Reinforcement Learning and Games

Temporal-Difference algorithms for Gameplay and their performance on playing 5x5 Go

Reindert-Jan Ekker

studentnr. 0943436

Supervisors:
Rijksuniversiteit Groningen:
Prof. Dr. L. Schomaker
Dr. Ir. J. Nijhuis
Universiteit Maastricht:
Dr. Ir. J. Uiterwijk
Ir. E. van der Werf

Kunstmatige Intelligentie
Rijksuniversiteit Groningen

Contents

1 Introduction

1.1 Artificial Intelligence and Machine Learning

Artificial Intelligence is a scientific discipline concerning computers, or any artificial object in general, and ways in which these may exhibit *intelligent behaviour*. This raises the question: what is intelligent behaviour, and how does it work? Maybe the easiest answer is that intelligence is a property of humans and other life forms. Most people seem to have a common implicit understanding of what this property 'intelligence' is and what kinds of behaviour it brings about. However, it is extremely difficult, or maybe even impossible, to formally define the concept of intelligence in a satisfactory way. To understand what it is, we study intelligence as it manifests itself in the living beings around us, as well as in our own behaviour. In these ways, the field of Artificial Intelligence touches on other fields like philosophy, psychology and biology.

But unlike these sciences Artificial Intelligence has as its goal not just to study, but to *create* systems that behave intelligently. It tries to create artificial machines that exhibit aspects of intelligence like perception, understanding, reasoning, and learning. It is with the latter that the discipline of *Machine Learning* is concerned.

Learning can be defined as an adaptation of a system's behaviour over time. Such a system might be a biological organism, a computer program, or even a complete population of organisms. In the case of Machine Learning, a computer program or *agent* changes its behaviour to perform better on a certain task. An intriguing aspect of this is that a computer can learn to solve a certain problem, even when we do not have the slightest idea about how to write a program to solve that problem. Instead of programming the computer explicitly to solve the task, we program it to learn the solution by itself. This is why Machine Learning is especially interesting when applied in complex environments, where solutions are very hard to find or to program manually.

1.2 Game-playing

Games form a special class of problems that traditionally attracts much interest from AI researchers. The most popular games in the AI community tend to be games like Chess or Go, which are difficult enough to be challenging for a computer, yet relatively easy to program, forming an ideal testing ground for Machine Learning.

Chess and Go are *perfect-information* games, meaning that all information in the environment is perceivable to the agent. As a consequence, the computer's representation of the environment can be correct in every detail. Furthermore, the dynamics of a game (its rules) are simple and we know them exactly. With real-world tasks, a description of the environment and its dynamics is usually far more complex. For example consider driving a car. It is hardly possible to completely know the current state, since too many different parameters are involved (from wind and temperature to the properties of tires, brakes and gears). And even if we would know all this information to the tiniest detail, as well as the dynamics involved (which we also do not know), then computing the next state would be a tremendously difficult task. Games on the other hand, are usually easy to formalise, and yet they can pose very complex and interesting problems.

It is possible to generate many thousands of trial games in a fast, cheap and easy way, making game-playing even more attractive as a research area. Sometimes it is even possible to use libraries of games played by humans, which can be re-played by the agent. Again, these things are simply not possible for most real-world tasks. All in all, we can say that games can be complex enough to require some form of intelligence, while being an ideal testing environment for Machine Learning techniques from a practical point of view.

1.3 Reinforcement Learning

The main focus of this thesis is on Reinforcement Learning, a Machine Learning discipline based on the mechanism of learning from *rewards*. These rewards are given to the agent as a source of information about how well its actions are. The agent can respond to this by changing its behaviour in order to achieve more rewards in the future. It can freely explore its world and try any action to see what its results are. There is a strong analogy here with the biological world, where organisms take actions and are rewarded by stimuli like pleasure and pain.

One of the nice properties of Reinforcement Learning is that the agent does not need explicit instruction. Because of the nature of the feedback given to the agent, a programmer needs relatively little knowledge about the solution to the learning task. No detailed feedback is needed about every action that the agent takes, nor does the agent need to be told what the correct solution is. At some point in time, it just needs to be rewarded, telling it how well it did until that moment.

A possible application of this is with problems where the desired solution is not known. In such a case, we cannot tell the agent what actions it should take, so it is very convenient to give feedback only at the end of a trial. Complex games often fall into this category, because we can tell who the winner of a game is, but the perfect move for a board position is usually unknown or very expensive to calculate. This makes those games very good candidates for applying Reinforcement Learning.

1.4 Reinforcement Learning and Gameplay

So games have a number of properties that make them an excellent environment for Reinforcement Learning. Their dynamics are clear and simple, large numbers of trials can be generated in fast and cheap ways, and yet they give us a very rich and complex learning environment. However, we find that the Reinforcement Learning model is not optimal for modeling deterministic, perfect information games like Chess and Go. The question arises whether adapting Reinforcement Learning algorithms to deal with the specifics of gameplay will facilitate training a game-playing program. We focus mainly on two important aspects of playing deterministic, perfect information games: game tree searching and imperfect opponents, which we will now describe shortly.

- **Game Tree Search**
  When a game is deterministic and both players have perfect information, all possible moves and countermoves are known to each player. This

makes it possible to think ahead, contemplating what the opponent's best response to a move is and how to continue after that, etc. In game-playing computer programs, this is done with *tree searching* algorithms like *minimax*. Determining the best move for games like Chess and Go without searching is many times more difficult than it is when searching is used.

Standard Reinforcement Learning uses a stochastic model of the environment in which there is no place for tree searching. Two adapted algorithms, TD-directed(λ) and TD-leaf(λ), have been described by Baxter et al.[2]. These algorithms use Temporal-Difference Learning in combination with game tree search.

- **Playing against an imperfect opponent**
  In the normal Reinforcement Learning model, changes in the world are caused either by the agent or by the environment. But when playing a game, we know there is an opponent, and we can tell the difference between his actions (moves) and the environment's dynamics (rules). This makes it possible to observe the opponent's moves, evaluate them, and reason about them. Standard Reinforcement Learning does not provide us with a means to do this.

  If the opponent is not perfect, he will sometimes make suboptimal moves. When we think a move is strong, reasoning about it should be different than when it was a bad move. Bad moves can lead to the learning of incorrect knowledge from imperfect gameplay. An algorithm called TD(μ), to detect bad play and prevent learning incorrect information from it, has been described by Beal[3].

Both of these aspects may have a great influence on game-playing performance, but Reinforcement Learning does not take them into account. In both cases, one of the most widely used Reinforcement Learning algorithms called *Temporal-Difference Learning* can be adapted to make better use of these properties of gameplay. Temporal-Difference Learning, for its own reasons, is very well suited for learning this kind of task[1]. We investigate whether training a game-playing agent is more successful when using an adapted algorithm than with standard Temporal-Difference Learning.

1.4.1 Go

To compare the performance of standard Reinforcement Learning methods and the adapted algorithms, we train an agent to play Go on a small board. Playing the game of Go has proven itself to be a difficult challenge for computers. The game is so complex that it has defied any attempt to build a strong computer player. Since the rules of the game allow it to be played on boards of almost any size while preserving its overall characteristics, we can make this challenge somewhat easier by letting our agent play on smaller boards than usual. The game of Go, its rules and current research into computer Go are discussed more in-depth in section 2.

[1]See section 3.1

1.4.2 Neural Networks

In many cases of Machine Learning, including Reinforcement Learning, the knowledge to be learned takes the form of a mathematical function. In the case of learning to play a game, the agent needs to learn an *evaluation function*, which gives a value to each possible situation, indicating how good it is to be in that situation. In our case, this function is represented by an *artificial neural network*. A neural network provides a mechanism to learn an arbitrary mathematical function, and has the important property of being able to *generalise*. Through generalisation, the network can deal with situations it has not seen during training. This is a very important property when the number of possible situations is so big that they cannot all be visited during training, which is the case for games like Go. Finding the correct way to use neural networks with Go and Reinforcement Learning is one of the major issues involved in training the agent.

1.5 Research Question

The research question of this thesis can be stated as follows:

> **Will adapting the Reinforcement Learning model to incorporate domain-specific knowledge about games improve learning performance ?**

To answer this question, we focus on two properties of deterministic, perfect-information games that are not taken into account by standard Reinforcement Learning techniques. These properties are game tree searching and learning from imperfect gameplay. Adapted algorithms are investigated for both of these, and the performance of these algorithms and the standard Temporal-Difference Learning algorithm are compared by training a neural network for playing Go on a small board. This also leads to some examination of the best way to train a neural network for playing Go, and how to combine this with Reinforcement Learning techniques.

2 Go

The game of Go originated in Asia more than 3000 years ago. It is one of the oldest and most complex board games in the world. Today, it is played by millions of players, mainly in the Far East, but its popularity in the western world is growing. The game has been extensively studied and played at professional levels for centuries. Its ruleset is much simpler than its western counterpart, Chess, but it is at least as challenging and complex to play. Although computers playing Chess have been very successful, Go-playing programs are still at weak amateur levels. In fact, playing Go is considered by many to be the next big challenge for Artificial Intelligence.

Because not all readers are familiar with Go, we give a short explanation of the rules of the game in this section. This provides some framework and introduces some of the terminology that is used in later sections. We also give a short overview of what has been done in the field of computer Go and what the major challenges in this field are.

2.1 Rules of the Game

Go is played by two players who take turns placing round stones on a board. One player uses black, the other white stones. Pass moves are allowed. On the board lines are drawn, and the stones are placed on empty intersections of these lines. Stones do not move after they are placed on the board; they stay in place until the end of the game unless they are captured. The usual size of a Go board is a square with sides of 19 intersections, but board sizes of 13 and 9 are also often used for practice. The goal of the game is to form as much territory on the board as possible.

There are some differences in the rulesets used by the various Go organisations worldwide. Although all agree on the general rules, there are some subtle differences, mainly concerning the ko rule, scoring, and suicide [9]. The major variants will be discussed below.

- **Groups**
  Stones that are horizontally or vertically connected form groups. All the stones in a group live or die together, so forming strong groups is an important element of the game. Groups are referred to with many different terms, like *block, string, unit, chain* or *worm.*

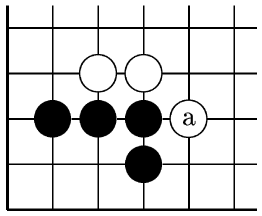
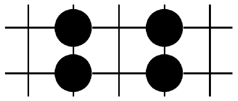
  

  Figure 1: The black stones form a group. So do the two white stones. The single white stone marked *a* is not connected to the white group.

  When two groups are not formally connected, they are often thought of as being connected into a single group when it is very difficult to prevent the joining of the two groups. In fact, human players use many different notions of groups, from the formally connected group to more abstract, fuzzier structures (*moyo's*) corresponding to the region of influence of stones. Some interesting work has been done on formalizing the various notions of connectedness, groups and influence in Go (see Müller[14], Bouzy[5]).

Figure 2: The so-called *bamboo joint* is a very strong connection between groups.



- **Capturing**
  Any empty intersections that a group touches are called its *liberties.* Liberties are shared between all the stones in the group. When the opponent places a stone on the last liberty of a group, this kills the group and it is removed from the board. These stones are now captured, or *prisoners.* A group with only a single liberty is in danger of being captured and said to be in *atari.* In practice, groups that can be captured are often left on the board until the end of the game instead of immediately being captured.

  It is not allowed to kill one's own groups (*suicide*). Placing a stone where it takes away its own last liberty is only allowed if this kills an enemy group, generating new liberties for itself.

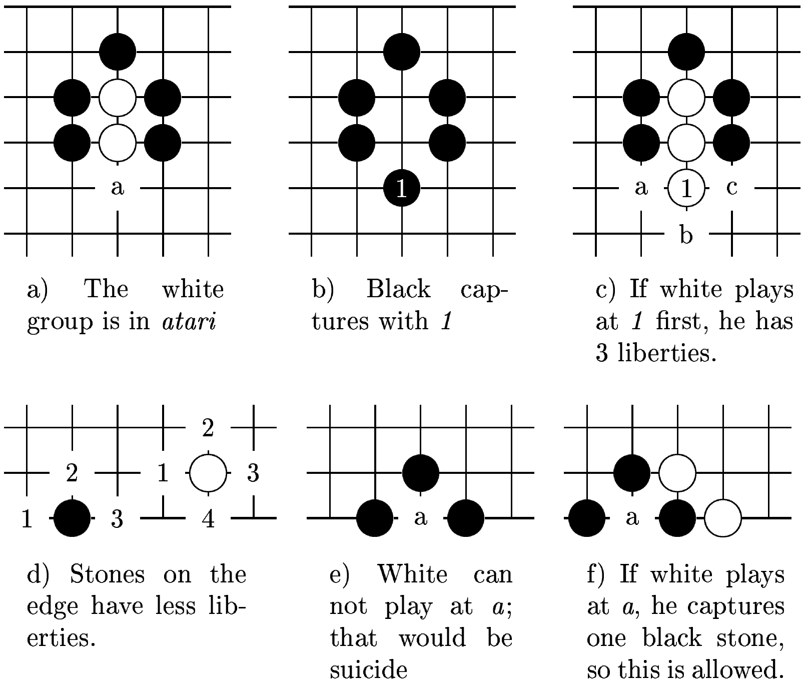
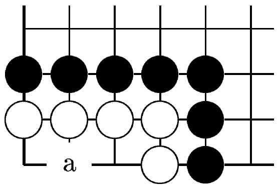

a
1
a
1
c
b
a) The white group is in *atari*
b) Black captures with *1*
c) If white plays at *1* first, he has 3 liberties.
2
2
1
3
1
3
4
3
d) Stones on the edge have less liberties.
a
e) White can not play at *a*; that would be suicide
a
f) If white plays at *a*, he captures one black stone, so this is allowed.

Figure 3: Liberties and capturing

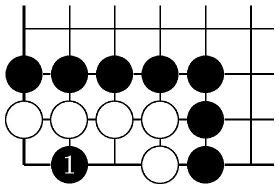
- **Living groups**
  An open space that is completely surrounded by a group is called an *eye.* A group with two or more eyes is *alive*; it cannot be killed. The struggle for life and death of a group can be quite complex, and it is very

important that a player can determine whether a group will be able to get two eyes. If a group will be able to get two eyes under all circumstances, it is considered to be alive. For example, groups with one large eye will usually be able to make a second eye by dividing the eye in two parts, and so they are classified as living groups.

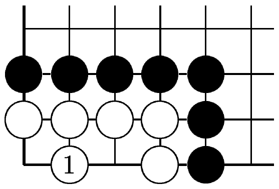
However, for single eyes up to a size of 6 spaces, this depends on the shape of the eye. See figure 4. The life of the white group in figure 4a depends on who gets to play at *a*. If black plays, the white group dies. It will not be able to make two eyes, and will eventually be captured. If white plays, the group has two eyes and is alive.



a) The life of the white group depends on who plays first.



b) If black plays *1*, the white group dies.



c) The group lives.

Figure 4: Life

There is a special form of life, called *seki*. It occurs when a white and a black group, neither of which has two eyes, are forced to coexist by sharing some of their liberties. Neither of the players can play on the shared liberties, because that would kill their own group. In case of a seki, both groups are considered alive, but no territory is awarded for the groups.

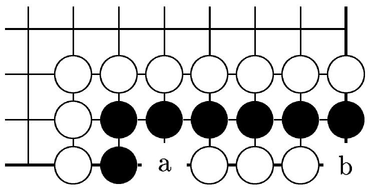


Figure 5: Black and white live in seki. Whoever plays at *a* or *b* will put his own group in atari.

The goal of the game can now be described as to construct living groups occupying as much space on the board (territory) as possible. Of course this also means that one has to prevent the opponent from making large living groups.

- **Ko**
  The concept of *ko* prevents loops in the game. It forbids capturing moves that bring the game back to a previous position. There are several different variations of this rule. The simplest form of ko (basic ko) just forbids immediate repetition. An example of this is shown in Figure 6. After black captures, white cannot immediately capture back, because this would restore the position before blacks move. White will have to play elsewhere

on the board, or pass. After this, black can end the ko by playing at *a*. Figure 3f shows a similar situation on the edge of the board.

If black plays *1*, it is forbidden for white to capture back at *a*.

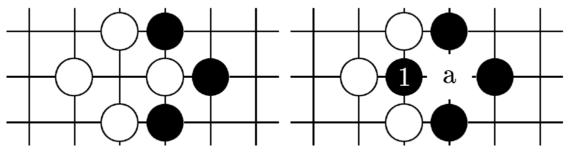


Figure 6: The Ko rule

Ko situations are a very common and important tactical aspect of the game. In a so-called *ko struggle*, the player that is not allowed to recapture the ko (white in figure 6) has to play elsewhere on the board. If he can, he might play at a position that threatens to gain him some points (a *ko threat*). The other player (black) then has to decide whether he wants to defend against the ko threat, or to end the ko. If black responds to the ko threat, white can reverse the situation by recapturing the ko. Now it is up to black to find a ko threat.

Correctly playing a ko struggle requires tactical evaluation of the entire board to find ko threats for both players. Not only the number of ko threats has to be known, but also whether each of them is worth more or less than winning the ko at hand. So more than just preventing repetition, the ko rule adds some very interesting tactical complications to the game.

The basic ko rule does not prevent repetition of cycles longer than 2 moves. There is a stronger version of the ko rule to deal with this kind of situation, called *superko*, but the various rulesets do not agree on this subject. The major variants of superko are positional superko, which forbids repetition of a board position, and situational superko, which only forbids repetition of a position if the overall board situation is the same. A situational superko only occurs if the player to move, any moves that cannot be played due to basic ko, and the number of prisoners are the same as a previous situation.

- **End of the game**
  The game ends when both players pass consecutively. This happens when there are no more sensible moves left for any of the players. All dead groups are then removed from the board and counted as prisoners. Now the score has to be determined. The player with the highest score wins.

  There are several different scoring systems, the major ones being Japanese and Chinese scoring. With Japanese or territory scoring, the number of empty points within living groups is counted and the number of captured enemy stones is added. With Chinese scoring or area scoring, the space inside living groups is counted and the number of stones on the board is added. The two systems give about the same result, usually with a difference of only a single point.

  It is possible that players do not agree about which groups are alive. Usually this means that play is continued until the dispute has been resolved. Another possibility is to let a referee determine the score.

2.2 Computer Go

Now that the game itself has been introduced, we turn to the issue of programming a computer player. Go proves itself to be a very difficult game for computers. The task of playing Go can be divided into many subtasks, each of which poses its own problems. It is the combination of all these problems which makes computer Go such a tremendous challenge. We will now give a short overview of the difficulties and achievements of computer Go.

2.2.1 Why Go is Difficult

The key to a game-playing program is to find a way to *evaluate* a board situation. Given a function that maps every possible board situation to a value denoting how good it is to be in that position, and enough time to calculate that function, we can achieve perfect play[2]. But finding even a moderately good evaluation function for Go has turned out to be many times more difficult than for many other games[14][5].

The first and foremost reason for this difficulty is the daunting complexity of the search space, caused by the large size of the board and the length of the game. The size of the state-space (the number of possible board positions) is estimated to be about 10^{170} for Go on a normal 19x19 board, compared to approximately 10^{50} for Chess[14]. The size of the game tree (number of possible games) is about 10^{600} vs. about 10^{123} for Chess. Because of the enormous search space, classical searching techniques fail with Go where they have been successful for other games.

But the size of the problem space is not the only reason why Go is so difficult. There are many other complicating matters. Evaluation is not a straightforward process like counting the number of stones on the board, but it relies on higher-level concepts like life and groups. This means that evaluating a position requires evaluation of many subproblems on the board. In any position, there might be a range of problems, from determining the life status of a group, capturing races (*semeai*) and ko struggles, to estimating how much territory an unfinished group might surround.

The problem is made even more difficult because a strong player tends to leave many of these subgames unfinished. Capturing a group can often wait until later, because there are urgent matters elsewhere, and threatening moves may be saved for a ko struggle. Even stones that are unconditionally dead can still have some influence on the game (*aji*).

And then there is the problem of determining the score at the end of the game. Players tend to pass in situations where many things are still implicit. To determine which groups live, searching is required and often it is not feasible to proof what the correct score is within reasonable time. This is in strong contrast with most other games, where the winning condition is relatively simple (like Chess or Checkers).

2.2.2 State of the Art

There has been a lot of effort into writing a computer Go player. Some excellent overviews have been written about the field[14][6][21][5], so we will not give such

[2]This, of course, would require a shallow search for the best move

an overview here. Many sophisticated techniques have been developed, like pattern matching and specialized searches. But even the most sophisticated program today has not been able to rank itself among strong human players.

There is a ranking system for Go players. *Kyu* levels are for novice players, ranging from 30 for the absolute beginner to 1 for the strongest novice. Next there are *dan* levels, ranging from 1 dan for a strong amateur to 9 dan for the strongest professionals. Players with different strengths can play a fair game with the use of handicap stones. The weaker player gets as much handicap stones as the difference in strength, up to 9 stones.

Currently, the best computer programs are ranked at the high kyu levels. However, stronger humans tend to beat these programs with more handicap stones than is reasonable with humans. Müller [14] gives an example where a human player beats Many Faces of Go, one of the strongest programs in the world, with a handicap of 29 stones. As Müller puts it: 'Judged by human standards, play of current programs looks *almost* reasonable, but certainly not impressive'([14], p.7). The fact that no strong computer programs exist, even after many years of research, gives an indication of the enormous challenge that Go poses.

2.2.3 Knowledge in Computer Go

We will be training an agent to play Go using Reinforcement Learning. The acquired knowledge will be represented by a neural network. But considering the complexity of the evaluation function to be learned, would it be reasonable to assume that this is feasible? The knowledge needed to play Go can be roughly divided in two parts. First, there is pattern-based, 'fuzzy' knowledge about territory, influence and many group-related concepts. Then there is logical reasoning, where local tactical searches have to be done to find the status of a group or the outcome of a local fight.

Clearly, it is the pattern recognition part that one expects to combine best with neural networks. Neural networks have been used for life/death problems[10], shape evaluation[7], scoring end positions[8], move prediction[26], and many other tasks. They have also been applied for whole-board evaluation with promising results[20][11]. This suggests that at least some valuable part of the knowledge needed for playing Go can be learned by neural networks.

On the other hand, there is logical reasoning. This includes formal proofs and algorithms for finding life or safety of a group[4][13]. Classical search algorithms also fit into this category. Many of the local evaluations that are necessary for a good full-board evaluations cannot be learned easily by a neural network, where a search algorithm can solve these in a well-known way. A good example of this are *ladders*, series of consecutive atari's with only a few possible countermoves every time. Even beginning human players are able to read these positions up to as much as 60 ply deep. Ladders can be solved by specialized narrow searches, but trying to evaluate them statically with pattern-matching is probably not sensible.

But searching is expensive and can be quite slow. It seems that maybe the best solution is a combination of the two approaches: pattern matching for large-scale territory evaluation and local searches for other, specific tasks. We will be taking a simpler approach with our program. A neural network will do full-board evaluation, but it will evaluate the leaf nodes in a minimax search. Because this approach is fairly simple, it cannot be expected to learn the full tactical intricacies of things like ko-struggles or ladders. Because we choose a simple opponent for the learning task, this will not be much of a problem. However, building a strong player without these specialised modules would probably not be possible.

3 Theoretical background

We will now discuss various Machine Learning concepts and algorithms. First we look into Reinforcement Learning, and specifically the technique called Temporal-Difference (TD) learning. We will introduce two popular TD-learning algorithms, TD(0) and TD(λ). Next, we investigate the application of Temporal-Difference Learning with games and introduce three additional algorithms that have been adapted for this purpose: TD-directed(λ) and TD-leaf(λ) use searching of the state space, and TD(μ) critically evaluates the opponents moves.

The evaluation function that is learned with TD-learning can be represented by a neural network. We explain several methods for training the weights of the network. Also, we introduce *residual algorithms*, which combine TD-learning with neural networks while addressing some problems regarding possible instability.

3.1 Reinforcement Learning

Reinforcement Learning lets an agent learn to perform a task by telling it about the consequences of its actions. It is rewarded for its actions through positive or negative reinforcements. Based on these rewards, the agent can change its behaviour. Reinforcement Learning has a clear biological counterpart, which can be observed in both humans and animals. Pleasure and pain are good examples of positive and negative rewards, and learning from these rewards is something every person experiences. But why would this be better than using supervised learning ?

3.1.1 Reinforcement Learning vs. Supervised Learning

The main difference between Reinforcement Learning and supervised learning techniques is the kind of information that the agent receives through feedback. In Reinforcement Learning, this feedback is purely *evaluative*, telling the agent about how good the result of its action was, but not what the correct action would have been. In supervised learning, feedback is *instructive*: independent of the action you try, it will tell you what your action should have been.

The following example is adapted from Reynolds([16],p.2). Consider the consequence of leaving home without an umbrella on a cloudy day. Supervised learning would give you the following: *'If it is cloudy, you should take an umbrella'*, whereas Reinforcement Learning would say: *'It was cloudy. Now you got wet. That was pretty bad.'* Reinforcement Learning tells you what the consequences of your actions are, but not how to choose a better action.

This has some nice practical implications. When doing supervised learning, one needs to have some knowledge about what the agent should learn, so the agent can be told what it should have done. In Reinforcement Learning, this kind of knowledge is not needed. The environment just presents the agent with rewards for its actions. Exact knowledge about how good a specific action is, is not needed because the agent will learn this from the rewards.

This is very important when learning to play Go. The exact value of any board position in the game is not known, nor is the knowledge of which move is best. We can try to make use of human experts to generate this knowledge, for example by rating moves, but this is in many ways not very practical[21].

When we use Reinforcement Learning, we can just tell the agent the score at the end of the game, and from that it will draw conclusions about how well its actions were.

The feedback signal that Reinforcement Learning gives to an agent, contains less information than the feedback given with supervised learning. In many cases, this means that Reinforcement Learning methods will give slower learning than supervised methods. However, it has the important advantage that we can use it on tasks where no target values for states or actions are known.

3.1.2 The Learning Task as a Markov Decision Process

We will now present a formal description of Reinforcement Learning. For a general task, we assume a discrete time scale $t = 0, 1, 2, \ldots$, with s_t denoting the state of the system at time t. In every state, the agent can choose an action a_t from the set of possible actions. This leads to a new state s_{t+1}. Figure 7 illustrates this.

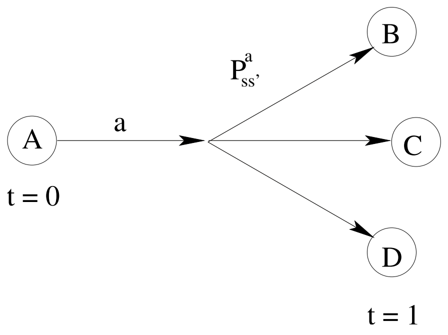


Figure 7: A simple example of a state transition

The diagram in Figure 7 might be a depiction of someone flipping a coin. At $t = 0$, he is in state $s_0 = A$, where he is still holding the coin in his hand. The action a he takes is to throw the coin into the air. Depending on the conditions of the environment, like wind, temperature and the weight distribution of the coin, he finds himself in some next state s_1. This could be state B, with the coin laying on the ground "heads" up. State C would give "tails", and state D would have the coin standing on its side. The chances to end up in each of these states are given by the *transition probabilities* $P^a_{ss'}$. The probabilities P^a_{AB} for getting tails and P^a_{AC} for heads are both about 0.5, and P^a_{AD} is very close to zero.

When we model a task like this, we call it a Markov Decision Process (MDP). An MDP is defined by S, the set of all possible states (in the example this is $\{A, B, C, D\}$), the set of possible actions in any state $A(s)$, and the transition probabilities $P^a_{ss'}$ of going from state s to s' when taking action a. These probabilities are a model of the world's dynamics, telling us what the possible consequences of an action are. As an agent performs his task, we can describe the events that occur as a path through the subsequent states he visits $\{s_0, s_1, s_2, \ldots, s_T\}$. This path depends on the action he takes in every state and the transition probabilities to every next state.

An important thing about MDP's is that they satisfy the so-called *Markov Property*, meaning that the transition probabilities to any next state are independent of the states we have visited in the past. In other words, the current state contains all the information the agent needs to decide which action is best. The Markov property is an important assumption in most theory about reinforcement learning. However, as Sutton[23] points out, when the state representation is non-Markov, it can be appropriate to think of it as an approximation of a Markov state when it provides enough information for predicting subsequent states.

For the agent to learn to accomplish a task, it needs feedback about the actions it takes. So in every state, we give it a reward r_t. For the example of flipping a coin, imagine that the agent has made a bet for \$100 that the result will be heads. This means that if he ends up in state B, he will get a reward of \$100, so $r_B = 100$. If the result is tails, he has to pay, so $r_C = -100$. If the coin ends up on its side, no one gets paid and $r_D = 0$. The goal of the agent is to try to maximise the total return R_t, which is the sum of all rewards to be received in the future. For a finite task with length T, the return is written as:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \tag{1}$$

The agent chooses an action in every state, according to a *policy* $\pi_t(s, a)$, which gives the chance of choosing action a when in state s.

$$\pi_t(s, a) = P(a_t = a \mid s_t = s) \tag{2}$$

This policy might, for example, be totally random, but for most tasks this will not result in desired behaviour. In practice we want to know how good it is to be in a certain state, so that we can decide which action is best. We might choose to use a *state value function* $V^\pi(s)$, which maps states to numbers. Formally, $V^\pi(s)$ is the expected return for being in state s when following policy π.

$$V^\pi(s_t) = E_{s_t,\pi}(R_t) \tag{3}$$

E here is the statistical symbol for the *expected return*, which is the sum over all possible next states of the reward received in that state multiplied by the probability of reaching that state[23]. The agent tries to make an estimate of $V(s)$ that is as close as possible to the optimal value function, enabling it to choose a good action in any state, so as to receive a lot of rewards.

So the agent learns how its world works, based upon the rewards it receives. Obviously, this makes the choice of a suitable reward function very important. It should provide the agent with the information it needs to learn to perform its task. While the agent explores its world, he continually changes its estimate $V(s)$ to reflect its experience. In simple cases, this can be implemented by keeping a table with a value estimate for every single state. Each entry in the table could, for example, hold the average of the returns received every time the agent visited that particular s[3].

Now suppose we have an agent that implements $V(s)$ in a lookup-table, like described above. An obvious choice for a policy might be the *greedy* policy, which means that in any state the agent chooses the action that leads to the best next state according to the current value estimate. The problem with this

[3]This specific technique is known as Monte Carlo policy evaluation[23].

approach is that when the agent learns that a certain state has a higher value than others, it will always try to go to that state. To prevent the agent getting stuck in suboptimal behaviour, we want to make sure that it samples all the states, especially those that have not been visited before, but also those that have been and yielded an inferior result at that time. In other words, we need the agent to *explore* its environment.

On the other hand, if the agent wants to be successful, it has to use its knowledge. The trade-off between making use of knowledge about known states and exploring unknown states is known as the exploration-exploitation dilemma. A popular approach to this is using an *ϵ-greedy* policy, which chooses greedy actions most of the time, but once in a while, with probability ϵ, it chooses a random exploration move. This guarantees the sampling of the entire state space when infinitely many samples are taken[23][16].

3.1.3 Temporal-Difference Learning

Temporal-Difference (TD) learning is one of the possible ways of estimating a state-value function from experience. It uses the rewards that are received in later states to update its value estimates. Because of the ability to learn from rewards that are given at a later time, this kind of learning is very suitable for tasks where actions do not have an immediate effect.

The simplest form of TD-learning is called TD(0). It uses only the information from the state that directly succeeds the current one. When V is implemented by holding a lookup-table with state-values, the value of s_t would be updated as follows:

$$\Delta V(s_t) = \alpha[r_{t+1} + V(s_{t+1}) - V(s_t)] \tag{4}$$

Here, α is a learning rate parameter that determines the size of the update. Because TD(0) uses only the next state for its update, information is only slowly propagated. This means it generally converges to an optimal V very slowly, taking a great number of episodes to learn. The more general algorithm, TD(λ), takes all the future states into account and is known to converge faster in many practical cases[22].

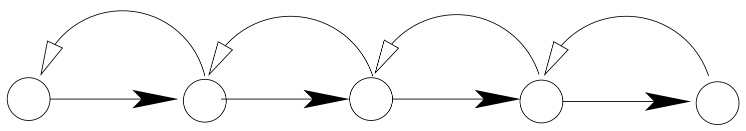
The TD(λ)-algorithm is based on the λ-return R_t^λ, which is a weighted variant of (1). It gives returns a smaller weight when they are further away in the future. For $\lambda = 0$, the algorithm is equivalent to TD(0). For $\lambda = 1$, it is equal to the Monte Carlo case, which just averages all rewards.

$$R_t^{(n)} = V(s_{t+n}) + \sum_{m=1}^{n} r_{t+m} \tag{5}$$

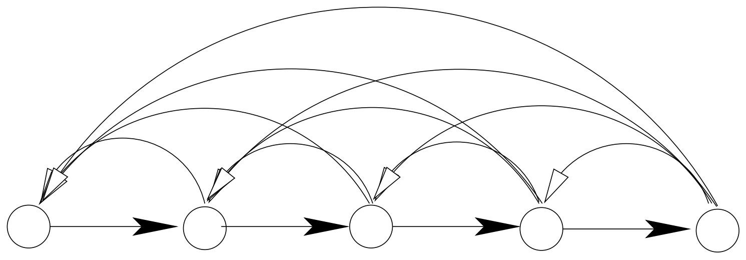
$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{6}$$

The quantity $R_t^{(n)}$ is appropriately called the *n-step return.* For the case of a task with finite length T, (6) can be written as:[4]

[4]This is shown in Sutton & Barto[23], p.170



TD(0) updates use only the next state



TD(λ) updates sum over all future states

Figure 8: Diagrams for updates as done by TD(0) and TD(λ). Black arrows depict state transitions, white arrows show the TD-updates.

$$R_t^\lambda = (1-\lambda)\left(\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)}\right) + \lambda^{T-t-1} R_t \tag{7}$$

The update rule for TD(λ) is:[5]

$$\Delta V(s_t) = \alpha[R_t^\lambda - V(s_t)] \tag{8}$$

There is one important note that we have to make here. In most theory about Reinforcement Learning, equations like (1) and (4) contain an extra *discount factor* γ, which is used to weigh future returns. When $\gamma < 1$, returns from further away in the future will be worth less to the agent than immediate returns.

For the task of playing games like Go and Chess, we don't want the agent to favour an immediate reward above the eventual outcome of the game (for example, sacrificing a stone might yield a negative reward for now, but a better result at the end of the game). This is why we set γ to 1. For simplicity, it isn't mentioned in any of the formulas in this paper. Since Go is always a finite game (assuming a ruleset with a Ko rule that forbids loops), this means that with a γ of 1, the total return (1) will still be finite.

3.2 Temporal-Difference Learning and Games

Now suppose we want to learn an evaluation function for a board game like Chess or Go. These are perfect-information games, in the sense that all information in the game is known to both players. This means that we can make a state

[5]This is the update rule for the so-called backward view of TD(λ). The forward view uses another equation, but can be shown to be equivalent when the updates are done off-line.[23]

representation that satisfies the Markov property. Such a state representation would include the current state of the board, as well as any information that has to be remembered from previous states. For example, in Chess it is not allowed to play a 'castling' move if the king has already been moved before, so the player has to be aware of whether this has happened or not. If we can devise a state signal that holds all this information, we can satisfy the Markov property.

In Reinforcement Learning the agent gets to choose an action in every move, after which the environment determines the next state. For two-player games this would mean that the opponent's move, as well as the board position in which he makes his move, are part of the workings of the environment. The opponent does not show up in our model of the world, even though we *know* that there is another state between the states where the agent makes its moves. For example, consider figure 9:

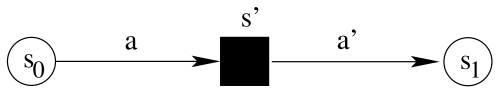


Figure 9: state transition in a game

In this example, our agent plays a move a in state s_0. We know what the board looks like after he plays this move. We call this 'afterstate' s'. This is the moment at which the opponent makes *his* move, a', leading to a new board position s_1 where the agent can play again. Reinforcement Learning normally considers only the states s_0 and s_1. In other words, normal Reinforcement Learning theory does not include the existence of an opponent in its model of the world.

A possible policy to choose a would be to select the most desirable afterstate by picking a move that maximises $V(s')$. This is a normal greedy policy, and it has been used to great success with backgammon[24]. This was mainly possible because of the stochastic nature of the game. The throwing of dice in backgammon makes it easier to give a good value estimate without extensively searching the game tree[2]. But for games like Chess and Go, which are completely deterministic, evaluating a position by only looking a single ply ahead is extremely difficult.

Also, when playing such a deterministic, perfect-information game, we can use our knowledge of the rules to look ahead. Not only do we know for certain which position will result from our action (s' in Figure 9), we also know the set of all possible responses from our opponent, and our responses to that, and so on. Normally, a game-playing program would use this knowledge by doing some sort of look-ahead search to determine what the best action is. For zero-sum, two-player games, a common search technique is *minimax* search[18].

We will now present two TD-learning algorithms that incorporate game tree search. Also, an algorithm is presented which includes the opponent in its model of the world's dynamics.

3.2.1 Temporal-Difference Learning and Game Tree Search

With minimax search, we look ahead into the game tree and make an estimate of $V(s_t)$ based upon states that are further ahead in the game. An example of a 2-ply minimax search is shown below. The evaluation values are shown beneath the leaf nodes of the search tree.

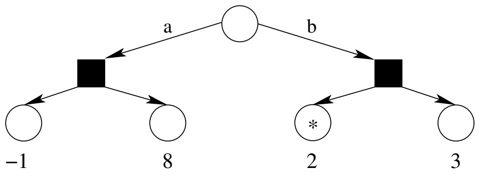


Figure 10: A small minimax tree

Here, the agent chooses action b. The opponent will play to minimise the agents result, so when the agent chooses a, the opponent will take the action leading to an evaluation of -1. If we choose b, the opponent will make the move that leads to the node marked by *, with a value of 2. The optimal path through the game tree, leading to *, is called the *principal variation.* The state value of the leaf node of the principal variation can be used as a new, more accurate estimate for the state value. Let us denote the leaf node of the principal variation starting at s as pv_s.

Note that the minimax algorithm implicitly assumes that we have a good evaluation function. If this is not really the case and our opponent's evaluation function is better, he might choose moves that seemed inferior to us, but win anyway.

We can make an adaptation to the TD-algorithm to make use of the information found by searching the game tree. Baxter et al.[2] tested two new variants, TD-directed(λ) and TD-leaf(λ), both of which were found to be more efficient for training the KnightCap Chess-playing program. TD-directed(λ) uses minimax search to choose moves to play. TD-leaf(λ) not only uses the 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.

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 value of V for the leaf node of its principal variation, $V(pv_s)$. Suppose we were to use this value $V(pv_s)$ instead of $V(s)$ to compute the TD error of equation (8). We cannot use this error to update the value for s, since it is in fact based on another state pv_s. Were we to feed this error to the backprop algorithm for a neural network, then the error would not match the feature set we update for. On the other hand, when we use this error for updating $V(pv_s)$, we have effectively the same algorithm as TD-leaf(λ). The correct way to implement the TD-directed(λ) algorithm is to use minimax search to *guide* gameplay, but not for evaluating states.

In other words, TD-directed(λ) can be viewed as TD(λ) with a policy that searches the gamespace. As such, it is the most straight-forward way of applying

TD(λ)when learning gameplay with minimax search.

TD-leaf(λ) 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.

3.2.2 Learning Gameplay with an Imperfect Opponent

TD-learning teaches an agent to perform some task in an environment. Since the opponent is a part of this environment, an agent will learn how to perform well against this particular opponent. This might not be what we intend to learn, though. 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 learning a perfect evaluation function in the game-theoretical sense, but it is learning to beat its current opponent.

The first thing that comes to mind to solve this, is to use a perfect opponent. Unfortunately, for most complex games perfect players do not exist. And playing against an imperfect or even bad opponent can lead to learning bad behaviour, since it might cause the agent to conclude that some play is good when it really is not. So we might consider modifying our learning algorithm to deal with imperfect play in a sensible way. A different approach would be to let the agent play against a population of various different opponents. Unfortunately, for Go there are not many programs available in the public domain, so we focus on learning from a single opponent instead.

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, which is reflected by a low value of V. 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 $V(pv(s))$. But the opponent unexpectedly plays another move m'_o, which gives us another state with value $V(s')$. The new state value is as least as good as the one we expected $V(s') \geq V(s_{pv})$. Let us presume the opponent's mistake was so bad that suddenly our agent has a fair chance of winning.

Now if 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 gets 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.

Next, we discuss an algorithm that uses current knowledge to determine whether a move is bad, and makes use of that judgement for updating the evaluation function.

3.2.3 TD(μ)

The issue of learning from playing against an imperfect opponent is addressed in a paper by Beal[3]. He points out that when playing TD(λ) against a "bad" opponent, TD(λ) 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.

TD(λ) assumes that the differences between the evaluations of successive states are caused only by an imperfect evaluation function. As Beal argues, this contains an implicit assumption 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.

With TD(μ), we use our understanding of the fact that there is another player, which makes moves in the same way we do. 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 (like in Figure 9), 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} (Figure 11).

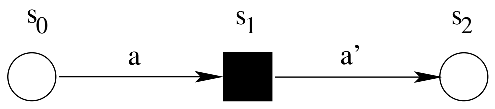


Figure 11: state transition for TD(μ)

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

$$d_0 = 0, \quad d_i = V(s_i) - V(s_{i-1}) \tag{9}$$

$$e_i = \begin{cases} \max(d_i, 0) & \text{if opponent played} \\ \min(d_i, 0) & \text{if agent played} \end{cases} \tag{10}$$

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 we make 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[6]. The formula for a_t^t is presented as a special case.

$$a_t^t = V(s_t) \tag{11}$$

$$a_i^t = V(s_i) + \sum_{m=t+1}^{i} r_m - \sum_{j=t+1}^{i} e_j \tag{12}$$

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.

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

$$P_i^t = \frac{1}{1+e^{-a_i^t}} \tag{13}$$

The update rule given by Beal is:

$$\Delta V(s_t) = \alpha[(1-\lambda)\sum_{k=t+1}^{T-1} \lambda^{k-t-1} P_k^t + \lambda^{T-t-1} P_T^t - P_t^t] \tag{14}$$

which is equivalent to (7) and (8), 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 (11). According to Beal, this makes it possible to learn from an opponent that plays poorly, or even from random play. This seems to suggest that the algorithm would be better at learning what moves are generally good. In other words, if we would train two agents against a certain opponent A, using standard TD(λ) for one and TD(μ) for the other, and we would test their performance against a new opponent B, then the agent trained with TD(μ) should perform better.

Beal does not provide any formal discussion of the convergence or stability of the TD(μ) algorithm. We, too, will not make any claims about the convergence of the TD(μ) algorithm. It is not clear if it will converge, or whether it will combine well with neural network training algorithms.

3.3 Combination with Neural Networks

TD-learning tries to produce a state-value function that maps each state to a number. For tasks where the state space is very large, it is infeasible to make a table with an entry for every possible state. In such a case, we can use a generalising function approximator to implement $V(s)$, such as a neural network. The generalisation has the added advantage that not every state needs to be sampled.

Of course, this assumes that the network used is capable of representing an approximation of $V(s)$. This includes choosing a sufficient featureset and making sure that the set of training data is both large enough and representative of the entire problem space. How we meet these criteria will be discussed in section 5.

3.3.1 The Backpropagation Algorithm

Consider a feed-forward multi-layer perceptron with m output neurons. When doing normal supervised learning, the desired output value for every output neuron j given input n is $t_j(n)$. The backpropagation algorithm defines a mechanism to update all the weights of the network [12].

First for every neuron i the activation $v_i(n)$ and output $y_j(n)$ corresponding to input n is calculated. This is the forward pass of the algorithm. The backward pass of the algorithm starts with calculating the error gradients $\delta_j(n)$ for the output neurons.

$$\delta_j(n) = \varphi'(v_j(n))\ (t_j(n) - y_j(n)),\ \forall n \text{ in output layer} \tag{15}$$

The function φ is the activation function of the network. The error is now propagated back through the hidden layers. Errors for the hidden neurons are calculated by weighing the errors from the neurons in the next layer.

$$\delta_j(n) = \varphi'(v_j(n)) \sum_k \delta_k(n) w_{kj}, \ \forall n \text{ in hidden layer} \tag{16}$$

The weights of the network are then updated according to

$$\Delta w_{ji} = \eta \ \delta_j(n) \ y_i(n) \tag{17}$$

Here, η is a learning rate parameter between 0 and 1.

3.3.2 The RPROP Algorithm

The backpropagation algorithm described in the previous section is not the only possible way to train a multi-layer perceptron. Many other algorithms are known. We have chosen to test the resilient backpropagation algorithm (RPROP) as described by Riedmiller and Braun [17], which is claimed to generally converge faster than standard backpropagation. Another advantage is that it is less dependent of the specific values of its learning parameters, since the algorithm is adaptive. Instead of using a fixed step size parameter like standard backpropagation does, RPROP adapts its step size automatically.

For every weight w_{ij}, RPROP keeps an individual update-value Δ_{ij}, which is the amount of change for that particular weight at that time step. The calculation of the partial derivatives of the error $\delta_j(n)$ is just like backpropagation with (15) and (16), but the rule for updating the weights is as follows:

$$\Delta w_{ij} = \begin{cases} -\Delta_{ij}, & \delta_j(n) > 0 \\ \Delta_{ij}, & \delta_j(n) < 0 \\ 0, & \delta_j(n) = 0 \end{cases} \tag{18}$$

As can be seen here, the size of the change in a weight is not dependent of the size of the error at that time, only on its direction. The amount of change is equal to the update value Δ_{ij}.

During training, the update value for each weight is continually adapted. Whenever the partial derivative of the error for a weight changes sign, the update value is decreased by a factor η^-. If the sign stays the same, the update value is increased by η^+. In this way, the step size is increased as long as the weight changes in the right direction. But a large step size also means we can step over the minimum we are looking for. So whenever the error changes sign, meaning we have jumped over a local minimum, we decrease the step size.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ \cdot \Delta_{ij}^{(t-1)}, & \delta_j(n)^{(t)} \cdot \delta_j(n)^{(t-1)} > 0 \\ \eta^- \cdot \Delta_{ij}^{(t-1)}, & \delta_j(n)^{(t)} \cdot \delta_j(n)^{(t-1)} < 0 \\ \Delta_{ij}^{(t-1)}, & else \end{cases} \tag{19}$$

Also, whenever $\delta_j(n)$ changes sign, the update value is reverted. This results in backtracking the last step, stepping back over the local minimum. At the next time step no change is allowed for that particular weight.

$$\Delta w_{ij}^{(t)} = -\Delta w_{ij}^{(t-1)}, \text{ if } \delta_j(n)^{(t)} \cdot \delta_j(n)^{(t-1)} < 0 \tag{20}$$

For a more in-depth discussion of the RPROP algorithm, we refer to the original paper[17].

3.3.3 State Evaluation with Neural Networks

We can use Artificial Neural Networks to represent $V(s)$ with TD-learning. To do this, the current state is mapped to a feature vector, which is used as input for the network. The corresponding output is used as the estimate for $V(s)$. But we do not know the desired value for $V(s)$, so we cannot use the normal way to compute the errors for each unit as in equation (15). What we *do* know is the current TD-learning error, which can be presented to the network as the desired change in the state value $\Delta V(s)$ given by (8). If we use this instead of the output error $(t(n) - y_j(n))$, we have a new equation for the updating of the weights for the output layer.

$$\delta_j(n) = \varphi'(v_j(n))\ \alpha[R_t^\lambda - V(s_t)] \tag{21}$$

Further backpropagation of the error through the hidden layers of the network is the same as in (16). This approach works for both normal backpropagation and RPROP.

3.3.4 Residual Algorithms

Convergence has been proven for the TD(0) and TD(λ) algorithms in the case of linear function approximation of $V(s)$ with on-line updating [23][25][16]. For the case of off-line updating, however, counterexamples have been presented by Baird[1] and Tsitsiklis and van Roy[25]. They show simple tasks on which off-line TD-learning with linear function approximations shows divergence of the weights of the function approximator to $\pm\infty$.

This instability is caused by the fact that, when an update is done for the value of a certain state, the values of other states will also change because of generalisation. Whenever the network weights are updated, not only the value of the current s is changed, but the response to many other possible inputs might change too. Also, the target value for a state is not fixed for every trial, but might change through time as the agent gains knowledge about the task and his behaviour evolves.

For example, imagine a Go-playing program which is in the first stages of learning about Go. In a certain position, it might be facing an endgame which it currently is not able to play correctly. The expected return for that position will be low for the current policy. Thousands of games later, the agent might have learned how to play this endgame. If it encounters the same (or a similar) position, the expected return will have a higher value.

This combination of moving targets and state values that change between updates can sometimes lead to instability. Because of this, Baird[1] introduced *residual algorithms* which are guaranteed to converge for any linear function approximator.

Consider training a network using TD(0) on a deterministic Markov Decision Process. For every state s, the optimal value function for this task satisfies the Bellman equation

$$V(s_t) = E_{s_t}\left(r_{t+1} + V(s_{t+1})\right) \tag{22}$$

In words, this means that the value function gives the expected value of the next state plus reward, given the stochastic transitions to all possible next states. The *Bellman residual* is the difference between the two sides of this equation, which is also the TD(0) learning error for the current state value $V(s_t)$[7]. TD(0) tries to minimise the Bellman residual. When used with a function approximator, it looks like:

$$\Delta w_d = \eta \Big[r_{t+1} + V(s_{t+1}) - V(s_t) \Big] \cdot \frac{\partial}{\partial w} V(s_t) \tag{23}$$

Baird calls this the *direct* approach (hence the subscript for w_d). As a solution to the problem of instability, he proposes to do a gradient descent on the *mean squared* Bellman residual instead. For a task with n states, the mean squared Bellman residual looks like

$$\mathcal{R} = \frac{1}{n} \sum_{s} \Big[E_{s_t}(r_{t+1} + V(s_{t+1})) - V(s_t) \Big]^2 \tag{24}$$

Minimising $\mathcal{R}$ can be done by updating weights using the partial derivative of V for both states in the Bellman equation:

$$\Delta w_{rg} = -\eta \Big[r_{t+1} + V(s_{t+1}) - V(s_t) \Big] \cdot \Big[\frac{\partial}{\partial w} V(s_{t+1}) - \frac{\partial}{\partial w} V(s_t) \Big] \tag{25}$$

Thus, the update of a weight is defined by multiplying an error (the Bellman residual), a partial derivative, and a learning speed η. This is equivalent to the updating of weights for backpropagation with TD(0)[8], with the difference that the partial derivative is replaced by the difference of the derivatives for states s_t and s_{t+1}.

The effect of the use of this combination of derivatives, is that information now flows both backward (through TD-updates) and forward (we use the gradient from the previous state) through time. This results in more 'gradual', stable learning, where not the error for the current state is directly minimised, but rather some kind of overall error over multiple states. This is also reflected by the use of the mean squared Bellman residual (24).

Baird argues that this approach, which he calls *residual gradient algorithms*, guarantees convergence for general function approximators with deterministic MDP tasks with a finite number of states. Baird also claims that when the approximator is general enough to represent any value function, and the mapping from value function to weight vectors is differentiable, then this algorithm is guaranteed to converge to the optimal value function[1]. He does not provide a formal proof of this, though.

However, he also argues that the residual gradient approach tends to converge much slower than the direct approach. As he states: 'direct algorithms can be fast but unstable, and residual gradient algorithms can be stable but slow.'([1],p.4) He introduces an adaptation of the algorithm, which tries to take advantage of the best of both. This variant he simply calls *residual algorithms*.

Let $\Delta\vec{\mathbf{w}}_d$ be the vector of updates for all weights w when using direct gradient descent, and $\Delta\vec{\mathbf{w}}_{rg}$ the same vector for residual gradient algorithms. These

[7]When the Bellman residual is multiplied by a learning speed α, this yields equation (4)
[8]See equations (4) and (17)

vectors could either be the update vectors for a single trial, or in the case of batch-training, they may be summed over an entire epoch. The update vector for residual algorithms is given by

$$\Delta\vec{\mathbf{w}}_r = (1 - \Phi)\Delta\vec{\mathbf{w}}_d + \Phi\Delta\vec{\mathbf{w}}_{rg} \tag{26}$$

The factor Φ, used as a weight for the two vectors, determines what the direction of the update will be. To guarantee stability, the residual update vector $\Delta\vec{\mathbf{w}}_r$ must make an acute angle with $\Delta\vec{\mathbf{w}}_{rg}$, the direction of steepest descent of $\mathcal{R}$:

$$\Delta\vec{\mathbf{w}}_{rg} \cdot \Delta\vec{\mathbf{w}}_r > 0 \tag{27}$$

On the other hand, for speed we want $\Delta\vec{\mathbf{w}}_r$ to be as close to $\Delta\vec{\mathbf{w}}_d$ as possible. This means we want to use the smallest possible $\Phi \in [0, 1]$, while still satisfying (27). A diagram of this is shown in Figure 12.

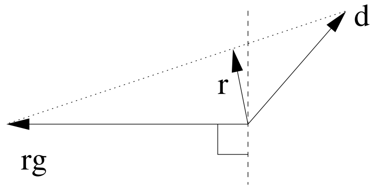


Figure 12: Update vectors for the residual gradient, residual, and direct algorithms.

Here, the dotted line represents the hyperplane that is perpendicular to the gradient $\vec{\mathbf{w}}_{rg}$. Weight change vectors lying to the left of the dotted line will result in a decreasing $\mathcal{R}$, while those to the right of it will increase $\mathcal{R}$. For maximal speed with stability, we need to find a $\vec{\mathbf{w}}_r$ close to the dotted line, but on the same side as $\vec{\mathbf{w}}_{rg}$. For finding such a $\vec{\mathbf{w}}_r$, we note that the Φ for which (27) is exactly zero is given by:

$$\Phi = \frac{\Delta\vec{\mathbf{w}}_d \cdot \Delta\vec{\mathbf{w}}_{rg}}{\Delta\vec{\mathbf{w}}_d \cdot \Delta\vec{\mathbf{w}}_{rg} - \Delta\vec{\mathbf{w}}_{rg} \cdot \Delta\vec{\mathbf{w}}_{rg}} \tag{28}$$

The following method can then be used for determining the Φ that gives the fastest learning with stability. First, the denominator of (28) is calculated. If it is 0, a Φ of zero is used. Else, (28) is evaluated, and a small constant ϵ is added to its result, while ensuring that $\Phi \in [0, 1]$[9].

An important limitation of the algorithms as discussed by Baird is that their application is only straight-forward for the case of TD(0) (this includes TD-leaf(λ) and TD(μ) with $\lambda = 0$). Unfortunately, it is currently not clear to us how to implement residual algorithms for the case of TD(λ). We should note here that Precup et al.[15] have introduced another algorithm for stable off-line TD-learning, which uses a technique called importance sampling to calculate returns. Their algorithm does apply to cases where $\lambda > 0$. However, it is an adaptation of the Q(λ) algorithm for learning state-action values, so it cannot be directly applied to our experiments.

[9]For an explanation of this method we refer to Baird's article[1].

3.3.5 Residuals with Other TD-learning Algorithms

When implementing Baird's residual algorithm with a neural network, one might simply view it as an adaptation of the backpropagation algorithm. Consider the update rule for the residual gradient algorithm as given by Baird (this is the same as eq. (25)):

$$\Delta w_{rg} = -\eta\Big[r_{t+1} + V(s_{t+1}) - V(s_t)\Big] \cdot \Big[\frac{\partial}{\partial w}V(s_{t+1}) - \frac{\partial}{\partial w}V(s_t)\Big] \tag{29}$$

This equation tells us to calculate the TD(0) error as usual, but the network training algorithm should use this error to descend on another gradient than normally (see eq.(23)). Practically, this means we do not have to change our TD-learning algorithm. We let the network training algorithm remember the gradient for the previous state. The 'gradient for the previous state' is set to 0 at the beginning of each game.

After the TD-error for the current state is calculated, both the direct and residual gradient update vectors for that state and error are calculated and the weights of the network are updated according to (this is the same as eq. (26)):

$$\Delta\vec{\mathbf{w}}_r = (1 - \Phi)\Delta\vec{\mathbf{w}}_d + \Phi\Delta\vec{\mathbf{w}}_{rg} \tag{30}$$

So we are effectively using Baird's residual algorithm as an adaptation of the backpropagation algorithm, which descents along a gradient based on two successive states. This training algorithm can be combined with the different TD-learning approaches, just like the RPROP and standard backpropagation algorithms. We combine the residual approach with TD-leaf(λ) and TD(μ) simply by changing (29) to contain the corresponding TD-error and setting $\lambda = 0$.

By using gradients for two successive states, the residual algorithm 'distributes' the updating of state values more evenly over the states in a game than TD-learning with normal backpropagation does. Although we do not formally show that this will guarantee convergence when used with TD(μ) and TD-leaf(λ) as well as with TD(0), this does not prevent us from testing it in combination with these algorithms.

3.3.6 The Residual-λ algorithm

In his article, Baird[1] only discusses the residual algorithm for $\lambda = 0$. For bigger values of λ, no stability has been proven, and to correctly apply the method of residual gradients to the general case of TD(λ), a much more complex combination of gradients should probably be used for updating the weights of the network (because of the large number of different states on which the TD(λ) error is based).

On the other hand, when viewing the residual algorithm purely as a network training technique, there is no reason why we cannot use it with the error as defined by TD(λ). Although this error is based on a larger number of states than with TD(0), still the combination of two gradients when updating the network weights might provide some extra stability.

From here, we refer to the combination of a network descending on the gradient $\vec{\mathbf{w}}_r$ and a TD-error using a $\lambda \neq 0$ as *Residual-λ* algorithms. Formally, Residual-λ uses the gradient from the residual algorithm for training the network, but the TD error is still calculated with a λ-return. So instead of (29), we used:

$$\Delta w_{rg} = -\eta\Big[R_t^\lambda - V(s_t)\Big] \cdot \Big[\frac{\partial}{\partial w}V(s_{t+1}) - \frac{\partial}{\partial w}V(s_t)\Big] \tag{31}$$

where $\lambda \neq 0$. Again, we can combine this in a straight-forward way with TD(μ) and TD-leaf(λ) for $\lambda \neq 0$. For the case of TD(μ) in particular, we suspect that using a λ bigger than zero is very important. If λ is zero, then for every state TD(μ) would only do a TD(0) update or no update at all. The special properties of TD(μ) might only be of real significance for larger values of λ, when 'good' and 'bad' information are propagated over a larger number of moves.

In our results section, we treat residual algorithms and Residual-λ algorithms as neural network training algorithms, as this is the role they actually play in our implementation. Unfortunately, providing a sound mathematical basis for their use, convergence or stability goes beyond the scope of this thesis. Before attempting to prove this, first the convergence and stability of TD-leaf(λ) and TD(μ) should be formally proven.

Summary

We have introduced the concept of Reinforcement Learning and the well-known TD(λ) algorithm. For large state spaces the evaluation function $V(s)$ can be represented by a neural network. The following methods for training the weights of the network were discussed:

- Standard Backpropagation
- Resilient Backpropagation (RPROP), which has been found to converge faster than backpropagation in many cases.
- Baird's residual algorithm, which guarantees stability for $\lambda = 0$, but is said to converge slowly.
- The Residual-λ algorithm, using Baird's residuals with $\lambda > 0$.

Normally, Reinforcement Learning methods do not take advantage of knowledge that is specific to the domain of gameplay. We discussed three adaptations of TD(λ), which do take this kind of information into account:

- TD-directed(λ), which uses minimax to select the agent's actions
- TD-leaf(λ), which uses minimax to select the agent's actions and does TD-updates for the leaf nodes of the principal variations of the search tree.
- TD(μ), which models the opponent and critically evaluates all the observed moves, to filter out incorrect information caused by 'bad play'.

To see whether using domain-specific knowledge about gameplay improves learning performance, we will test the TD(μ), TD-leaf(λ) and TD-directed(λ) algorithms. In the next section we discuss the learning task on which these algorithms will be tested.

4 Implementation of TD-learning with Go

We have introduced several TD-learning algorithms that take knowledge about gameplaying into account. Our goal is to test whether using this domain-specific knowledge yields better results than algorithms that do not use this knowledge. We will now discuss the task on which the performance of these algorithms was tested.

First, we will discuss the task of playing Go on a small board. Then we will provide some more details about our implementation of TD-learning for Go on a small board, and we will touch on some practical issues, like how to score end positions and how to use randomness to improve learning.

4.1 Choice of Learning Task

The learning task on which we choose to test our algorithms is playing Go on a 5x5 board. This is the smallest board size at which the game still has enough tactical variations and the state space is big enough to make the game an interesting machine learning challenge, also considering the fact that we wish to learn a quite complex evaluation function.

One of the reasons for choosing a small board was the large amount of time needed for training. We would typically be able to play about 15000 games in a day, using a computer with two 1.2 GHz processors. In other words, training three different algorithms for 50000 games each would take more than a week. A great part of this time was spent calculating the scores for the games.

The program was trained by playing against WALLY, a weak public-domain program. It has the advantages of being well-known and readily available, and quite fast as a move generator. There has also been other research with learning from WALLY, like Schraudolph[20]. We also use another, stronger program, GNUGO, to be 'referee' and determine the score at the end of every game. We also used GNUGO as an opponent to test the agent's performance after it was trained against WALLY.

Training from observation by re-playing games from human experts was not possible, because the smallest board size on which humans compete is 9x9, so there is insufficient data to train on. Furthermore, even for a 9x9 board the quality of human games in online databases is questionable, because most games of that size are played for practice by novices. Self play, where the network is trained by playing against itself, was attempted but yielded very poor results. As Schraudolph puts it, "learning from self-play is sluggish as the network must bootstrap itself out of ignorance without the benefit of exposure to skilled opponents."([20],p.11) Self play might be interesting with a net that has been primed by playing against another opponent, or by re-playing games from other players.

4.2 Scoring

In Temporal-Difference Learning, the agent uses rewards as an information source to learn from. At the very least, there should be a final reward at the end of each game. A common approach in games is to use rewards of 0 throughout the game, and only after the game has ended give a reward corresponding with the outcome of the game.

As opposed to Chess, where the outcome can only be 0, 0.5 or 1, Go games can be scored (unless one of the player resigns). But automatically calculating the scores for a game is quite a problem. There is no known algorithm that can correctly score every possible endgame position within a reasonable amount of time[8][13].

Calculating the score is especially difficult when the players pass before the life and death of every group on the board has been clearly established. Efficient algorithms like Bensons algorithm for determining the unconditional life or death of a group exist[4], but turn out to be of little use. In practice players tend to pass well before all the groups on the board have reached such a status. This leaves the status of large parts of the board to be determined in another way.

We decided to use GnuGo, another public-domain program, to calculate the scores of our games. Wally also has a scoring mechanism, but this turned out to be highly inaccurate. GnuGo, on the other hand, is the best Go-playing program available in the public domain, and as such plays significantly better than Wally. Furthermore, it is able to score games heuristically. When necessary, it will play some extra moves at the end of the game to determine the most probable outcome. Although we cannot trust the scores given by GnuGo to always be correct, it was the best automatic scoring method available to us at the time.

We have also tried the program MiGos, which was used to solve 5x5 Go[9], for determining the score. This program needed more time than GnuGo to calculate scores, and its ruleset differed somewhat from that used by our program, which was partly based on the GnuGo source code. This made scoring with GnuGo the most practical scoring method. We used the Japanese ruleset for scoring, GnuGo's default.

The score as reported by GnuGo was presented as the final reward, given at the end of the game. We found that using the score in stones (ranging from -25 to 25 on a 5x5 board, not counting any captures) results in better learning than using only a value of 1 for winning and 0 for losing, because it holds more information about how well the agent played. Since evaluations and rewards are expressed in stones, they were scaled with a sigmoid to the range [0,1], which is the output range of the network.

4.3 Implementing a Go-playing program

At the first attempt, TD(0)-learning was used for training. Moves were chosen by searching a single ply deep and evaluating all the next states with the network. This search was done using only the moves that were legal in the current position, so that the network would not be concerned with learning the rules of the game, but only with deciding which one of the legal moves is best. Also, moves that would fill any single-space eyes of the agent were not taken into account, since it is virtually never sensible to play such a move[10]. This was found to dramatically improve performance, which is not surprising because it keeps the agent from killing his own living groups by filling all the eyes.

We also found that performance improved when giving immediate rewards for captures. After a move in which a group was captured, a reward was given

[10]In fact, there are some very exotic positions where such a move might be sensible[9], but we consider it safe to assume that the agent does not run into any of these.

(positive if it was an enemy group, negative if it was friendly). It is necessary to substract these rewards from the final score, to make sure the captures are not counted twice. After changing the program to give these immediate rewards, the agent showed a stronger tendency towards making groups and defending them against capture.

Next, a two-ply alpha-beta search was implemented. The alpha-beta search algorithm is a common optimisation for minimax search[18]. The network was used to evaluate the leaf nodes of the search tree. The first move of the principal variation found by searching would be played by the agent.

Using the 2-ply search by moving from TD(0) to TD-directed(λ), the network was successfully trained to beat WALLY. The first 10000 games were played against a WALLY that was weakened even more by using random moves 50% of the time. The number of random moves for WALLY was then slowly decreased to 0. In about 100000 games, the network then learned to beat WALLY 100% of the time.

The main reason for adding randomness to WALLY's behaviour at the start, is to prevent the learning of a self-fulfilling prophecy of bad play. If we train the network against an opponent that is too strong, all the games would be lost from the start, resulting in learning a value of 0 for all states very quickly. This evaluation function results in losing all the games, confirming the predictions made by the network. The way out of this cycle is to make sure that the network and its opponent are evenly matched (see also Schraudolph[20]).

Upon examining the net's knowledge of the game, it was found that it did not learn to play any better than WALLY. Instead, it had learned a single pattern of moves that would beat WALLY every time. Typically, it would always try to form a group in a diagonal line across the board, preventing WALLY from making a living group. This demonstrates the ability of the network to learn an evaluation function for the task at hand.

However, this particular case is an undesirable kind of overtraining since it does not show any actual understanding of the game. To prevent this from happening in the future, all subsequent training games were started with 2 random moves. This effectively changed the task at hand from learning a single, simple strategy to beat WALLY's standard tactics, to learning to beat WALLY's general behaviour.

Summary

The learning task for which we will compare different TD-learning algorithms is playing Go against a simple opponent on a small board. We have discussed some of the practical issues of applying TD-learning for this task:

- Scoring is being done by the GNUGO program
- We give both final rewards for the outcome of a game, and intermediate rewards when captures are made
- The ability of our neural network to represent an evaluation function was verified by overtraining the network to win from WALLY
- Randomness was added to WALLY's behaviour at the beginning of training to make sure the players were evenly matched

- Every game was started with 2 random moves to ensure sufficient exploration of the state space.

Next, we will discuss the use of a neural network as an evaluation function for 5x5 Go.

5 Evaluation of Go Positions with a Neural Network

The knowledge of our game-playing agent will be represented by a neural network. Of course this means that the task should be learnable for such a neural network. In this section we test various methods of training the network to find out in what way a neural network can best be trained to represent an evaluation function for 5x5 Go.

First we will discuss our network architecture and the input feature set. Then we will show the observed performance for the various training mechanisms that were discussed in section 3.3.

5.1 Network Topology

A simple fully connected multi-layer perceptron with a single hidden layer of 75 units and a single output unit with a range of [0,1] was used for learning. This topology was chosen somewhat arbitrarily, but with keeping in mind that the network should be able to represent a sufficiently complex function for learning the desired mapping from board positions to evaluation values.

Schraudolph[20] trained a net for beating WALLY on a 9x9 board with only 40 hidden units. This suggests that results similar to ours can be achieved with a smaller net. Generally we suspect that finding a more sophisticated network topology might very much aid both learning and playing performance, for example by letting the network lay-out reflect the symmetries of the Go board, or the ways in which different points of the board are connected. The main reason for choosing such a large size for the hidden layer was to make sure that the network had sufficient capacity to represent an evaluation function.

5.2 Feature Set

When we view the task at hand as a TD-learning problem, the output of our network represents the state value function, so finding a suitable state representation is equivalent to finding a suitable feature set for training the network. A good featureset facilitates learning the prediction of the outcome of the game. The simplest kind of feature set would just represent the current board position. The problem with this kind of approach is that, although it provides the network with all the information that is present in the current board position, it does not make the classification of positions as 'good' or 'bad' any easier. A good featureset has the effect that states that are close to each other in the featurespace should also have evaluations that are similar.

There is a large number of possible extra features, like the number of liberties of groups, atari's, and all kinds of topological features of the board position. Many others have investigated the possible features for learning Go[14][20][8][26]. We chose to use the following features:

1. For every point on the board, if there is a stone there, the number of liberties of the group that stone belongs to is calculated. If that group has more than 5 liberties, that number is rounded down to 5. If it is an enemy stone, multiply by -1. If the position is empty, use 0. This number tells us whether there is a stone on this position, who it belongs to and how

strong it is in terms of its liberties. Also, this contains implicit information about the topology of the board, since stones in the middle of the board can have more liberties than on the edge. This gives us 25 features for a 5x5 Go board.

2. The total number of liberties for all of the friendly groups.
3. The total number of liberties for all of the enemy groups.
4. The number of friendly stones on the board minus the number of enemy stones (this roughly corresponds with the number of stones that have been captured).
5. The number of friendly stones on the edge of the board. This is an interesting feature, because it adds some information about the geometry of the board.
6. The number of enemy stones on the edge of the board
7. Whether there is a friendly group currently in atari.
8. Whether there is a enemy group currently in atari.

This amounts to a total number of 32 features for a 5x5 board (25 instances of feature 1, and 7 others). We tried adding a feature for ko, but this did not make a significant difference in performance. It is probably very difficult for our network to learn about the tactical intricacies of ko-struggles from playing Go on a small board against a weak opponent like WALLY. What's more, we would probably also need a way to indicate where on the board the ko is located, which would mean adding some extra features[11]. So, for the task at hand, it was decided not to use ko as a feature. If one would want to build a strong Go player, however, this would have to be reconsidered.

From a TD-learning perspective, a feature for ko would add the necessary information to make the state representation satisfy the Markov Property. But, as argued above, Sutton[23] states that a state signal that approximates the Markov property can also be sufficient for reinforcement learning. Furthermore, when one encounters a ko position, doing a specialised search into the tactical implications of the ko might yield better results than trusting that the network can oversee these implications.

Using the right feature set turns out to be very important. Figure 13 shows the results for training with only the first feature for every point on the board as input, as compared to training with the total feature set. When using only the board position, the performance of the network never even came close to the winning percentage achieved when the extended feature set was used. With the extended feature set, the network passes a 60% winning percentage after only 1000 games, and reaches 70% within 10000 games. Without extra features, the learning was very slow, and even after training for several hundreds of thousands of games the winning percentage was still below 40%.

The quantities shown in the figure are the percentage of games won and the number of stones won, both averaged over 1000 games. This averaging explains

[11]We estimate that to represent the position of a ko on a 5x5 board, at least 10 features would be needed. The most straight-forward approach would be to add a ko feature for every point, which would amount to adding 25 features.

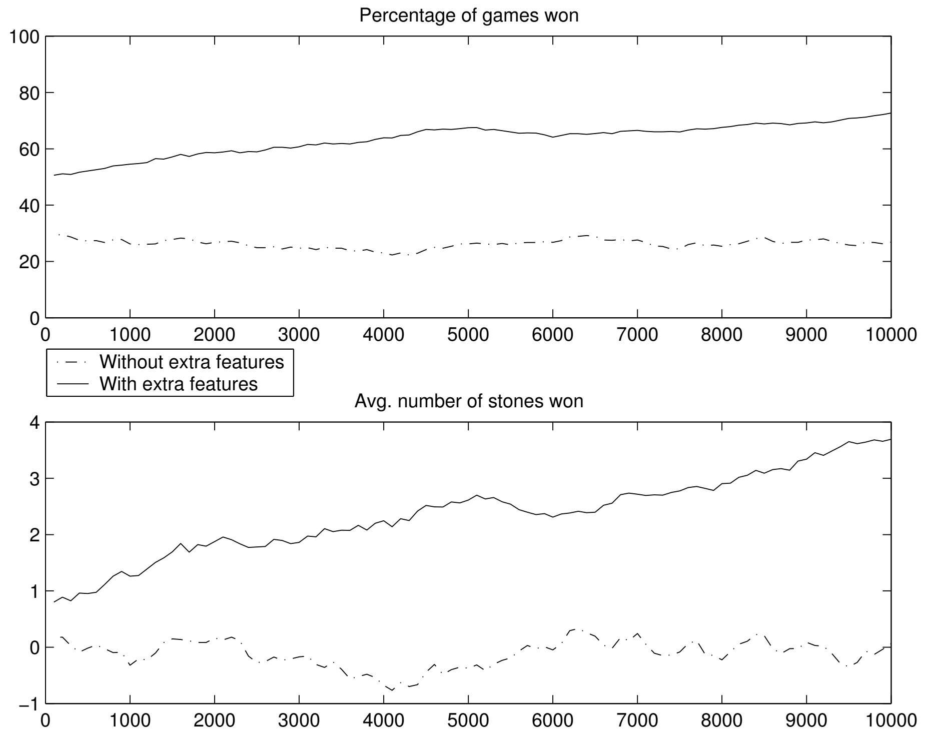


Figure 13: Performance differences for training with and without an extended feature set

the fact that performance seems to start quite high for the extended feature set. In reality, the network starts with randomised weights and very poor behaviour, but the winning rate increases very fast over the first 1000 games.

5.3 Effects of Training Algorithm on Network Performance

To find the best way of training our network, the following algorithms for training the neural network were tested and implemented: Standard Backpropagation, RPROP, Baird's residual algorithm and Residual-λ. With all of the methods batch-training was done, updating the network after every 10 games.

For RPROP, we used the parameter values as proposed in the original paper[17]: $\eta^+ = 1.2, \eta^- = 0.5$ and $\Delta_{max} = 50$. We set the minimal update step a little larger than the $1e^{-6}$ mentioned in the paper: $\Delta_{min} = 0.001$. For the residual and backpropagation algorithms, we used a learning rate of 0.2. In all of the experiments, a λ of 0.5 was used, except for the residual algorithm, which uses TD(0) updates. The exploration rate for ϵ-greedy exploration of the search space was set fixed to 0.1. This ensures exploration after the percentage of random moves for WALLY is decreased to 0.

For each run the network was initialized with random values for the weights in the range $[-0.1, 0.1]$, and the random number generator was seeded with a unique value. The residual and Residual-λ algorithms were combined with

TD(μ), TD-directed(λ) and TD-leaf(λ) by substituting the TD(0) error in (25) for the TD errors defined by those algorithms. For residual algorithms, λ had to be set to 0, while for Residual-λ it was set to 0.5.

Figure 14 shows the results of training against WALLY, using the TD(μ) algorithm, for the first three algorithms. It shows both the percentage of games won and the average score. Both quantities are measured over a thousand games, which explains the fact that for RPROP and residuals, the performance plot starts at quite a high level. In reality, the network started with poor performance, but this very rapidly increases over the first thousand games. The data shown in the plots is averaged over three runs of training the network.

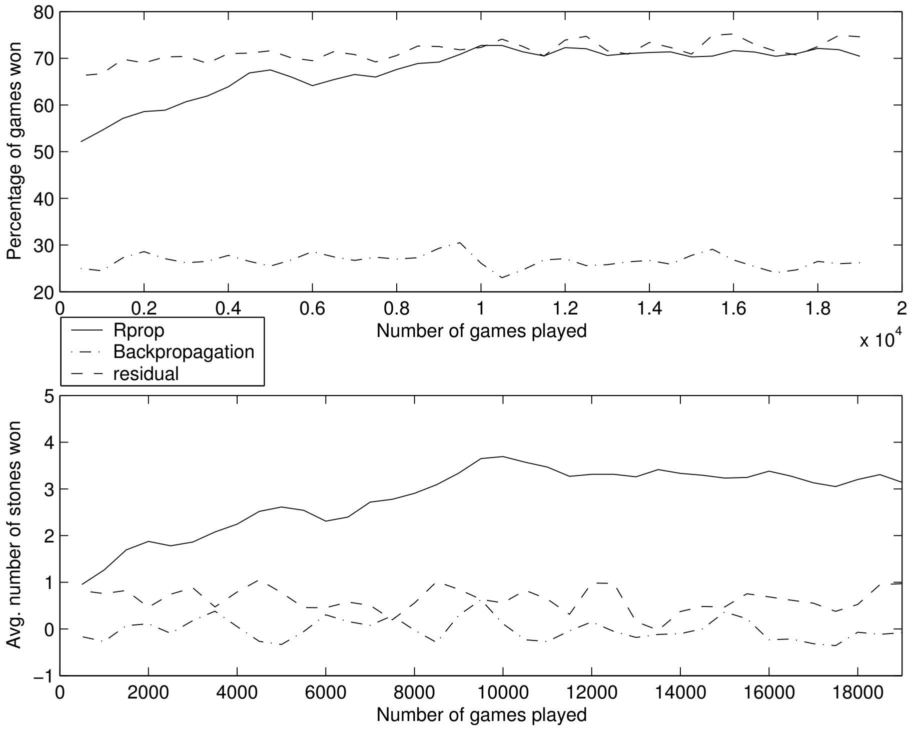


Figure 14: Comparison of different NN training algorithms - Standard backpropagation, RPROP and Baird's residuals

As we see in the figure, standard backpropagation performed quite poorly. Even over longer periods of time, performance stayed low. With backpropagation, it took more than 100000 games to reach a winning percentage of 60%.

The RPROP algorithm is known to converge faster than backpropagation in many cases, and we see this confirmed in our test. A performance level of 70% wins was reached by RPROP within 10000 games, with an maximum average score of about 4 points. This is the difference in territory, meaning that our player on average gets 4 points more that WALLY.

We also see that Baird's residual algorithm reached a winning percentage of 70% within 2000 games. This came somewhat as a surprise, since Baird suggests that the algorithm would be slower even than standard backpropagation. In fact, this algorithm, which is a simple adaptation of standard backpropagation, was even faster than the accelerated RPROP algorithm. Apparently the way in which the combined gradients descend on an overall error speeds up the process. However, the average score did not reach the same level as it does with RPROP. This is probably caused by the fact that for the residual algorithm, we had to use a λ of 0.

This brings us to another interesting result. In Figure 15 we compare the results for RPROP, residuals and Residual-λ. We see that the Residual-λ algo-

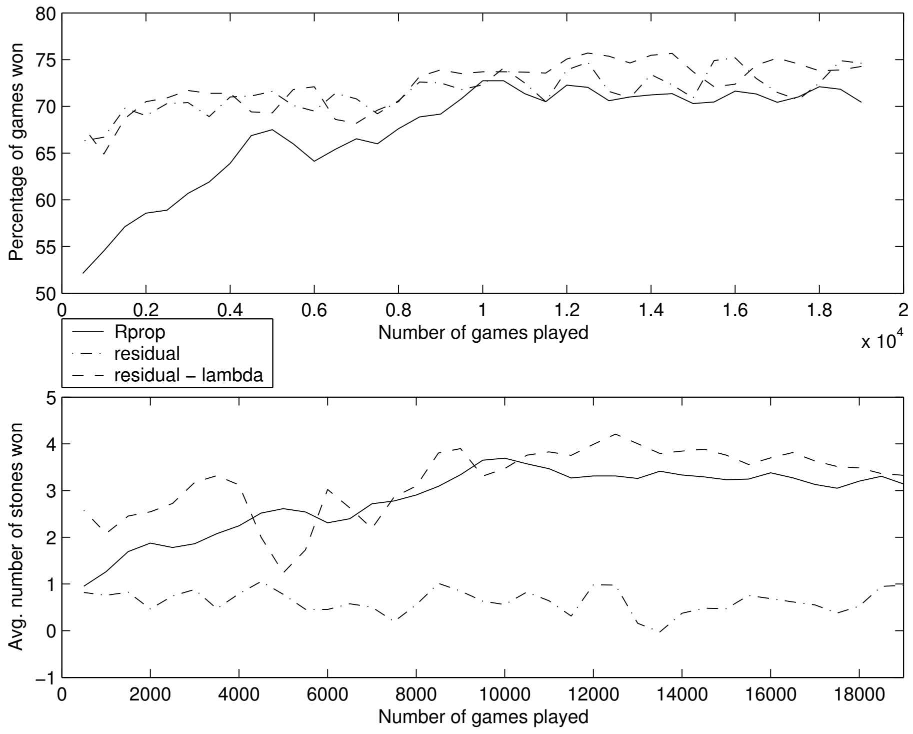


Figure 15: Comparison of NN training algorithms, with the Residual-λ variant added

rithm lead to even better performance than Baird's residuals. Not only did it perform better in terms of winning percentage, but it also reached a higher average score than RPROP. This supports the idea that the low scores for Baird's residuals were caused by the value of λ. When we use a higher value of λ, the agent gets feedback from further in the future, which will give more accurate information about the actual outcome of the game. It seems that this makes it easier for the agent to not only learn to win from WALLY, but also maximise its score.

5.4 Discussion

We have tested four different training algorithms for the network. Standard backpropagation yielded poor results, but with both RPROP and Residual-λ, winning rates of about 70 to 75% were achieved, with an average score of about 4 stones.

It is very interesting to see that the residual algorithms, which we implemented as an adaptation of standard backpropagation, can yield faster learning than the RPROP algorithm which was specifically developed for fast learning. According to Baird's paper about the algorithms, slower learning than normal backpropagation was to be expected. But our tests show that using the combination of gradients actually speeds up learning on this task.

The residual algorithm achieved about the same winning percentage as RPROP, but with maximal scores of about 1 stone. This seems to be caused by the λ parameter being 0, since the Residual-λ algorithm does not show these low scores. Actually, Residual-λ performs best among the training algorithms.

Summary

We have discussed how to train a neural network to represent an evaluation function for 5x5 Go, the network architecture and the input feature set. One important point is the large impact that finding a good feature set has on the efficiency of learning.

To find the best way of training our network, we have compared the performance of the standard backpropagation, RPROP, residual and Residual-λ training algorithms. It was found that

- Standard Backpropagation performance is poor
- RPROP learns significantly faster than backpropagation
- Baird's residual algorithm learns even faster than RPROP, without using any kind of accelerated convergence. However, the average score stays low, probably caused by the value of λ being zero.
- Residual-λ performs best of all, using the properties of both residual's gradients, and a higher value of λ.

Next, we will discuss the testing of various TD-learning algorithms that use gameplaying knowledge. With the results from these tests, we will try to answer our main research question.

6 Comparison of TD-learning Algorithms

Previously we have introduced Temporal-Difference Learning and shown three variants that use gameplaying knowledge. A learning task was chosen and implemented, and we have shown that it is feasible to train a neural network for this task. We will now try to determine whether TD-learning techniques adapted for gameplay give better performance when learning Go than TD-algorithms that do not.

We will test TD(μ), which evaluates the strength of its opponents moves, TD-leaf(λ), which does its updates for the leaf nodes of the minimax tree, and TD-directed(λ), on the task of playing 5x5 Go. The latter is the most straightforward application of TD(λ) with a gameplaying agent that does minimax search. We will also test the knowledge of the network after training by using it on two other tasks.

6.1 Expectations

Generally, we expect that incorporating game-specific knowledge into the Reinforcement Learning model will yield better learning. We expect to find faster learning in the case of TD-leaf(λ), since according to the article from Baxter et al.[2], it should take advantage of game tree search. We also expect increased performance in the case of TD(μ). Beal seems to suggest that the TD(μ) algorithm might lead to better learning of 'general knowledge' of the game, in the sense of improved performance for playing against a different opponent than the one that was trained against[3].

6.2 Comparison of TD-learning algorithms

We have trained our network on playing 5x5 Go against WALLY with three different TD-learning algorithms. For training the weights of the network, we had several algorithms at our disposal. We will focus here on the results that were achieved with RPROP and Residual-λ, the two network training algorithms that gave best performance.

We did use standard backpropagation, but this yielded very slow learning, so no further testing with it was done. Baird's residual algorithm was also implemented, but showed almost no performance differences between the different TD-learning techniques. This can be understood by realizing that, with $\lambda = 0$, there is not as much difference between the updates for the various algorithms. For example, a λ-return with TD(μ) can have quite a large number of e_j's subtracted from it during a game, which gives it a different update value. With $\lambda = 0$, there is only a single possible e for every state that is updated, which leaves little room for differences between the algorithms.

Figure 16 and 17 show our results for the first 20000 rounds of training for the three different TD-learning algorithms. The data in figure 16 is for training with RPROP and in figure 17 for training with the Residual-λ algorithm. The information shown in the plots is the same as with the comparison of the network training algorithms. The top part shows the percentage of games won, while the bottom part shows the average score. All data shown is averaged over 1000 games and 3 training runs.

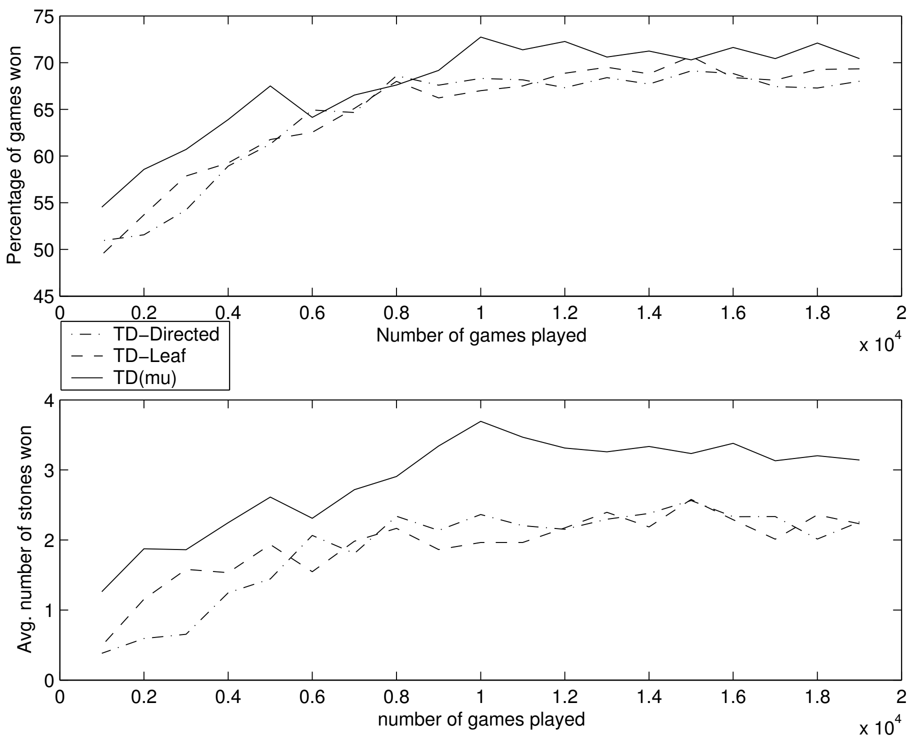


Figure 16: Performance for three different TD-learning algorithms, with the RPROP algorithm. TD(μ) clearly shows best performance.

The performance differences between the TD-algorithms are small but noticeable. TD(μ) performed best in combination with both neural network training algorithms. It achieved 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 was 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.

Another interesting result is that TD(μ) performed better when combined with Residual-λ, but TD-leaf(λ) and TD-directed(λ) performed better with RPROP. Also, in both cases, TD(μ)'s score tended to go down when training for longer periods of time (see figures 19 and 20 in the appendix). We will say something more about this in the next section.

The best performance that was observed was about 80% wins against WALLY. It might well be that achieving better performance than this was extremely difficult, because every game was started with two random moves. Not only does this vastly expand the problem space, it also adds games that are started at a position that gives the agent a disadvantage. This makes it virtually impossible to win 100% of the games. As noted before, when no random start positions were involved the network could easily be trained to beat WALLY every single game. So there is a trade-off between using randomisation to guarantee the sampling of the entire problem space, and getting high winning averages against the current opponent.

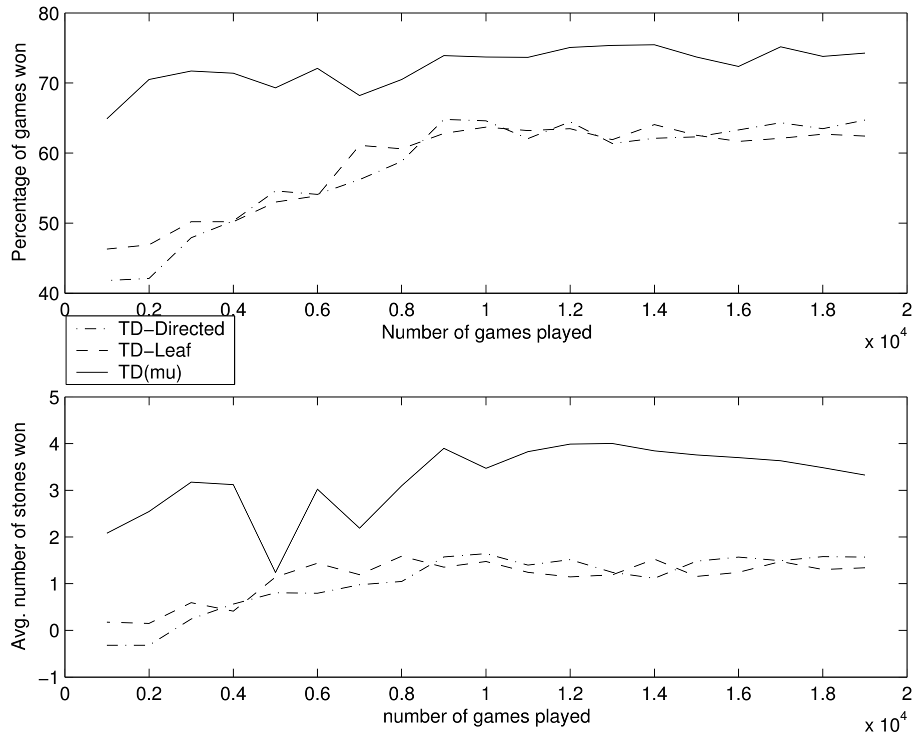


Figure 17: Performance for three different TD-learning algorithms, with Residual-λ. Again, TD(μ) performs best.

The exploration of the state space is also very important if we want to learn 'general' Go-knowledge. If we do insufficient exploration, the agent might learn a simple strategy that will beat WALLY, but reflects no real knowledge of the game. Unfortunately, it is not possible to draw strong conclusions about the Go knowledge of our agent from the data shown above. To test whether our agent had learned any valuable information about Go, we have implemented two other tasks. We used these tasks to test the performance of our network after training it against WALLY.

6.2.1 Performance on Other Tasks

We implemented the following two tasks for testing the Go knowledge of the network: the scoring of end positions, and playing against another opponent, GNUGO. While training against WALLY, the state of the networks weights would be saved every 500 games, making it possible to test the networks after training on these other tasks.

1. Scoring endpositions

A testset of 1000 endgame positions was generated by playing the best version of our program against WALLY. Then a network was loaded and used to evaluate these board positions. Their evaluation was compared with the evaluation from GnuGo. Every position was presented for evaluation once as black and once as white. Figure 18 shows the results for this test. The quantities shown are the percentage of positions that were correctly classified as a win or a loss, and the mean squared error in the output of the network.

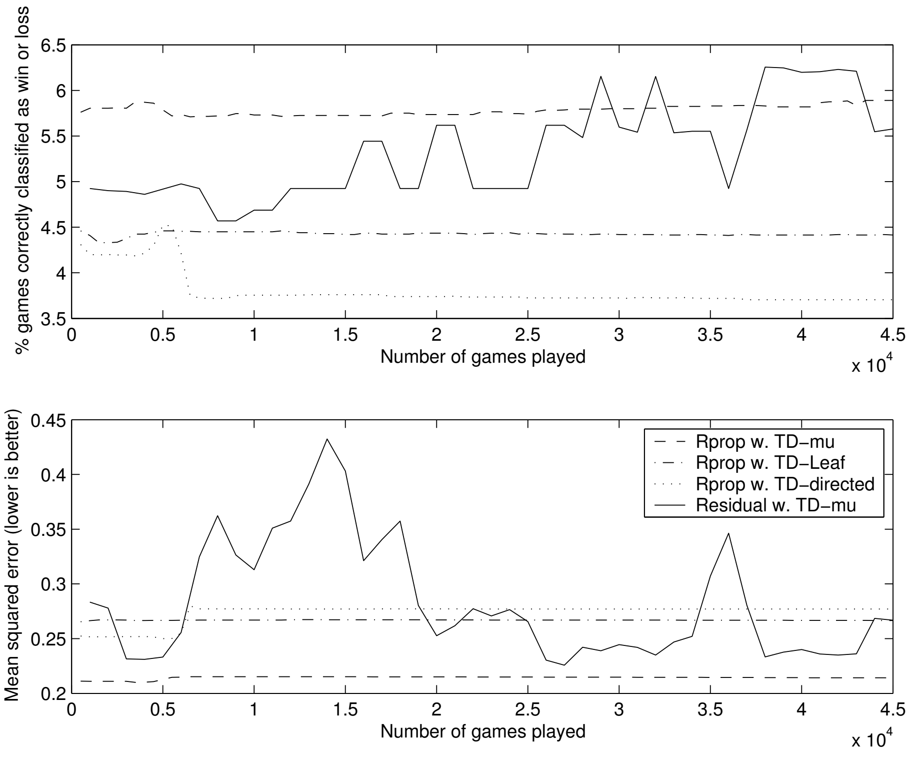


Figure 18: Performance of the network on scoring endpositions

In figure 18 we see the results for this task with the three TD-learning algorithms combined with RPROP, and for TD(μ) with Residual-λ. The other TD-algorithms showed very similar results with Residual-λ. We did not add the results for these two in this plot, but instead chose to compare this single one with the results from the networks trained with RPROP.

Again, TD(μ) shows the best results, supporting our expectations and Beal's claim that TD(μ) would learn 'good' information from a 'bad' opponent. Also, the difference between TD-leaf(λ) and TD-directed(λ) is more clear (but still very small).

It seems like no progress at all is made by the algorithms when using RPROP. Only TD(μ) shows a very slight increase, but TD-directed(λ) even shows a steep drop in performance at about 700 games. On the other hand, with Residual-λ we see that the number of correctly classified games does increase over time. Actually, all three of the TD-variants trained with Residual-λ start off worse than TD(μ), but rise to a performance of more than 60% correct classifications (this is not shown in the figure).

This can be understood if we realise that the residual algorithm prevents the network from directly following the error gradient for the last position in the game, even though there is a direct feedback for that position. Instead, it always takes the gradient for the previous position into account too. However, the network eventually reaches similar results as with RPROP. For the TD-leaf(λ)and TD-directed(λ) algorithms, it even performs better than RPROP.

Apparently scoring endpositions is not the part of the Go playing task that causes the agent to learn to beat WALLY. It seems that even at the very beginning, when training against a nearly random playing opponent, the network learned to score endpositions as well as it could. The explanation for this is that feedback for the endposition is always given immediately. For all other positions in the game, information has to be propagated back, and thus will take longer to learn.

In the previous section we mentioned that the average score for TD(μ) tends to go down over time, but its winning percentage stays approximately the same. In figure 18 we see that the performance for the scoring of endpositions goes up. This might be supportive of Beal's claim that TD(μ) is able to learn from a player that plays worse than itself. Is the increasing performance at scoring endgame positions the effect of the agent learning 'general' Go knowledge beyond the level that WALLY has to offer? If so, we might explain the decreasing score by saying that the agent is not always choosing the move which will maximise the score against WALLY, but instead it chooses a 'solid' move which maximises a more general chance of winning. Unfortunately, these questions are hard to answer, and we can currently only speculate on the right interpretation of these results.

2. Playing against Gnugo

As another test of how much our network has learned, we let the networks that were trained against WALLY play against another opponent, GNUGO. Because it uses some sophisticated knowledge about Go, like pattern databases and specialised searches for common tactical situations, GNUGO has quite an advantage, so we decided to set it to the lowest level. Furthermore, we set the alpha-beta search used by our program for playing to a level of 3 ply. The table below shows the best results for each combination of learning algorithms.

NN Algorithm	TD Algorithm	% of games won	Avg. score
Residual	TD-directed(λ)	11	2.7
Residual	TD-leaf(λ)	11	2.7
Residual	TD(μ)	19	4.5
Residual-λ	TD-directed(λ)	45	7.0
Residual-λ	TD-leaf(λ)	41	6.2
Residual-λ	TD(μ)	44	7.7
RPROP	TD-directed(λ)	42	8.5
RPROP	TD-leaf(λ)	44	8.3
RPROP	TD(μ)	49	7.4

Table 1: Performance for testing against GNUGO. The networks were trained against WALLY.

Although none of the algorithms was trained to beat GNUGO, the results show that scores close to 50% were achieved, indicating that the network evaluation function represented by our network plays almost as well as GnuGo. Again, there are clear differences in performance between the different algorithms. RPROP and Residual-λ show much better performance than the residual algorithm. Also, we see that again TD(μ) tends to perform best, and the scores of TD-leaf(λ) and TD-directed(λ) are quite close to each other. In the case of Residual-λ, TD-directed(λ) performs best, but the difference is too small too draw any conclusions from this.

6.3 Statistical Significance

From the results presented so far, can we conclude that the differences between the various algorithms are significant? Some statistical analysis of the data is needed to answer this question. First, let us consider the data for training the agent against Wally. This includes the data presented in section 5 about network training algorithms. For all the plots shown, the data has been averaged by using a running average of 1000 points. If we assume these samples are independent we can calculate confidence intervals for this data.

The 95% confidence interval for the winning percentages lies around 1.3%. In other words, the real winning percentage has a 95% chance of being in an interval of $[-0.6, +0.6]$ around the percentage shown in the plot. For the scores, the 95% confidence interval is $[-0.2, +0.2]$. These numbers are representable for all plots. We can interpret this by saying that the performance differences we found are significant, and the choice of a neural network training algorithm or a Temporal-Difference Learning algorithm really makes a significant difference. Only where the differences are very small, like TD-directed(λ) and TD-leaf(λ) in figure 17, we cannot speak of a significant performance difference.

We have also done a Chi-square test on the data of testing the agent against GnuGo. For the choice of network training algorithm we find a p-value of 0.0000 and for the choice of Temporal-Difference Learning algorithm, we find a p-value of 0.0005. These values indicate the chance of finding a difference in performance while there is no difference in reality. They are extremely small, indicating a very high level of significance. In other words, the choice of a right network training algorithm clearly matters, and so does the choice of a Temporal-Difference Learning algorithm.

6.4 Discussion

We have tested different TD-learning algorithms for playing Go on a small board. As expected, we find that the TD(μ) algorithm offers better learning and, in most cases, performs better than the other two algorithms. For TD-leaf(λ), some small difference with TD-directed(λ) was found, but this was only marginal. It might be that the difference is bigger when using deeper search.

When we only look at the data from training against Wally, we see that with both RPROP and Residual-λ, winning rates of about 80% are achieved, with an average score of about 4 stones. In both cases, TD(μ) is clearly the best choice of TD-algorithm.

From the two other tests, we can conclude that there is at least some learning of genuine Go knowledge. The clearest evidence of this is the fact that we reach almost 50% wins agains GnuGo. Furthermore, TD(μ) again yields best performance. This seems to indicate that this algorithm does indeed learn more real Go-playing information than the other algorithms.

We also find that the choice of TD-learning algorithm does not matter that much when training with Baird's residual algorithm, since the differences observed are only very small. This is probably due to the fact that we can only use $\lambda = 0$. The results with Residual-λ seem to confirm this.

Generally, it seems that the biggest performance gains are to be found in providing the network with the right feature set, and choosing the right network training algorithm. This makes the problem 'learnable' for the network. Once these choices are made appropriately, one can try to optimise learning by choosing a good TD-learning algorithm.

7 Discussion

We have investigated a number of Machine Learning algorithms for use with gameplay. Testing these algorithms was done by training a neural network to play 5x5 Go against a weak opponent. We will now discuss our main findings.

7.1 Implementation

As our opponent, we chose WALLY, a simple public-domain Go-playing program. We used another program, GNUGO, to score the games and determine the winner. The scores were given as rewards at the end of the game, and during the game rewards were given for captures. Using a 2-ply search, the network could be trained to beat WALLY every time, showing its ability to learn. By starting every game with two random moves, we made the learning task more challenging and ensured learning of 'real' Go knowledge.

7.2 The Network

It is very important that the neural network is able to represent the evaluation function. This amounts to finding a suitable network topology and feature set. A multi-layer perceptron with 32 inputs, 75 hidden neurons and a single output was used. Finding a good featureset turned out to be crucial for being able to learn.

When a good featureset was found, we proceeded to test several network training algorithms. The RPROP algorithm learned significantly faster than standard backpropagation, which did not learn to beat WALLY even after very long periods of training. Although it was expected to perform very slow, Baird's residual algorithm learned even faster than RPROP. The best algorithm was Residual-λ, an adaptation of Baird's algorithm for use with TD(λ).

7.3 Temporal-Difference Learning and Gameplay

Having succeeded to train the network to play Go, we could test various Temporal-Difference Learning algorithms. We found TD(μ), an algorithm which prevents learning from bad moves, to give significantly better performance than other algorithms. The two other algorithms that were tested were TD-directed(λ) and TD-leaf(λ), both of which incorporate tree searching in the learning algorithm. TD-leaf(λ) was found to perform marginally better than TD-directed(λ), but not enough to make strong claims. This might be different with deeper tree searches. The TD-directed(λ) algorithm gave poorest performance of all.

After training, we tested the agents on two other tasks to verify the results. Both tests, scoring endpositions and playing against a stronger opponent, supported our findings.

8 Conclusion

We will now answer our main question: will adapting the Reinforcement Learning model to incorporate domain-specific knowledge about games improve learning performance?

We have found that incorporating the existence of an opponent into the model of the worlds dynamics, as done by the TD(μ) algorithm, significantly increases performance compared to other TD-learning algorithms that do not use this knowledge. For TD-leaf(λ), which takes into account the searching of the game space, some performance differences were found, but these are too small to draw serious conclusions. This means that the answer to our research question should be positive. We can improve Reinforcement Learning performance by using domain-specific knowledge. But apparently some knowledge makes a bigger difference than other knowledge.

An important note is that one should first optimise the training of the network through the combination of the right feature set, topology and training algorithm. There is more performance to be gained by making these choices in the right way, than by the choice of TD-algorithm. When an optimal network training strategy is chosen, the right TD-algorithm does make a significant difference, though.

Residual algorithms are found to combine quite well with TD-learning, as they were designed to do. It is surprising to see that training with it was even faster than RPROP, even though the literature suggests otherwise. Although the residual algorithm yielded good results against WALLY, the limitation to $\lambda = 0$ is an important disadvantage.

One of our most interesting findings is the performance of the Residual-λ algorithm. The fact that it performs so well, makes the combination of residuals and TD(λ) an interesting theoretical issue. It is not clear whether stability can be guaranteed when combining these methods.

8.1 Future work

During the project we came across several issues which might be interesting subjects for futher scientific investigation.

- **TD-leaf(μ)**
  Given the performance of TD(μ) and TD-leaf(λ), the question arises whether we can combine these algorithms to make use of both game tree search and the critical evaluation of gameplay. Combining these algorithms should not be too difficult, and might give even better performance.

- **Residual-λ**
  Combining Baird's residual algorithm with a $\lambda > 0$ gave some very promising results. However, there is no strong mathematical basis for this, so we cannot make any claims about it being stable or converging. Since it performed so well on our task, it would be very interesting to see some mathematical analysis of the method. One of the major disadvantages of Reinforcement Learning is its slow learning speed in comparison to supervised learning, so it is important to explore methods to accelerate learning.

- **Residuals and accelerated gradient descent**
  Baird's residual algorithm as an adaptation of backpropagation shows faster learning than the accelerated RPROP algorithm. This raises the question whether a combination of residuals with accelerated gradient descent might give even better performance. However, for the combination of RPROP and residuals, proving convergence is not trivial. Both methods depend on error gradients in previous time steps, and both descend on a different gradient than the actual error gradient. As with residual-λ, it would be interesting to see whether we can use these methods to speed up learning.

- **Network topology**
  The Go board is symmetric, rotation and translation invariant (except at the edges). Also, some points are connected to each other, and others are not. It would be nice to have the topology of a neural network reflect these properties. Techniques that may be used include weight sharing, convolutional networks, and lateral connections.

- **TD(μ) and other domains**
  The TD(μ) algorithm was designed with the gameplaying domain in mind. But there is no clear reason why it could not be used in other domains. The algorithm might just as well be used in an environment without an opponent. It would then just observe and criticise the agent's behaviour. This may be particularly useful in tasks where the policy for choosing actions or the outcome of an action are non-deterministic, or when learning from observation. TD(μ) might be able to learn more from suboptimal behaviour than other algorithms.

- **Vector-valued reinforcement**
  Scoring a Go position means giving a value to every point on the board, designating whether it belongs to black, white, or to none of them. This means that we have more information than the score, and a neural network might give such a value for every point on the board as its output. The reason we do not do this, is that Temporal-Difference Learning does not give us a method to propagate vectors back in time, so our reinforcement signal is limited to a scalar value. However, Schmidhuber[19] describes a method for vector-valued reinforcement, and Schraudolph[20] also mentions it in his paper about training a neural network for Go. Having such a rich reinforcement signal might make the learning of an evaluation function many times easier.

- **Go: bigger board, stronger opponent**
  The learning task that we used to compare the performance of algorithms on, was very limited. It would be very interesting to try these techniques with Go on bigger boards and against stronger opponents. To create a strong game-playing agent, some extra functionality should be implemented like modules for detecting ladders.

A Some extra figures

The two extra figures included here show performance of the agent on when playing 5x5 Go against WALLY. The data presented is for the same experiment was shown in section 6.2, but this time we show results for training over a longer period of time. These plots are also discussed in that section.

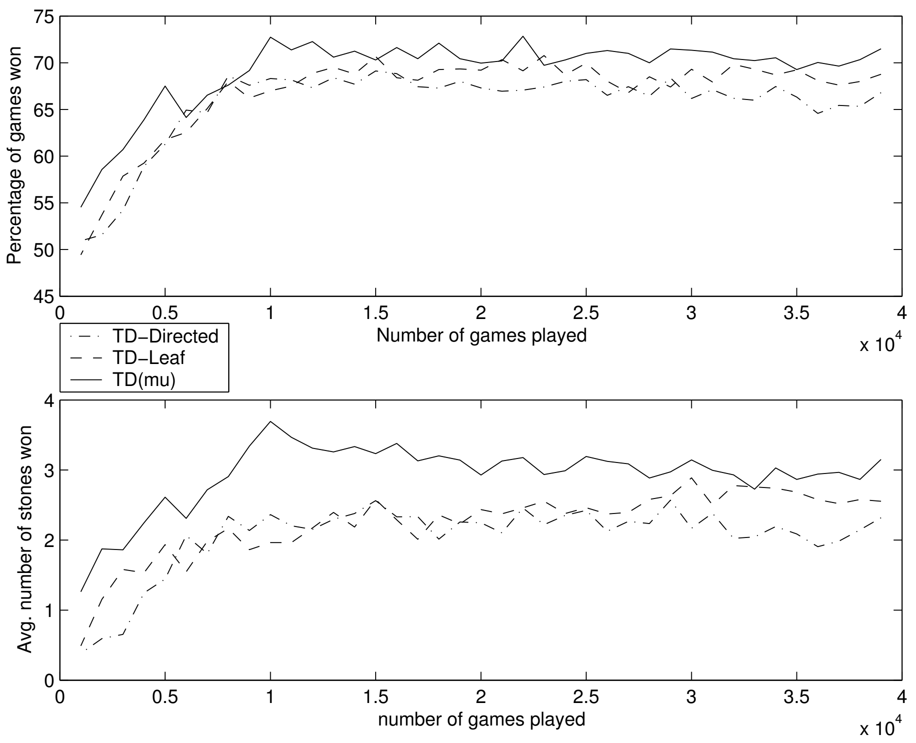


Figure 19: Performance for three different TD-learning algorithms

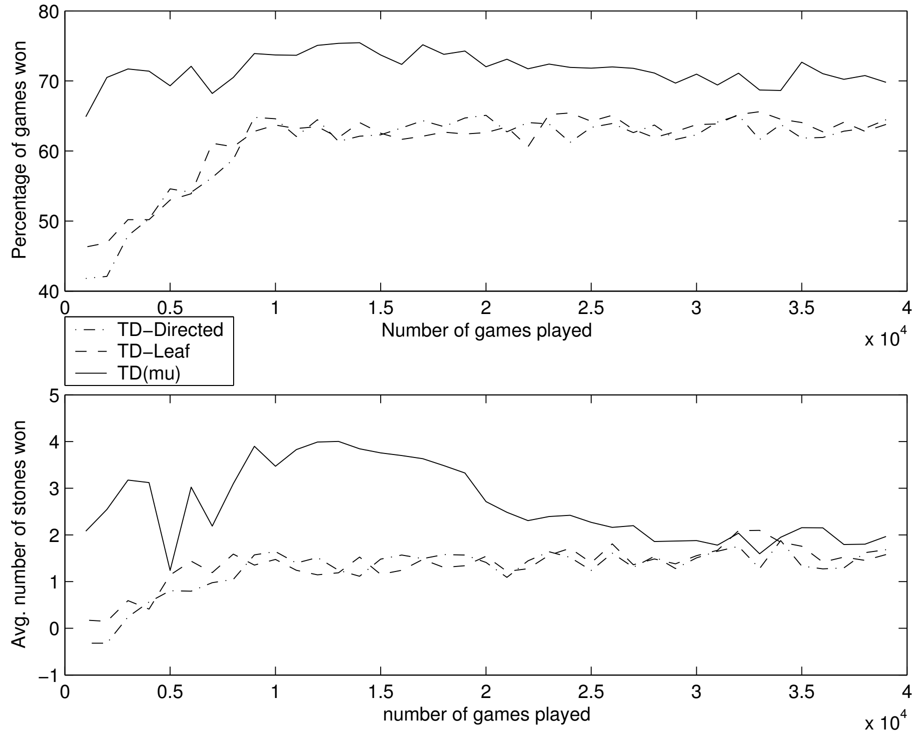


Figure 20: Performance for three different TD-learning algorithms with residual-lambda

References

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

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

[3] D.F. Beal. 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, 2002.

[4] D. B. Benson. *Life in the Game of Go*, pages 203–213. Springer-Verlag, 1988.

[5] B. Bouzy and T. Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132:39–103, October 2001.

[6] J. Burmeister and J. Wiles. An introduction to the computer Go field and associated internet resources. Technical Report 339, Department of Computer Science, University of Queensland, 1995.

[7] F. A. Dahl. Honte, a Go-playing program using neural nets. In *16th International Conference on Machine Learning (ICML-99)*, June 1999.

[8] E. Van der Werf, J. Van den Herik, and J. Uiterwijk. Learning to score final positions in the game of Go. *To appear*, 2003.

[9] E. Van der Werf, J. Van den Herik, and J. Uiterwijk. Solving Go on small boards. *To appear in ICGA Journal*, 26(3), June 2003.

[10] E. Van der Werf, M. Winands, J. Van den Herik, and J. Uiterwijk. Learning to predict life and death from go game records Go. *To appear*, 2003.

[11] Herbert D. Enderton. The Golem Go program. Technical Report CMU-CS-92-101, School of Computer Science, Carnegie Mellon University, december 1991.

[12] S. Haykin. *Neural Networks, a comprehensive foundation.* Prentice Hall, 2nd edition edition, 1999.

[13] M. Müller. Playing it safe: Recognizing secure territories in computer go by using static rules and search, 1997.

[14] M. Müller. Computer Go. *Artificial Intelligence*, 134:145–179, January 2002.

[15] D. Precup, R. S. Sutton, and S. Dasgupta. Off-policy temporal-difference learning with function approximation. In *Proc. 18th International Conf. on Machine Learning*, pages 417–424. Morgan Kaufmann, San Francisco, CA, 2001.

[16] S. I. Reynolds. *Reinforcement Learning with Exploration.* PhD thesis, School of Computer Science, University of Birmingham, december 2002.

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

[18] S. Russel and P. Norvig. *Artificial Intelligence, a modern approach.* Prentice Hall, 1995.

[19] J. Schmidhuber. Reinforcement learning in markovian and non-markovian environments. In Richard P. Lippmann, John E. Moody, and David S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 500–506. Morgan Kaufmann Publishers, Inc., 1991.

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

[21] D. Stoutamire. Machine learning, game play and Go. Technical report, Case Western Reserve University, Cleveland, Ohio, 1991.

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

[23] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* Bradford, 1998.

[24] G.J. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38:58–68, 1995.

[25] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, Massachusets Instutute of Technology, 1996.

[26] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In *Proceedings of the 3rd International Conference on Computers and Games*, July 2002.