy Exploration in Reinforcement Learning Based on Value Difference. In: Dillmann, R., et al. (eds.) KI. Lecture Notes in Computer Science, vol. 6359, pp. 203–210. Springer (2010)

[72] Tsitsiklis, J., Roy, B.V.: An Analysis of Temporal-Difference Learning with Function Approximation. IEEE Transactions on Automatic Control 42(5), 674–690 (1997)

[73] Wilson, D.: Dots-and-Boxes Analysis Programs. `http://wilson.engr.wisc.edu/boxes/program.shtml` (2002), last Access: 20.03.2015

[74] Zobrist, A.L.: A new hashing method with application for game playing . Tech. rep., Computer Sciences Department, The University of Wisconsin, Madison (1970)

Appendices

Appendix A

Derivations

A.1 Derivation of the TD(λ) Algorithm

In [59, Sec. 7.2 & 7.3], Sutton & Barto describe two possible views on the TD(λ) algorithm, namely the *forward* and the *backward* view. The forward view is more theoretical and not directly implementable since it includes an acausal target signal. The backward view is more mechanistic and allows the TD(λ) algorithm to be implemented in an incremental manner by utilizing the already mentioned eligibility traces [59, Sec 7.3].
Although the backward view is sufficient for the general understanding of the effects and the advantages of eligibility traces, the derivation of the backward view – starting with the forward view – will be helpful for the derivations of the algorithms nl-IDBD(λ) and SMD(λ) in the Appendices A.3 and A.4.

In the following we derive the backward view of TD(λ), based on Sutton's & Barto's equivalence proof of the forward and backward view [59, Sec. 7.4]. The aim is to derive a weight update-rule that incorporates eligibility traces. Starting point is the sample squared error $\left[T_t^\lambda - V_t(s_t)\right]^2$, which is minimized by gradient descent.

We briefly summarize the relations which emerged from the discussions in Section 3.2.2 and which are required for the later derivations:
The main component in the *forward view* of TD(λ) is the λ-return [59, Sec. 7.2], which is one particular approach to average the n-step returns $R_t^{(n)}$ and which depicts the target value of $V(s_t)$:

$$\begin{aligned} T_t^\lambda &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}T^{(n)} \\ &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\left(R_t^{(n)} + P_t^{(n)}\right) \end{aligned} \tag{A.1}$$

with:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} \tag{A.2}$$

$$P_t^{(n)} = \gamma^n V(s_{t+n}). \tag{A.3}$$

The squared error loss function for the TD(λ) algorithm is given by:

$$L_{mse}^{\lambda}(\vec{w}_t) = \frac{1}{2}\big[T_t^{\lambda} - V_t(s_t)\big]^2. \tag{A.4}$$

The gradient of the loss-function is computed with:

$$\vec{g}_t = \nabla_{\vec{w}_t} L_{mse}^{\lambda}(\vec{w}_t) = \frac{1}{2}\nabla_{\vec{w}_t}\big[T_t^{\lambda} - V_t(s_t)\big]^2 = -\big[T_t^{\lambda} - V_t(s_t)\big]\nabla_{\vec{w}_t} V_t(s_t). \tag{A.5}$$

The backup step, using the λ-return as target value can then be written as:

$$\vec{w}_{t+1} = \vec{w}_t - \alpha\vec{g}_t = \vec{w}_t + \alpha\big[T_t^{\lambda} - V_t(s_t)\big]\nabla_{\vec{w}_t} V_t(s_t), \tag{A.6}$$

The problem with this formulation is that it is acausal, since future rewards are required in order to compute the λ-return in each time step t. One possibility for overcoming this problem is to wait until the episode is completed and then perform a full backup for the whole episode (off-line backup), which can be expressed as:

$$\vec{w}_{T+1} = \vec{w}_T - \alpha\vec{g}_T = \vec{w}_T - \alpha\sum_{t=0}^{\infty}\vec{g}_t, \tag{A.7}$$

where $\vec{g}_T$ is the (negated) recommended weight change (RWC) for the overall episode:

$$\vec{g}_T = \sum_{t}^{\infty}\vec{g}_t = -\sum_{t=0}^{\infty}\big[T_t^{\lambda} - V_t(s_t)\big]\nabla_{\vec{w}_t} V_t(s_t). \tag{A.8}$$

With a few simple transformations we can obtain a causal representation of the above relation, which also allows infinite episode lengths and, more important, which can be implemented in an on-line incremental manner. This representation is referred to as the backward view of TD(λ) [59, Sec. 7.3].

We begin the derivation of the backward view of TD(λ) by simplifying the λ-return. According to Equation (A.1), one can express the λ-return as:

$$T_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)} + (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}P_t^{(n)}. \tag{A.9}$$

The first sum in Equation (A.9) can be re-written, by inserting (A.2) and using the identity

$$\sum_{k=1}^{m}\sum_{j=1}^{k} a_{kj} = \sum_{j=1}^{m}\sum_{k=j}^{m} a_{kj}, \tag{A.10}$$

and several properties of the geometric series in Appendix A.5:

$$\begin{aligned}
(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)} &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\big(r_{t+1}+\gamma r_{t+2}+\gamma^2 r_{t+3}+\cdots+\gamma^{n-1}r_{t+n}\big)\\
&= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\sum_{i=1}^{n}\gamma^{i-1}r_{t+i}\\
&= (1-\lambda)\sum_{n=1}^{\infty}\sum_{i=1}^{n}\lambda^{n-1}\gamma^{i-1}r_{t+i}\\
&= (1-\lambda)\sum_{i=1}^{\infty}\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^{i-1}r_{t+i}\\
&= (1-\lambda)\sum_{i=1}^{\infty}\gamma^{i-1}r_{t+i}\sum_{n=i}^{\infty}\lambda^{n-1}\\
&= (1-\lambda)\sum_{i=1}^{\infty}\gamma^{i-1}r_{t+i}\frac{\lambda^{i-1}}{1-\lambda}\\
&= \sum_{i=1}^{\infty}(\gamma\lambda)^{i-1}r_{t+i}
\end{aligned} \tag{A.11}$$

After a few modifications of the second sum in Equation (A.9), the following result can be obtained:

$$\begin{aligned}
(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}P_t^{(n)} &= \sum_{n=1}^{\infty}\lambda^{n-1}P_t^{(n)} - \lambda^n P_t^{(n)}\\
&= \sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n V(s_{t+n}) - \lambda^n\gamma^n V(s_{t+n})\\
&= \sum_{n=1}^{\infty}(\gamma\lambda)^{n-1}\Big[\gamma V(s_{t+n}) - \gamma\lambda V(s_{t+n})\Big].
\end{aligned} \tag{A.12}$$

Plugging back Equations (A.11) and (A.12) into (A.9) yields:

$$\begin{aligned} T_t^\lambda &= \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} r_{t+n} + \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} \Big[\gamma V(s_{t+n}) - \gamma\lambda V(s_{t+n})\Big] \\ &= \sum_{n=1}^{\infty} \left(r_{t+n} + (\gamma\lambda)^{n-1} \Big[\gamma V(s_{t+n}) - \gamma\lambda V(s_{t+n})\Big] \right). \end{aligned} \tag{A.13}$$

This new representation of the λ-return allows us to re-write the term $T_t^\lambda - V_t(s_t)$ in Equation (A.8), by dragging $-V_t(s_t)$ into the sum (according to [59, Sec. 7.4]):

$$\begin{aligned} T_t^\lambda - V_t(s_t) &= \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} \Big[r_{t+n} + \gamma V(s_{t+n}) - \gamma\lambda V(s_{t+n})\Big] - V_t(s_t) \\ &= \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} \Big[r_{t+n} + \gamma V(s_{t+n}) - \gamma\lambda V(s_{t+n})\Big] - (\gamma\lambda)^0 V_t(s_{t+0}) \\ &= \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} \Big[r_{t+n} + \gamma V(s_{t+n}) - V(s_{t+n-1})\Big]. \end{aligned} \tag{A.14}$$

We can now identify $\delta_{t+n} = r_{t+n+1} + \gamma V(s_{t+n+1}) - V(s_{t+n})$, which is the standard TD error signal as defined in Equation (3.6). By additionally applying an index transformation to the sum, this leads to:

$$\begin{aligned} T_t^\lambda - V_t(s_t) &= \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} \delta_{t+n-1} \\ &= \sum_{i=t}^{\infty} (\gamma\lambda)^{i-t} \delta_i. \end{aligned} \tag{A.15}$$

Combining the above Equation (A.15) with (A.8) allows us to re-formulate the recommended weight change $\vec{g}_T$ – again with the identity (A.10) – in the following way:

$$\begin{aligned}
-\vec{g}_T &= \sum_{t=0}^{\infty} \big[T_t^\lambda - V_t(s_t)\big] \nabla_{\vec{w}_t} V_t(s_t) \\
&= \sum_{t=0}^{\infty} \sum_{i=t}^{\infty} (\gamma\lambda)^{i-t} \delta_i \nabla_{\vec{w}_t} V_t(s_t) \\
&= \sum_{i=0}^{\infty} \sum_{t=0}^{i} (\gamma\lambda)^{i-t} \delta_i \nabla_{\vec{w}_t} V_t(s_t) \\
&= \sum_{i=0}^{\infty} \delta_i \sum_{t=0}^{i} (\gamma\lambda)^{i-t} \nabla_{\vec{w}_t} V_t(s_t) \\
&= \sum_{t=0}^{\infty} \delta_t \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \nabla_{\vec{w}_k} V_k(s_k).
\end{aligned} \tag{A.16}$$

In the last row of above equation, the index labels were simply interchanged. For tasks with finite episodes, per definition $\delta_t = 0$ for $t \leq T$, which results in a small modification of the sum-boundaries in (A.16):

$$\begin{aligned}
\vec{g}_T &= -\sum_{t=0}^{T-1} \delta_t \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \nabla_{\vec{w}_k} V_k(s_k) \\
&= -\sum_{t=0}^{T-1} \delta_t \vec{e}_t,
\end{aligned} \tag{A.17}$$

where $\vec{e}_t$ is the so called eligibility trace vector. The above definition of the episodic recommended weight change $\vec{g}_T$ can now again be transformed into an iterative form, which allows a backup in every time step t and therefore an on-line implementation of the TD(λ) algorithm. The corresponding recommended weight change (gradient) g_t for each step is given by:

$$\begin{aligned}
\vec{g}_t &= \nabla_{\vec{w}_t} L_{MSE}(\vec{w}_t) = \frac{1}{2} \nabla_{\vec{w}_t} \big[T_t^\lambda - V_t(s_t)\big]^2 = -\big[T_t^\lambda - V_t(s_t)\big] \nabla_{\vec{w}_t} V_t(s_t) \\
&= -\delta_t \vec{e}_t.
\end{aligned} \tag{A.18}$$

With the following relation it is possible to compute the eligibility trace vector in an iterative manner:

$$\begin{aligned} \vec{e}_t &= \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{\vec{w}} V_k(s_k) \\ &= \lambda\gamma\vec{e}_{t-1} + \nabla_{\vec{w}} V_t(s_t), \\ \vec{e}_0 &= \nabla_{\vec{w}} V(s_0). \end{aligned} \tag{A.19}$$

The trace decay parameter λ and the discount parameter γ decay (or discount) 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.
Finally, we can express the update rule of the TD(λ) algorithm, by inserting (A.18) back into (A.6):

$$\vec{w}_{t+1} = \vec{w}_t + \alpha\delta_t\vec{e}_t. \tag{A.20}$$

In this appendix, we derived the relations (A.19) and (A.20), which are the central elements of the TD(λ) algorithm. Furthermore, the new definition of the (negated) recommended weight change $\vec{g}_t$ in Equation (A.18) will be important for later derivations in the Appendices A.3 and A.4.

A.2 Derivation of nl-IDBD

In [34, pp. 31] & [60], it is not explicitly shown how the gradient of the cross-entropy loss (which is used as objective function in Koop's non-linear version of the IDBD algorithm) is derived, therefore, we briefly illustrate in this appendix how to compute the gradient and we summarize the update rules of nl-IDBD.

Koop's nl-IDBD algorithm uses a logistic sigmoid squashing function σ in the output:

$$V_t(s_t) = \sigma\big(a(s_t)\big), \tag{A.21}$$

$$a(s_t) = \vec{w}_t^T \vec{x}(s_t), \tag{A.22}$$

$$\sigma(\nu) = \frac{1}{1 + \mathrm{e}^{-\nu}}. \tag{A.23}$$

Instead of the MSE, the loss is measured based on the cross-entropy between the target signal $T_t = T_t(s_t)$ and the current prediction $V_t = V_t(s_t)$:

$$L_{CE}(\vec{w}_t) = -T_t \log\big(V_t\big) - (1 - T_t) \log\big(1 - V_t\big), \tag{A.24}$$

Based on the above loss function, required for a weight update in the direction of the steepest descent, can be computed as follows:

$$\begin{aligned}
-\frac{\partial L_{CE}(\vec{w}_t)}{\partial \vec{w}_t} &= T_t \frac{\partial \log(V_t)}{\partial \vec{w}_t} + (1 - T_t) \frac{\partial \log(1 - V_t)}{\partial \vec{w}_t} \\
&= T_t \frac{1}{V_t} \frac{\partial V_t}{\partial \vec{w}_t} + (1 - T_t) \frac{1}{1 - V_t} \frac{\partial (-V_t)}{\partial \vec{w}_t} \\
&= \frac{T_t}{V_t} V_t (1 - V_t) \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t - \frac{1 - T_t}{1 - V_t} V_t (1 - V_t) \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t \\
&= T_t (1 - V_t) \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t - (1 - T_t) V_t \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t \\
&= \Big[T_t (1 - V_t) - (1 - T_t) V_t \Big] \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t \\
&= \Big[T_t - V_t \Big] \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t \\
&= \Big[T_t - V_t \Big] \vec{x}_t.
\end{aligned} \tag{A.25}$$

The derivations for the memory traces $\vec{\beta}$ and $\vec{h}$ can be found in [34, pp. 31]. With $\delta_t = T_t - V_t$, the update rules for nl-IDBD can be summarized as follows:

$$\begin{aligned} \beta_{i,t+1} &= \beta_{i,t} + \theta \delta_t x_i h_i \\ \alpha_{i,t+1} &= \mathrm{e}^{\beta_{i,t+1}} \\ w_{i,t+1} &= w_{i,t} + \alpha_{i,t+1} \delta_t x_{i,t} \\ h_{i,t+1} &= h_{i,t} \big[1 - \alpha_{i,t} x_{i,t}^2 \sigma_t'(a)\big]^+ + \alpha_{i,t} \delta_t x_{i,t} \end{aligned} \tag{A.26}$$

where

$$\sigma_t'(a) = V_t \cdot \big(1 - V_t\big). \tag{A.27}$$

The above update rules are nearly the same as before for the linear case, only the decaying trace h_i (line 9 in Algorithm 3) has to be adjusted slightly; the general procedure and operational order of nl-IDBD remains the same as in Algorithm 3.

A.3 Derivation of nl-IDBD(λ)

The nl-IDBD algorithm [34, pp. 31], [60] is originally only defined for supervised learning tasks and cannot be used in conjunction with the TD(λ) algorithm, since eligibility traces are not supported. In this appendix we extend Koop's nl-IDBD to eligibility traces.

In a naive approach, one could simply replace the input vector $\vec{x}_t$ in the defined update rules by the eligibility trace vector $\vec{e}_t$. This is, however, not correct. The key idea in order to derive correct update rules is to use the λ-return T_t^λ (Section 3.2.2) as target signal instead of the standard TD target $T_t = r_{t+1} + \gamma V_t(s_{t+1})$. At first glance, this may appear not straightforward, since the provided target signal T_t^λ in the forward view of TD(λ) is acausal, as already discussed in Section 3.2.2. For this reason, the forward view is not directly implementable. To overcome this issue, eligibility traces were introduced in a backward view of TD(λ), yielding a causal, implementable formulation of the algorithm. Similar to the explanations in Section 3.2.2, we can also re-formulate the cross-entropy loss of nl-IDBD with the λ-return as target value and then subsequently derive (in Appendix A.3) new update rules for $\vec{w}_t$, $\vec{\beta}_t$ and $\vec{h}_t$: Similarly to the derivations in Appendix A.2, the gradient of the loss is given by

$$\frac{\partial L_{CE}^\lambda(\vec{w}_t)}{\partial \vec{w}_t} = -\Big[T_t^\lambda - V_t\Big] \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t = -\Big[T_t^\lambda - V_t\Big] \vec{x}_t, \tag{A.28}$$

with the difference, that now the target signal $T_t = T_t^\lambda$ is used. A comparison of (A.28) with (A.18) and (A.19) reveals the relation:

$$\frac{\partial L_{CE}^\lambda(\vec{w}_t)}{\partial \vec{w}_t} = -\delta_t \vec{e}_t, \tag{A.29}$$

where the eligibility trace vector is now:

$$\begin{aligned} \vec{e}_t &= \lambda\gamma\vec{e}_{t-1} + \vec{x}_t \\ \vec{e}_0 &= \vec{x}_0. \end{aligned} \tag{A.30}$$

The weights of the system can therefore be updated according to the following rule:

$$w_{i,t+1} = w_{i,t} - \alpha_{i,t} \frac{\partial L_{CE}^\lambda(\vec{w}_t)}{\partial w_{i,t}} = w_{i,t} + \alpha_{i,t}\delta_t e_{i,t}, \tag{A.31}$$

which is identical to the backup performed in the plain TD(λ) algorithm with a linear approximation function. For $\lambda = 0$ we again retrieve the same weight-update rule as before in (A.26). Similarly to [34, p. 32] and [58, 60], the β-update rule can be derived, using

Leibniz's notation and inserting (A.29)[1]:

$$\begin{aligned}\vec{\beta}_{t+1} &= \vec{\beta}_t - \theta \frac{\partial L^{\lambda}_{CE}(\vec{w}_t)}{\partial \vec{\beta}} \\ &= \vec{\beta}_t - \theta \frac{\partial \vec{w}_t}{\partial \vec{\beta}} \frac{\partial L^{\lambda}_{CE}(\vec{w}_t)}{\partial \vec{w}_t} \\ &= \vec{\beta}_t + \theta \delta_t \frac{\partial \vec{w}_t}{\partial \vec{\beta}} \vec{e}_t.\end{aligned} \tag{A.32}$$

where the remaining vector-by-vector derivative (both vectors have the length n) generates a $n \times n$ matrix:

$$\frac{\partial \vec{w}_t}{\partial \vec{\beta}} = \begin{bmatrix} \frac{\partial w_{1,t}}{\partial \beta_1} & \frac{\partial w_{2,t}}{\partial \beta_1} & \cdots & \frac{\partial w_{n,t}}{\partial \beta_1} \\ \frac{\partial w_{1,t}}{\partial \beta_2} & \frac{\partial w_{2,t}}{\partial \beta_2} & \cdots & \frac{\partial w_{n,t}}{\partial \beta_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_{1,t}}{\partial \beta_n} & \frac{\partial w_{2,t}}{\partial \beta_n} & \cdots & \frac{\partial w_{n,t}}{\partial \beta_n} \end{bmatrix}. \tag{A.33}$$

The individual trace β_i is then updated with following rule:

$$\begin{aligned}\beta_{i,t+1} &= \beta_{i,t} + \theta \delta_t \sum_{j=1}^{n} \frac{\partial w_{j,t} e_{j,t}}{\partial \beta_i} \\ &\approx \beta_{i,t} + \theta \delta_t e_{i,t} \frac{\partial w_{i,t}}{\partial \beta_i} \\ &= \beta_{i,t} + \theta \delta_t e_{i,t} h_{i,t}\end{aligned} \tag{A.34}$$

As in [34, pp. 32], [58, 60], the approximation here is performed under the assumption, that the effect on weight w_j for changes of the i-th learning rate should be negligible for $i \neq j$. The update rule for $h_{i,t} = \frac{\partial w_{i,t}}{\partial \beta_i}$ can be derived analogous to [34, p. 33], [58], using

[1] Throughout this thesis the denominator layout convention is used for the matrix calculus.

the product rule of calculus and $V_{t+1} = V_t(s_{t+1})$:

$$\begin{aligned}
h_{i,t+1} &= \frac{\partial w_{i,t+1}}{\partial \beta_i} \\
&= \frac{\partial}{\partial \beta_i}(w_{i,t} + \mathrm{e}^{\beta_{i,t+1}} \delta_t e_{i,t}) \\
&= \frac{\partial w_{i,t}}{\partial \beta_i} + \mathrm{e}^{\beta_{i,t+1}} \delta_t e_{i,t} + \mathrm{e}^{\beta_{i,t+1}} e_{i,t} \frac{\partial \delta_t}{\partial \beta_i} \\
&= h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} + \alpha_{i,t+1} e_{i,t} \frac{\partial}{\partial \beta_i}[r_{t+1} + \gamma V_{t+1} - V_t] \\
&= h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} - \alpha_{i,t+1} e_{i,t} \frac{\partial V_t}{\partial \beta_i} \\
&= h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} - \alpha_{i,t+1} e_{i,t} \frac{\partial V_t}{\partial a_t} \frac{\partial a_t}{\partial \beta_i} \\
&= h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} - \alpha_{i,t+1} e_{i,t} V_t(1 - V_t) \frac{\partial \vec{w}_t^T \vec{x}_t}{\partial \beta_i} \\
&= h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} - \alpha_{i,t+1} e_{i,t} V_t(1 - V_t) \sum_{j=1}^{n} \frac{\partial w_{j,t} x_{j,t}}{\partial \beta_i}.
\end{aligned} \tag{A.35}$$

By using the same approximation as in (A.34), we can write

$$\begin{aligned}
h_{i,t+1} &\approx h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} - \alpha_{i,t+1} e_{i,t} V_t(1 - V_t) x_{i,t} \frac{\partial w_{i,t}}{\partial \beta_i} \\
&= h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} - \alpha_{i,t+1} e_{i,t} V_t(1 - V_t) x_{i,t} h_{i,t} \\
&= [1 - \alpha_{i,t+1} V_t(1 - V_t) x_{i,t} e_{i,t}] h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t}.
\end{aligned} \tag{A.36}$$

As before, $h_{i,t}$ is a decaying trace, although the factor $[1 - \alpha_{i,t+1} V_t(1 - V_t) x_{i,t} e_{i,t}]$ only then causes a decay, when the corresponding input x_i is active. This could prevent a too large decrease of h_i in a short time, since h_i is adjusted in each time step of the remaining episode, once the trace e_i is triggered.
Finally, after adding the positive bounding rule to (A.36), we have everything together for the nl-IDBD algorithm with eligibility traces, which in the following will be denoted as nl-IDBD(λ):

$$\begin{aligned}
\beta_{i,t+1} &= \beta_{i,t} + \theta \delta_t e_{i,t} h_{i,t} \\
\alpha_{i,t+1} &= \mathrm{e}^{\beta_{i,t+1}} \\
w_{i,t+1} &= w_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} \\
h_{i,t+1} &= [1 - \alpha_{i,t+1} V_t(1 - V_t) x_{i,t} e_{i,t}]^+ h_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t} \\
e_{i,t+1} &= \lambda \gamma e_{i,t} + x_{i,t+1}.
\end{aligned} \tag{A.37}$$

The operational order should be kept as specified above. Note, that we again retrieve Koop's & Sutton's original update rules if we set $\lambda = 0$.

In this appendix, we derived slightly modified update rules for Koop's nl-IDBD algorithm [34, pp. 31], [60], resulting in the nl-IDBD(λ) algorithm, which can be used in conjunction with the TD(λ) algorithm, hence, incorporates eligibility traces.

A.4 Derivation of SMD(λ)

In this appendix we derive update rules from Schraudolph's Stochastic Meta Descent (SMD) [48, 49] algorithm, which can be used together with the TD(λ) algorithm and which are – similarly to Koop's nl-IDBD [34, 60] – based on the cross-entropy loss function. The general update rules of the original SMD algorithm are shown in (A.39). Since SMD is not restricted to a specific objective function and value function approximation, one has to specify these and subsequently, compute the gradient $\vec{g}_t$ and the instantaneous Hessian H_t and insert the results into (A.39).

In Sutton's derivation of the IDBD algorithm, the instantaneous Hessian $H_t = \vec{x}_t \vec{x}_t^T$ is diagonalized to $H_t \approx \text{Diag}(\vec{x}_t \vec{x}_t^T)$ under the assumption, that the effect on weight w_j for changes of the i-th learning rate should be negligible for $i \neq j$ [48, 49]. This approximation allows to save some computation time in every time step. When using the exact Hessian, the update rule for the trace h_i changes to:

$$\begin{aligned} h_{i,t+1} &= h_{i,t} - x_{i,t}\alpha_{i,t} \sum_j x_{j,t} h_{j,t} + x_{i,t}\alpha_{i,t}\delta_{i,t} \\ &= h_{i,t} + x_{i,t}\alpha_{i,t}\Big(\delta_{i,t} - \sum_j x_{j,t} h_{j,t}\Big) \end{aligned} \tag{A.38}$$

where the sum adds up $x_j h_j$ for all active inputs of the system. It is sufficient to evaluate the sum only once in every time step, so that the above formulation can be used for systems with sparse activation without significantly increasing the computation time. By neglecting all terms of the sum for $j \neq i$ and adding the positive bounding rule, we again get Sutton's original update rule.
With a few modifications of (A.38), one can obtain Schraudolph's SMD (Stochastic Meta-Descent) algorithm [47, 49], which is basically an extension of Sutton's IDBD:

$$\begin{aligned} \vec{\alpha}_{t+1} &= \vec{\alpha}_t \text{max}\Big(\frac{1}{2}, 1 - \theta \vec{g}_t^D \vec{h}_t\Big), \\ \vec{w}_{t+1} &= \vec{w}_t - \vec{\alpha}_{t+1}^D \vec{g}_t, \\ \vec{h}_{t+1} &= \mu \vec{h}_t - \mu \vec{\alpha}_{t+1}^D (H_t \vec{h}_t) - \vec{\alpha}_{t+1}^D \vec{g}_t, \end{aligned} \tag{A.39}$$

where the operation $\vec{v}^D$ generates a diagonal matrix[2] of $\vec{v}$, $\mu \in [0, 1]$ is a discount factor for non-stationary tasks, $\vec{g}_t$ is the gradient $\nabla_{\vec{w}_t} L(w_t)$ of a suitable objective (loss) function L (such as the MSE) and H_t is the square Hessian matrix, containing the second-order partial derivatives of the objective function, with $H_t = \frac{\partial^2 L(\vec{w}_t)}{\partial^2 \vec{w}_t}$.

[2]This is required here in order perform a component-wise multiplication of two vectors. Other notations may be confusing or misleading, e.g. the notation used in [48] may be misunderstood as a dot-product of the two vectors, which results in a scalar value.

Schraudolph replaced the exponential function for computing the step-sizes in $\vec{\alpha}_t$ by the approximation (first-order Taylor expansion about the point zero)

$$\mathrm{e}^u \approx \max(\frac{1}{2}, 1+u), \tag{A.40}$$

in order to avoid costly exponentiation operations and reduce the effect of outliers for $|u| \gg 0$ [49].

In this work, we use a similar setup for the SMD algorithm as for Koop's nl-IDBD (derived in Appendix A.2), which uses a logistic sigmoid squashing function in the output:

$$V_t(s_t) = \sigma\big(a(s_t)\big), \tag{A.41}$$
$$a(s_t) = \vec{w}_t^T \vec{x}(s_t), \tag{A.42}$$
$$\sigma(\nu) = \frac{1}{1+\mathrm{e}^{-\nu}}. \tag{A.43}$$

Furthermore, we choose the loss function to be the cross-entropy loss as described in Section 3.5.2, with the λ-return (Section 3.2.2) as target signal:

$$\frac{\partial L_{CE}^{\lambda}(\vec{w}_t)}{\partial \vec{w}_t} = -\Big[T_t^{\lambda} - V_t\Big] \frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t = -\Big[T_t^{\lambda} - V_t\Big] \vec{x}_t, \tag{A.44}$$

According to Equation (A.29), the gradient of the loss is:

$$\vec{g}_t = \frac{\partial L_{CE}^{\lambda}(\vec{w}_t)}{\partial \vec{w}_t} = -\delta_t \vec{e}_t, \tag{A.45}$$

with the eligibility trace vector, which is updated as:

$$\begin{aligned} \vec{e}_t &= \lambda\gamma\vec{e}_{t-1} + \vec{x}_t \\ \vec{e}_0 &= \vec{x}_0. \end{aligned} \tag{A.46}$$

The Hessian matrix can be computed as follows[3], with $V_t = V_t(s_t)$ and $V_{t+1} = V_t(s_{t+1})$:

$$\begin{aligned} H_t &= \frac{\partial^2 L(\vec{w}_t)}{\partial \vec{w}_t \partial \vec{w}_t^T} = \frac{\partial}{\partial \vec{w}_t^T} \vec{g}_t \\ &= -\frac{\partial}{\partial \vec{w}_t^T} \delta_t \vec{e}_t \\ &= -\frac{\partial}{\partial \vec{w}_t^T} \Big(V_{t+1} + r_{t+1} - V_t\Big) \vec{e}_t \end{aligned}$$

[3]Throughout this work we use the denominator layout for matrix calculus.

$$
\begin{aligned}
&= \frac{\partial}{\partial \vec{w}_t^T} V_t \vec{e}_t \\
&= V_t(1-V_t)\Big(\frac{\partial}{\partial \vec{w}_t} \vec{w}_t^T \vec{x}_t \vec{e}_t\Big)^T \\
&= V_t(1-V_t)\Big(\vec{w}_t^T \vec{x}_t \frac{\partial \vec{e}_t}{\partial \vec{w}_t} + \frac{\partial \vec{w}_t^T \vec{x}_t}{\partial \vec{w}_t} \vec{e}_t^T\Big)^T \\
&= V_t(1-V_t)\Big(0 + \vec{x}_t \vec{e}_t^T\Big)^T \\
&= V_t(1-V_t)\vec{e}_t \vec{x}_t^T
\end{aligned} \tag{A.47}
$$

Inserting the gradient from (A.45) and the Hessian from (A.47) back into (A.39) yields a set of update-rules, which are as follows:

$$
\begin{aligned}
\vec{\alpha}_{t+1} &= \vec{\alpha}_t \max\Big(\frac{1}{2}, 1 + \theta \delta_t \vec{e}_t^D \vec{h}_t\Big), \\
\vec{w}_{t+1} &= \vec{w}_t + \vec{\alpha}_{t+1}^D \delta_t \vec{e}_t, \\
\vec{h}_{t+1} &= \mu \vec{h}_t - \mu \vec{\alpha}_{t+1}^D V_t(1-V_t)\vec{e}_t \vec{x}_t^T \vec{h}_t + \vec{\alpha}_{t+1}^D \delta_t \vec{e}_t, \\
\vec{e}_{t+1} &= \lambda \gamma \vec{e}_t + \vec{x}_{t+1},
\end{aligned} \tag{A.48}
$$

In vector-element form, the rules can be expressed as:

$$
\begin{aligned}
\alpha_{i,t+1} &= \alpha_{i,t} \max\Big(\frac{1}{2}, 1 + \theta \delta_t e_{i,t} h_{i,t}\Big), \\
w_{i,t+1} &= w_{i,t} + \alpha_{i,t+1} \delta_t e_{i,t}, \\
h_{i,t+1} &= \mu h_{i,t} - \mu \alpha_{i,t+1} V_t(1-V_t) e_{i,t} \sum_j h_{j,t} x_{j,t} + \alpha_{i,t+1} \delta_t e_{i,t}, \\
e_{i,t+1} &= \lambda \gamma e_{i,t} + x_{i,t+1}.
\end{aligned} \tag{A.49}
$$

The operational order should be kept as specified above. In Algorithm 5 it is described how SMD(λ) can be used in conjunction with the incremental TD(λ) algorithm for board games (Algorithm 5).

In this appendix we specified the update rules for the SMD(λ) algorithm, which can be derived from the original update rules of the SMD algorithm, once a suitable loss function (here: the cross-entropy loss) and parametrized approximation function (here: linear network & with a logistic sigmoid function as activation function) are found.

Algorithm 5 Adjustment of the incremental TD(λ) algorithm for board games in Algorithm 1 when used in conjunction with SMD(λ). In the original algorithm, the lines 21 – 27 have to be replaced by the following pseudo-code:

```
1 for (every weight index i) do
2     if (q ≥ ε  or  s_{t+1} ∈ S_Final) then
3         α_{i,t+1} ← α_{i,t} max(1/2, 1 + θ δ_t e_{i,t} h_{i,t})
4         w_{i,t+1} ← w_{i,t} + α_{i,t+1} δ_t e_{i,t}
5         h_{i,t+1} ← μ h_{i,t} − μ α_{i,t+1} V_t (1 − V_t) e_{i,t} Σ_j h_{j,t} x_{j,t} + α_{i,t+1} δ_t e_{i,t}
6     end if
7     Δe_i ← x_i(s_{t+1})
8     e_{i,t+1} ← { Δe_i,            if x_i(s_{t+1}) ≠ 0 ∧ REP
                  { γλ e_{i,t} + Δe_i, otherwise
9 end for
```

A.5 Geometric Series

In this appendix we briefly mention several properties of the geometric series which are required for the derivations in other dependencies. For $q \neq 0$ and $j < N$ the geometric series can be derived as:

$$\begin{aligned} S &= q^j + q^{j+1} + \cdots + q^{N-1} \\ qS &= q^{j+1} + q^{j+2} + \cdots + q^N \\ qS - S &= q^N - q^j \\ S(q-1) &= q^N - q^j \\ S &= \frac{q^N - q^j}{q-1} = \frac{q^j - q^N}{1-q} \end{aligned} \tag{A.50}$$

More generally, the geometric series can be written as:

$$a_0 \sum_{i=j}^{N-1} q^i = a_0 \frac{q^N - q^j}{q-1} \tag{A.51}$$

Special cases If the series starts with $i = 0$ we retrieve:

$$a_0 \sum_{i=0}^{N-1} q^i = a_0 \frac{1 - q^N}{1-q} \tag{A.52}$$

For $q = 1$ we obtain:

$$a_0 \sum_{i=j}^{N-1} 1^i = a_0 (N - j) \tag{A.53}$$

For an infinite series with $N = \infty$, convergence is achieved for values $|q| < 1$:

$$a_0 \sum_{i=j}^{\infty} q^i = a_0 \frac{q^j}{1-q} \tag{A.54}$$

Appendix B

Connect-4 Game Playing Framework

In the following we briefly describe the Connect-4 game playing framework (C4GPF) that emerged from the work on several studies [69, 67, 68, 8] over the last years. The framework provides a GUI with which the user can train, interact with and measure the strength of various agents and more. The whole software is written in Java and can be easily extended by experienced programmers. The Java framework for Connect-4 is available as open source from GitHub and can be downloaded from:

`http://github.com/MarkusThill/Connect-Four`.

Authors

- Markus Thill (markus.thill "at" fh-koeln.de)
- Wolfgang Konen (wolfgang.konen "at" fh-koeln.de)

Features

- Direct interaction between user and agents: By playing against different agents, the user can get an own impression of the playing strength of the individual agents.
- Animated matches between different agents: The user can select two agents and follow matches of these two agents against each other on the board. If desired, both agents interact automatically with each other and the user can watch the animated match. It is also possible to use a step-by-step mode in which the user decides when an agent performs the next move.
- Due to the fast inbuilt Minimax agent, the user can analyze the exact game-theoretic values and state-action values of arbitrary positions in a matter of a few milliseconds.
- Benchmark options: It is possible to set up competitions between agents, where both opponents play a certain number of matches against each other (swapping the sides after each match if desired). After the competition is completed the user is provided with some statistics about the competition and the user can then also analyze individual matches.

Another option is to use a simple benchmark for an agent, where the framework determines a scalar value, indicating the strength of the agent, based on a predefined number of matches against a perfect playing agent. This benchmark can also be used to estimate the strength of completely deterministic agents.

- Built-in Reinforcement Learning (RL) agent, as described below, which learns with n-tuple systems.
- Simple tools for visualizing, creating, adjusting and deleting n-tuples.
- GUI-based inspection of the look-up tables (LUTs) of the n-tuple system.
- Loading/saving of parameter files, which contain the exact settings of an agent. These files can be used in order to start a training process with pre-defined settings.
- Trained agents with all look-up tables and all other configurations can be saved to and loaded from small compressed files.
- Import/Export only the weights of an n-Tuple RL-Agent from/to a compressed ZIP file.
- Already included pre-trained agents and pre-defined parameter files.
- Help file that can be called from the GUI.

Agents The framework already provides several inbuilt agents which can be selected as opponents for the user or which can play matches against each other. It is fairly simple for other developers to plug in new agents, if the interface "Agent.java" is used.

- Random Agent: Plays completely random moves and can be constantly defeated even by weak agents.
- Monte Carlo Tree Search (MCTS) [17]: A fairly simple agent in an early development phase, based on the general MCTS approach, without any special enhancements. With 500 000 iterations, the playing strength is comparable to a standard Minimax agent with a search depth of depth 12.
- Perfect playing Minimax agent supported by a 8-ply and 12-ply database and a transposition table (25 MB by default) and many other enhancements [67]. Currently, this agent is one of the fastest tree-search algorithms available for Connect-4: Even without the support of the databases, the best move for the empty board can be found in less than 4 minutes on a Pentium-4 computer (using only a 25 MB transposition table). The strength of the agent can be controlled by the search-depth.

- Reinforcement Learning (RL) agent: The agent is based on an n-tuple system [36] and trained with TD-Learning. The TD-Learning algorithm supports eligibility traces and can be augmented with several step-size adaptation algorithms, such as TCL, IDBD, Autostep and others (listed below).
- RL-Minimax agent: Performs a classical tree search. On the leafs of the tree an RL agent is used to estimate the corresponding state value.

Step-size Adaptation Algorithms All step-size adaptation algorithms that are described in this thesis can be selected and configured by the user in the C4GPF (as shown in Figure B.2). At the time this work was created, the following step-size adaptation algorithms were supported by our framework: α-bounds [22] , Autostep [39, 38] , IDBD [58] , nl-IDBD [60, 34] , nl-IDBD(λ) A.3 , K1 [61] , ELK1 [48] , TCL [14, 13] , TCL-EXP [8] , SMD [48, 49] , SMD(λ) (as described in Appendix A.4) , RProp [42] , ALAP [5].

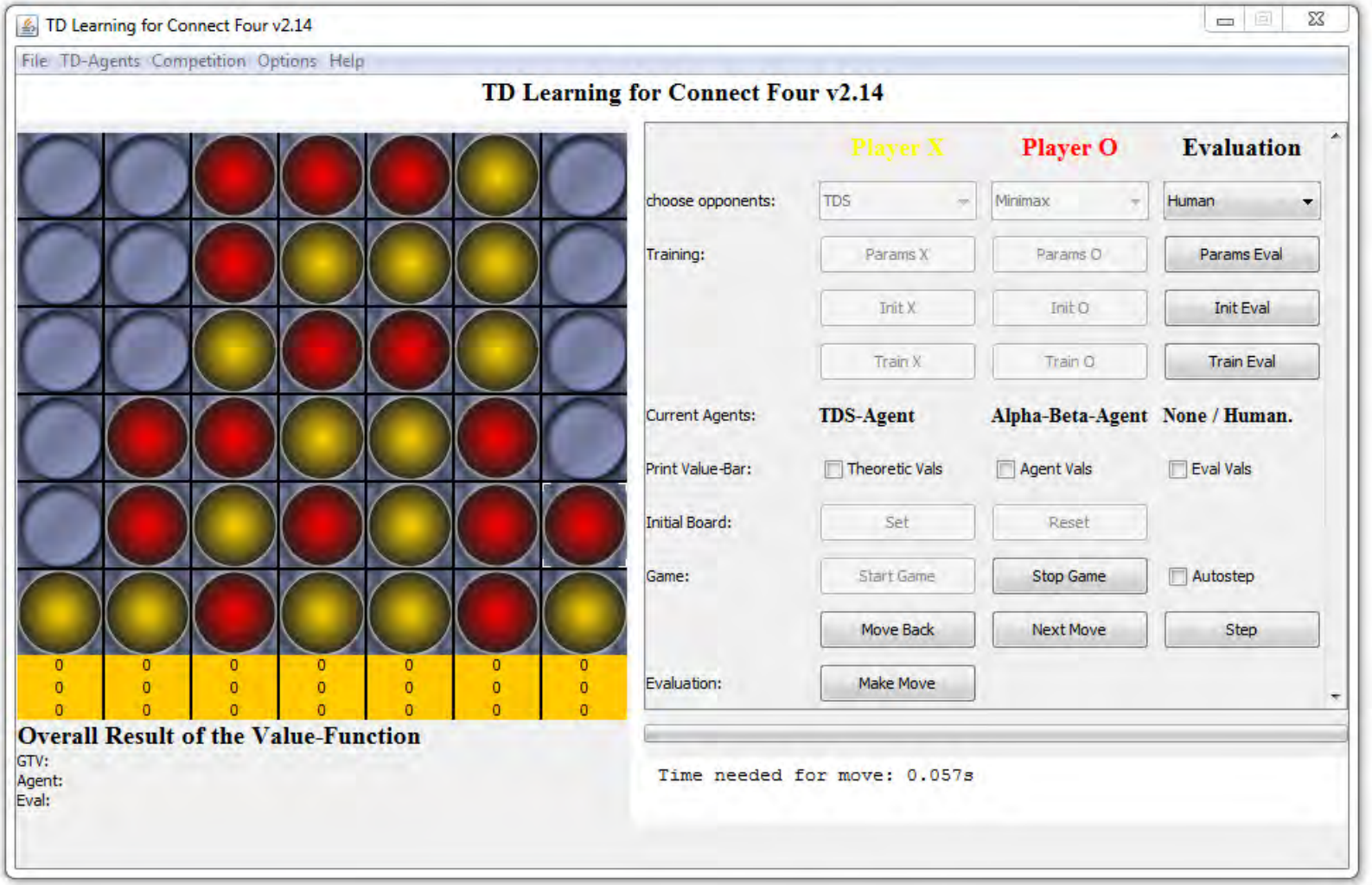

Figure B.1: Main window of the Connect-4 Game Playing Framework (C4GPF).

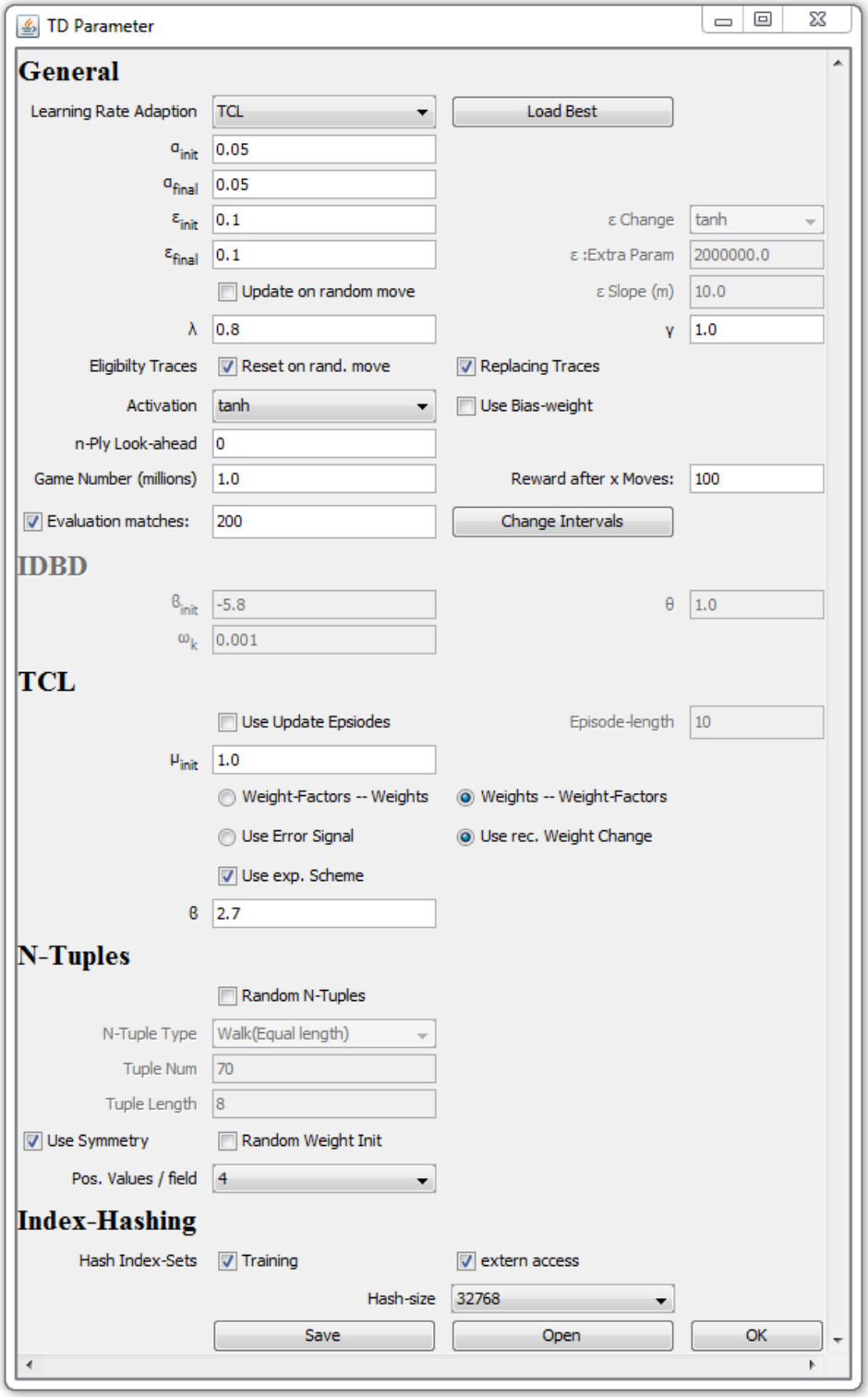

Figure B.2: TD-parameter window of the Connect-4 Game Playing Framework (C4GPF).

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.
Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht.
Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.
Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

(Ort, Datum) Markus Thill