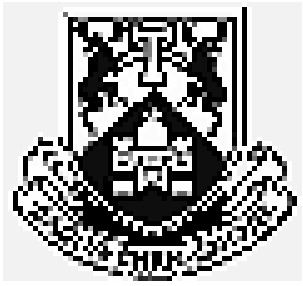

NeuroDraughts

An Application of Temporal Difference Learning to Draughts

by

Mark Lynch

Dept. of CSIS, University of Limerick, Ireland.

Email: mark@lynch.cc

Supervisor: Niall Griffith

Email: niall.griffith@ul.ie

Abstract

NeuroDraughts is a draughts playing program which follows the approach of both NeuroGammon (G.Tesauro) and NeuroChess (S.Thrun). It uses an artificial neural network trained by the method of temporal difference learning. It achieves a high level of play simply by self play, with minimal search, and without any expert game analysis.

Keywords: Temporal Difference Learning, Input modelling, Game Theory, Draughts, Checkers.

Acknowledgement

I would like to thank all those who have indirectly or directly influenced and helped me during the duration of this project, most especially my Family and Friends for putting up with my incomprehensible ranting about various aspects of NeuroDraughts. Many thanks to my supervisor, Niall Griffith, for his unbounded help and encouragement. To Dave Harte for his invaluable play testing which helped me rate the level of play. My brother Daniel, and sister Caragh, for their great art work, which made everything look so much nicer!

Table of Figures

1. Introduction

1.1 Purpose

The purpose of this project was to create a good draughts player with as little expert tuning as possible. Most game playing programs today avail of fined-tuned feature sets, or heuristics, as well as incredibly efficient search algorithms. The aim of this project was to create a 'more human' opponent, more specifically, an opponent which could learn from self-play given the rules of the game and a few simple features. It was also the purpose of this project to show that this opponent could compete at an acceptable level with a minimum of search. An acceptable level of play in this case could be defined as being able to beat it's creator.

1.2 Why Draughts?

Draughts was chosen over Chess because the relative simplicity of it's rules allowed for greater emphasis to be put on the learning and representational issues. Anyone doubting the complexity of the game should refer to Oldsbury's great book on the game, Move-Over [1] or to [14].

1.3 Why Temporal Difference Learning (TD)?

The TD(lambda) family of learning procedures have been applied with astounding success in the last decade. Most notable among these successes is G. Tesauro's NeuroGammon [2] which plays backgammon at world champion level. S. Thrun also created Neuro-Chess which played a relatively strong game [3]. The TD procedure is nothing new however having been first applied by A.L. Samuels in his famous checkers program all the way back in 1959 [4]. The procedure has since been formalised and convergence proven by Sutton [5].

TD is particularly well suited to game playing because instead of forming pairs between the actual outcome and each state encountered in the game, instead it updates it's prediction at each time step to the prediction at the next time step. This also means that a record doesn't need to be kept of the game as only the current state and the previous evaluation is needed to calculate the error at any particular stage.

TD provides many advantages over traditional supervised learning methods, most important of which is that TD can achieve a high level of play simply from playing against itself. Because it requires no human intervention tournaments can set up and left running for thousands of games resulting in a high standard of player.

Problems such as over training are also reduced as the non-terminal boards are decayed through time. This means that no intermediate boards are likely to have extreme values associated with them, which could lead to the network getting 'stuck' during training.

2. Background

2.1 Samuels's Checkers Program

It was Samuels's who pioneered the idea of updating evaluations based on temporally successive predictions in his checkers program. The fact that his experiments where carried out in the early fifties on an IBM 704 makes it all the more remarkable. He also details some ingenious methods for saving processor time and memory space, not one CPU tick or Byte of RAM is wasted.

"… we are attempting to make the score, calculated for the current board position, look like that calculated for the terminal board position of the chain of moves which most probably will occur during actual play. Of course, if one could develop a perfect system of this sort it would be the equivalent of always looking ahead to the end of the game." A.L. Samuels

His program made use of a polynomial evaluation function as opposed to an ANN as used in NeuroDraughts, with the coefficients of the terms being adjusted instead of weights.

As well as devising this ingenious scheme for training he also created a very good feature set which has been cut down and adapted for NeuroDraughts (see Appendix A). Samuels's showed in his experiments that binary combinations of features produced better results, making an Artificial Neural Network (ANN) a perfect platform given it's inherent non-linearity due to the hidden layer.

He also details an alternative internal board representation which is detailed in Appendix C that makes many calculations much simpler. Indeed it is easy to see why his Checkers program is, and will remain to be for quite some time, one of the most referenced works in Machine Learning.

2.2 Supervised Learning

The Multi Layered Perceptron

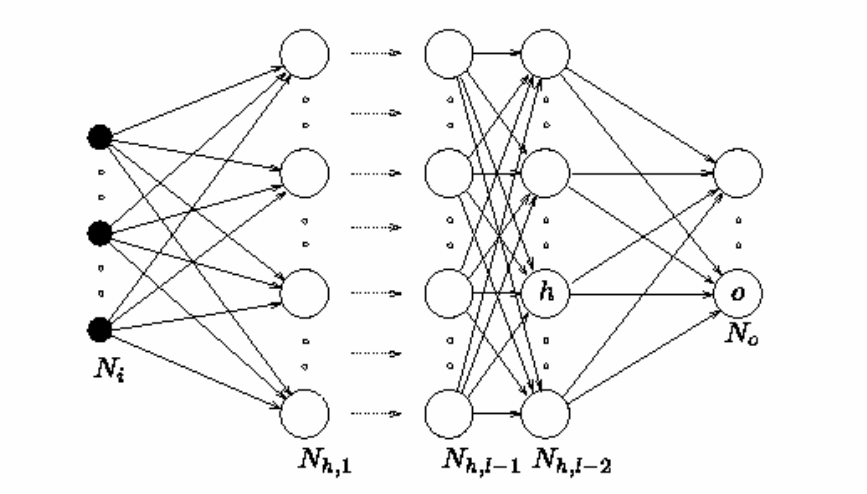

Figure 1: *A Multi Layered Perceptron. Taken from [12].*

A multi layered perceptron (MLP) consists of a layer of inputs, a layer of hidden units and a layer of outputs. Each input is connected to each hidden unit and each hidden unit is in turn connected to each output. It can be used to approximate almost any function.

Without the hidden unit an ANN would only be able to approximate linearly separable functions. A linearly separable function is one such as AND or OR whose output graph, when plotted, has a clear line between the different value groups. This is best demonstrated by the XOR problem[1], whose error space has no hyperplane (see Fig 2). i.e. On the third graph of figure 2 there is no single straight line that will place the open and filled circles in separate regions.

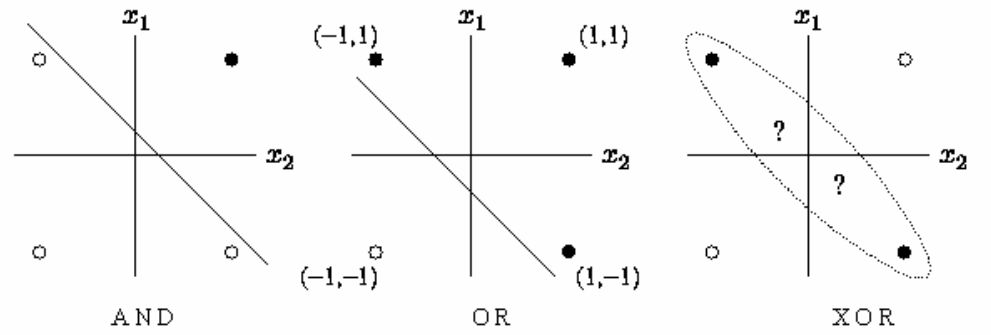

Figure 2: Limitations of the single layer perceptron. Taken from [12].

[1] For a more comprehensive treatment of this problem please see [7].

Training a MLP becomes a lot more complex due to the addition of the hidden layer. The backpropagation algorithm is basically a generalisation of the delta rule as used in the single layer perceptron.

Squashing Function

To control the range of values coming from each hidden unit, it is necessary to apply a squashing function. This facilitates learning as values of each hidden unit (and the output unit) are always within a certain range, thus nullifying any side affects due to extreme values. Standard choices for this function are the sigmoid (range 0 to +1) and hyperbolic tangent (range -1 to +1) functions. As the hyperbolic tangent function fits the needs of NeuroDraughts best (-1 for loss, +1 for win), it was chosen over the more popular sigmoid.

Backpropagation

Again it cannot be stressed enough that for a full explanation and derivation of this algorithm to see the references [7],[12].

In order to make the weight updates it is first necessary to calculate the error signal of the network. This is given by:

$$\sigma_k = (t_k - o_k)\, f'(net_k)$$

where f' is the derivative of the activation function (or squashing function) f. The formula to change the weights between the output unit, k and input unit j is:

$$w_{jk} \leftarrow w_{jk} + \eta \delta_k o_j$$

where η is some relatively small constant called the learning rate, usually set to 1/n where n is the number of units in the layer the weight originated from.

The error signal for the hidden unit j (or, more correctly, it's contribution to the error) is given by:

$$\sigma_j = f'(net_j) \sum \sigma_k w_{jk}$$

The weight update rule is essentially the same for the weights between input units i and hidden units j.

$$w_{ij} \leftarrow w_{ij} + \eta \delta_j o_i$$

Momentum

Momentum in backpropagation tries to speed up the process of finding the correct weights by adding a fraction of the previous weight change to the current update. If the weights changes are on the right track, this process speeds things up enormously.

2.3 Learning in a Dynamic Environment

In order for backpropagation to work it must be presented with training pairs. These pairs consist of an input vector and an expected outcome. This method is very unsuitable for game playing where it might be impossible, or at least time consuming, to assign correct values to intermediate boards. Methods such as this are called supervised learning methods, and are in stark contrast to reinforcement learning methods which require the teacher simply to define the conditions for receiving a reward.

It is in this area that reinforcement learning methods win hands down. By concentrating on getting the learner to 'learn for itself' they eliminate the need for a teacher and thus automate the process. The only information that is needed to perform such training is to know what is a good outcome or state and what is a bad one, as well as rules governing state transitions (in this case legal moves). Given this information it is entirely feasible that the system can learn simply by exploring the state space using the transition rules given, as well as applying the appropriate rewards.

A system created by Michael Gherrity, SAL[13], is just such a system. It can theoretically learn to play any game based on a board. The user simply provides the system with the game structure and a list of rules governing play. Through self play it then develops above average levels of play.

2.4 Reinforcement Learning & Temporal Difference Learning

In reinforcement learning the learner is rewarded for performing well (in this case winning) and given negative reinforcement for performing badly (i.e. losing). In between the starting board and the final board when no specific reward is available the TD mechanism tries to update the prediction for the current state to that of the next state. For all the non-terminal boards states the program forms a training pair between the current board state and the prediction of a win for the next state.

TD can be basically viewed as an extension to backpropagation[2] and gives the following weights change formula:

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

where Alpha is the learning rate. Lambda in this equation controls how much the final reward trickles back to the starting states, this is generally referred to as the eligibility trace. If set to zero this is eliminated and we have the special case TD(0) in which each state is trained simply to it's successor.

An important part of TD is that it doesn't have to wait until the final outcome to train. This means that only one state (e.g. board state) must be kept in memory. Thus time is saved as the actual learning is occurring throughout the game. This would be particularly beneficial if multi processor machines were available. One thread could be working out the next move whereas the other could be training for the current move.

The famous example of why TD is superior to ordinary supervised learning approach's is the following game playing example as given in [5]. See Figure 3.

Supposing you land in the novel state (which you have previously never encountered) and subsequently end up in the bad position but still win the game. What value is then associated with the novel state?

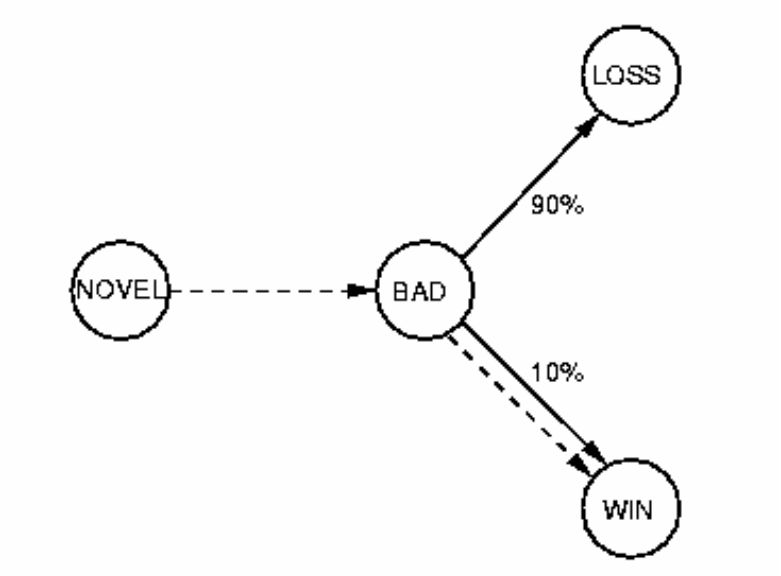

Figure 3: *A game-playing example showing the inefficiency of supervised learning methods.*

[2] For a more detailed description of TD please see [2],[5].

A supervised leaning method would associate the novel state with a win which is obviously wrong as we have found from experience that it has a high chance of resulting in a loss. The TD method, however, would form a pair between the novel and bad state which immediately followed it. Assuming that the evaluation of the bad state is correct then we have correctly evaluated the novel state. TD will consider a state good if the most likely state following it is good and vice versa, thus after playing several games one can expect the TD trained network to be a very accurate prediction of winning/losing.

3. NeuroDraughts - Explained

3.1 Architecture

At the heart of NeuroDraughts are two, meticulously designed, C++ classes. These classes are EvalNet, which implements the ANN and TD aspects of the project and DirectNet, which itself creates and manipulates EvalNet objects. DirectNet's sole purpose is to mirror the functionality of EvalNet whilst providing additional functions necessary to allow board states to be passed as parameters. It supports several input representations as detailed in section 3.3.

The EvalNet class was designed so as to be easily re-usable in any ANN/TD project and this was demonstrated by it's successful use in solving the Gridworld puzzle, without any modification. As it's name suggests EvalNet was designed with only a single output in mind. This meant a large saving in complexity with little loss of generality.

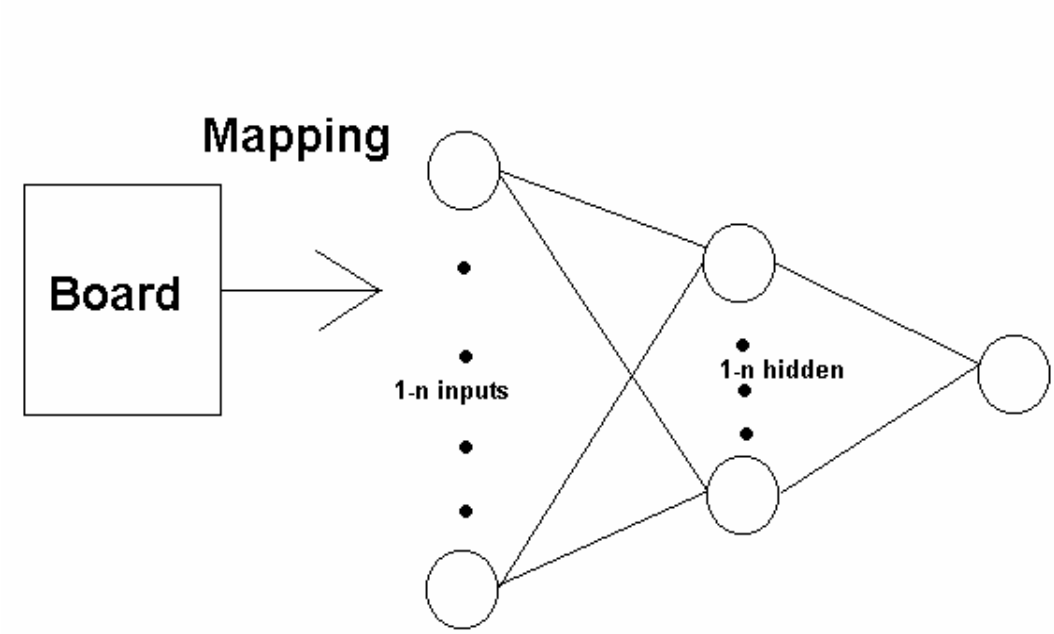

Figure 4: *How a board is evaluated. Mapping a board to the input vector is handled by DirectNet which then passes this to EvalNet which evaluates the position.*

The evaluation network and the temporal difference procedures are incorporated into the EvalNet class. The mapping of the board to the inputs is handled by the DirectNet class which sits above EvalNet.

Two types of internal board structures were used, an ordinary board (1-32) and an extended board (1-35) (see appendix C) which makes a lot of computations much simpler. Both boards are implemented in a simple C fashion to increase performance. A library of functions was written to manipulate the boards.

Another aspect of the project was of course the feature mapping. This is provided again by a simple library of functions, each of which accepts the board structure and returns either true or false. These functions were designed to be completely independent of one another to allow for full flexibility, albeit at the cost of performance.

3.2 The Evaluation Network

At the base of NeuroDraughts is a neural network. The network consists of a layer of inputs, hidden and a single output unit. The network is trained using back propagation with momentum [6] [7]. This net is trained to evaluate boards after the player in question has moved.

3.3 Representation of Board

How the board is fed to the evaluation network was a major issue for NeuroDraughts. Although as little domain specific knowledge as possible was the aim, in practice this was impossible. In order to learn to play at an acceptable level some sort of feature mapping was required. If just the raw input is used, small changes in the board can produce drastically different evaluations which goes against the smooth nature of the game. By mapping the board to a set of features (adapted from those provided by Samuels [4]) we increase the ability of the Evaluation network to generalise and thereby glean more information from each game.

As part of the purpose of NeuroDraughts was to show the advantages (or plain necessity) of feature mapping several different strategies were tried. These were as follows:

- NET_BINARYMAP - Each board square is represented by 3 inputs. Each set either to 1 or -1 depending on what piece occupies it.
- NET_DIRECTMAP - Each board square is represented by a single input which is set to 0 for empty, 0.25 for black man, 0.5 for red man, 0.75 for black king and 1 for red king.
- NET_FEATUREMAP - The board is now represented by a given number of features. A full description of all features can be found in Appendix A.

3.4 Automatic Feature Generation

Due to time constraints this area has not been explored. Originally it was planned to incorporate some form of feature optimisation into NeuroDraughts. Much work has been done on automatic feature generation but without very much success [9],[10],[11]. One promising way forward however is hybrid systems which are described in detail by Martin Schmidt[8].The features would be represented by a Genetic Algorithm and mating selection process would consist of a tournament of an arbitrary number of games.

Working on the success or failure of this approach it was then planned to see if any inroads could be made into automatic feature generation as opposed to simple optimisation.

4. Developing NeuroDraughts

NeuroDraughts was developed in several stages. I did a lot of research before starting coding to make sure that the project was feasible. This involved reading many journal papers and consulting with both Gerald Tesauro and Sebastian Thrun, who are probably the world's leading authorities on the use of TD methods in game playing. Several papers exist which share some similar elements to NeuroDraughts, namely [13], [15], [17], [18], [19], [20], [21] and [22].

4.1 Design Strategy

The first priority in the design of NeuroDraughts was to create and test an MLP. The TD elements would then be incorporated and the board mapping functions added. It was an important design consideration that the evaluation network be as independent as possible, with no specific references to draughts. The results of this was a highly re-usable class which can be implemented in a variety of programs. This is demonstrated by how easily it slotted into the Grid World problem with no modification. Lastly the User Interface (UI) had to be created in order to allow human opponents to compete against the computer.

4.2 Developing the MLP

Developing the MLP was a fairly simple process as a wealth of documentation and sample source code was available. Once the code was written it was tested on the XOR problem, already described. The results achieved were in line with those reported by others. Once this was working, momentum was added and significant improvements noted. One problem to note here is that the momentum term was implemented straight*. i.e.* no checks were made to see if the momentum term was in fact going in the same direction as the current weight update. It was found however, that performance increased when a check was added, but at the expense of more CPU time. This is discussed further in section 5.1.

4.3 TD and Gridworld

When coding began a lot of difficulties arose and there was a lot of uncertainty as to whether or not the TD mechanism was working as it should. It was decided to try a simple problem with which we could clearly see whether or not all was well. The problem selected was the grid world puzzle, in which the network tries to learn the fastest path to the exit or goal state. The original implementation of this puzzle uses a network with four outputs and at any step the network must select the direction. Because we were more interested in evaluating positions we restructured the problem so that it would consider each neighbouring square and select to move to the one with the highest evaluation. Solving the problem in this way meant it was now analogous to selecting good moves in draughts.

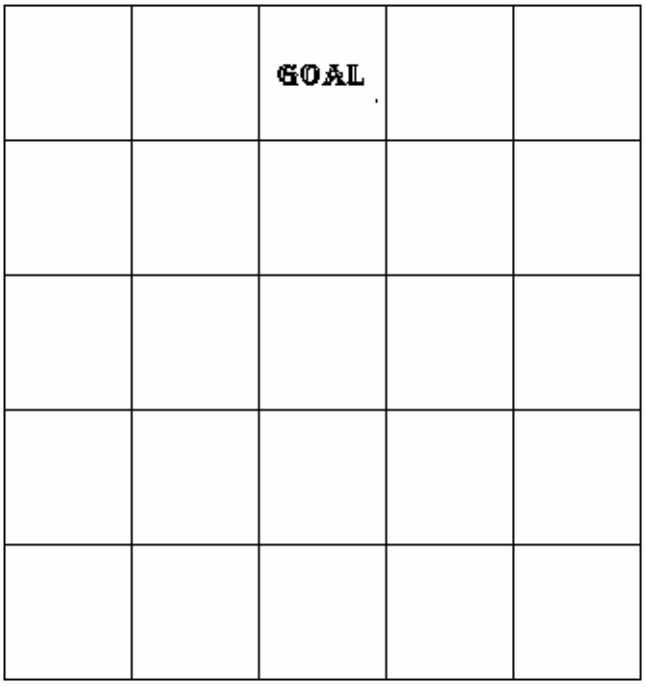

Figure 5: *The Gridworld Puzzle.*

The GridWorld puzzle consists of a 5*5 world. Upon reaching the goal (in the optimal time) it receives a reward of +1 whereas getting stuck (not reaching the goal in 10 moves - the extra leeway giving it a chance to reach the goal in sub-optimum time) gets it a reward or -1.0. The program is continually started at random positions in the world and ‘let wander’. The program selects it’s moves by evaluating all possible moves, up, down, left or right, and choosing the one with highest evaluation.

It was found that training time decreased considerably when the starting point's distance from the goal was gradually increased over time, allowing the net to learn the squares close to the goal first. This lesson proved very valuable in teaching NeuroDraughts how to play.

As the GridWorld puzzle used the same evaluation network class and TD procedures, it was safe to assume after this experiment that all was in order with TD.

4.4 Results of Gridworld

The following results show how the reinforcement given at a particular stage affects the learning. Success refers to the reward given after success arrival at the goal, failure when after 10 moves the goal still hasn't been reached and Side is the reinforcement given when a side square is moved into. Result is the number of random starts required for the optimal solution to be found, the number in brackets reflecting how many where not learned in the case of failure.

Lambda = 0.1 Gamma = 0.9

SUCCESS	FAILURE	SIDE	MOVES	RESULT
+1	0	0	1050	All Learned
+1	-0.2	0	900	All Learned
+1	0	-0.1	1500	1 Failed
+1	-0.2	-0.1	1500	1 Failed
+1	-0.2	-0.2	1500	18 Failed
+1	-1	0	1050	All Learned

5. Important Elements

This is a brief discussion on some of the elements which were pivotal in Neuro Draughts development. They include such additions as momentum with test and adding Direct Links.

5.1 Momentum

As a result of choosing the hyperbolic tangent activation function it was necessary to employ a direction check when applying momentum. If a weight update was in the opposite direction to the previous weight update, then both could cancel one another out. This resulted in the network sometimes getting stuck in extreme values, as it would take quite a while to reverse the direction of the changes, especially if a high momentum term was being used. This necessitated the implementation of a check when applying the momentum term. So if the momentum term is in the same direction it is applied, otherwise it is not.

5.2 Direct Links

Again this was a major error in my judgement. Because evaluating a draughts board is in most implementations a linear function it only made sense that correctly evaluating a draughts board was indeed at least a partly linear function. It was found by Samuels, whose original evaluation function was a simple linear polynomial that after the addition of some binary connective terms, performance significantly improved. This clearly shows that some elements of draughts cannot be approximated by linear means alone, thus making a network with both linear and non-linear (via hidden units) capabilities the ideal choice. This assumption proved true after initial training provided a player that, using only 2 ply look ahead, could beat most human opponent of moderate skill level.

5.3 Modular Networks

After reading some impressive results in [16][19] it was decided to try incorporate modular networks into NeuroDraughts. The approach taken was that three networks would be used, one for attacking play (a piece advantage), one for defending (a piece disadvantage) and one for mid-game play. It was found however, through analysing what was being trained, that the mid-game was getting well over half of the training time. The idea was left out of the current version of NeuroDraughts, but with more thought and consideration on how better to utilise it, it could prove invaluable.

6. Training NeuroDraughts

6.1 How NeuroDraughts was trained

At first it was attempted to train NeuroDraughts using expert games. However this approach failed as the number of games available was small (about 200), as well as the fact that expert games tend to terminate with seven or more pieces still on the board. In the end a self-play regime of learning was finalised upon. Several different strategies were tried before finally settling on the current one. I will now describe some of them and why they failed.

6.2 Straight Play

This method simply let two opponents play against each other, both learning for a set number of games. Because no benchmarks are available at any stage of the training, it is impossible to know if the nets are improving. One net might hit a slump and then the other network can't be judged as being good because it is beating a weak opponent.

6.3 Cloning

Using this method, a set number of games was played with only one network training. If the training network succeeded in winning a set percentage of games, usually 80-90%, then the non-training network copied it's weights. Using this method meant that the standard of play was forced to increase. There was however one flaw with this approach.

Using the results of, say 50 games, is not reliable because even if the training net loses the last 5 games after winning the first 45, he will be cloned. This means that although it was good, it has since degraded, and a much poorer network is in fact being cloned. The solution to this was to play 10 games training, after which two games are played, one as black, one as red. If both of these games are successful then the network is cloned. This technique quickly showed much promise, outdoing previous networks. This is demonstrated in Figure 6.There was however still one flaw in the process which is dealt with in the next, and final, method.

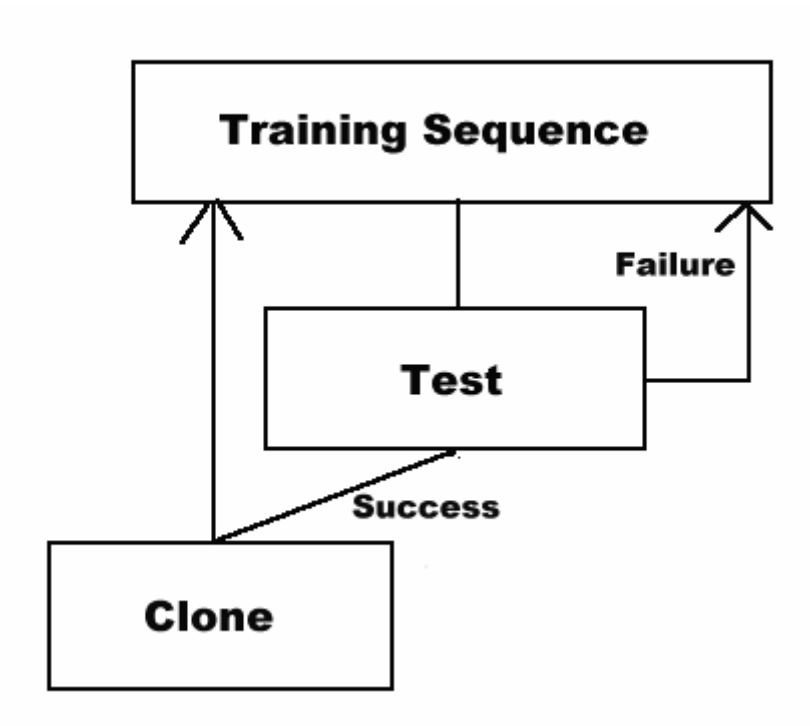

Figure 6 - How the Training/Cloning Process Works

6.4 Cloning with Look Ahead

One of the problems with the features used is that none of them could be very accurate at a given time step. In order to get a better evaluation, a slight amount of look ahead was required. All previous attempt failed because, although the networks realised the value of pressuring the opponent and eliminating his pieces, it also saw sacrificing pieces as good at times, not realising that this was only good in certain circumstances. To eliminate the difficulty in evaluating exchanges a 1 move (2 ply) look ahead was added to the training. By doing so we narrowed the search space of the problem significantly and thus increased the performance dramatically. The results with this method have been the best so far with the network performing to above average standard after only 500 training games.

6.5 What gets Trained?

There are three distinct areas to discuss here. Training on expert play, self play and against a human opponent.

Expert Play

Before training from expert play the expert games are converted into an easily digestible format. This involves swapping all red boards around and grouping the boards into red and black boards. Black boards being those when black had just played and vice versa. The games are then read in and the net trained as if they were just happening. The major problem, as mentioned earlier, is that expert games never tend to go down to the wire. They always finish with six or more pieces left. Because many of the strategic nuances seen in expert play rarely make much impact on an ordinary game it is very difficult to learn them. This coupled with the fact that the games terminate very early makes learning from the masters a poor choice for NeuroDraughts.

Self Play

During self play there is only ever one network being trained as is the nature of the cloning mechanism adopted. This means that when the training network is red all boards must be swapped before being used. Self play is probably the best method of training as it is fully automated and should consistently improve from generation to generation. This said it is also true to say that play can degenerate under circumstances, especially when no look ahead is used when playing. This is particularly true of draughts, were many moves are forced and a simple one move look ahead can reveal a lot.

Human Play

It has yet to be seen what, if any, improvement can be gained from training when playing against a human opponent. If the player is of a high standard then it would be beneficial to learn from and it is possible (v. likely) that the network would evolve to a more capable match for the human. On the other hand performance would undoubtedly degrade if playing against a poor human opponent. I'm afraid it will have to be up to the individual whether or not to turn learning on when playing.

7. Final Training Sequence Results

In order to see how different values of Lambda affected the training, 5 networks were created with exactly the same seed and parameters. The only thing that was varied was Lambda. 100 sequences of 10 games were played with a test after each one. A league was then played between the last clone of each network (most reached 5th generation).

Lambda	0.0	0.1	0.2	0.3	0.4	Total
0.0	x	2	2	2	4	10
0.1	2	x	1	4	1	8
0.2	2	3	x	2	2	9
0.3	2	0	2	x	2	6
0.4	0	3	2	2	x	7

The following shows the graphs of Average Pieces plotted for both the best (Lambda 0.0) and worst (Lambda 0.3) cases. Only the first 40 games are shown, after which point most of the networks oscillated, only one or two actually cloning again in the later stages. The gaps indicate cloning took place.

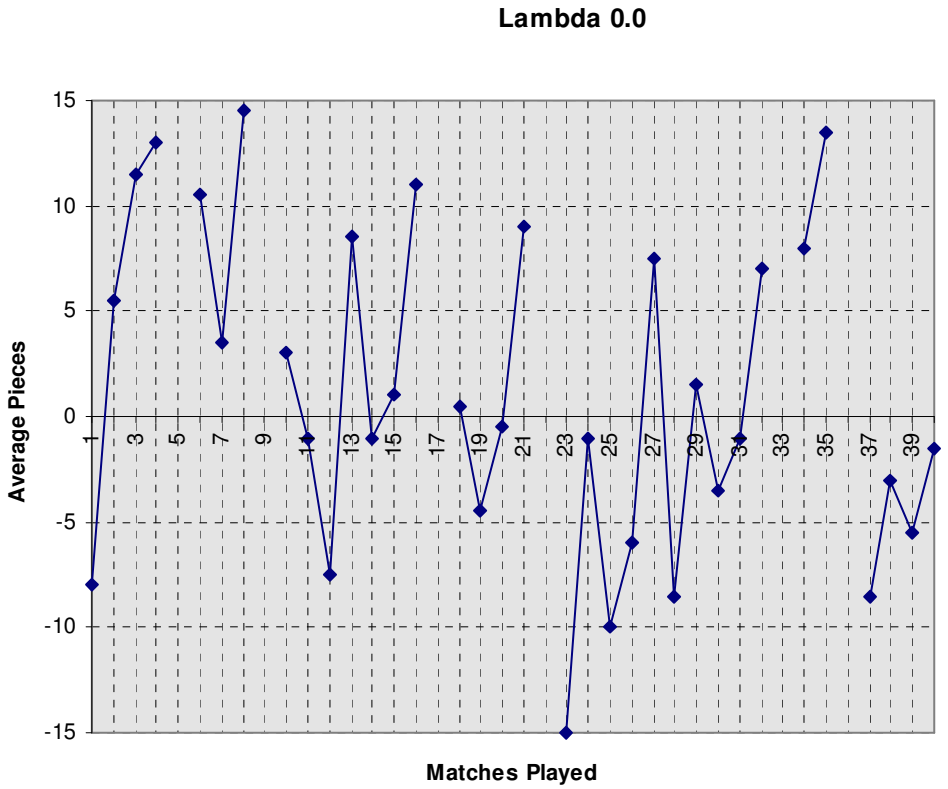

Figure 7 - Lambda 0.0 Results

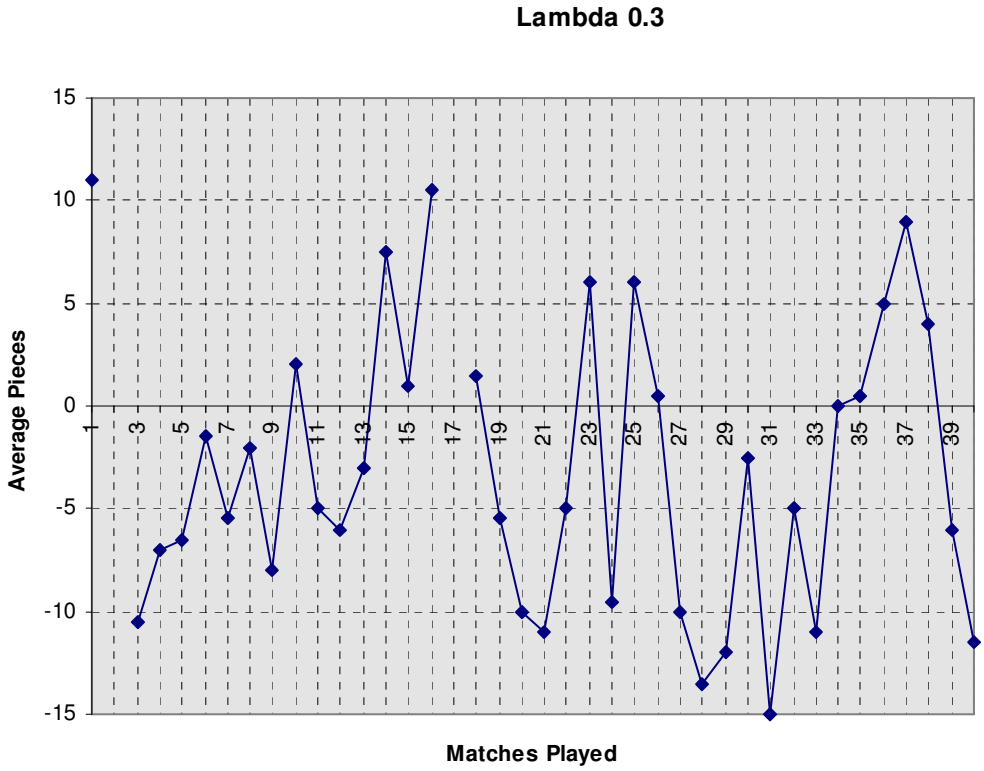

Figure 8 - Lambda 0.3 Results

Another interesting graph to see is the correlation between Average Pieces and Moves Taken as demonstrated here. Note: Lambda = 0.1

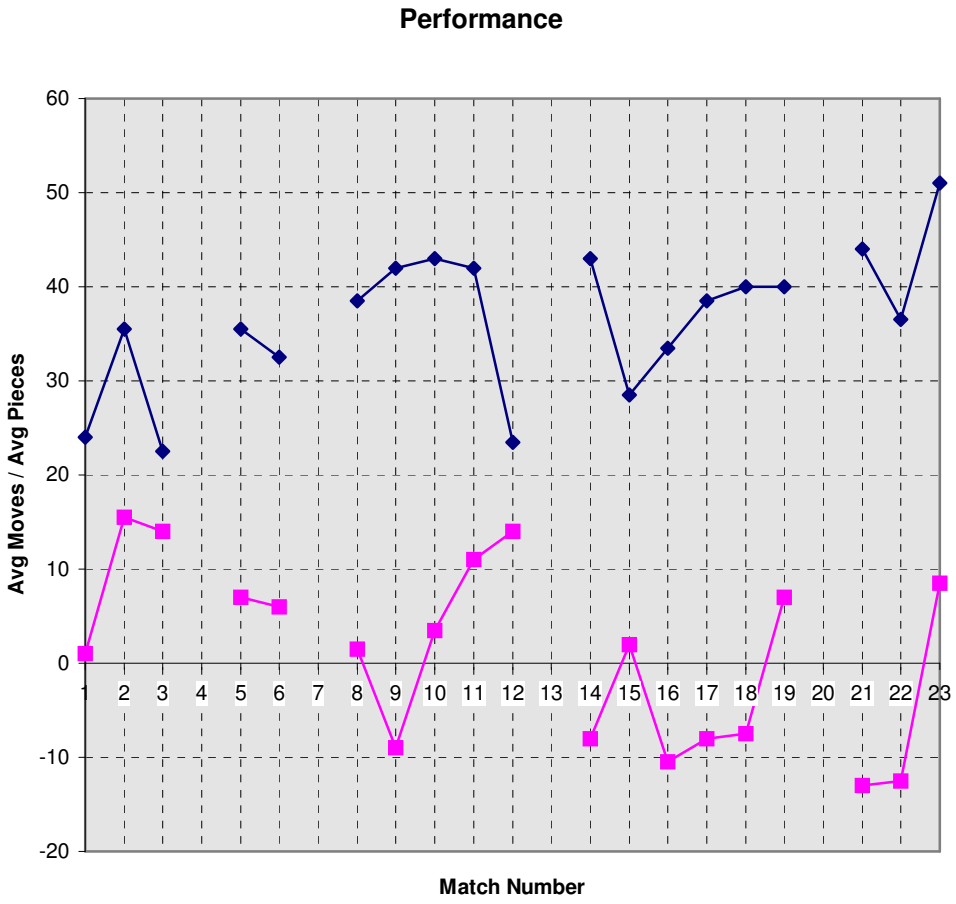

Figure 9: Performance Graph of Training

In this graph the Blue (higher of the two) line represents the average moves taken and the pink line the average number of pieces left (for the net being trained). It can be clearly seen from this graph how for each segment of games, play steadily (with only some jitters) improves until cloning. If one looks at the larger picture you can see that number of pieces is going down and moves is going up as the network is continually finding it harder to beat it's clones.

8. Walk-Through

This section hopes to provide an overview of what goes on when NeuroDraughts is learning through self play. It can be split into the following sections: Tournament & Test, Move Selection & Evaluation and Final Reward.

8.1 Tournament & Test

At the basis of self play is the idea of a tournament. This basically means a set number of games during which training is occurring followed by two games, the test, which show whether or not the level of play has improved enough to beat it's clone. The only differing condition between the players, is that one is being trained during the initial ten games whereas the other (clone) is not. Each opponent receives an equal number of starts (ie. black starts) and both use the same search techniques, if any. At the end of these ten games, two games are played (one as black, one as red), with training turned off, so both players remain static making any further games pointless. If the opponent that we are training completes both successfully then his weights are copied to the non-training opponent (he's cloned) and the process is started again.

8.2 Move Selection & Evaluation

Move Selection is made in much the same way as an ordinary heuristic approach, the rules simply replaced by a neural net. Each possible board is generated from the current position and then evaluated. If the player is red then the boards must first be swapped before being evaluated. If search is on then each moves game tree is first searched as well, to the depth specified. Once complete the move with the highest evaluation is passed back the tournament which then executes the move for the player in question.

8.3 Final Reward

At the end of the game depending on who has won, a reward is given to the network being trained. This is the most important element of the training as it here were the behaviour is either positively or negatively reinforced.

9. Setting up NeuroDraughts

There are several programs included with NeuroDraughts to allow anyone to create neural nets to their specifications and also to train them in various ways. I will now outline the basic file structures used by the different programs and then detail how to use the programs.

9.1 File Structures

When specifying a NeuroDraughts network, one can either specify a Creation File or a Pre Created Network.

Creation File

NumInputs:	37	-1 = BinaryMap, 0 = DirectMap
NumHidden:	12	How many hidden units
DirectLinks:	1	Use Direct In-Out Links
NetworkSeed:	210476	Seed for Random numbers
LowRange:	-0.2	Low Range of Random Numbers
HighRange:	+0.2	High Range of Random Numbers
NetworkName:	FeatureNetwork	Give your Network a Name

Pre Created Network

SuperFeatureNetwork	Name
37 12 1	Inputs, Hidden, DirectLinks
In - Hid Weights	All the Input - Hidden Weights
Hid - Out Weights	All the Hidden - Output Weights
In - Out Weights	All the Direct In - Out Weights (if applicable)

When specifying the inputs in a creation file, a 0 means BINARY_MAP and a 1 DIRECT_MAP, using any higher number then the network will assume that number corresponds to the number of features to be used.

Parameter File

LRate1:	0.0	Learning Rate for 1st Layer (also in-out layer)
LRate2:	0.0	Learning Rate for 2nd Layer (hid-out)
Momentum:	0.9	Momentum Term
Gamma(Decay):	0.98	TD Decay
Lambda(TD?):	0.1	Eligibility Trace value.

By Setting a Learning Rate to zero you ask the program to calculate an appropriate value for you. The program uses the formula 1/inputs for the first learning rate and 1/hidden for the second.

Board File

2 - How many boards are in the file

```
                          The Board                                  Params
/--------------------------------------------------------------------------\  /----------\
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2  -0.1 +0.1 4
0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 3 0 0  -0.5 +0.5 3
```

The Parameters correspond to, draw reward if black, draw reward if red and points to clone respectively. This allows to train networks on situations where drawing is actually good (when at piece disadvantage) or where it is very bad. The points to clone represents how many points are needed from the two game test sequence in order for the network to be passed fit for cloning. 2 points are awarded for a win, 1 point for a draw, none for a loss.
Numbers are used to represent the pieces. There are as follows: 0 = Empty, 1 = BlackMan, 2 = RedMan, 3 = BlackKing, 4 = RedKing.

9.2 How to Use the Programs

NeuroDraughts uses five programs for various functions, the following is a descriptions of how to use them and what to use them for.

Cloning Tournament Runner

Usage: ND create_file, param_file, board_file, savegame_file, testafter, gamelimit

This is the main program in the NeuroDraughts family. It takes as it's parameters a Creation File or Pre Created File, a Parameter File and a Board File. It also lets you specify where to save the games, how many games to test after and how many tests to make.
Using the creation file is creates two equal networks. It then uses the parameters file to set up the learning rates and momentum. It then plays the specified number of games on each board from the Board File. If the network one (which is being trained) meets the criteria specified by the Board File it then gets cloned and the process is repeated until the limit is encountered.

Fight Simulator

Usage: FIGHT opponent1, opponent2, board_file, savegame_file

In order to match two differing networks against one another the Fight program is provided. This again take a Board File as a parameter. It also take two Pre Created Files (the two opponents) with which it creates the networks.
Using these network it then goes through each of the boards in the Board File, playing two games on each (each player getting to start once) and showing the results. This is a simple method of determining the strength of two different networks.

PDN to ND Converter

Usage: CONVPDN PDN_file, output_file, ignore_draws

PDN is a standard Draughts Notation and therefore it is useful to have a program which will convert between PDN and a list of boards, which is what ND uses for training. You must give it the PDN and output file and it also allows you to specify if draws are to be ignored or not. This allows a user to take their favourite championship games and train a network with them using the following program.

Expert Trainer

Usage: XTRAIN create_file, param_file, training_file, training_epochs

This program takes a Creation File or Pre Created File, a Training Data File and a Parameter File as it's input. Using this training File it trains the network for the specified number of epochs.

NeuroDraughts

This is the interface which allows you to play against one the created networks. Before playing you must copy the network you want to play against to opp1.net and then run the program. Prepare to be beaten!

10. Conclusions

Some of the goals I set out in my initial aims for this project have, for various reasons, not made it to implementation. The main purpose was to see how good a draughts player could be created with as little as possible domain specific knowledge. In the end I had to concede and use feature mapping but if more processing power was available, as well as more time, more might have been achieved. It was also an aim of this project to investigate the possibilities of automatic feature discovery. Although none of the ideas have made it into the current version of NeuroDraughts, some promising progress has been made.

10.1 NeuroDraughts, a success?

As set out above, the aim was to create a draughts playing program that could beat it's creator. The program achieved this easily, with the creator only managing the occasional draw. It has also performed well against Dave Harte, an Irish U18 champion and other competent draughts players. Given that my knowledge of both ANN and TD, as well as other AI methods was minimal when starting out on this project a year ago I feel that a lot has been achieved, both from a personal point of view as well as a technical and theoretical point of view.

10.2 Future Development

As already mentioned, a lot of ideas for NeuroDraughts remain un-implemented. It will however remain in development, with the emphasis being put on reducing the domain specific knowledge whilst trying to keep the level of play constant. The most promising development to be tried is that of Hybrid Systems, which avail of Genetic Algorithm technology to attempt to automatically discover features.

11. References

[1] Derek Oldbury. "Move Over or How to win at Draughts".

[2] G.J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8, 1992.

[3] S. Thrun. Learning to play the game of chess. In G. Tesauro, D.Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, San Mateo, CA, 1995. MIT Press.

[4] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3, pages 210-229, 1959.

[5] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 1988.

[6] D.E. Rummelhart, G.E. Hinton and R.J. Williams. Learning internal representation by error propagation. In D.E. Rummelhart and J.L. McClelland, editors, *Parallel Distributed Processing.* Vol. I + II. MIT Press, 1986.

[7] Don Tveter. How to succeed at backprop. Available at http://www.mcs.com/~drt/software/backprop.zip

[8] Martin Schmidt. A unification of Genetic Algorithms, Neural Networks and Fuzzy Logic. The GANNFL approach.

[9] Paul E. Utgoff. Feature function learning for value approximation. Technical Report 96-09. University of Massachusetts, Amherst, MA. Jan 20, 1996.

[10] Tom E. Fawcett. Feature discovery for problem solving systems. CMPSCI Technical Report 93-49. University of Massachusetts, Amherst, MA. May 1993.

[11] Tom E. Fawcett, Paul E. Utgoff. Automatic feature generation for problem solving systems. COINS technical report 92-9, University of Massachusetts, Amherst, MA. Jan 1992.

[12] Ben Kröse, Peter van der Smagt. An Introduction to Neural Networks.

[13] Michael Gherrity. A Game Learning Machine. Doctorate Thesis, University of California, San Diego. 1993.

[14] Jonathan Schaeffer and Robert Lake, Solving the Game of Checkers. University of Alberta.

[15] Robert Levinson, General Game-Playing and Reinforcement Learning. UCSC-CRL-95-06. University of California, Santa Cruz.

[16] Justin A. Boyan. Modular Neural Networks for Learning Context-Dependant Game Strategies. Masters Thesis, Cambridge University.

[17] David Kenneth Olson. Learning to Play Games from Experience: An Application of Artificial Neural Networks and Temporal Difference Learning. Dec, 93.

[18] David Stoutamire. Machine Learning, Game Playing and Go.

[19] Marco A. Weiring. TD Learning of Game Evaluation Functions with Hierarchical Neural Architectures. University of Amsterdam, April 95.

[20] Peter Sommerlund, Artificial Neural Networks applied to strategic games. May 96.

[21] David Carmel and Shaul Markovitch. Learning Models of Opponents Strategy in Game Playing. CIS Report #9318. June 93. Technion, Israel Institute of Technology.

[22] Barney Pell, A Strategic Metagame Player for General Chess-like Games. RIACS/NASA Research Centre.

12. Bibliography

George Thimm, Perry Moerland and Emile Fiesler, Interchangeability of Learning Rate and Gain in Backpropagation Neural Networks. Neural Computation 8(2), Feb. 1996.

David L. Elliot, A Better Activation Function for ANN. Technical Report TR-93-8. Institute for Systems Research.

Christopher D. Rosin and Richard K. Belew. Methods for Competitive Co-Evolution : Finding Opponents Worth Beating. ICGA 95.

Barry L. Kalman and Stan C. Kwansy. TrainRec: A System for Training Feedforward and Simple Recurrent Networks efficiently and correctly.

Charles L. Isbell, Explorations of Practical Issues of Learning Prediction Control Tasks Using Temporal Difference Learning Methods. Masters Thesis, MIT, Dec 92.

Christopher D. Rosin and Richard K. Belew. A Competetive Approach to Game Learning.

Mary Jo Ratterman and Susan L. Epstein. Skilled Like a Person, A Comparison of Human and Computer Game-Playing.

Jean-Christophe Weill, How Hard is the Correct Coding of an Easy Endgame. University of Paris.

Jonathan Schaeffer, Robert Lake, Paul Lu and Martin Bryant. Chinook - The World Man-Machine Checkers Champion.

Marco Weiring and Jürgen Schmidhuber. HQ-Learning: Discovering Markovian SubGoals for Non-Markovian Reinforcement Learning. Technical Report IDSIA-95-96.

13. Appendix A - List of Features

The Features were designed to work in binary groups. Because of the nature of ANN's it was necessary for each feature to be represented by a set of binary inputs instead of one scalar input. To achieve this features are organised into groups. The first in a group calculates the actual scalar value of the feature and sets the first bit or the bits allocated, whereas those following simply set successive bits. Most features have 3 inputs allocated, allowing them to represent a range of 0 to 7 but some have 4, allowing 0 to 15.
The following list owes a heavy debt to A.L. Samuels who provided a similar list in his paper [4]. As well as providing the features, Samuels also provided charts detailing how important certain features proved to be. Armed with that information I was able to significantly prune the feature set to the following.

PieceAdvantage 4 inputs allocated
Pieces were given the standard values of 2 for a man and 3 for a king. This feature calculated how much of an advantage it had over it's opponent.

PieceDisadvantage 4 inputs allocated
This feature is the opposite of PieceAdvantage

PieceThreat 3 inputs allocated
This calculated how many pieces where under threat from the opponent.

PieceTake 3 inputs allocated
This calculated how many of the opponents pieces are under threat from us.

BackRowBridge 1 input allocated
This is set on if the bridge is in place. i.e. squares 1 and 3.

CentreControl 3 inputs allocated
This is credited with 1for each piece we have in the centre squares.

XCentreControl 3 inputs allocated
This is credited with 1 for each piece the opponent has in the centre squares or could move to in his next move.

TotalMobility 4 inputs allocated

This is credited with 1 for each square to which the opponent can move.

Exposure 3 inputs allocated

This is credited with 1 for each piece that is flanked on each side of either diagonal by empty squares.

Advancement 3 inputs allocated

This is credited with 1 for each piece in the 5^{th} and 6^{th} rows and debited with one for each piece in the 3^{rd} and 4^{th} rows.

DoubleDiagonal 4 inputs allocated

This is credited with 1 for each piece on a double diagonal square.

KingCentreControl 3 inputs allocated

This is credited with 1 for each king on a centre square.

14. Appendix B - Object Descriptions

14.1 EvalNet - The TD Evaluation Neural Network Class.

This class contains all the code to implement a neural network trained by TD and momentum. It could be used for any application which requires a neural network evaluation function. This has been proven by it's use, entirely unmodified, in solving the Grid World problem. It's only limitation is that it has been designed *only* for use in evaluation functions, as it is fixed with a single output.

Constructor 1

EvalNet(numinputs, numhidden, dlinks, seed, lowrange, highrange, bias, name)

This constructor allows you to specify the number of inputs and hidden units in the network. It also allows the specification of whether or not direct links between input and output units should be used. The low and high range values are the limits within which the weights will be initialised. The bias variable allows you to specify the value of the bias unit for both layers. The name variable allow you to give the net a name to distinguish it from other nets. The name's purpose is twofold. It is useful in debugging to keep track of what's being trained and serves as a tag on saved networks.

Constructor 2

EvalNet(loadfile, bias)

This constructor loads all the details of a network from a file, except the bias.

Evaluating the Network

EvaluateNet(inputvector)

To evaluate a network is simple. An input vector, which MUST match the dimensions of the network created, it passed to it. The network is then evaluated in the same fashion as an ordinary neural network.

Initialising TD training

InitTDTrain(iv, gamma, lambda)

Before TD training can begin a several things must be initialised. These include momentum variables and the partial derivative (or eligibility trace). It also requires the starting input vector (iv) as well as the gamma and lambda values that will be used until the sequence terminates.

Training the intermediate boards

TDTrain(iv)

At each time step between the start and finish, TDTrain is called with the intermediate input vector. After calculating the error between the previous and current output, it adjusts the weights accordingly. Once it has done this is recalculates the output of the network as well as the eligibility's for the next run.

Final Reward for TD Training

TDFinal(evalue)

When the final outcome is reached a reward is given to the network.

Saving the Network

SaveNet(savefile)

This routine saves the network for later use. The name, dimensions and weights are all that is saved. Bias, Learning Rates, gamma, lamdba etc are all NOT saved.

Loading weights from a File

LoadWeights(loadfile)

This routine loads the networks weights from a file. Care must be taken that the dimensions match as otherwise results are unpredictable. It's intended use is in the cloning process, where nets swap weights regularly.

14.2 DirectNet - The TD Eval Network Class with Draughts Specifics.

This code is designed to further abstract the EvalNet class. It basically mirrors all the functions of EvalNet except that is accepts Boards instead of array's of floats. It then does one of three provided mapping before passing it on to EvalNet as an array of floats.

Constructor 1

DirectNet(typeofnet, nhidden, direct_links, seed, lowrange, highrange, bias, name)

The main difference between this constructor and that of EvalNet is that instead of specifying a number of inputs, you specify the type of network. Three types are catered for:

- NET_BINARYMAP - Each board square is represented by 3 inputs. Each set either to ON or OFF depending on what piece occupies it. The code's where as follows: 000 = Empty, 001 = BlackMan, 010 = RedMan, 011 = BlackKing, 100 = RedKing.
- NET_DIRECTMAP - Each board square is represented by a single input which is set to 0 for empty, 0.25 for black man, 0.5 for red man, 0.75 for black king and 1 for red king.
- NET_FEATUREMAP - The board is now represented by a given number of features. A full description of all features can be found in Appendix A. The number specified indicates the number of features to use.

Constructor 2

DirectNet(fname, bias)

This provides the exact same functionality as the EvalNet 2nd constructor.

TD Training Functions.

InitTDTrain(board, gamma, lambda, swapboard)
TDTrain(board, swapboard)
TDFinal(evalue)

These functions call their conterparts in EvalNet. The only difference is that they accept Boards as parameters, and before they call the EvalNet functions they perform the required mapping. They also have an extra parameter, swapboard, which is used to signify a red board which must be swapped before being evaluated.

Evaluating a Board

EvaluateNet(board)

As with the training functions, evaluating a board simply requires that the board be mapped to an input vector first. Then it calls the EvalNet function.

Mapping the board to the Input Vector

The following two private functions perform the bulk of DirectNet's work.

ComputeIV(pboard)
mapfeatures(iv, board)

Firstly ComputeIV creates the appropriate size float array and then passes it along with the board to mapfeatures, which performs the mapping from board to float array.

14.3 DraughtsGame - Draughts Game Managing Class

This class forms the basis of the training process.

Constructor 1
DraughtsGame()

No parameters are required to initialise a DraughtsGame. It simply acts as a convenient tournament hall. It keeps track of the current game as well as games played, which networks are being trained, who's black, who's red, how many moves have been made, average moves made in a particular sequence as well as average pieces left in a sequence.

Play a Tournament
AutoPlay(fname, train, numofgames, board, gamma, lambda, opp1, opp2)

This function allows one to specify the number of games to play, the initial board configuration, which networks to train, where to store the games being played as well as, most importantly, the network to be associated with each opponent. This function forms the core of the training regime.

Make a Move
MakeBlackMove(void)
MakeRedMove(void)

Using the networks provided by AutoPlay these functions call the function SuggestMove (in player.cpp) to decide which move to make. After making the suggested move these function also handle the training (if any is be performed) of the networks.

Initialising the Game
InitGame(char* fname, BOARD* board, int train)

This function set's up the file for recording, saving any preliminary information already available such as the names of players etc. It also initialises the board as well as setting up variable for training.

Cleaning Up after a Game

EndGame(whowon)

This function administers the reward to those nets being trained as well as recording the information in the PDN file.

14.4 Miscellaneous Headers.

Several other headers exist in NeuroDraughts. All of them are described in their appropriate header file. They are:

Board.cpp - Contains definition of Board and XBoard structs as well those functions relating to them.

Features.cpp - Contains the code for calculating all the various features.

Player.cpp - Contains the code for SuggestMove which implements a basic Minimax search routine.

Train.cpp - Contains the code for converting PDN files to training boards for NeuroDraughts as well as routines to implement training from this data.

Appendix C - Internal Board Representation

This is taken exactly from Sameuls's paper. It differs from standard 1-32 notation in that it skips numbers 9,18 and 27. It allows for very easy move/exchange calculation as any square to the north east is +4, south east is -5, north west +5 and south west -4. This is called an XBOARD in the program.

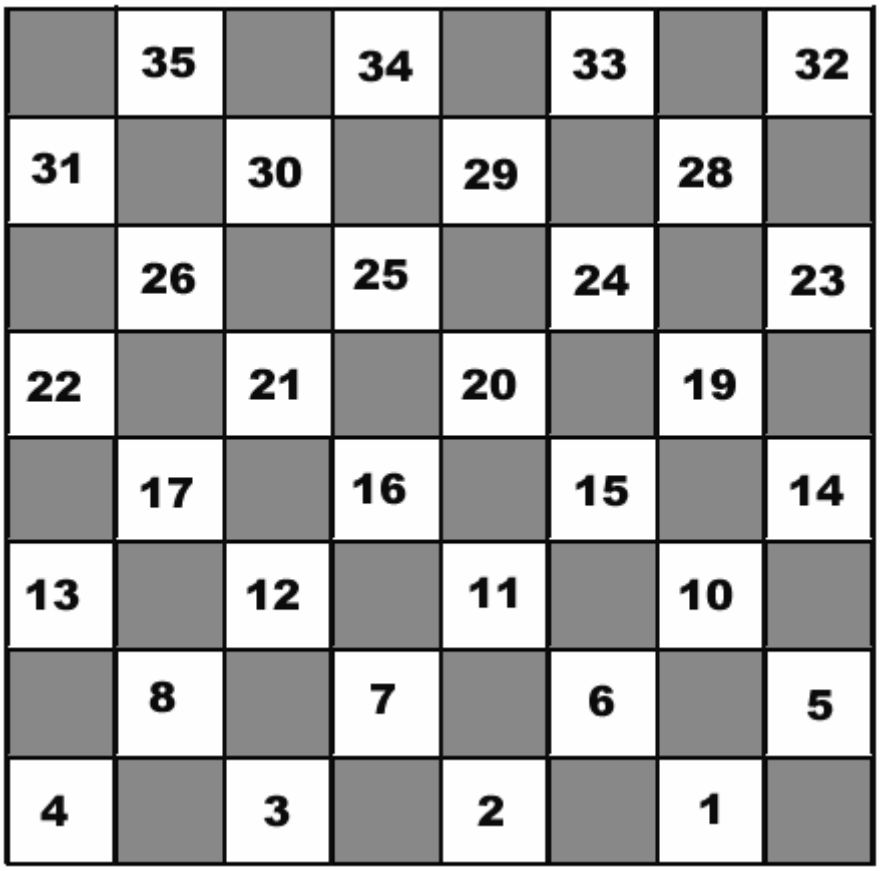

Figure 10 - Internal Board Representation