Dedicated TD-learning for Stronger Gameplay: applications to Go

R. Ekker[1] *r.ekker@ai.rug.nl*

E.C.D. van der Werf[2] *E.vanderWerf@cs.unimaas.nl*

L.R.B. Schomaker[1] *l.schomaker@ai.rug.nl*

[1]Artificial Intelligence, Rijksuniversiteit Groningen

[2] IKAT, Department of Computer Science, Universiteit Maastricht

Abstract

This paper presents a study of several dedicated Temporal-Difference (TD) learning algorithms for deterministic zero-sum games of perfect information such as the Game of Go. The algorithms include TD(μ) by Beal (2002), which separates good play from bad play, TD-leaf(λ) and TD-directed(λ) by Baxter *et al.* (1998), which exploit game tree searching, and Baird's residual algorithms (1995), for preventing instability during training. We show that dedicated TD learning algorithms provide faster training and acquire more 'genuine' knowledge of the game resulting in a significantly higher playing strength than players trained by standard TD.

1 Introduction

Games have a number of properties that make them an excellent environment for the application of Temporal Difference (TD) Learning. Game dynamics are usually clear and simple, large numbers of trials can be generated in fast and cheap ways, and yet games can give us a very rich and complex learning environment. However, TD learning does not take optimal advantage of the structure of deterministic perfect information games like Chess and Go. Although the properties of these games make it possible to use sophisticated reasoning (like minimax search), TD learning only uses a stochastical model of the environment. The question arises whether adapting Temporal Difference Learning algorithms to deal with the specifics of gameplay will improve learning performance.

Several adaptations to Temporal Difference Learning for application in the game-playing domain have been described. The TD-leaf(λ) algorithm (Baxter et al., 1998) focuses on using game-tree search with TD learning, the TD(μ) algorithm (Beal, 2002) deals with learning from suboptimal play, Baird's residual algorithms (1995) prevent instability during training, and the RPROP algorithm (Riedmiller and Braun, 1993) provides significantly faster training compared to standard network training algorithms. In this paper we compare the performance of these algorithms, and test them by training a neural network for playing Go on a small board against a weak opponent. To our knowledge this is the first time that the performance of TD(μ) is compared to that of other dedicated algorithms.

We will now introduce TD learning and the adapted algorithms. Then our experimental setup is described, followed by the results and a short discussion of our findings.

2 Temporal-Difference Learning

This section presents a brief description of the TD(λ) algorithm (for more information see: Sutton and Barto, 1998). Consider the case of an agent learning a state evaluation function $V(s)$, predicting the sum of future rewards, which should be maximised. At every time step t, the agent visits a state s_t, chooses an action, and receives a reward r_t. In the case of a 2-player game, the action is the move to play, and the next state depends upon the rules of the game and the opponent's move. Depending on that action and the world's dynamics, the agent winds up in a next state s_{t+1}.

For improving the predictions of $V(s)$ a quantity called the λ-*return* R_t^λ is defined as follows:

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\left(V(s_{t+n}) + \sum_{m=1}^{n} r_{t+m}\right) \quad (1)$$

The λ-return is commonly used to balance n-step returns expressed in the forward view of TD() (Sutton and Barto, 1998). The value of λ should be chosen between 0 and 1 because it generally provides faster convergence than the extreme cases of TD(0), which only looks at the 1-step return, and TD(1), which is just the Monte Carlo backup.

For learning the simple tabular case where all possible states can be stored independently (no generalisation) the value for a state is then updated with a learning rate α according to:

$$\Delta V(s_t) = \alpha[R_t^\lambda - V(s_t)] \tag{2}$$

For most interesting domains, which have a large state space, independently storing $V(s)$ for all possible states is not an option. As an alternative general function approximators are used to approximate $V(s)$ for all possible states. For well-chosen representations this also provides generalisation to unseen states.

The function V that is learned by TD(λ) is a model of the world, telling us how much future reward is to be expected in a state. In its simplest form, this model does not contain an explicit model of the opponent nor does it use reasoning or tree searching. The adapted algorithms we describe in this paper *do* contain these kinds of functionality.

3 Temporal Difference Learning and Game Tree Search

Baxter et al. (1998) describe two new TD learning variants, TD-directed(λ) and TD-leaf(λ), for use with game tree search. Their results show both algorithms to be more efficient for training the KNIGHTCAP Chess-playing program than standard TD(λ).

3.1 TD-directed(λ)

TD-directed(λ) uses minimax search to select moves, and as such it can be viewed as TD(λ) with a policy that searches the gamespace. TD-directed(λ) is the most straight-forward way of applying TD(λ) when learning gameplay with minimax search.

For TD-directed(λ), a minimax search is used to guide play, but the evaluation and updating of play is just like normal TD(λ). It might seem logical to use the minimax-value of a state as the evaluation value for TD-directed(λ), but this is not the correct approach. The minimax-value of the state s is the evaluation V of the leaf node of its principal variation s_{pv}. (The principal variation is the sequence of moves that the program thinks represents best play for both sides.) $V(s_{pv})$ is generally a more reliable prediction than $V(s)$. However, we cannot simply replace $V(s)$ by $V(s_{pv})$ to compute the TD error of equation (2) because the feature set of state s does not match the feature set of state s_{pv}. The correct way to implement the TD-directed(λ) algorithm is therefore to use minimax search to *guide* gameplay, but not for evaluating states.

3.2 TD-leaf(λ)

TD-leaf(λ) not only uses search to guide gameplay, but learns and updates its values for the leaf position of the principal variation, instead of the state that the agent is in. It only considers the principal variations. The leaf nodes of the principal variations are the states that are evaluated and also the states for which V is updated. Whenever the principal variation gets played, the two mechanisms evaluate and update the same state. But whenever this is not the case, the state for which V is updated differs from the state observed in play.

4 Learning Gameplay with an Imperfect Opponent

Standard TD learning does not use an explicit model of the opponent. Instead, the opponent is simply perceived as a part of the environment. In other words, the learning task is to play against that particular opponent. But this might not be what we intend to learn. What we really desire is an evaluation function for playing a certain game in general, against any opponent. The problem here is that the TD-algorithm is not optimising an evaluation function in the game-theoretical sense, but simply to beat its current opponent. This becomes an important problem especially when playing against a small number of weak opponents (which is usually the case in Computer Go).

As an example of the problems that can be encountered when learning from an imperfect opponent, suppose an agent is playing a game and not doing very well. It has only a small chance of winning. The agent then makes a move, and according to the search it did, the opponent's best continuation would be move m_o, leading us to a leaf node with evaluation $V(s_{pv})$. But the opponent unexpectedly plays another move m_o', leading to another state s' with $V(s') \geq V(s_{pv})$. Now if the opponent's move was a mistake, and the agent wins the game, the positive return will be propagated back throughout the game, past the bad move m'_o. So the feedback the agent obtains is that it was doing well even *before* the opponent made its mistake. We want to prevent this kind of learning of incorrect information from bad play.

4.1 TD(μ)

The issue of learning from playing against an imperfect opponent is addressed in a paper by Beal (2002). He points out that when TD(λ) trains against a "bad" opponent, it will learn to produce bad play, even if it was trained to play well before. He describes a new algorithm called TD(μ), which should perform better when learning from bad play. Since the description of the algorithm in the original article contained some minor errors (Beal, 2003), we describe it here in more detail.

TD(λ) assumes that the differences between the evaluations of successive states are caused only by an imperfect evaluation function. As Beal argues, this implicitly assumes that play is perfect. In reality play is not perfect, and some of the changes in the evaluation value during a game come about because of mistakes made by one of the players. TD(μ) tries to separate the changes due to bad play from the changes due to a bad evaluation function.

TD(μ) also adds the opponent to the learning algorithm. Both sides of play are observed, and the opponent is no longer thought of purely as part of the environment. Instead, its moves are evaluated too, taking into account not only the states where one player is to move but also the states that lie in between. In other words, when it is Black's move in state s_t, it will be White's move in state s_{t+1}.

Whenever one of the players makes a move, the evaluation change d_i is observed, and an error e_i is calculated:

$$d_0 = 0, \quad d_i = V(s_i) - V(s_{i-1}) \tag{3}$$

$$e_i = \begin{cases} \max(d_i, 0) & \text{if opponent played} \\ \min(d_i, 0) & \text{if agent played} \end{cases} \tag{4}$$

The equation for e_i can be explained as follows. Both players try to maximise their profit. Whenever a player makes a move that happens to *decrease* the value of the board position for that player, it is considered a mistake. The amount of change is then stored in e_i. On the other hand, if it was a "good" move, the state value will have increased or stayed the same.

Note that this does not only filter out bad play from the opponent. When the player makes an exploration move which happens to be bad, then TD(μ) reacts in the same way as it does when the opponent makes a bad move.

Now, we use the values e_i to calculate a corrected n-step return a_i^t, with all the score drops subtracted from it[1]. The formula for a_t^t is presented as a special case.

$$a_t^t = V(s_t) \tag{5}$$

$$a_i^t = V(s_i) + \sum_{m=t+1}^{i} r_m - \sum_{j=t+1}^{i} e_j \tag{6}$$

Next, a sigmoid squashing function is applied to a_i^t, yielding P_i^t, which can be interpreted as the probability of winning when in state t, as seen from state i.

$$P_i^t = \frac{1}{1 + e^{-a_i^t}} \tag{7}$$

The update rule given by Beal is:

$$\begin{aligned} \Delta V(s_t) &= \alpha[(1-\lambda) \textstyle\sum_{k=t+1}^{T-1} \lambda^{k-t-1} P_k^t \\ &\quad + \lambda^{T-t-1} P_T^t - P_t^t] \end{aligned} \tag{8}$$

To understand this equation, we note that for the case of a task with finite length T, equation (1) can be written as:[2]

$$R_t^\lambda = (1-\lambda)\left(\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)}\right) + \lambda^{T-t-1} R_t \tag{9}$$

So (8) is equivalent to (9) and (2), with the difference that $R_t^{(n)}$ and $V(s_t)$ are now substituted by respectively P_n^t and P_t^t.

The TD(μ) algorithm uses the agent's current knowledge to judge gameplay from both sides. Whenever a move is encountered that is considered to be sub-optimal, the corresponding drop in V is subtracted from the n-step returns in

[1]In the formulas in Beal's paper, no rewards r_t are mentioned. They have been added here.

[2]This is shown in (Sutton and Barto, 1998), p.170

(5). According to Beal, this makes it possible to learn from an opponent that plays poorly, or even from random play. This suggests that the algorithm would be better at learning what moves are generally good.

In his paper, Beal tests his algorithm by learning an evaluation function for chess from random play. Although it shows some learning, it is not clear from the paper just how well the algorithm performs, and how much knowledge it acquires. In this paper we compare its performance with other TD learning algorithms and try to make a stronger case about learning from bad play by testing the agent's knowledge after training on two new tasks.

5 Experiments

We have compared the performance of various adapted versions of the TD algorithms, applied to training networks on a Go-playing task. The game of Go has yet defied any attempt to create a strong player. Conventional methods that are successful with other games tend to perform poorly with Go. In particular, the problem of finding a good board evaluation function is extremely difficult. Given the strong pattern-matching component of the game, Neural Networks and Machine Learning become particularly interesting for application in Go.

The combination of Neural Networks and TD learning has been applied on Go with interesting results (Schraudolph et al., 2000; Dahl, 2001). Apparently it is possible to acquire Go knowledge in these ways. However, training tends to take a long time and can result in a strongly biased playing style (lacking 'genuine' Go knowledge). This raises the question whether a dedicated TD learning algorithm can provide better performance. To answer this we have tested the various algorithms on playing 5×5 Go against WALLY, a weak Go-playing program. 5×5 is the smallest board size for which the game is still tactically interesting.[3] The game of Go is very scalable, in the sense that its structure remains very much the same with different board sizes, and results obtained for small boards can be quite relevant for use on bigger boards.

[3] 5x5 Go has recently been solved (van der Werf et al., 2003). If the first move is played in the centre a full-board win can be proven by a 22-ply deep search. Other openings can require searches up to 40-ply deep, and sometimes surprise even professional players.

5.1 Training the network

In our case, the evaluation function $V(s)$ is represented by a neural network. We used a fully-connected MLP with 32 inputs, a single hidden layer of 75 units and a single output. Input features include: number of liberties for every stone on the board (with a maximum of 5 liberties and negative values for enemy stones), the total number of liberties for all friendly stones, the total number of liberties for all enemy stones, the number of friendly stones minus the number of enemy stones on the board, the number of stones on the edge, whether there is a friendly group in atari, and whether there is an enemy group in atari. After some experimentation this combination of network architecture and feature set was found to be sufficient to learn the desired evaluation function. It is probably possible to improve the network architecture by using symmetries or weight sharing, but this is not our main concern.

During games rewards were given directly for captured stones. At the end of each game rewards were given proportional to the total score minus the captures already rewarded during the game. Every game was started with 2 random moves to ensure sufficient exploration of the state space. This also made the task of playing against WALLY more challenging. The network could not learn to beat WALLY every single game because some of the starting positions gave the agent too much disadvantage. As another means of exploring the state space, an ϵ-*greedy* policy was used. This made the agent choose a random exploration move a small fraction ϵ of the time.

If the randomisation was decreased at the end of training, the network could be trained to beat WALLY every time, regardless of being black or white (black gets to play the first move). However, this turned out to be an undesirable kind of overtraining. The network would learn a single 'trick' to beat WALLY every time, but forget all its 'real' knowledge, so performance against GNUGO would go down. There is an important trade-off here between obtaining maximal performance on the current task, and acquiring more general knowledge independent of the current opponent.

While the TD learning algorithm gives us an error value that can be used for updating the

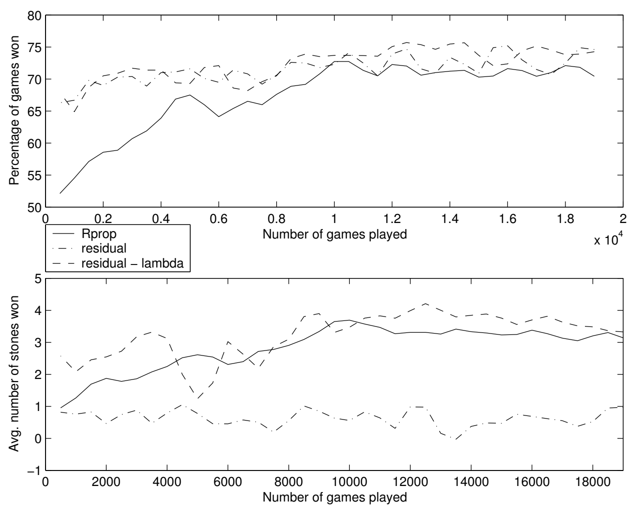

Figure 1: Comparison of NN training algorithms. The top figure shows winning percentage, the bottom figure shows average score. On the horizontal axis is the number of games played.

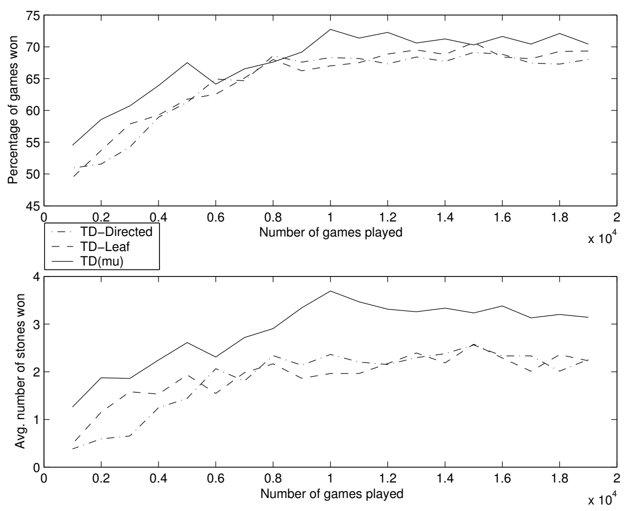

Figure 2: Performance for three different TD-learning algorithms, with the RPROP algorithm. TD(μ) clearly shows best performance.

network, this still leaves us with the choice of a network training algorithm. Standard TD(λ) with backpropagation was found to be very slow, taking in the order of 10^5 games to beat WALLY. Instead we selected the RPROP algorithm (Riedmiller and Braun, 1993), which generally converges faster, is well suited for complex error surfaces, and has the advantage of not having to optimise its parameters because it adapts these automatically.

It has been shown that using TD learning with neural networks can be unstable even with very simple tasks. The reason for this is that, unlike in a lookup table, states are not independent. A possible solution to this problem has been described in the form of the residual algorithm (Baird, 1995). This algorithm was designed specifically to prevent instability for the combination of Neural Networks with TD learning. According to Baird, the residual algorithm should only be used with TD(0), and it should be expected to learn quite slowly. However, it can be combined with other algorithms like TD-leaf(λ), as long as λ is set to 0. Despite of this, we also tested an adaptation of Baird's algorithm using TD(λ) with $\lambda > 0$, which we call Residual-λ[4]. (We always used $\lambda = 0.5$ unless stated otherwise.)

The performance of the network training algorithms (with TD(μ)) is shown in Figure 1. It is shown that Baird's residual algorithm reaches high winning percentages faster than RPROP, which was selected for speed. This is a remarkable result since Baird describes his algorithm as being rather slow. Its average score (in stones won) is significantly lower than RPROP, however. The Residual-λ algorithm shows learning performance comparable to that of Baird's residual algorithm, but it reaches the highest average score.

5.2 TD learning algorithms

We now proceed to the comparison of dedicated Temporal Difference Learning algorithms. Figures 2 and 3 show our results. Performance is shown for three algorithms. The TD-directed(λ) algorithm is just standard TD(λ) with a policy that does a two-ply minimax search. TD-leaf(λ) and TD(μ) use the same policy for choosing their moves, but use different rules for updating the evaluation function. Standard TD(λ), using only a single-ply search, performed very poorly and did not learn to defeat WALLY within a time comparable to the other algorithms. Schraudolph (2000) describes more successful learning with TD(λ), but this seems to be strongly related to using a more sophisticated network architecture.

The performance differences between the TD-algorithms are small but noticeable. TD(μ) performs best in combination with both neu-

[4]For a more elaborate description of these algorithms, see (Baird, 1995) and (Ekker, 2003)

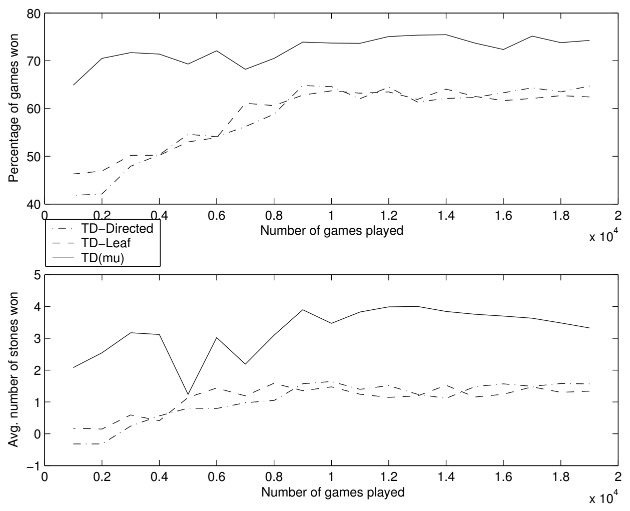

Figure 3: Performance for three different TD-learning algorithms, with Residual-λ. Again, TD(μ) performs best.

ral network training algorithms. It achieves the best winning percentage and the highest average score. TD-leaf(λ) does tend to learn a bit faster than TD-directed(λ) at the start, but the difference is only very small. It might be that the performance gain for TD-leaf(λ) would be larger when training is done with a deeper search.

Although the networks learn to beat WALLY, these figures do not tell us much about the amount of Go knowledge that is acquired. Therefore, after training, we used the trained network on two other tasks to test how much it had learned about Go. We let it play against GNUGO, a stronger opponent, at its lowest level and we let it score endpositions. Table 1 shows the results.

A Chi square analysis of these figures gives a p-value of 0.0005, indicating a high level of significance, supporting our previous findings.

TD Algorithm	% wins vs. GNUGO	% correct classifications
TD-directed(λ)	42	37
TD-leaf(λ)	44	45
TD(μ)	49	58

Table 1: Performance of the network after training against WALLY on two other Go-related task. The winning percentage against GNUGO and the percentage of games correctly classified as wins or losses are shown.

TD(μ) again shows the best performance, followed by TD-leaf(λ). Apparently, some 'real' Go knowledge has been learned, which is most clearly shown by the 49% winning rate against GNUGO, which is a much stronger opponent than WALLY. This seems to support Beal's claim that TD(μ) is better suited to learn 'good' information from a 'bad' opponent.

6 Discussion

We have shown that using dedicated Temporal Difference Learning algorithms can improve performance significantly. Using standard TD(λ), our networks did not learn to beat WALLY within a reasonable number of games, compared to the other algorithms. The dedicated algorithms reach 70% wins within 10000 games, and up to 80% within 20000 games. Of the algorithms we have tested, TD(μ) clearly performs best, followed by TD-leaf(λ).

We believe that a fair amount of 'genuine' Go knowledge was learned. The clearest evidence of this is in reaching about 50% wins agains GNUGO at its lowest level, just by training against WALLY. Again, TD(μ) yields the highest playing strength and the best performance at the classification of endpositions. This indicates that TD(μ) indeed learns more 'genuine' Go knowledge than the other algorithms.

We have tried several network training algorithms. The RPROP algorithm proved to be much faster than standard backpropagation. We also tried Baird's residual algorithm and found that it performed even better than RPROP. It remains unclear whether residuals might yield fast learning on other tasks, too, or whether the observed performance is related to the structure of our learning task. Furthermore, combining residuals with TD(λ), which we call Residual-λ, improved performance even more. It will be interesting to see whether these results hold also for other learning tasks.

In this paper, we have focused on a comparison of temporal-difference reinforcement learning algorithms in a scaled-down version of the game Go. As a consequence the implementation of the features which are derived from the game state has not been explicated in great detail. It should be noted that in order to improve computer-based playing of Go both the design of this feature set and the choice of the learning algorithm need to be optimised. It is expected that the use of better features will improve the overall performance of gameplay, while maintaining the performance rank order of TD-learning variants which has been found in this study.

References

L. C. Baird. 1995. Residual algorithms: Reinforcement learning with function approximation. In Morgan Kauffman, editor, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37.

J. Baxter, A. Tridgell, and L. Weaver. 1998. TDLeaf(λ): Combining temporal difference learning with game-tree search. *Australian Journal of Intelligent Information Processing Systems*, 5(1):39–43.

D. F. Beal. 2002. Learn from you opponent - but what if he/she/it knows less than you? In J. Retschitzki and R. Haddad-Zubel, editors, *Step by Step. Proceedings of the 4th colloquium "Board Games in Academia"*, pages 123–132. Editions Universitaires, Fribourg, Suisse.

D. F. Beal. 2003. Personal communication.

F. A. Dahl. 2001. Honte, a go-playing program using neural nets. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 10, pages 205–223. Nova Science Publishers, Huntington, NY.

R. Ekker. 2003. Reinforcement learning and games. Master's thesis, RijksUniversiteit Groningen, Groningen, The Netherlands.

M. Riedmiller and H. Braun. 1993. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA.

N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. 2000. Learning to evaluate Go positions via temporal difference learning. Technical Report IDSIA-05-00, IDSIA, February.

R. S. Sutton and A. G. Barto. 1998. *Reinforcement Learning: An Introduction.* MIT Press, A Bradford Book.

E. C. D. van der Werf, H. J. van den Herik, and J. W. H. M. Uiterwijk. 2003. Solving Go on small boards. *ICGA Journal*, 26(2):92–107, June.