Reinforcement learning in board games.

Imran Ghory

May 4, 2004

Abstract

This project investigates the application of the $TD(\lambda)$ reinforcement learning algorithm and neural networks to the problem of producing an agent that can play board games. It provides a survey of the progress that has been made in this area over the last decade and extends this by suggesting some new possibilities for improvements (based upon theoretical and past empirical evidence). This includes the identification and a formalization (for the first time) of key game properties that are important for TD-Learning and a discussion of different methods of generate training data. Also included is the development of a TD-learning game system (including a game-independent benchmarking engine) which is capable of learning any zero-sum two-player board game. The primary purpose of the development of this system is to allow potential improvements of the system to be tested and compared in a standardized fashion. Experiments have been conduct with this system using the games Tic-Tac-Toe and Connect 4 to examine a number of different potential improvements.

1 Acknowledgments

I would like to use this section to thank all those who helped me track down and obtain the many technical papers which have made my work possible. I hope that the extensive bibliography I have built up will help future researchers in this area.

I would also like to thanks my family, friends and everyone who has taken time out to teach me board games. Without them I would never have had background necessary to allow me to undertake this project.

Contents

2 Why research board games ?

> "If we choose to call the former [Chess-Player] a pure machine we must be prepared to admit that it is, beyond all comparison, the most wonderful of the inventions of mankind."
>
> Edgar Allan Poe, *Maelzel's Chess-Player* (c. 1830)

As the Poe quote shows the aspiration towards an AI that can play board games far pre-dates modern AI and even modern computing. Board games form an integral part of civilization representing an application of abstract thought by the masses, the ability to play board games well has long been regarded as a sign of intelligence and learning. Generations of humans have taken up the challenge of board games across all cultures, creeds and times, the game of chess is fourteen hundred years old, backgammon two thousand years but even that pales to the three and half millennia that tic-tac-toe has been with us. So it is very natural to ask if computers can equal us in this very human pastime.

However we have far more practical reasons as well,

- Many board games can seen as simplified codified models of problems that occur in real-life.
- There exist tried-and-tested methods for comparing the strengths of different players, allowing for them to be used as a testbed to compare and understand different techniques in AI.
- Competitions such as the Computer Olympiad have made developing games AIs into an international competitive event, with top prizes in some games being equal to those played for by humans.

3 What is reinforcement learning ?

The easiest way to understand reinforcement learning is by comparing it to supervised learning. In supervised learning an agent is taught how to respond to given situation, in reinforcement learning the agent is not taught how to behave rather it has a "free choice" in how to behave. However once it has taken its actions it is then told if its actions were good or bad (this is called the reward – normally a positive reward indicates good behaviour and a negative reward bad behaviour) and has to learn from this how to behave in the future.

However very rarely is an agent told directly after each action whether the action is good or bad. It is more usual for the agent to take several actions before receiving a reward. This creates what is known as the "temporal credit assignment problem", that is if our agent takes a series of actions before getting the award how can we take that award and calculate which of the actions contributed the most towards getting that award.

This problem is obviously one that humans have to deal with when learning to play board games. Although adults learn using reasoning and analysis, young

children learn differently. If one considers a young child who tries to learn a board game, the child attempts to learn in much the same way as we want our agent to learn. The child plays the board game and rather than deducing that they have won because action X forced the opponent to make action Y, they learn that action X is correlated with winning and because of that they start to take that action more often. However if it turns out that using action X results in them loosing games they stop using it. This style of learning is not just limited to board games but extends many activities that children have to learn (i.e. walking, sports, etc.) .

The technique I intend to concentrate on, one called TD(λ) (which belongs to the Temporal Difference class of methods for tackling the temporal credit assignment problem) essentially learns in the same way. If our TD(λ) based system observes a specific pattern on the board being highly correlated with winning or loosing it will incorporate this information into its knowledge base. Further details of how it actually learns this information and how it manages its knowledge base will be discussed more in depth in section 6.2.

4 The history of the application of Reinforcement Learning to Board Games

Reinforcement learning and temporal difference have had a long history of association with board games, indeed the basic techniques of TD-Learning were invented by Arthur Samuel [2] for the purpose of making a program that could learn to play checkers. Following the formalization of Samuel's approach and creation of the TD(λ) algorithm by Richard Sutton [1], one of the first major successful application of the technique was by Gerald Tesauro in developing a backgammon player.

Although Tesauro developed TD-Gammon [12] in 1992 producing a program that amazed both the AI and Backgammon communities equally, only limited progress has been made in this area, despite dozens of papers on the topic being written since that time. One of the problems this area has faced is that attempts to transfer this miracle solution to the major board games in AI, Chess and Go, fell flat with results being worse than those obtained via conventional AI techniques. This resulted in many of the major researchers in board game AIs turning away from this technique.

Because of this for several years it was assumed that the success was due to properties inherent to Backgammon and thus the technique was ungeneralizable for other games or other problems in AI, a formal argument for this being presented in [40].

However from outside the field of board game AI, researchers apparently unaware of the failures of the technique when applied to chess and go decided to implement the technique with some improvements for the deterministic game of Nine Men's Morris [11], and succeeded in not only producing a player that could perform not only better than humans, but could play at a level where it

lost less than fifty percent of games against a perfect opponent (developed by Ralph Gasser). There have been a number of attempts to apply the technique to Othello[28] [31] [47] however results have been contradictory. A number of other games were also successfully tackled by independent researchers.

Since that time the effectiveness of the TD(λ) with neural network is now well accepted and especially in recent years the number of researchers working in this area have grown substantially.

One result of this has been the development of the technique for purposes which were originally unintended, for example in 2001 it was shown by Kanellopoulos and Kalles [45] that it could be used by board game designers to check for unintended consequences of game modifications.

A side effect of a neural network approach is that it also makes progress on the problem of transferring knowledge from one game to another. With many conventional AI approaches even a slight modification in game rules can cause an expert player to become weaker than a novice, due to a lack of ability to adapt. Due to the way the neural network processes game information and TD(λ) learns holds the potential to be able to learn variants of a game with minimal extra work as knowledge that applies to both games is only changed minimally as only those parts which require new knowledge change.

5 Traditional approaches to board game AIs and the advantages of reinforcement learning

The main method for developing board game playing agents involves producing an evaluation function which given the input of a "board game position" returns a score for that position. This can then be used to rank all of the possible moves and the move with the best score is the move the agent makes. Most evaluation functions have the property that game positions that occur closer to the end of a game are more accurately evaluated. This means the function can be combined with a depth-limited minimax search so that even a weak evaluation function can perform well if it "looks ahead" enough moves. This is essentially the basis for almost all board game AI players.

There are two ways to produce a perfect player (a game in which a perfect player has been developed is said to have been solved),

1. Have an evaluation function which when presented with the final board position in a game can identify the game result and combine this with a full minimax search of all possible game positions.

2. Have an evaluation function which can provide the perfect evaluation of any game board position (normally know as a *game theoretic* or *perfect* function. Given a perfect function, search is not needed.

While many simple games have been solved by brute-force techniques involving producing a look-up table of every single game position, most commonly

played games (chess, shogi, backgammon, go, othello, etc.) are not practically solvable in this way. The best that can be done is to try and approximate the judgments of the perfect player by using a resource-limited version of the above listed two techniques.

Most of the progress towards making perfect players in generalized board game AIs has been in the first area, with many developments of more optimal versions of minimax, varying from the relatively simple alpha-beta algorithm to Buro's probabilistic Multi-ProbCut algorithm [42].

Although these improvements in minimax search are responsible for the worlds strongest players in a number of games (including Othello, Draughts and International checkers) they are of limited use when dealing with games with significantly large branching factors such as Go and Arimaa due to the fact that the number of boards that have to be evaluated increase at an exponential rate.

The second area has remained in many ways a black art, with evaluation functions developed heuristically through trial-and-error and often in secrecy due to the competitive rewards now available in international AI tournaments. The most common way for an evaluation function to be implemented is for it to contain a number of hand-designed feature detectors that identify important structures in a game position (for instance a King that can be checked in the next move), the feature detectors returning a score depending on how advantageous that feature is for the player. The output of the evaluation function is then normally a weighted sum of the outputs of the feature detectors.

Why Tesauro's original result was so astounding was because it didn't need any hand-coded feature vectors; it was able with nothing other than a raw encoding of the current board state to produce an evaluation function that could be used to play at a level not only far superior to previous backgammon AIs but equal to human world champions. When Tesauro extended TD-Gammon adding the output of hand-designed feature detectors as inputs to TD-Gammon it played at an then unequaled level and developed unorthodox move strategies that have since become standard in international human tournaments.

What is most interesting is how advanced Tesauro's agent became with no information other than the rules of the game. If the application of TD(λ) to neural networks could be generalized and improved so that it could be applied so successfully to all games it would represent a monumental advance not only for board game researchers but also for all practical delayed-reward problems with finite state spaces.

6 How does the application of TD-Learning to neural networks work ?

The technique can be broken down into two parts, evaluation and learning, evaluation is the simplest and works on the same standard principles of all neural networks. In previous research on TD-Learning based game agents virtually all work has been done using one and two layered neural networks.

The reason for using neural networks is that for most board games it is not possible to store all possible boards, so hence neural networks are used instead. Another advantage of using neural networks is that they allow generalization, that is if a board position has never been seen before they can still evaluate it on the basis of knowledge learnt when the network was trained on similar board positions.

6.1 Evaluation

The network has a number of input nodes (in our case these would be the raw encoding of the board position) and an output node(s) (our evaluation of our board position). A two-layered network will also have a hidden layer which will be described later. For a diagram of these two types of network see figure 1.

All of the nodes on the first layer are connected by a weighted link to all of the nodes on the second layer.

In the one-layer network the output is simply a weighted sum of the inputs, so with three inputs $Output = I_1W_1 + I_2W_2 + I_3W_3$. This can be generalized to $Output = \sum_{i=1}^{N} I_iW_i$ where N is the number of input nodes.

However this can only be used to represent a linear function of the input nodes, hence the need for a two-layer network. The two-layer network is the same as connecting two one-layer networks together with the output of the first network being the input for the second network, with one important difference - in the hidden nodes the input is passed through a non-linear function before being outputted. So the output of this network is $Output = f(I_1W^I_{1,1} + I_2W^I_{2,1} + I_3W^I_{3,1})W^O_1 + f(I_1W^I_{1,2} + I_2W^I_{2,2} + I_3W^I_{3,2})W^O_2$. This can be generalized to,

$$Output = \sum_{k=1}^{H} f(\sum_{j=1}^{N} I_{j,k}W^I_j)W^O_k$$

N is the number of input nodes.

H is the number of hidden nodes.

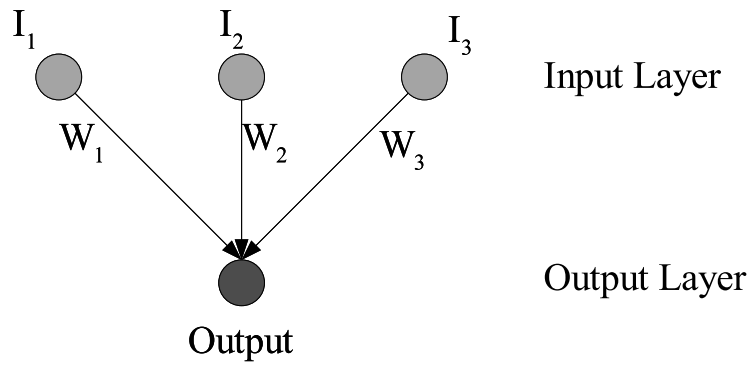
f() is our non-linear function.

It is common to use $f(\alpha) = \frac{1}{1+e^{-\alpha}}$ for the non-linear function because this function has itself as its derivative. In the learning process we often need to calculate $f(x)$ and also $\frac{df}{dx}(x)$. With the function given above $\frac{df}{dx}(x) = f(x)$ so we only have to calculate the value once and hence halve the number of times we have to calculate this function. Profiling an implementation of the the process reveals that the majority of time is spent calculating this function, so optimization is an important issue in the selection this function.

6.2 Learning

Learning in this system is a combination of the back-propagation method of training neural networks and TD(λ) algorithm from reinforcement learning. We

One layer network



Two layer network

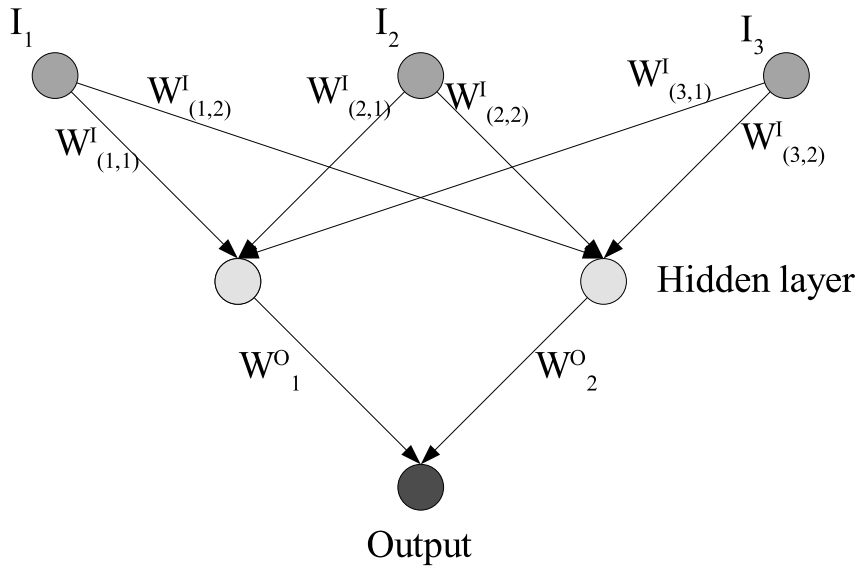


Figure 1: Neural network

use the second to approximate our game theoretic function, and the first to tell the neural network how to improve. If we had a game-theoretic function then we could use it to train our neural network using back-propagation, however as we don't we need to have another function to tell it what to approximate. We get this function by using the TD(λ) algorithm which is a methods of solving the temporal credit assignment problem (that is when you are rewarded only after a series of decisions how do you decide which decisions were good and which were bad). Combining these two methods gives us the following formula for calculating the change in weights,

$$\Delta W_t = \alpha \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k d_t$$

t is time (in our case move number).

T is the final time (total number of moves).

Y_t is the evaluation of the board at time t when $t \neq T$.

Y_T is the true reward (i.e. win, loss or draw).

α is the learning rate.

$\nabla_w Y_k$ is the partial derivative of the weights with respect to the output.

d_T is the *temporal difference*.

To see why this works consider the sequence $Y_1, Y_2, ..., Y_{T-1}, Y_T$ which represents evaluations of all the boards that occur in a game. Let us define Y'_t as the perfect game-theoretic evaluation function for the game. The *temporal difference*, d_t is defined as the difference between the prediction at time t and $t+1$, d_t=$Y_{t+1} - Y_t$. If our predictions were perfect (that is $Y_t = Y'_t$ for all t) then d_t would be equal to zero. It is clear from our definition we can rewrite our above formula as,

$$\Delta W_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

If we take $\lambda = 0$ this becomes,

$$\Delta W_t = \alpha(Y_{t+1} - Y_t) \nabla_w Y_k$$

From this we can see that our algorithm works by altering the weights so that Y_t becomes closer to Y_{t+1}. It is also clear how α alters the behaviour of the system; if $\alpha = 1$ then the weights are altered so that $Y_t = Y_{t+1}$. However this would mean that the neural network is changed significantly after each game, and in practice this prevents the neural network from stabilizing. Hence in practice it is normal to use a small value of α or alternatively start with a high value of α and reduce it as learning progresses.

As $Y_T = Y'_T$, that is the evaluation at the final board position is perfect, Y_{T-1} will given enough games become capable of predicting Y_T to a high level of accuracy. By induction we can see how Y_t for all t will become more accurate as the number of games learnt from increase. However it can be trivially shown that Y_t can only be as accurate as Y_{t+1}, which means that for small values of t in the early stages of learning Y_t will be almost entirely inaccurate.

If we take $\lambda = 1$ the formula becomes,

$$\Delta W_t = \alpha(Y_T - Y_t)\nabla_w Y_k$$

That is rather than making Y_t closer to Y_{t+1}, it make Y_t become closer to Y_T (the evaluation of the final board position). For values of λ between 0 and 1, the result falls between these two extremes, that is all of the evaluations between Y_T and Y_{t+1} influence how the weights should be changed to alter Y_t.

What this means in practice is that if a board position frequently occurs in won games, then its score will increase (as the reward will be one), but if it appears frequently in lost games then its score will decrease. This means the score can be used as a rough indicator of how frequently a board position is predicted to occur in a winning game and hence we can use it as an evaluation function.

6.3 Problems

The main problem with this technique is that it requires the player to play a very large amount of games to learn an effective strategy, far more than a human would require to get to the same level of ability.

In practice this means the player has to learn by playing against other computer opponents (who may not be very strong thus limiting the potential of our player) or by self-play. Self-play is the most commonly used technique but is known to to be vulnerable to short-term pathologies (where the program gets stuck exploiting a single strategy which only works due to a weakness in that same strategy) which although over the long term would be overcome, result in a significant increase the number of games that have to be played. Normally this is partially resolved by adding a randomizing factor into play to ensure a wide diversity of games are played.

Because of this problem the most obvious way to both improve the playing ability for games with small state-spaces and those yet out of reach of the technique is to increase the (amount of learning)/(size of training data) ratio.

7 Current State-of-the-art

In terms of what we definitely know, only little progress has been made since 1992, although we now know that the technique certainly works for Backgammon (There are at least seven different Backgammon program which now use TD-learning and neural networks, including the open source GNU Backgammon,

and the commercial Snowie and Jellyfish. TD-Gammon itself is still commercially available although limited to the IBM OS/2 platform.)

The major problem has not been lack of innovation, indeed a large number of improvements have been made to the technique over the years, however very few of these improvements have been independently implemented or tested on more than one board game. So the validity of many of these improvements are questionable.

In recent years researchers have grown re-interested in the technique, however for the purposes of tuning hand-made evaluation functions rather then learning from raw encoding. As mentioned earlier many conventional programs use a weighted sum of their feature detectors. This is the same as having a neural network with one layer using the output of the feature detectors as the inputs.

Experiments in this area have shown startling results Baxter, et al.[9] found that using TD-Learning to tune the weights is far superior to any other method previously applied. The resulting weights were more successful than weights that had been hand-tuned over a period of years. Beal [19] tried to apply the same method using only the existence of pieces as inputs for the games of Chess and Shogi. This meant the program was essentially deciding how much weight each piece should be worth (for example in Chess conventional human weights are pawns: 1, knight: 3, bishop: 3, rook: 5), his results showed a similar level of success. Research is currently continuing in this area, but current results indicate that one of the major practical problems in developing evaluation functions may have been solved. The number of games that need to be played for learning these weights is also significantly smaller than with the raw encoding problem. The advantage of using TD-Learning over other recently developed methods for weight tuning such as Buro's *generalized linear evaluation model* [42] is that it can be extended so that non-linear combinations of the feature detectors can be calculated.

An obvious extension to this work is the use of these weights to differentiate between effective and ineffective features. Although at first this appears to have little practical use (as while a feature has even a small positive impact there seems little purpose in removing it), Joseph Turian [14] innovatively used the technique to evaluate features that were randomly generated, essentially producing a fully automated evolutionary method of generating features.

The success in this area is such that in ten years time I would expect this method to be as commonplace as alpha-beta search in board game AIs.

8 My research

Firstly in section 9 I produce an analysis of what appear to be the most important properties of board games, with regards to how well TD-Learning systems can learn them. I present new formalizations of several of these properties so that they can be better understood and use these formalizations to show the importance of how the board is encoded.

I then in section 11 go on to consider different methods of obtaining training data, before moving onto a consideration of a variety of potential improvements to the standard TD-Learning approach, both those developed by previous researchers and a number of improvements which I have developed based upon analyzing the theoretical basis of TD-Learning and previous empirical results.

The core development part of my research will primarily consist of implementing of a generic TD(λ) game-learning system which I will use to test and investigate a number of the improvements earlier discussed, both to verify or test if they work and to verify that they are applicable to a range of games. In order to do this I develop a benchmarking system that is both game-independent and able to measure the strength of the evaluation function.

Although the game-learning system I developed works on the basis of a raw-encoding of the board and using self-play to generate games, the design is such that it could be applied to other encodings and used with other methods of obtaining training data with minimal changes.

9 Important game properties

Drawing on research from previous experiments with TD-Learning and also research from the fields of neural networks and board games I have deduced that there are four properties of games which contribute towards the ability of TD-Learning based agents to play the game well. These properties are the following,

1. Smoothness of boards.
2. Divergence rate of boards at single-ply depth.
3. State-space complexity.
4. Forced exploration.

9.1 Smoothness of boards

One of the most basic properties of neural networks is that a neural network's capability to learn a function is closely correlated to mathematical smoothness of the function. A function $f(x)$ is smooth if a small change in x implies a small change in $f(x)$.

In our situation as we are training the neural network to represent the game-theoretic function, we need to consider the game-theoretic function's smoothness. Unfortunately very little research has been done into the smoothness of game-theoretic functions so we essentially have to consider them from first principles.

When we talk about boards, we mean the "natural human representation" of the board, that is in terms of a physical board where each position on a board can have a number of possible alternative pieces on that position. An alternative way of describing this type of representation is viewing the board as an array

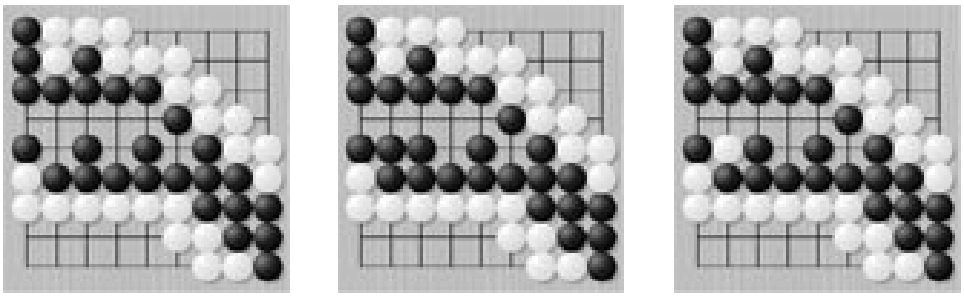


Figure 2: Smooth Go boards. Images taken using the "Many Faces of Go" software.

where the pieces are represented by different values in the array. A non-human representation for example could be one where the board was described in terms of piece structures and relative piece positions,

The reasons for using "natural human representations" are twofolds, firstly as it allows us to discuss examples that humans will be able to understand at an intuitive level and secondly because it is commonplace for programmers designing AIs to actually use a representation that is very similar to a human representation.

If we let $A-B$ be the difference between two boards A and B and let $|A-B|$ be the size of that difference, then a game is smooth if $|A-B|$ being small implies that $|gt(A) - gt(B)|$ is small (where $gt()$ is our game-theoretic function).

An important point to note is that the smoothness is dependent on the representation of the board as much as on the properties of the game itself. To understand the difference consider the following situation: As humans we could consider the three Go boards shown in figure 2 where the difference is only one stone, the "natural" human analysis of these boards would result in one saying that the closest two boards are the first and second or first and third, as changing a black piece to white is a more "significant" change than simply removing a piece.

To show why this is not the only possibility consider the following representation. Let R_A and R_B be binary representations of A and B then we can say that $|R_A - R_B|$ is the number of bits equal to 1 in $(R_A \oplus R_B)$. If we represented our Go pieces as 00 (Black), 01 (White) and 11 (Empty) we would find that the closest boards are the second and third as they have a difference of one while the others have a difference of two.

This highlights the importance of representation to the smoothness of our function. While in theory there exists a perfect representation of the board (that is a representation that maximizes the smoothness of our function) which is derivable from the game, it is beyond our current abilities to find this representation for any but the simplest of games. As a result most board representation that are chosen for smoothness are chosen based upon intuition and trial-and-error. In many cases smoothness is not considered at all and boards are chosen

for reasons other than smoothness (for example optimality for the purpose of indexing or storage).

9.2 Divergence rate of boards at single-ply depth

Although similar to smoothness this property applies slightly differently. Using the same language as we did for smoothness, we can describe it in the following sense.

Let $A1, A2, ..., An$ be all distinct derived boards from A (that is all of the boards that could be reached by a single move from a game position A), then we can define $d(A) = (\sum(|A1 - A2|))/2 * n!$ for all pairs of derived boards $A1, A2$. The divergence rate for a game can now be defined as the average of $d(A)$ for all possible boards A.

For optimality in most games we want the divergence rate to be equal to 1. The reason for this is that if we have two boards A1 and A2 that derive from A, the minimum values for $|A - A1|$ and $|A - A2|$ will both be 1. Although in general the distance between boards isn't transitive (that is $|A - B| + |B - C|$ do not necessarily equal $|A + C|$), the distance is transitive if the changes $A - B$ and $B - C$ are disjoint. In non-mathematical terms two differences are disjoint if they affect non-overlapping areas of the board.

As A1 and A2 have a distance of 1 from A, A1 and A2 must be either disjoint or equal. By the definition of "derived board" we know they cannot be equal so they must be disjoint. Hence $|A1 - A2| = |A - A1| + |A - A2|$ and so $|A1 - A2|$ has minimal value of 2. As we have n derived boards we have $n!$ pairs of derived boards. Combining these two we know that the optimal value for the derived rate will occur when $\sum(|A1 - A2|)$ equals $2 * n!$ hence our definition of $d(A)$.

In non-mathematical terms, we can describe this property as saying for the average position in a game all possible moves should make as little a change to the board as possible.

It is clear that some games satisfy our optimal divergence rate, for example in Connect 4 every move only adds a single piece to the board and makes no other change whatsoever to the board. Considering other major games,

- Backgammon and Chess have a low to medium divergence rate. The difference between a position and the next possible position is almost entirely limited to a single piece move and capture (two moves and captures in backgammon) and so is tightly bounded.
- Go has a medium to high divergence rate. Although initially it seems to be like Backgammon and Chess in that each move only involves one piece, many games involve complex life and death situations where large numbers of pieces can be deprived of liberties and captured.
- Othello has a high divergence rate. One of the most notable properties of Othello that a bystander watching a game will observe is rapid changing of large sections of the board. It is possible for a single move to change as many as 20 pieces on the board.

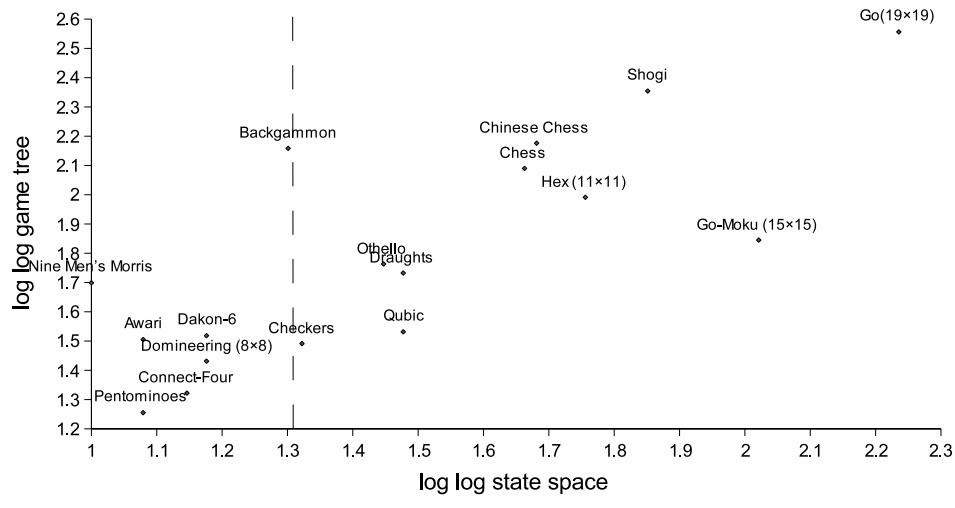


Figure 3: State-space and Game-Tree complexities of popular games

The theoretical reason for the importance of a low divergence rate was developed by Tesauro [12] while trying to explain his experimental results that his program was playing well despite the fact it was making judgment errors in evaluation that should have meant that it could be trivially beaten by any player.

The theory is that if you have a low divergence rate the error in a evaluation function becomes less important, because a low divergence rate will mean that because of the properties of a neural network as a function approximator evaluations of similar positions will have approximately the same absolute error. So when we rank the positions by evaluation score the ranks would be the same as if the function had no errors.

9.3 State space complexity

State space complexity is defined as the number of distinct states that can occur in a game. While in some games where every possible position on the board is valid it is possible to calculate this exactly for most games we only have an estimated value of state space complexity.

The state space complexity is important as it gives us the size of the domain of the game-theoretic function which our function approximator is trying to learn. The larger our complexity is the longer it will take us to train our function approximator and the more sophisticated our function approximator will have to be.

Although many previous papers have noted its importance in passing, the relationship between state space complexity and the learnability of the game hasn't been considered in depth. To illustrate the importance of state-space

complexity I produced a graph showing state-space complexity versus game-tree complexity, shown here in figure 3. The values for the complexities are primarily based upon [26] and my own calculations. The dashed line shows an approximation of the state-of-the-art for featureless TD-Learning system. All of the games to the left of the line are those for which it is thought possible to produce a strong TD-Learning based agent. Just to the right of the line are a number of games in which researchers have obtained contradictory results (for example Othello), from this graph it is easy to see that small variations in the learning system may bring these games into and out-of reach of TD-Learning.

Additional evidence for the importance of state-space complexity comes from the failure of TD-Learning when applied to Chess and Go. I would hypothesize that with these two games the technique does work, however due to their massively larger state-space complexities that these games require a far larger amount of training to reach a strong level of play. Nicol Schrudolph (who worked on applying the technique to Go [3]) has indicated via the computer-go mailing list that the Go agent they developed was becoming stronger albeit slowly, which seems to support my hypothesis.

From this evidence and based upon other previous empirical results, it seems likely that state-space complexity of a game is the single most important factor in whether it is possible for a TD-Learning based agent to learn to play a game well.

9.4 Forced exploration

One problem found in many areas of AI is the exploration problem. Should the agent always preform the best move available or should it take risks and "explore" by taking potentially suboptimal moves in the hope that they will actually lead to a better outcome (which can then be learnt from). A game that forces explorations is one where the inherent dynamics of the game force the player to explore, thus removing the choice from the agent. The most obvious example of this class of games are those which have an important random element such as Backgammon or Ludo, in which the random element changes the range of possible moves, so that moves that would have been otherwise considered suboptimal become the best move available and are thus evaluated in actual play.

It was claimed by Pollack [40] that the forced exploration of Backgammon was the reason Tesauro's TD-Gammon (and others based upon his work) succeeded while the technique appeared failed for other games. However since then there have been a number of successes in games which don't have forced exploration, in some cases by introducing artificial random factors into the game.

10 Optimal board representation

While board representation is undoubtedly important for our technique, as it is essentially virgin territory which deserves a much more thorough treatment

than I would be able to give it, I have mostly excluded it from my work.

However the formalization of board smoothness and divergence rate allows us to consider how useful alternate board representations can be and also allows us to consider how to represent our board well without having to engage in trial-and-error experimentation.

An almost perfectly smooth representation is theoretically possible for all games. To create such a representation one would need to create an ordered list of all possible boards sorted by the game-theoretic evaluation of the boards, then one could assign boards with similar evaluations similar representations.

Representation for a minimal divergence rate is also possible for all games, but unlike with smoothness is actually practical for a reasonable number of games. Essentially optimality can be achieved by ensuring that any position A' derivable from position A has the property that $|A - A'| = 1$. One way of doing this would be to create a complete game search tree for a game and for each position in the game tree firstly assign a unique number and secondly if there are N derivable positions create a N bit string and assign each derivable position a single bit in the string.

The representation of a derived position will now be the concatenation of the unique number and the N bit string with the appropriate bit inverted. It is clear from our definition that this will be minimal as $|A - A'| = 1$ for all A and A'. Unlike smooth representation, the usage of a representation that gives a minimal divergence rate has been done in practice [4].

In some cases (especially in those games where human representation has minimal divergence) it is possible to calculate an optimal representation system without having to do exhaustive enumeration of all board positions. For example as our natural human representation of Connect 4 is optimal, we can convert this into a computer representation by creating a 7 by 6 array of 2 bit values with 00 being an empty slot, 01 red and 10 yellow. It can be trivially shown that this satisfies the requirement that $|A - A'| = 1$.

The board representation will also affect the domain of our function, however given that game state complexity is very large for most games the board representation will only cause insignificant difference. An optimal binary representation in terms of game state complexity would be of size $log_2 game - complexity$, although in practice the representation is almost never this compact as the improvements that can be obtained from representations with more redundancy normally far outweigh that advantage of a compact representation.

One problem with using the natural human representation is that some games may be easier to learn than others with this form of encoding, so not all games will be treated on an equal basis. For example returning to Tesauro's TD-Gammon, Tesauro first chose to have one input unit per board position per side (uppps), the value of the unit indicating the number of pieces that side has on that position. Later on he switched to having 4 uppps (indicating 1 piece, 2 pieces, 3 pieces, or more than 3) which he found to work better. The argument he presented for this was that the fact that if there was only one piece on a board position this was important information in itself and by encoding the 1 piece code separately it was easier for the agent to be able to recognize

its importance.

Intuitively it seems separating the data in to as many units as possible would allow for better training given the nature of neural networks, however having more units leads to a longer training time.

11 Obtaining training data

To use the TD-learning method you need to have a collection of games which you can process with the TD(λ) algorithm. An important question is where you obtain this collection of games, as like with humans, the games the agent learns from have an impact on the style and ability of play the agent will adopt.

11.1 Database play

One way for our technique to learn is from a database of (normally human played) games, one of the major advantages of which is that of speed. Learning from a database game only takes a fraction of the time taken when actually playing. This is because playing involves a large number of evaluations, normally in the region of $gamelength/2 * (branchingfactor)^{(searchdepth)}$ when the AI is playing one side - double that if it is playing both sides (i.e. self-play).

Wiering and Patist [4] recently found in experimentation with Draughts that database play was stronger than random play but weaker than when playing against an expert. Using a database of 200,000 games they were able to produce a strong player in only 5 hours.

In recent years the amount of interest into using databases for training has increased, undoubtedly related to the growth in availability of large databases of human played games caused by the increasing popularity of Internet gaming servers which record games.

Other approaches involving databases include NeuroChess[39] by Sebastian Thrun which was trained by using a unusual combination of database play and self-learning. Games were started by copying moves from a database of grand-master games but after a set number of moves the agent stopped using the database and then continued the game using self-play. Despite Thrun stating that "this sampling mechanism has been found to be of major importance to learn a good evaluation function is reasonable amount of time", there does not seem to have been any other investigation of this approach.

11.2 Random Play

Random play methods fall into two types firstly a method in which we produce training data by recording games played between our agent and a random-move generator. While this takes approximately half the time of self-play the disadvantages seem in general to outweigh any benefits.

Among the disadvantage are,

- In many games there exist strategies which can beat random-play but fail against even the most novice of human players (i.e a very basic strategy could be sufficient to beat a random player). This essentially creates a "strategy trap" our agent can fall into.
- Random play is often far different from the way normal players (both computer and human) play the game, and valid techniques learned against random players may not serve a practical purpose. For example in the game Liar's Poker (a game in which bluffing plays an important role) as bluffing is meaningless when a player plays completely randomly, it is impossible to learn an effective strategy from observing random players.
- Dependent on the game in question, random play may not cover important basic knowledge of the game (captures, threats, forks, etc.)
- In some games random play may result in a game that lasts longer (by several orders of magnitude) than a human game.

The second type of random play is where we play a random-player against itself to produce a database of games which we use for training. Although only limited work has been undertaken in this area there is some evidence that it may be beneficial for initial training to be performed using random player data before moving onto another training method. Dayan, et al. [3] in producing a Go agent found that learning from data produced by a partially-random player (it was altered so as not to perform the "suicidal" move of filling in the eyes of a live group) the agent was able to learn "a surprising amount of basic Go knowledge" in only a few thousand games.

11.3 Fixed opponent

Fixed opponent play is where our agent plays one side and a single specific opponent (be it human or computer) plays the other side.

This has the problem that our agent may learn to beat the opponent but still fail to develop a good general evaluation function.

One of the reasons for this is because standard TD-learning technique were developed on the basis of an agent learning to improve their behaviour in a fixed environment, so when we introduce an opponent into the environment it is just treated as part of that environment. That is, the agent has no concept that there is a separation between environment and opponent, so the game and the opponent are treated as the same.

Because of this testing the agent by examining how many games are won against an opponent which the agent was trained against as was done in [4] isn't reliable in determining if the agent has been successful in learning a good general evaluation function.

11.4 Self-play

Self-play is when the agent plays against itself to generate games to train from. This technique has proved highly-popular among researchers. One of the major reasons for this is that it doesn't need anything apart from the learning system itself to produce the training data. This has several advantages including that the agent will not be biased by the strategy of other players (that is it will be able to learn strategies that work effectively rather than strategies that other players think work effectively). Research with fixed opponents has shown that learning appears to be most effective when the opponent is roughly the same strength as the agent; with self-play the agent obviously satisfies this criteria.

However the technique also has several disadvantages which contribute to making learning take place at a slower rate. The most important of these disadvantages is that takes a long time to play a large number of games as for each game we have to perform $gamelength * (branchingfactor)^{(searchdepth)}$ board evaluations. A related disadvantage is that when we start learning for the first time the agent has no knowledge and plays randomly and so it may take a substantial number of training games before the agent can correctly evaluate basic concepts in the game.

11.5 Comparison of techniques

There has only been a limited amount of research in this area, so I will concentrate in describing the comparison of database play to other techniques as this is the area for which we have the most solid evidence.

11.5.1 Database versus Random Play

In this case all evidence point towards Database being far superior to random play for learning. A large number of reasons have been suggested for this, the most common being that from database games the system will learn from games that are similar to how humans play and thus when it plays against humans it will be able to perform better.

11.5.2 Database versus Expert Play

In experimentation with tic-tac-toe Wiering and Patist [4] found that training against an expert resulted in better performance than when learning from databases. Their explanation for this was that the databases held a significant number of "bad" games which negatively impacted the agent's learning. Testing the agent's ability against the expert came out in favour of the agent that was trained by playing against the expert as one would expect.

11.5.3 Database versus self-play

One of the major disadvantages of learning from a database is that our agent does not know if the behaviour it is learning is good or bad. That is, it can

learn bad behaviour as well as good and without actually playing games it cannot differentiate between the two.

A story that has become anecdotal in the world of Backgammon illustrates this well. TD-Gammon played an opening move that was almost universally considered incorrect by experts. However TD-Gammon had repeatedly won games when it used this move, and this led to a heated debate in the Backgammon community as to the value of this move. Eventually a thorough analysis of the move supported TD-Gammon and since that time the move has become a standard opening play. This highlights a potentially major advantage of self-play, that is, self-play avoids learning what is commonly regarded as effective and rather learns what is effective in practice.

One of the problems with self-play is that with some games a common problem is that it takes a significant time for the network to be able to recognize fundamental concepts which are easy to learn from databases. The same applies to sophisticated tactics which may be non-intuitive (such as in Chess the boards leading up to a pawn becoming promoted to a queen may be poor, but once the pawn has become a queen the position is actually much stronger).

12 Possible improvements

Based upon previous work of other researchers and my own theoretical and empirical investigations, I produced a list of possible improvements to the standard TD-Learning system.

12.1 General improvements

12.1.1 The size of rewards

One factor which hasn't been examined in any depth is the impact of the reinforcement value assigned to a win or loss. Most applications of reinforcement learning have arbitrarily decided on one of two options, assigning a win or loss a fixed value or assigning a value based upon a standard method associate with the game. I would suggest that the second method may result in greater success than the first, as rather than assuming that all wins are equal it gives greater weight to "strong" wins.

To explain by example with the game of Go, let us assume two agents A and B, with A being trained with 1/0/-1 representing win/draw/loss, and B being trained with a weight equal to the difference between the number of intersections each side holds at the end of the game. Now in a game where the differences are ± 1 (i.e just one piece away from a draw) A would regard both as absolutely different assigning the weights as 1/-1, while B on the other hand would regard them as very similar and only regard the +1 win as slightly better than the -1 loss which is much closer to reality. At the other end of the scale a board position which led to a win of +50 is much more likely to be a stronger position than that which led to a win of +20.

12.1.2 How to raw encode

As covered earlier altering the board encoding can have a significant impact on learning.

12.1.3 What to learn

The selection of which games (and game positions) to learn from is an important factor as it defines what function is learnt. If the agent learns from all moves that the agent makes then the function will learn to predict the outcome of the play based upon its own playing style (i.e it will essentially become a self-fulfilling prophesier). If it instead learns from a wide variety of games then it will learn the game-theoretic version. When pseudo-random moves are introduced should it learn from these or not, or only for those where it has evaluated the move as the best move ?

12.1.4 Repetitive learning

The principle of repetitive learning can be described as the following. Rather than learning from 300,000 different game sessions, will learning 3000 different game session 100 times do just as well ? Intuition would surely argue against this however empirical evidence from Ragg, et al. [11] has shown that at least for Nine Men's Morris this is true. The theoretical argument behind this approach is that neural networks are slow to learn functions and repeating learning for the same position means that evaluation for positions similar to that position will be better at the cost of having explored less board positions overall.

12.1.5 Learning from inverted board

In the standard method of learning, the agent essentially learns two games, that played by the first player and that played by the second player.

However if we switched the colours of the two players (that is altered the representation so that the first player's pieces appears as if they belong to the second player and vice versa) for all of the boards in a game, we would then have a game which while technically the same as the original would appear different to the agent.

If this colour-inverted game was learnt, would the agent benefit from this or would this be the same as learning the same game again ?

12.1.6 Batch Learning

If instead of learning after every game, what would happen is instead the agent only had a learning session every 5,10,20 or more games. That is if we collected a large number of games, before we updated the weights using back-propagation. Would the overall performance decrease, and if so how much ? This is an important issue, as if learning from a batch of games doesn't significantly reduce learning then the game generation rate can be speeded up using parallelization.

The vast majority of time in training a network comes from the playing of games (as every position in a one-ply search has to be evaluated), so if we can have a large number of client machines playing the game and sending the results back to a server which learns from all these games and sends out the new neural network out to the clients the process of learning could be greatly speeded up.

12.2 Neural Network

12.2.1 Non-linear function

The most commonly used non-linear function in neural networks for games is the sigmoidal logarithmic function, however a wide variety of functions would work just as well.

Ito [28] takes an interesting approach by using Radial Basis Functions. These have another variable not found in most other functions, known as the *centre*. The centre allows the neural network to be better tuned when the domain has uneven complexity across its range. In the context of board games this can be viewed as allowing the network to be more accurate for board patterns that occur frequently.

12.2.2 Training algorithm

Conventionally TD(λ) has been used with standard back-propagation, however there have been many improvements to back-propagation that speed up learning, most notably RPROP which has been shown for some games to be significantly faster (by several orders of magnitude)[5]. In the same paper Ekker also investigated Residual-λ a training algorithm designed to reduce the instability that is inherent in the training of neural networks with TD (caused by a combination of states being non-independent and the problem of trying to learn a moving target).

12.3 Random initializations

Many neural network algorithms only find the local optimum, and while in many situations the local optimum may be sufficient it is worth considering if starting with different initial configurations will improve the agent. Theoretical and empirical evidence from TD-learning so far has indicated that given enough games TD-Learning will always find the global optimum, however in practice this may not be the case as an agent could get "stuck" in a local optimum from which the agent will need to play millions of games to leave.

So it may be worth researching if differently initialized networks learn or peak at significantly different speeds and if so, is this detectable at an early stage ? (I.e if we can predict after 100 games of self-play how strong the player will be after 10,000 games it may be worth testing hundreds of different initial randomized weightings to find the one that will give us the best performance in the long term.)

A related issue is how the random numbers should be bounded (some papers recommend a range of 0.1 others a far larger range).

12.4 High-branching game techniques

These are techniques which would be most useful in games which have a high branching factor (and have thus not normally used slow algorithms such as neural networks as evaluators).

These could be even more important in games where the branching factor is so high that even a one-ply search isn't possible (for example the mancala variant Omweso in which the opening player has $7.5 * 10^{11}$ possible moves).

12.4.1 Random sampling.

Rather than evaluate each possible position from a one-ply search, instead evaluate arandomly selected sample of those positions and pick the highest scored move (perhaps requiring that the move have a positive score, else picking another sample group).

The main question is, will the faster run time (which will obviously vary from game to game depending on the size of the sample and the average branching factor of the game) offset the weakness introduced by not playing the optimal move every time ?

12.4.2 Partial-decision inputing.

Another potential improvement I designed, which I am not aware of anyone suggesting before, is that if instead of training the function approximator by using just the board as an input, also add an input suggesting the region in which the best possible next move may be found.

So for instance in the game of Tic-Tac-Toe, the input could be (current-board, top-row), (current-board, middle-row) and (current-board, bottom-row). Which ever one of these scored the highest could then be used to reduce the number of future board positions that have to be evaluated to decide upon a move.

Although this will severely reduce the smoothness of the function, the benefits may outweigh the weaker learning.

12.5 Self-play improvements

12.5.1 Tactical Analysis to Positional Analysis

Although it is normal to discuss decisions based upon a game-tree search as "tactical" and those based upon the analysis of a board state as "positional", both are essentially the same. With a perfect positional analysis you arrive at exactly the same decision with a full game-tree search. So the obvious question is, is it possible to convert between them ?

I think this is not only possible, but also the solution to self-play problems. Previous studies have shown that the agent improves if it plays against an opponent that is slightly stronger than itself (as the latter is able to "force" certain situations). A new agent could be created which uses a game-tree search (using the function approximator for board evaluation) to oppose the agent being trained.

This would mean that the stronger tactical agent will be able to "teach" the learner, and as the learner improves the tactical agent will also improve, thus ensuring the learner always has an opponent slightly stronger than itself.

12.5.2 Player handling in Self-play

Another closely related issue to raw encoding is how you treat the two opposing players. Tesauro in TD-Gammon took the route of adding an additional unit to indicate the side that the agent was playing. This seems to me to be a non-optimal strategy, however I found only a few papers such as [46] that even gave the issue any consideration. Other alternative techniques could include multiplying the score by $(-1)^x$ where $x = 0$ or $x = 1$ depending on whether the agent was playing the first or second player (this approach would essentially assume a zero-sum game, so a board ranked for white as +0.5 should be ranked as -0.5 for black) or arranging the Neural Network so changing the value of the bias unit would cause the neural network to "change sides".

One of the primary advantages of using of these techniques is that for every game the same approximator would be trained for both a loosing and a winning game, which could increase the speed of learning two-fold.

12.5.3 Random selection of move to play.

In order to increase the amount of exploration and to prevent self-play problems, agents often don't select the best move, but pick another strong move. The methods for picking the move normally involve a random selection function biased towards the strong moves, however no real investigation has taken place into how the choice of the random selection function affects learning. Gibbs sampling is the most popular technique (used in [27] and [3] among others) but does it have any real advantages over other techniques ?

12.5.4 Reversed colour evaluation

In conventional board game systems (for zero-sum board games) it is common for one evaluation function to be used for both players, this is because the zero-sum property of the game means that what is the worst possible board for one player is the best possible board for the other player.

However an alternative method to evaluate both players using one evaluation function is to develop it to evaluate the board from player A's perspective and when it comes to evaluating a board for player B switch the colours of both sides and treat the board as if it was player A's turn.

12.5.5 Informed final board evaluation

Normally in self-play end-of-game positions are evaluated using a perfect function designed to tell if the game has been won, drawn or lost. The advantage of this is that the agent is "guided" to play better than if it was using its own evaluation function, another advantage is that games tend to be shorter if at any point in the game it is possible for the agent to make a winning move then it will do so.

However there are some disadvantages as well. Shorter games mean less exploration takes place and it may take longer for the agent to be able to evaluate end-of-game positions accurately for itself.

12.6 Different TD(λ) algorithms

12.6.1 TD-Leaf

TD-Leaf is a variant of the standard TD algorithm developed by Baxter, et al. [9] that allows it to be integrated with game-tree search. One of their primary reasons behind the development of this technique was a claim that TD-Gammon was able to perform so successfully because humans weren't capable of performing deep searches in Backgammon in the same way they were in Chess and Go.

The essential basis of this variant is that instead of defining the temporal difference as the difference between the evaluation of two sequential board position, it defines it as the difference between the evaluation of the principal variation (the best move discovered by a limited ply game-tree search) of the two sequential board position. This means that the function approximator instead of trying to predict the outcome of the game, instead tries to predict the outcome of the principle variation of the board position.

12.6.2 Choice of the value of λ

Although Tesauro's original work[12] found that the value of lambda appeared to have no significant impact on learning, later researchers[18] working on chess have found otherwise. So investigating how important the value of λ is to other games may be useful.

12.6.3 Choice of the value of α (learning rate)

Choosing a large α can make it hard for the network to converge as the changes made to the network will tend to be large. On the other hand too small an α can mean the network will take a very long time to reach an optimum.

One middle route that has been suggested is using a decaying rate, that is starting with a large α and slowly decreasing it as "experience" increases.

12.6.4 Temporal coherence algorithm

The temporal coherence algorithm developed by Beal [20] attempts to solve the problem α. It does this by assigning each weight in our neural network its own learning rate and provides an algorithm which automatically tunes the learning rates. The initial research conducted by Beal shows that it outperforms the use of a static learning rate and also performs better than other dynamic weight tuning methods.

12.6.5 TD-Directed

TD-Directed like TD-Leaf combines game-tree search with TD-Learning. Like TD-Leaf it was developed by Baxter [9]. It is the same as conventional TD(λ) learning, except that when playing it uses a minimax search on a multiple-ply depth tree (in the conventional approach only a one-ply search is conducted).

12.6.6 TD(μ)

TD(μ) is an algorithm designed by Beal [43] to compensate for the problems caused by a fixed opponent (see section 11.3). The problem can essentially be reduced to the following: Training against a weak opponent can result in bad habits. Beal's original paper contained some errors which were corrected in [5].

The basic idea behind TD(μ) is to separate the opponent from the environment. This is done by using our evaluation function to examine the moves made by the opponent (as most board games are zero-sum, a move made by the opponent is good if it minimizes our agent's potential score) and not learning from them if they are considered not-good. The distinction between not-good and bad is intentional here as exploratory moves may also be regarded as not-good without actually being bad.

Although Beal's original paper shows that this algorithm does work in that learning occurs, it does not provide comparative evidence against algorithms. This however is provided in Ekker et al. [5] where they compare TD(μ), TD-Leaf and TD-Directed in the game of 5x5 Go. There results found that while TD-Leaf was roughly equivalent to TD-Directed they were both outperformed by TD(μ).

12.6.7 Decay parameter

Ragg et al. [11] have suggested that for deterministic gamesa decay rate parameter (generally know as a discount-rate parameter) should be used to make the network better able to handle move selection based upon a long term goal. Their research has shown it to be successful with Nine Men's Morris, a result confirmed by Thrun [39] who argued from empirical evidence that without a discount-rate no learning occurred.

Use of a decay rate parameter means changing the temporal difference so that $d_t = \gamma Y_{t+1} - Y_t$ (γ is the decay rate parameter). This changes our learning algorithm to,

$$\Delta W_t = \alpha(\gamma Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

13 Design of system

13.1 Choice of games

I decided to concentrate on two games, although as my system is game-independent it is fairly easy for additional games to be added to the system. The two games I have chosen are Tic-tac-toe and Connect 4. One of the important factors in choosing these two games was that according to the properties I listed earlier they were both likely to be amenable to TD-Learning and have human representations which have near optimal divergence rates. This should minimize the negative impact of board representation on the learning process, and thus allow us to measure the performance of the learning process more accurately.

13.1.1 Tic-tac-toe

Tic-Tac-Toe (also known as Noughts and Crosses) is a classical game (dating from the second millennium BCE) which involves a 3x3 board on which alternate players place their pieces in an attempt to get three pieces in a line. It has a state-space complexity of around 5000 and can be trivially solved using a min-max search of depth nine.

13.1.2 Connect-4

A relatively recent game based on similar principles to tic-tac-toe but using a 6x7 board with gravity (each player can only place their piece in the lowest available slot in any column). The aim is to get four in a row. Despite the game's relatively simplicity it was only solved in 1988 by Victor Allis [38] using a branch-and-bound technique.

Although there has been no previous work on this game with neural networks, [33] mentions that they had success in the game when using TD(λ) with an alternative function approximator.

13.2 Design of Benchmarking system

Traditionally for benchmarking game evaluation functions there are two basic methodologies. The first is to compare the evaluation of boards with expert evaluation (normally from human experts or from databases for games which have been fully solved) and the second is to use the evaluation function in actual gameplay. The first approach is of limited practicality, in the human case due to the fact that only a relatively small number of board positions can be evaluated in this way and because it will only work while human evaluation is superior to

that of the agent, and in the "solved games" approach because only a limited number of games have been solved.

This means we have little choice but to follow the second methodology. Almost universally this has been put into practice by playing the agent against a fixed computer opponent. There are a few exceptions mostly being those evaluated by playing on online servers against a variety of humans where they can be ranked and scored using conventional raking systems such as ELO in Chess.

While playing the agent against a computer opponent is a effective measure of improvement of the agent it has several problems,

1. Rather than developing a good generalized evaluation function the agent may have developed an evaluation function that that results in a strategy which is only effective against a particular opponent. This is especially a problem if the agent was trained based upon games played by the opponent such as in [31].

2. Often little information is given about the opponent meaning that the results are hard to reproduce accurately. For example [3] states "A small sample network that learnt to beat Many Faces (*at a low playing level*) in 9x9 Go", without specific information about the level that the opponent Many Faces of Go was set at it would be hard for someone developing an alternative Go agent to discover if their agent was superior to the one described in the paper.

3. The limit of learning ability may be higher than the limit of measurability when playing a non-perfect opponent. That is once the agent has become sufficiently advanced it will no longer loose to the opponent , and from that point onwards any further learning will be unmeasurable.

4. For every game you need a separate opponent, and there is no way to compare performance of agents between games without the opponent being a significant factor.

5. If an evaluation function allows us to play a perfect strategy it would not lose a single game but the evaluation function could still be weak. For example in the game of three-dimensional Tic-Tac-Toe there is a trivial strategy which can be used by the first player so as to always win, but knowing this strategy is of no use if asked to evaluate a random game position that does not occur in the normal course of that strategy.

As the system is intended to be used for the comparison of TD-Learning systems across a number of games, the need to have a separate opponent for each game is particularly a hindrance in the development of a standardized testing system.

The obvious solution is the development of an algorithm that can play all games and play them all roughly to the same standard. Due to the context of the benchmarking system as part of the TD learning system I considered

creating the benchmarking system using the game functions that are available to the agent.

This led me to two possibilities the first being to use a neural network evaluation function in our opponent. However this has two problems, firstly if the network is the same as the one used by the agent we will have the same problem as occur when the opponent is used to produce training data and secondly the specific random initialization of the network is not reproducible by a third party.

The second possibility was to use our end-of-game state evaluation function as opponent, that is the function which given a game state will tell us if the game is won or loss. However this will only give us the evaluation for positions which are at the end of the game or are close enough to the end for us to use minimax search of low ply depth to reach the end. This problem can be solved however by having the evaluation of the rest of the positions return a random value. So the opponent plays randomly except near end states. Since I started my project this method of benchmarking has been independently developed by Patist and Wiering [4] although curiously they only applied it to tic-tac-toe and didn't consider generalizing for other games, nor did they consider the problems with this system.

While this system does indeed avoid the majority of the problems list above, it has one serious flaw, that the player essentially plays randomly (except near end positions) and so a fairly simplistic evaluation function could beat it. To solve this flaw I developed a technique called random-decay in which a percentage of our agent's moves are replaced by random moves. The percentage of moves that are replaced I have called the benchmark decay-rate.

Although not common in current practice, in the 13th century this was a common system of handicapping in Arab countries especially in the games of Chess and Nard (a game of the Backgammon family).

This technique works as a good evaluation function should still win more games than a random evaluation function even if 90 percent of its moves are replaced with random moves. As the percentage of random moves increases, the percentage of games won by our agent converges to fifty percent (i.e. the number of games won by playing two random move evaluators against each other).

The better an agent is the more moves we need to replace with random moves in order to be able to successfully measure its strength. However increasing the number of moves replaced will make agents of sufficiently different strength incomparable. After experimenting with different approaches the most effective solution I was able to find was to test every agent with several decay rates and to plot the data.

In practice I implemented this by testing with decay rates at 10 percent intervals (i.e. 0,10,20,...90,100 percent). The graph of this data represents the strength of an agent. For every game there exists a "maximum" graph which we would get if we used this procedure with a perfect evaluation function. As our agent improved we would expect our graphs to converge to the maximum graph. However without having a perfect evaluation function there is no known way to establish where the maximum graph would be located.

A side-effect of this representation is that it is possible to derive an intuitive

understanding of the differences between two agents by comparing their graphs. For example, if we had two graphs A and B, with graph A starting higher than B but crossing at a low decay-rate this would mean that B better represented a perfect evaluation function but A represented an evaluation function that resulted in a better strategy. The reason for this is that more perfect evaluation functions are more robust when given forced random moves when compared against worst evaluation functions (even if they give better strategies). This gives us a methods of differentiating between strong (i.e. more perfect) and weak evaluation functions without being subject to the risk that the weaker evaluation function is giving us a better strategy, hence it could be used for diagnosing some of the problems discussed in the training method.

This approach manages to solve all of the problems listed earlier.

13.3 Board encoding

As discussed in the earlier section on board representation, the representation used can have a major impact on the ability of our system to learn, so to avoid this I developed a standardized board representation form. Although in my experiments I will be using the representation described here for the boards, the learning system is mostly independent from the board representation.

The representation will be space-based (as opposed to piece-based), that is we will describe the board in terms of physical locations on the board, assigning each discrete location an integer which changes depending on the piece that is located on that location. As such this representation falls into the class of "human representations" discussed earlier.

Although this is the representation I will be using in my experiments, I have designed the system so all the learning systems can observe is a series of arrays and an evaluation of the final game state as a win, draw or loss. This means that essentially any form of encoding can be used with our learning system.

13.4 Separation of Game and learning architecture

From the start of my work one of the most important considerations was to develop a system in which the game and learning architecture are separated. That is that there should be a strict separation of game and learning function so that it would be possible to change the game the agent plays without any need to modify any of the learning functions or change the learning function without having to modify the game. The primary purpose of this is to allow different learning systems to used on a variety of games with a minimum of difficulty and also ensuring that no changes were made for reasons of compatibility that unintentionally affect gameplay.

This led me to analyse how different TD-learning systems worked and to separate all of their functionality into three separate categories, game-related, learning-related and learning system to game system interactions. Of those in the last category I further broke them down into game related and learning

components establishing an artificial interface via which these functions would communicate with each other.

Game related functions

- Generation of all possible moves at one-ply depth.
- Win/loose/draw detector.
- Board manipulation functionality.

Learning related functions

- Neural network initialization.
- Learning system.
- Evaluation function.
- Benchmarking system.

Interface

- Combination of evaluation function and depth search.
- Transfer of board states.
- Information about the board game the learning agent needs to know (such as board size).

These last three items were achieved as follows. The one-ply depth search is conducted as a game function and for each board generated the game function calls the evaluation function and returns the board which has the best evaluation. The transfer of board states was simply done by passing an array encoded in the standard method given earlier. A record of a game was stored as an array of these arrays. The information about the board game is stored in a game.h, a header file which is accessible to the learning system.

13.5 Design of the learning system

As I intend to make the code publicly available after the completion of my project, many of my design decisions were made so as to keep the learning system as simple as possible so it is easier for other interested parties to learn from my code. When I started my project I was unable to find a publicly available simple TD-learning based game system. Those that were publicly available tended to be tied in to complex AI systems such as in GNU Backgammon.

I decided to make the neural network a fully connected network (that is all input nodes are connected to all hidden nodes and all hidden nodes are connected to the output node) as that is the standard practice among TD-Learning game agents. Due to the need to implement a specific learning system

I decided to develop the neural network code myself rather then use a separate library. Another advantage of taking this approach is that the code is totally independent of any third party libraries so it is easier for someone to follow through the code and fully understand what is happening.

The learning system was based upon a pseudo-code version of TD(λ) developed Richard Sutton and Allen Bonde Jr. [36] (although my implementation used a different reward structure which was more appropriate for board games). I decided to stay with the standard back-propagation algorithm that was used in the pseudocode, as although variants of back-propagation (such as using momentum) are faster, they may also introduce instability into the network and unnecessary complexity into the code.

Rather than use a fixed α (learning rate) for all the weights in my network, I decided to use a smaller value for the weights in the first layer and a larger value for those in the second layer. It is commonly accepted that this results in more effective learning. The actual values used for α (learning rate of first layer) and β (learning rate of second layer) were found by a trial-and-error approach based upon values used by previous researchers.

The sizes of network used (excluding bias nodes) were 42-80-1 for Connect 4 and 9-80-1 for Tic-tac-toe. These were chosen arbitrarily. It is important to note that network size is unimportant. The primary reason for this is that we are comparing different training methodologies and are not developing agents to compete against other players. As the different training methodologies will all use the same size networks they can be compared on a fair basis.

13.6 Implementation language

All of the core system is developed in C with the code being compiled with GNU C. The compilation process is automated via a shell script and gawk scripts were used to post-process the output data from the benchmark program.

The system is comprised of a number of programs (anal, learn, selfplay, initnet, etc.) with the separate components of the system having their own programs. Although this was originally intended so as to allow for the user to be able to change part of the learning system by just switching a single binary, it also has the advantage that these separate component do not have to be programmed in the same language.

The program for which it is most easy to change the language of development is anal, which is the analysis program which acts as the glue between the other program. It could easily be rewritten in any language including shell scripts without causing any serious detrimental effect on the efficiency of the system.

13.7 Implementation testing methodology

Due to the complexity of the algorithms being used it was important to ensure the implementation was error free. On a number of previous occasions failure of TD-learning has been attributed to implementation errors. Unlike with most other programs where an error in output can be tracked to its source fairly

easily, this is not possible in neural network based TD-Learning systems, as any error will only be noticeable when the system fails to learn to play a game. If a system fails to learn how to play a game, it means the error could be virtually anywhere in the code or could even be down to a single parameter being set incorrectly. Hence it was very important to thoroughly test the system.

Each function used was tested independently to ensure correct performance, wherever possible I hand generated sample input and output data and compared it with that of the function. For the neural network learning function I also tested it by having it learn a static function and verifying that the neural network converged towards our function.

I also implemented the core TD-lambda code in Maple and used it to verify the results of my C code, although the extent of this testing was limited by the slow speed of Maple.

All of the C code was also examined using the valgrind system to identify any memory leaks which were then corrected.

The final test was to see if it was capable of learning the games. The final code successfully passed all of these tests.

14 Variant testing

Once the system was developed I tested a number of the variants described earlier. The ones I chose to test were based on a number of factors including how significant they were likely to be and the amount of research previously done on them. I concentrated especially on those improvement which show large potential but have not been the subject of much investigation.

Unless otherwise noted the benchmarking system was run with a decay rate of zero percent (that is no decay rate - no random moves were introduced), in order to allow the data to be displayed in two-dimensions (number of training games versus percentage of games won). When the benchmarking system was run it played one thousand games and recorded the percentage of games that were won (that is not lost or drawn). In most of the tests the modified agent is compared with a control, which is the base implementation of the system. Both the control and the modified agent in every test start with the same set of initialization weights.

All settings others than those specifically stated as being changed are constant throughout all of the experiments.

All of the tests were conducted using a one-ply depth search and self-play for game generation.

14.1 Random initialization

Due to the potential importance of random initialization on learning this was my first item I tested. As the impact of random initialization is likely to be the same for all games I chose to test it using Tic-Tac-Toe. I randomly initialize

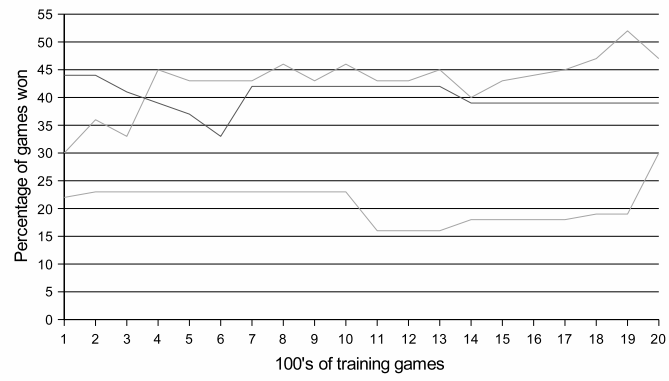


Figure 4: Agents with different random initializations

three neural networks with values from -1 to 1 and trained them with 2000 games, testing performance every 100 games.

Although 2000 seems a small amount of games, the first few thousand games are some of the most significant training games. This is because it is in these games the agent goes from "random" knowledge to having an established knowledge base containing the basic concepts of the game. So the most significant changes in our agents ability would occur at this time and after this time learning occurs at a much slower pace, so it is important to get a "good start".

The results are displayed in figure 4. From this graph it is clearly visible that random initialization has a significant impact on learning. Because of this I will need to take the effect of random initialization into account when conducting the other tests, I will do this by running each test a number of times with a number of different randomly initialized weights and use the average of the performances for the results.

However this means for practical reason I will only be able to run most of my tests with Tic-Tac-Toe, as to perform a single test for Connect 4 takes 10 hours, so doing it repeatedly for different weights for all of the improvements is not practical. All of the tests are conducted at least once with both games, and only if from these tests or from theoretical evidence it appears as if the results from the two games will be different will the test be repeated for Connect 4 with different weight initializations.

14.2 Random initialization range

Having established the importance of the random initialization, the next step is naturally to consider the range of our initialized number. On the basis of previous work I decided to concentrate on two ranges, the first -0.05 to 0.05 (range 0.1) and the other between 0.5 and -0.5 (range 1). For each range a set of 20 random weights within the range were generated and trained using the standard training method for 2000 games. For each set of weights after every

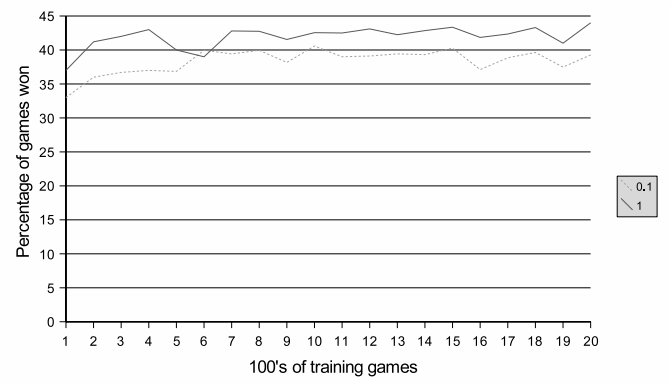


Figure 5: Comparing ranges of random initializations

100 training games they were benchmarked.

The results with tic-tac-toe are graphed in figure 5. The results have been averaged across all the training weights in the range. Although it appears from the graph that the range of 1 was slightly superior, this graph is misleading as it fails to show the variance of the results. Taking variance into account there appears to be little difference between using the two ranges. One notable feature of the graph is a distinct "flatness" which seems to indicate little learning took place, again this is misleading, in this case being due to some randomly initialized agents which didn't learn obscuring the results of those agents which did learn. Further experimentation using the same ranges led me to discover that using 5 sets of randomly chosen weights gave optimal balance between showing learning while still taking into account variance in performance due to initialization weights, hence for all later experiments 5 sets of random weights were used.

Although 5 sets of weights seem to be enough to approximately show the performance of a technique, this number is still low, and it may be possible that random variation obscure results, especially when the difference between two agents is only small. However using small number of sets seems to be standard practice in this area of research, perhaps due to the long run time of the training process.

14.3 Decay parameter

If we set $\gamma = 1$ in the decay parameter formula it becomes the same as our original formula. Hence γ was included in my standard implementation and set to 1. So for this test I modified gamma to 0.5 and 0.7 to see the impact, as control an agent with $\gamma = 1$ is also displayed. As previous research has shown the impact of gamma varies dependent upon the game being played I have conducted full tests using both Connect 4 and Tic-Tac-Toe.

The system was benchmarked every 10 games for the first 100 games, and

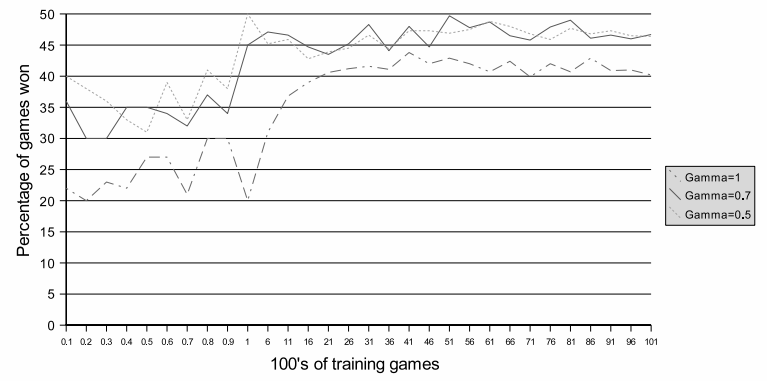


Figure 6: Agents with different decay parameters in Tic-Tac-Toe

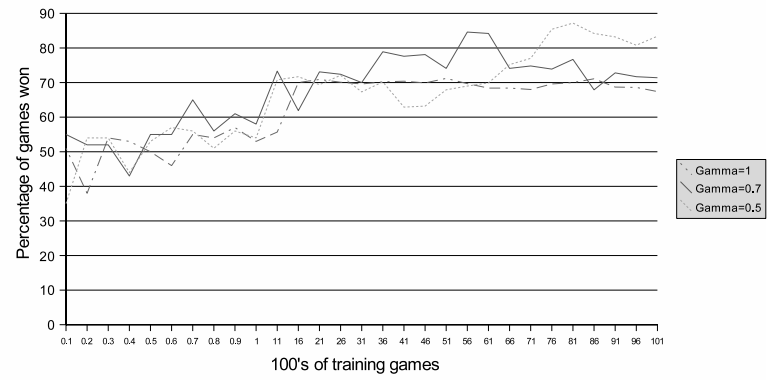


Figure 7: Agents with different decay parameters in Connect 4

every 500 games after that. In total approximately 10000 training games were played.

The results from Tic-Tac-Toe are displayed in figure 6 and those for Connect 4 in figure 7. From Tic-Tac-Toe it seems that having a decay parameter (i.e. $\gamma < 1$) does effect learning, although there seems to be little difference between having $\gamma = 0.7$ and $\gamma = 0.5$. In Connect 4 while having a decay parameter does seem to produce slightly better learning, the effect is almost negligible, and could be due to random noise.

However both of these results disagree with the earlier work of Thrun [39] who claimed on the basis of empirical evidence from Chess that without a decay parameter no learning would occur at all. An area of potential investigation that arises from this is to try and discover if there exist some games in which a decay parameter is necessary and to attempt to identify the common property shared by these games that makes it necessary.

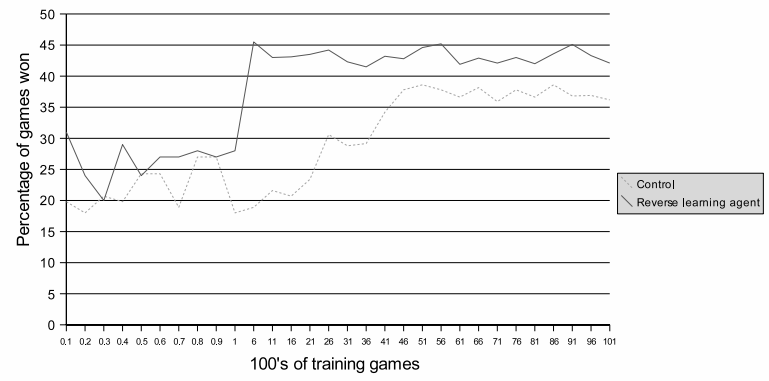


Figure 8: Reversed board learning agent

14.4 Learning from inverted board

For this test the learn.c was altered so that if a command line parameter was passed to the program it would invert the board before learning. This test was only conducted fully with Tic-Tac-Toe. Preliminary results from Connect 4 indicate the results are likely to be similar to that for Tic-Tac-Toe.

The system was benchmarked every 10 games for the first 100 games, and every 500 games after that. In total approximately 10000 training games were played.

The results are graphed in figure 8. The learning process shows an unusual growth pattern when this method is used, learning in the first 1000 games is significantly greater than in the control agent, but after these early games the rate of learning returns to the same level as the control. The most likely theoretical explanation behind this is that the reversed boards help the agent to learn the basic concepts underlying the game. If this technique works similarly for others games then it could be used to speed up the initial training of agents.

14.5 Random selection of move to play.

Rather than allowing the move to be chosen completely at random or using a sophisticated technique like Gibbs sampling, I decided to investigate a middle ground approach. I set up the self-play system so that ninety percent of the moves (found by a one-ply search) would be evaluated using our neural network, the other ten percent of the moves were assigned random evaluation random score.

The system was benchmarked every 10 games for the first 100 games, and every 500 games after that. In total approximately 10000 training games were played.

The results displayed in figure 9. Although this technique seems to start significantly better than the control, virtually no learning occurs over the 10,000

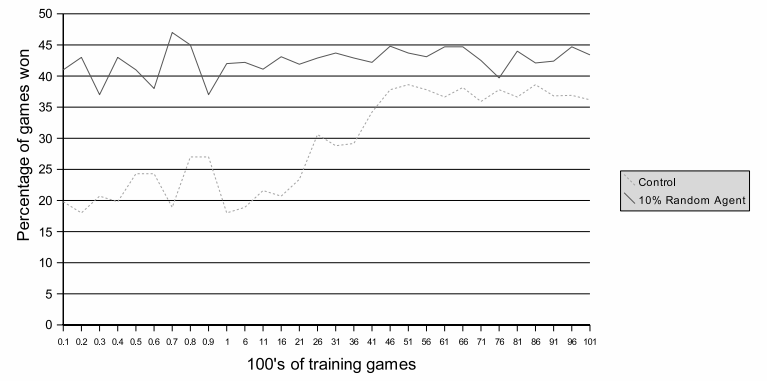


Figure 9: Random Exploratory agent

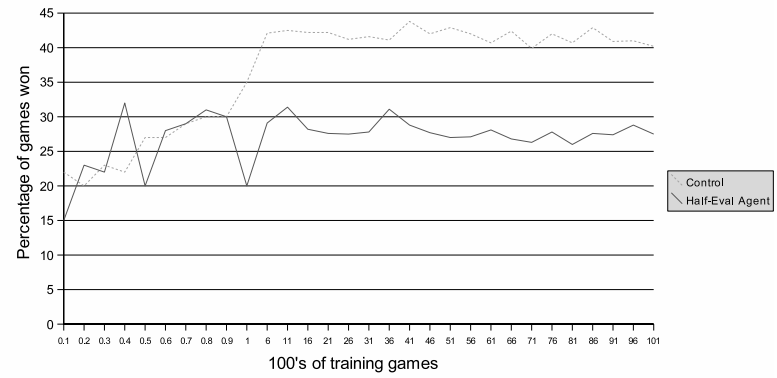


Figure 10: Half-board evaluation agent

game period, so I conclude that this technique fails to work in practice and that more sophisticated random selection techniques are required.

14.6 Random sampling.

To test the idea of random sampling I altered the learning system so that only half of the boards were evaluated. I only performed this test on Tic-Tac-Toe, the reason for this is that in Tic-Tac-Toe even if a player is limited to playing half of the possible moves it is still perfectly possible to win or at least force a draw in the vast majority of cases.

The results are displayed in figure 10, from this graph it is clearly visible that this method is far weaker than the control and so this technique is unlikely to be of practical use.

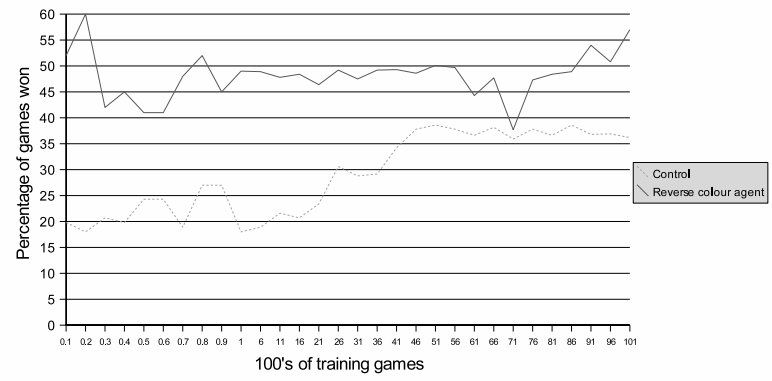


Figure 11: Reversed colour evaluation agent

14.7 Reversed colour evaluation

This test was essentially a straight implementation of what was described in the variants section 12.5.4.

The system was benchmarked every 10 games for the first 100 games, and every 500 games after that. In total approximately 10000 training games were played.

The results are displayed in figure 11. The results show that this form of evaluation is significantly better than the control. Although only a limited amount of learning appears to be taking place from the graph, investigating the trained agents indicated that learning is occurring although at a slow rate. Whether this is because of the reversed colour evaluation or because at its relatively high level of performance (generally as performance ability increases the rate of learning decreases) is unknown and warrants further investigation.

I also conducted some preliminary tests combing this method with the "Learning from inverted board" method, the results indicated that the "Learning from inverted board" method did not cause any improvement when used with this evaluation technique.

14.8 Batch learning

To test the batch learning idea the system was altered so that rather than learning after each game, five games were played between each learning phase.

The system was benchmarked every 10 games for the first 100 games, and every 500 games after that. In total approximately 10000 training games were played.

The results are displayed in figure 12, the results show us that this method of learning is as successful as the control method. Although this is promising, the result might not extend to a large number of games and thus may limit how much we can parallelize the game generation process. However it does show

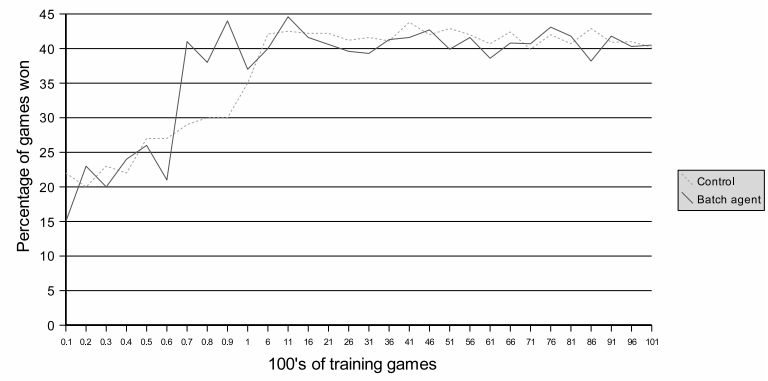


Figure 12: Batch learning agent

that in principle this idea works.

14.9 Repetitive learning

To test the repetitive learning idea the system was altered so that rather than learning once after each game, learning from a game would take place five times. The results from Nine Men's Morris [11] indicate that we should expect the learning to occur five times faster.

The system was benchmarked every 10 games for the first 100 games, and every 500 games after that. In total approximately 10000 training games were played.

The results are graphed in figure 13. The results from Tic-Tac-Toe are disappointing as they fail to show any significant difference between our repetitive learner and the control. As this technique is known to work with Nine Men's Morris, the technique might only work on specific games, so I decided to also conduct full tests with Connect 4. These results are shown in figure 14, however the results for Connect 4 are similar to the results for Tic-Tac-Toe; the improvement in learning we expected to see is not there. From this evidence I suspect that the technique may have worked for Ragg et al. due to some specific property of Nine Men's Morris rather than due to a property of TD-Learning, however experiments with other games need to be done to better understand if this is the case.

14.10 Informed final board evaluation

As the standard learning system I implemented used a perfect final position evaluator this became the control in this test, and a modified version where final board positions were evaluated by the agent became our experimental agent.

The system was benchmarked every 10 games for the first 100 games, and every 500 games after that. In total approximately 10000 training games were

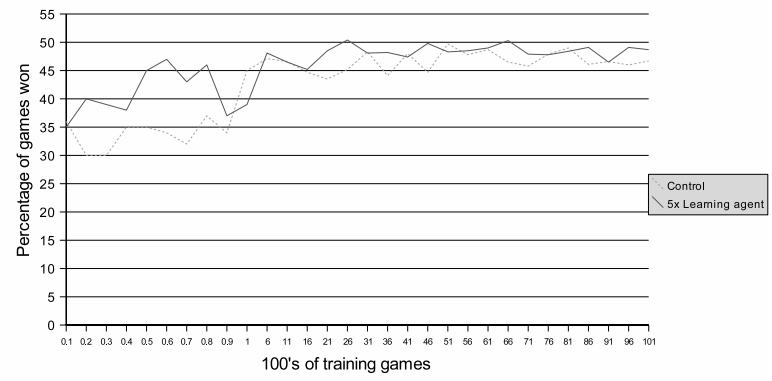


Figure 13: Repetitive learning in Tic-Tac-Toe

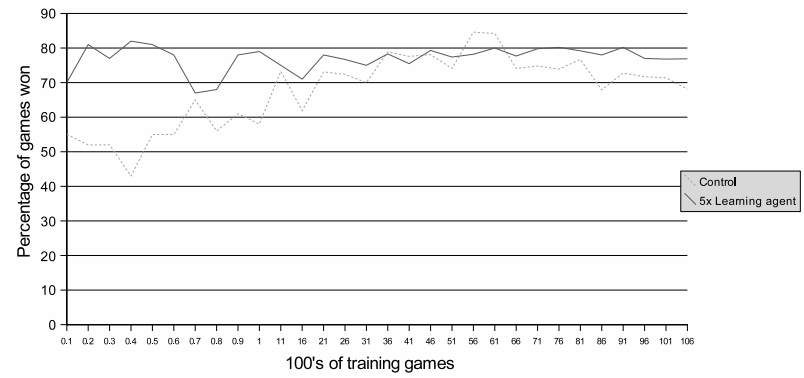


Figure 14: Repetitive learning in Connect 4

played.

The results are displayed in figure 15. Of the experiments this is perhaps the one with the most conclusive evidence. The control clearly outperforms the modified agent in both terms of rate of learning and in terms of strength after 10,000 learning games.

15 Conclusion of tests

From the tests I conducted I hope to have contributed to a better understanding of TD-Learning techniques applied to games.

I have shown Informed final board evaluation, Reversed colour evaluation, Decay parameters and Learning from inverted boards all improve our rate of learning and should be considered by anyone developing TD-Learning based agents. My tests into Batch learning show that TD-Learning has the potential

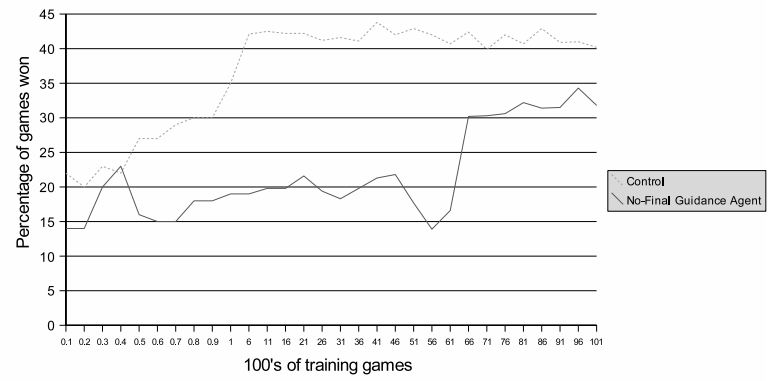


Figure 15: Half-board evaluation agent

to be parallelized and thus reduce learning time.

Also highlighted is the importance of random initialization and how different initialization can have a major impact on the agent's strength. Perhaps equally important is showing that the range of the initialization is only of minimal importance.

The negative results are also important as they show us what fails and perhaps thus better allow us to understand why some techniques work and others don't. The Random selection of move and Random sampling of moves both fail to work effectively, indicating that research should move towards investigating biased randomization techniques.

Our negative results for Repetitive learning, while disappointing, tell us that this technique may not have the great potential it once showed.

With regards to Connect 4, the results from the various experiments clearly indicate that a TD(λ) and neural network approach can work successfully to produce a good evaluation function for the game.

16 Future work

There are a number of possible extensions to this work, testing the other variants described in my work but not tested and running all of the tests over a larger number of games being the most obvious extension. To allow for this I intend to make all of my code publicly available under an open source license, which I hope will encourage others to carry on the work I have started. I also intend to continue developing this work myself after completion of my project.

Adding extra games to the system would also be useful, as once the game code has been developed all of the variants I have developed can be easily retested with the new game. I hope that this functionality will enable future TD-Learning developers to quickly discover which improvements are likely to be useful to the games that they are working on.

A Appendix A: Guide to code

A.1 initnet.c

Creates a neural network of size specified in game.h and initializes it with uniformly distributed random values. The range of the random variables can be controlled as a command line parameter, else the program will prompt you for a size. The range will be centered around 0, so if the range is set to 1 then the network values will be set between -0.5 and 0.5.

It has the following functions:

InitNetwork Initializes the neural network with random values.

outputweights Outputs the weights to a file.

A.2 anal.c

A glue program that calls the other programs, this file will be different for every test I conduct. In all of the tests the core code will be approximately the following:

```
for(i=0;i<100;i++)
  {
    for(j=0;j<500;j++)
      {
        system("./selfplay");
        system("./learn");
      }
   system("./rbench 1000 2");
  }
```

These loops are responsible for running 50000 training games (selfplay generates the game and learn learns from the game just played) and every 500 games it will run the benchmarking system. This is one of the key files in the modular design of our system to allow different learning systems to be tested. For example if we implemented an alternative learning systems we would just need to replace the *learn* binary, if we wanted to use it along with the normal learning systems we would just add a call to it after the *learn* command.

A.3 ttt-compile and c4-compile

This is a shell script which compiles the core parts of our learning system. Quoted in full:

```
gcc -o learn -lm ttt.c shared.c learn.c -Wall
gcc -o rbench -lm ttt.c shared.c rbench.c -Wall
gcc -o initnet -lm ttt.c shared.c initnet.c -Wall
gcc -o selfplay -lm ttt.c shared.c selfplay.c -Wall
```

When changing the game being played the filename ttt.c needs to be replaced by the filename of the C file containing the code for the new game.

A.4 selfplay.c

This contains the code which makes the agent play against itself and record the game. It uses the findbestmove() function to identify the best move and movep() to make the move. It is also responsible for updating data required by these two functions such as the move number and which player's turn it currently is.

It records the game as a multidimensional array which contains every board position that occurs in the game in order.

A.5 rplay.c

A modified version of selfplay which is designed to play pseudo-randomly.

A.6 learn.c

The learning part of our code.

It has the following functions:

initlearn Initializes the arrays used for learning.

updatelig Updates the eligibility traces.

tdlearn The code that actually updates each weight by calculating the change required using the formula learning-rate*error*eligibility trace.

loadgame Loads the game record produced by selfplay or rplay.

reward Returns a reward for the final game position by calling isend() to find out which player has won and then assigning an award appropriately.

A.7 shared.c

This file contains a number of functions that are often used by the other files.

randomnumber returns a randomnumber between 0 and 1 which had six significant figures.

score If the board is a non-final position the function uses the neural network to evaluate the board. If it is a final position it returns the true reward it obtains from the isend() function.

inputweights Loads the neural network weights from a file.

A.8 rbench.c

Implements the benchmarking system described in section 13.2. As a command line parameter this program takes a decay rate value (otherwise it defaults to 0.1). If the decay rate is greater than 1 then no decay will occur. An example of where this could be used would be if the pseudo-random player was strong enough not to be comprehensively beaten by the agent, as in this case it would not be necessary to handicap our agent. Although we get less information without the decay rate we do obtain a significant speed increase of the benchmarking system (10-fold compared to the default decay rate of 10 percent).

The following data is outputted from the benchmarking system:

- Decay.
- Games won as first player.
- Games won as second player.
- Games lost as first player.
- Games lost as second player.
- Games drawn.

A.9 ttt.c and c4.c

The game-specific code for Tic-tac-toe and Connect 4. Although implemented differently they are both required to have the following functions which are used by the learning system.

isend Checks if the game has been won and returns 1 or -1 to indicate win or loss.

isdraw Checks if the game has been drawn.

movep Takes as input a board state, a player identifier, and a move number and uses this information to change the board to make the given move for the given player.

pboard Prints the board to screen, mainly used for debugging purposes. Not required.

copyboard Copies one board state array to another. This is part of the game-specific code so that it is possible to optimize the copying for a particular game.

randomscore Assigns a random score to a non-final board position. If it is a final position then returns the value of isend.

findrandommove Calculates all the possible boards that could be produced by the next move and assigns each board a score using randomscore() and returns the best score. Note that this does not select a move randomly, as if it is possible to make a winning move it will do so and it will never make a "suicidal" move, that is a move that will result in it losing immediately.

colourreverse Switches the side of the two players. This function only makes sense in a zero-sum game. While it isn't strictly necessary for our learning system, it is used by several of the modifications I investigated.

findbestmove Calculates all the possible boards that could be produced by the next move and assigns each board a score using score() and returns the best score.

The implementation of Tic-tac-toe has only these functions as it is a fairly simplistic game and doesn't need other subfunctions in order to be able to produce the information required for the above functions. However the implementation of Connect 4 has several extra functions. The primary reason for this is that our learning system requires the board be stored in a one-dimensional array, and while it is possible to detect situations such as game wins in Tic-tac-toe when it is represented as a one-dimensional array without much difficulty, it is much more difficult to do so with more complex games. Hence the need for extra functions that allow us to manipulate the board in a data-structure of our choice and to convert to and from that data structure to our one-dimensional array.

When implementing a new game for the system apart from this file and its header and compilation files the only other file that needs to be modified is game.h which contains information about the game that the learning system needs such as the size of the neural network (which depends on the board size).

B Appendix B: Prior research by game

B.1 Go

Ekker et al.[5] investigated the application of TD-Directed(λ), TD-Leaf(λ) and TD(μ) with RPROP and Residual-λ to 5x5 Go. Training and testing is performed using Wally and GNU Go. A 7x7 Go board was used in [27] where a variant implementation of TD(λ) was used, however as they didn't compare their system to any other playing methodology (either directly or indirectly) it is difficult to assess how useful their approach was.

Other approaches have involved breaking the game of Go into simpler smaller problems. Examples of this approach include [23] who use TD-learning to develop a network which can evaluate piece structures (known as shapes in Go) and [3] in which a sophisticated network architecture is developed which exploits the symmetry of structures in Go.

B.2 Chess

The earliest applications of TD to chess were to specific situations such as endgames [44]. The first successful TD based chess agent was NeuroChess [39] by Sebastian Thrun which after 120,000 games was able to win 13 percent of games against a conventional chess agent GNU Chess.

Beal [43] uses chess as a testbed for using his TD(μ) algorithm to learn from a random opponent.

Mannen and Wiering [6] investigated database learning using TD(λ). They used this technique to obtain material weights for Chess and also to produce a general Chess playing agent.

However perhaps the most significant results have been those from Baxter, Tridgell and Weaver [7] who use TD-Leaf and hand-developed evaluation features to produce a very strong chess player.

B.3 Draughts and Checkers

Little work has been performed on Draughts, but [16] has shown that it is possible to develop a successful TD Draughts player using a combination of hand-made features and self-play.

For the game of Checkers, Schaeffer et al. [35] investigated using the TD-Leaf algorithm for tuning the feature weights used by Chinook (the World Man-Machine Checkers Champion). Their results indicated the TD-Leaf weights performed as well as the previously used hand tuned weights.

B.4 tic-tac-toe

The game of tic-tac-toe has perhaps received the most attention. Due to its simplicity it is often used to test algorithms have been correctly implemented. A standard approach to tic-tac-toe can be found in [24]. Interesting results from

tic-tac-toe include [4] in which Patist investigated the use of database learning using TD(λ) versus learning from playing against a fixed opponent.

B.5 Other games

For many games only preliminary or basic research has been done. These games include Chinese Chess in which Trinh et al. [22] successfully used TD-Learning on a simplified version of the game. In Shogi, Beal and Smith [19] used TD-Learning combined with minimax search to generate material weights that resulted in performance equal to hand-tuned weights.

B.6 Backgammon

Backgammon was the game that made TD(λ) famous, almost every paper listed in the bibliography cites one of Tesauro's papers[37] [13] [12] on Backgammon. There now exist over ten independent Backgammon programs based upon Tesauro's work, these include most strong commercially available programs and the open source GNU Backgammon.

Other research into Backgammon includes Turian [14] who investigated methods of automating feature generation using TD(λ) training to identify good feature (i.e. by using a neural network with no hidden layers and examining the weights assigned to each feature) and Wiering [41] who investigated using a multiple network architecture to allow for better approximation of the game theoretic function. One of Wiering's interesting results was that his system did not seem capable of overtraining.

Baxter et al. investigated the effectiveness of TD-Leaf and the TD-directed algorithms in [8], their results indicated that these algorithms were not able to outperform standard TD(λ) learning in Backgammon.

B.7 Othello

Othello has received a significant amount of attention, but it appears to be a question of some doubt as to how effective TD learning is in the game, with results from different groups often providing contradictory results. Given its complexity as shown by the earlier graph (figure 3) that the game may only just be within reach of our technique, it is possible that different implementations with only minor underlying theoretical differences could produce different results. Important approaches to this problem have included [28] which used RBF functions and self-play and [31] in which the Othello board was broken down into subsections.

The TD-Leaf algorithm seems to also be applicable to Othello with Wang [25] using it to tune the feature weights in his Othello agent.

B.8 Lines of Action

The only application of TD learning to Lines of Action I was able to find was that of [30] where it was used to tune the weights of an already existing function. As such it was successful and the learnt weights were able to correct several mistakes that had been made by the human developers.

B.9 Mancala

Despite the inherent mathematical structure underlying Mancala, which is likely to make the game theoretic function very smooth, the only work in this area appears to be that of Bardeen [17] who investigates combining TD-Learning with evolutionary techniques.

References

[1] Richard S. Sutton (1988) *Learning to predict by the methods of temporal difference*. Machine Learning 3, 9-44.

[2] Arthur L. Samuel (1959) *Some studies in machine learning using the game of checkers*. IBM Journal of Research and Development 3, 210-229.

[3] Peter Dayan, Nicol N. Schraudolph, and Terrence J. Sejnowski (2001) *Learning to evaluate Go positions via temporal difference methods*. Computational Intelligence in Games, Springer Verlag, 74-96.

[4] Jan Peter Patist and Marco Wiering (2004), *Learning to Play Draughts using Temporal Difference Learning with Neural Networks and Databases*. Benelearn'04: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning.

[5] R. Ekker, E.C.D. van der Werf, and L.R.B. Schomaker (2004) *Dedicated TD-Learning for Stronger Gameplay: applications to Go*. Benelearn'04: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning.

[6] Henk Mannen and Marco Wiering (2004) *Learning to play chess using TD(λ)-learning with database games*. Benelearn'04: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning.

[7] Jonathan Baxter, Andrew Tridgell, and Lex Weaver (1997) *KnightCap: A chess program that learns by combining TD(λ) with minimax search*. Technical Report, Learning Systems Group, Australian National University.

[8] Jonathan Baxter, Andrew Tridgell, and Lex Weaver (1998) *TDLeaf(λ) Combining Temporal Difference Learning with Game-Tree Search*. Australian Journal of Intelligent Information Processing Systems (Autumn 1998), 39-43.

[9] Jonathan Baxter, Andrew Tridgell, and Lex Weaver (2000) *Learning to Play Chess using Temporal Differences*. Machine Learning, v.40 n.3 (Sept. 2000), 243-263.

[10] Louis Victor Allis (1994) *Beating the world champion The state of the art in computer game playing*. New Approaches to Board Games Research.

[11] Thomas Ragg, Heinrich Braunn and Johannes Feulner (1994) *Improving Temporal Difference Learning for Deterministic Sequential Decision Problems*. Proceedings of the International Conference on Artificial Neural Networks - ICANN '95, 117-122.

[12] Gerald Tesauro (1992) *Practical issues in temporal difference learning*. Machine Learning 4, 257-277.

[13] Gerald Tesauro (1995) *Temporal Difference Learning and TD-Gammon*. Communications of the ACM 38, 58-68.

[14] Joseph Turian (1995) *Automated Feature Selection to Maximize Learning in Artificial Intelligence*. Technical Report, MIT.

[15] J. Schaeffer (2000) *The games computers (and people) play*. Advances in Computers 50, Academic Press, 189-266.

[16] Mark Lynch (2000) *NeuroDraughts An Application of Temporal Difference Learning to Draughts*. Undergraduate thesis, University of Limerick.

[17] Matthew Bardeen (2002) *TD-Learning and Coevolution*. M.Sc. Thesis, University of Sussex.

[18] Donald F. Beal and Martin C. Smith (1997) *Learning Piece values Using Temporal Differences*. Journal of The International Computer Chess Association (September 1997), 147-151.

[19] Donald F. Beal and Martin C. Smith (2001) *Temporal difference learning applied to game playing and the results of application to shogi*. Theoretical Computer Science 252, 105-119.

[20] Donald F. Beal and Martin C. Smith (2000) *Temporal difference learning for heuristic search and game playing*. Information Sciences 122, 3-21.

[21] Justin A. Boyan (1992) *Modular Neural Networks for Learning Context-Dependent Game Strategies*. Master's thesis, Cambridge University.

[22] Thong B. Trinh, Anwer S. Bashi, Nikhil Deshpande (1998) *Temporal Difference Learning in Chinese Chess*. Proceedings of the 11th International Conference on Industrial and Engineering Applications of ArtificialIn telligence and Expert Systems: Tasks and Methods in Applied Artificial Intelligence, 612-618.

[23] Fredrick A. Dahl (1999) *Honte, a Go-Playing Program using Neural Nets*. Machines that learn to play games (2001).

[24] Daniel Kenneth Olson (1993) *Learning to Play Games from Experience: An Application of Artificial Neural Networks and Temporal Difference Learning*. M.S. thesis, Pacific Lutheran University, Washington.

[25] Philip Wang *Deep Fritz: A Championship Level Othello Program*. Technical Report, Standford University.

[26] H. Jaap van den Herik, Jos W.H.M. Uiterwijk and Jack van Rijswijck (2002) *Games solved: Now and in the future*. Artificial Intelligence 134, 277-311.

[27] Horace Wai-kit Chan, Irwin King and John C. S. Lui (1996) *Performance Analysis of a New Updating Rule for TD(λ) Learning in Feedforward Networks for Position Evaluation in Go*. Proceedings of the IEEE International Conference on Neural Networks, vol. 3, 1716-1720.

[28] Minoru Ito, Taku Yoshioka and Shin Ishii (1998) *Strategy acquisition for the game "Othello" based on reinforcement learning*. Technical report, Nara Institute of Science and Techology.

[29] Shin Ishii and M. Hayashi (1999) *Strategy acquisition for the game "Othello" based on reinforcement learning*. ATR Technical report, TR-H-159, ATR.

[30] Mark H.M. Winands, Levente Kocsis, Jos W.H.M. Uiterwijk, H. Japp van den Herik (2002) *Learning in Lines of Action*. Proceedings of the Fourteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2002), 371-378.

[31] Anton Leouski (1995) *Learning of Position Evaluation in the Game of Othello*. Master's thesis, University of Massachusetts.

[32] Bryan McQuade (2001) *Machine Learning and the Game of Go*. Undergraduate thesis, Middlebury College.

[33] Justin A. Boyan and Andrew W. Moore (1996) *Learning Evaluation Functions for Large Acyclic Domains*. Machine Learning: Proceedings of the Thirteenth International Conference.

[34] Curtis Clifton and Mark Slagell *Quixote: The Quixo Temporal-Difference Environment*. Technical Report, Iowa State University.

[35] Jonathan Schaeffer, Markian Hlynka and Vili Jussila (2001) *Temporal Difference Learning Applied to a High-Performance Game-Playing Program*. Proceedings of the 2001 International Joint Conference on Artificial Intelligence (IJCAI-2001), 529-534.

[36] Richard Sutton and Allen Bonde Jr. (1992) *Nonlinear TD/Backprop pseudo C-code* GTE Laboratories.

[37] Gerald Tesauro (2002) *Programming backgammon using self-teaching neural nets*. Artificial Intelligence 134, 181-199.

[38] Louis Victor Allis (1988) *A knowledge-based approach of Connect-Four. The game is solved*. Masters Thesis, Free University of Amsterdam, Amsterdam.

[39] Sebastian Thrun (1995) *Learning to Play the Game of Chess*. Advances in Neural Information Processing Systems 7.

[40] Jordan B. Pollack and Alan D. Blair *Why did TD-Gammon Work ?*. Advances in Neural Information Processing Systems 9, 10-16.

[41] Marco A. Wiering (1995) *TD Learning of Game Evaluation Functions with Hierarchical Neural Architectures*. Master's thesis, University of Amsterdam.

[42] Michael Buro (2000) *Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello*. Games in AI Research.

[43] D. F. Beal (2002) *Learn from your opponent - but what if he/she/it knows less than you?*. Step by Step. Proceedings of the 4th coloquium "Board Games in Academia".

[44] Johannes Schäfer (1993) *Erfolgsorientiertes Lernen mit Tiefensuche in Bauernendspielen*. Technical report, Unversität Karlsruhe.

[45] Dimitrios Kalles, Panagiotis Kanellopoulos (2001) *On verifying game designs and playing strategies using reinforcement learning*. Proceedings of the 2001 ACM symposium on Applied computing, 6-11.

[46] Charles L. Isbell (1992) *Exploration of the Practical Issues of Learning Prediction-Control Tasks Using Temporal Difference Learning Methods*. Master's thesis, MIT.

[47] Jean-Francois Isabell (1993) *Auto-apprentissage, l'aide de rseaux de neurones, de fonctions heuristiques utilises dans les jeux stratgiques*. Master's thesis, University of Montreal.