Learning to Play Othello with N-Tuple Systems

Simon M. Lucas
Department of Computer Science
University of Essex, Colchester, UK
sml@essex.ac.uk

Abstract

This paper investigates the use of n-tuple systems as position value functions for the game of Othello. The architecture is described, and then evaluated for use with temporal difference learning. Performance is compared with previously developed weighted piece counters and multi-layer perceptrons. The n-tuple system is able to defeat the best performing of these after just five hundred games of self-play learning. The conclusion is that n-tuple networks learn faster and better than the other more conventional approaches.

Keywords: Othello, n-tuple network, temporal difference learning.

1 Introduction

Games provide an ideal test-bed for the study of artificial intelligence. Early pioneers of computing and information theory such as Alan Turing and Claude Shannon were intrigued by the idea that computers might one day play grand-master level chess, and sketched out ideas of how this might be achieved. Computers know play at super-human levels on many complex games. Remarkably, checkers is now a solved game [23], the result being a draw if both players play optimally. Chess is far from being solved, but computers play at world-leading level.

The immediate goal of the research described in this paper is not to produce world leading AI players. Rather, it is to study the effectiveness of machine learning approaches to game playing: how well a machine can learn to play, rather than how well we can program it to play. In particular we are interested in how well the system can learn to play without any expert tuition, and without recourse to an expert opponent to practice against. The two main ways to achieve this are with temporal difference learning (TDL), and with co-evolution. For board games such as Othello, these techniques usually work by learning a value function that operates within a game-tree search algorithm.

Temporal difference learning (TDL) was applied by Samuel as far back as 1957 [22] and Michie in 1961 [18]. A famously successful application of TDL was Tesauro's TD Gammon [25], which was followed up by an evolutionary approach to the same problem by Pollack and Blair. In recent years there has been a surge of interest in evolutionary approaches to this type of learning. Much of this was probably inspired by the work of Pollack and Blair [19], and Chellapilla and Fogel [4] [5] [7].

All the systems under test in this paper play at one-ply. This puts the emphasis entirely on the quality of the learning, not on the details of the game-tree search, and provides the most efficient way to compare a set of learners. The learner aims to learn a good position value function that when combined with a one-ply search algorithm will encode a strategy for playing the game. Limiting to one-ply does overlook the computational cost of the method, which might be viewed as an oversight, since higher computational cost would lead to more limited game-tree search if these learners were to be used for real. It does however provide an interesting challenge.

The most popular methods for approximating value functions in games are linear functions (perceptrons), multi-layer perceptrons (MLPs), and spatially arranged MLPs as used in Blondie [5]. This paper describes a new approach to value function learning based on n-tuple systems. While n-tuple systems date back to the late 1950s, their use for learning game strategies is novel, and was recently introduced by the author [16]. This work is still in its initial stages, but has already proved to be remarkably successful. An n-tuple network trained with a few hundred of self-play games was able to significantly outperform the CEC 2006 champion.

The rest of this paper is structured as follows. Section 2 describes the game of Othello, and the randomised version of the game used in this paper. Section 3 gives a brief overview of n-tuple systems, and describes how n-tuples are used as position evaluators for Othello. Section 4 explains how temporal difference learning can be used to train n-tuple networks. Section 5 reports the results and section 6 concludes.

2 Othello

This section gives a brief description of the nature of the game, and then summarises previous approaches to learning to play it. Othello is a challenging unsolved game, where the best computer players already exceed human ability. Othello is played on an 8x8 board between two players, black and white (black moves first). At each turn, a counter must be placed on the board if there are any legal places to play, else the player passes. At each move, the player must place a counter on an empty board square to 'pincer' one or more opponent counters on a continuous line between the new counter and an old counter. All opponent counters that are pincered in this way are flipped over to the color of the current player. The initial board has four counters (two of each color) with black to play first. This is shown in figure 1, with the open circles representing the possible places that black can play (under symmetry, all opening moves for Black are identical). The game terminates when there are no legal moves available for either player, which happens when the board is full (after 60 non-passing moves, since the opening board already has four counters on it), or when neither player can play. The winner is the player with the most pieces of their color at the end of the game.

Counters placed in one of the four corners can never be flipped and therefore play a vital role in the game. Placing a high value on the corners tends to be the first thing learned, a fact that can be seen easily by inspecting the evolution of weight values in a weighted piece counter (WPC). Indeed the WPC [27] used as a benchmark in that study also reflects this. The highest value of 1 is given to all four corners. To hinder the possibility of an opponent getting a corner, the squares next to them should be avoided. For this reason they are given the lowest value -0.25. As a consequence the WPC encourages the players to place its counter at advantageous squares. The total set of

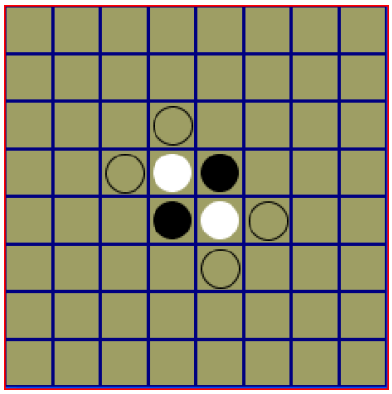

Figure 1: The opening board for Othello.

weights for this heuristic player is given in Figure 2, and depicted in Figure 3. These weights are symmetric under reflection and rotation, and have just 10 distinct values out of a possible 64. Experiments by the author (but not presented in this paper) show that enforcing this symmetry increases the learning speed. High levels of play can be learned more quickly when symmetry is enforced. However, Lucas and Runarsson did not enforce symmetry, and were able to learn a weighted piece counter that outperformed the standard symmetric weights given below, but only after a large number of games. Symmetry is exploited by the n-tuple system described in this paper, and does seem to enable very rapid learning.

```
 1.00 -0.25 0.10 0.05 0.05 0.10 -0.25  1.00
-0.25 -0.25 0.01 0.01 0.01 0.01 -0.25 -0.25
 0.10  0.01 0.05 0.02 0.02 0.05  0.01  0.10
 0.05  0.01 0.02 0.01 0.01 0.02  0.01  0.05
 0.05  0.01 0.02 0.01 0.01 0.02  0.01  0.05
 0.10  0.01 0.05 0.02 0.02 0.05  0.01  0.10
-0.25 -0.25 0.01 0.01 0.01 0.01 -0.25 -0.25
 1.00 -0.25 0.10 0.05 0.05 0.10 -0.25  1.00
```

Figure 2: The weights ($\boldsymbol{w}$) for the heuristic player [27].

As play proceeds, the piece difference tends to oscillate wildly, and some strategies

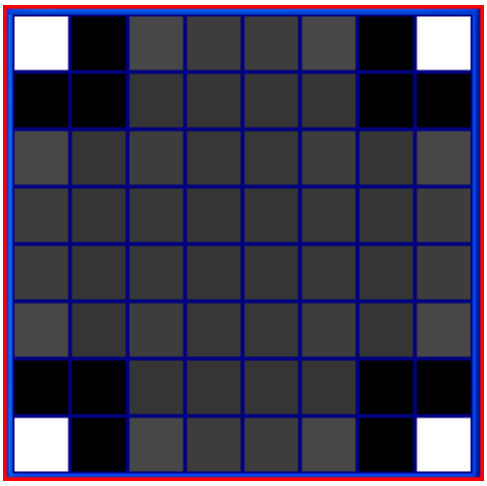

Figure 3: The standard heuristic weights, with lighter shades corresponding to more positive numbers.

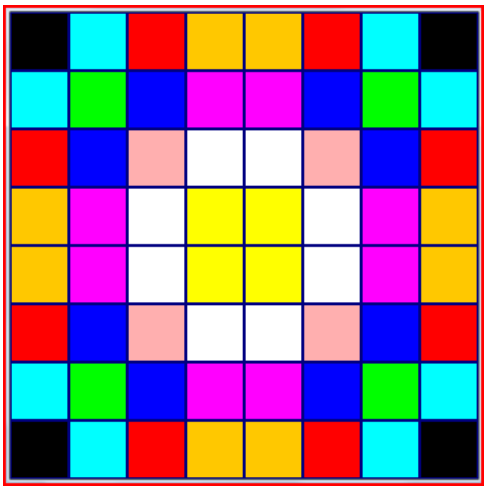

Figure 4: The Othello board, shaded to show squares that are equivalent under reflection and / or rotation.

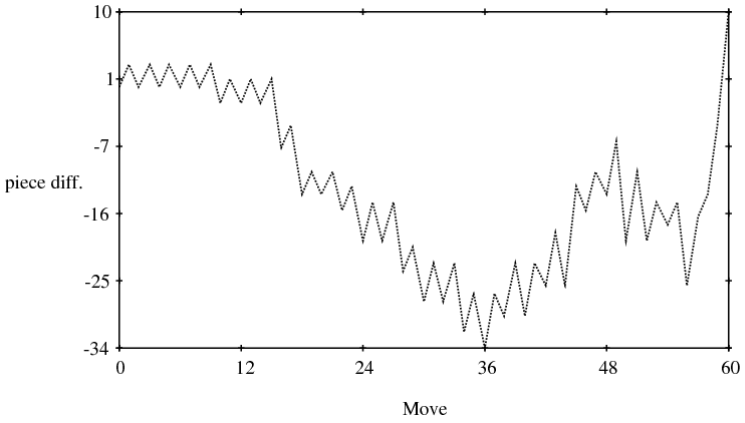

Figure 5: Typical volalite trajectory of piece difference during the game of Othello.

aim to have few counters during the middle stages of the game to limit possible opponent moves. Figure 5 shows how piece difference can change during the course of a game. This shows the player based on the heuristic weights shown above versus a pure random player. This piece difference trajectory is fairly typical of a match between these two players.

2.1 AI Othello Players

The first strong learning Othello program developed was Bill [10, 11]. Later, the first program to beat a human champion was Logistello [3], the best Othello program from 1993–1997. Logistello also uses a linear weighted evaluation function but with more complex features than just the plain board. The weights were initially estimated from a large database of games, and then tuned automatically using self-play. Logistello also uses an opening book based on over $23,000$ tournament games and fast game tree search [2].

More recently, Chong *et al* [6] co-evolved a spatially aware multi-layer perceptron (MLP) for playing Othello. Their MLP was similar to the one used by Fogel and Chellapilla for playing checkers [5], and had a dedicated input unit for every possible sub-square of the board. Together with the hidden layers this led to a network with

$5,900$ weights, which they evolved with around one hundred thousand games. The n-tuple systems described below are randomly constructed but typically have around 15,000 weights, yet can learn highly effective Othello strategy in a few hundred games of self-play.

2.2 Othello for Computational Intelligence Research

Due to its extremely simple rules yet significant complexity and engaging gameplay, Othello makes an excellent benchmark for machine learning algorithms and trainable architectures. Most trainable architectures used in game strategy learning go through two phases: learning, and then testing. In the learning phase an algorithm is used to adjust the parameters of the architecture, which are then fixed during testing against other players. This is quite unlike human competition play, where players learn from their mistakes during a series of games against an opponent, and in particular, they will build some form of opponent model in order to optimise their play. A human player will try to avoid losing in the same way twice against the same opponent. While it is certainly possible to use on-line learning with neural networks, most previous research uses the two-phase approach of separating learning from testing.

The upshot of this is that when playing two trained (but then fixed) function approximators against each other in a perfect knowledge noise-free game such as Othello, there are only two possible outcomes, depending on which player moves first. This might give a poor estimate of the true relative ability of two players; the weaker player might just happen to beat the stronger player on both occasions.

To overcome this problem a simple modification to any such noise-free game is to force random moves with a given probability. This is the methodology adopted by Runarsson and Lucas [21] and Lucas and Runarsson [12]. This can also be used for evaluation of weak players against very strong players, where the stronger player can be handicapped by the occasional forced random move.

For this paper all position value functions have been evaluated at one-ply. Each value function under test is used as follows. The computer player expands the current board to all possible next boards, by making all possible legal single moves. If this set

is empty, then the player passes. If it is not empty, then the value function is applied to each next board, and the move is made that leads to the board with the highest value.

One-ply players are at a significant disadvantage against players searching to greater ply, but providing all functions play at one-ply, then it is a level playing field. Furthermore, one-ply is especially easy to implement and fast to compute. When conducting comparisons with value functions developed by other researchers it also makes matters simpler. For high-ply minimax search it is harder to make direct comparisons specifically on the performance of the value function as there are many details of the minimax search (alpha-beta pruning, variable depth search etc.) which greatly affect the standard of play.

The author has been running an Othello neural network web server for the past two years. During that time, well over one thousand neural networks have been uploaded to the site. When a network is uploaded, it is played against the standard heuristic weighted piece counter for many games (initially 1,000, but this has been reduced to 100 to reduce load), and this gives it a ranking in the trial league. Then, for particular competition events, entrants are allowed to nominate two of their best networks to participate in a round-robin league.

The best network found in this way so far was an MLP. Co-evolution finds it hard to learn MLPs for this task, and for a long time the best network was an MLP trained by Runarsson[1] using TDL. For the 2006 IEEE CEC Othello competition, however, a new champion was developed by Kyung-Joon Kim and Sung-Bae Cho. They seeded a population with small random variations of the previous best MLP, and then ran co-evolution for 100 generations. This was able to produce a champion that performed in the round-robin league significantly better than the other players, and than the TDL-trained MLP that it was developed from. This points toward the value of TDL / Evolution hybrids.

[1]The weights for which are available here: http://algoval.essex.ac.uk:8080/othello/html/Othello.html.

3 N-Tuple Architectures

N-Tuple networks date back to the late 1950s with the optical character recognition work of Bledsoe and Browning [1]. More detailed treatments of standard n-tuple systems can be found in [26] and [20]. They work by randomly sampling input space with set of n points. If each sample point has m possible values, then the sample point can be interpreted as an n digit number in base m, and used as an index into an array of weights. The n-tuple works in a way somewhat similar to the kernel trick used in support vector machines (SVM)s, and is also related to Kanerva's sparse distributed memory model [9]. The low dimensional board is projected into a high dimensional sample space by the n-tuple indexing process. There are many varieties of n-tuple systems. Original n-tuple systems were often implemented in hardware, since the indexed look-up process is easy to implement using RAM chips. The very simplest of these used a 1-bit wide memory configuration, also known as binary n-tuples. Each memory location in a binary n-tuple records whether an address has occurred during training or not. Such systems suffer the risk of *saturation*, where excess training can make test-set performance worse, since given noisy training data, all addresses will eventually occur. For this reason, modern n-tuple systems tend to store continuous value weights, or probabilities. When trained on supervised data, probabilistic n-tuple systems can be trained using single-pass maximum likelihood techniques, where the probability of each address occurring is estimated as the number of times it occurred during training, divided by the number of occurrences of all addresses in the n-tuple.

While the basic idea of n-tuple systems is wonderfully simple, getting high performance from them in practice may involve significant design effort. Examples of this include the continuous n-tuple used for face recognition [13], the scanning n-tuple used for sequence recognition [17], and the scanning n-tuple grid used for OCR [15]. Interesting results have also been achieved with bit-plane decomposition methods [8].

More recently Lucas [14] introduced a back-propagation training rule based on optimising a cross-entropy measure. The same back-propagation update rule is used in this paper, though the error criterion is based on minimising the mean-squared error, with the target values being set according to the temporal difference training rule.

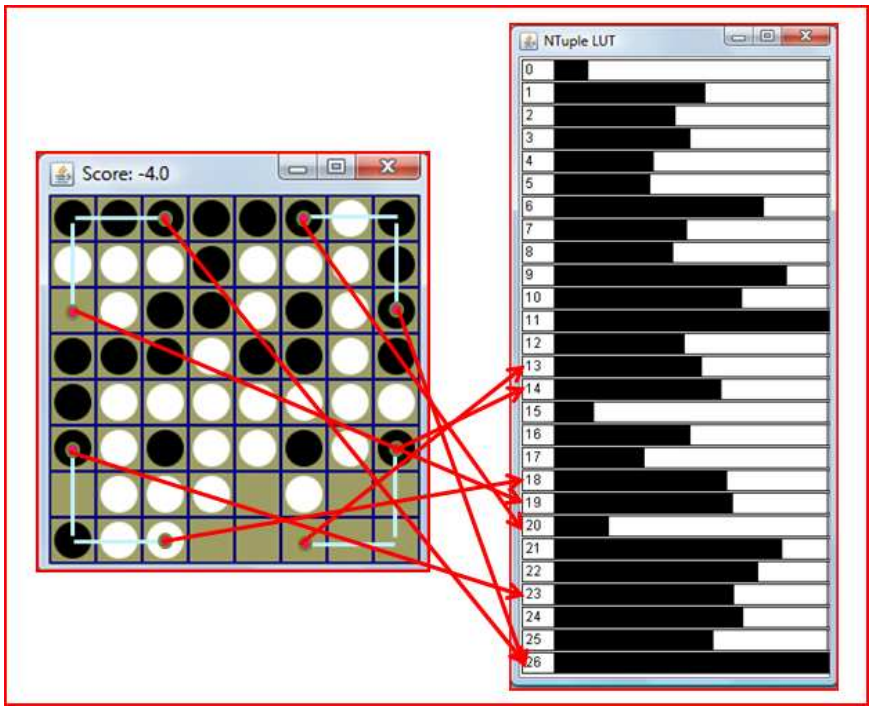

Figure 6: The system architecture of the N-Tuple-based value function, showing a single 3-tuple sampling at its eight equivalent positions (equivalent under reflection and rotation).

3.1 Application to Othello

To apply an n-tuple system to Othello, we first introduced symmetric sampling. Each square on an Othello board belongs to a group of either 4 or 8 squares that are all equivalent under reflection and / or rotation, as was illustrated in Figure 4.

The value function for a board is then calculated by summing over all table values indexed by all the n-tuples.

Figure 6 illustrates the system architecture but shows only a single n-Tuple. Each n-Tuple specifies a set of n board locations, but samples them under all equivalent reflections and rotations. The Figure shows a single 3-tuple, sampling 3 squares along an edge into the corner.

Each n-tuple has an associated look-up table (LUT). The output for each n-tuple

is calculated by summing the LUT values indexed by each of its equivalent sample positions (eight in the example). Each sample position is simply interpreted as an n digit ternary (base three) number, since each square has three possible values (white, vacant, or black). The board digit values were chosen as (white=0, vacant=1, black=2). By inspecting the board in the Figure, it can be seen that each n-tuple sample point indexes the look-up table value pointed to by the arrow. These table values are shown after several hundred self-play games of training using TDL. The larger the black bar for a LUT entry, the more positive the value (the actual range for this figure was between about $+/-0.04$. Some of these tables entries have obvious interpretations. Good for black means more positive, good for white means more negative. The LUT entry for index zero corresponds to all sampled squares being white: this is the most negative value in the table. The LUT entry for index twenty six corresponds to all sampled squares being black: this is the most positive value in the table.

The value of a board $v(b)$ based on a single n-tuple is defined in the following equation, where b is the board, d is a sampled n digit number in the set $D(b)$ of symmetric samples given the n-tuple, and l is the indexed vector of values in the LUT.

$$v(b) = \sum_{d \in D(b)} l[d] \tag{1}$$

The value function for a board is simply the sum of the values for each n-tuple. For convenient training with error back-propagation the total output is put through a $tanh$ function.

3.2 Choosing the Sample Points

The n positions can be arranged in a straight line, in a rectangle, or as random points scattered over the board. The results in this paper are based on random snakes: shapes constructed from random walks. Each n-tuple is constructed by choosing a random square on the board, and taking a random walk from that point. At each step of the walk, the next square is chosen as one of the eight immediate neighbours of the current square. Each walk was for six steps, but only distinct squares are retained. So each randomly constructed n-tuple had between 2 and 6 sample points. The results in this

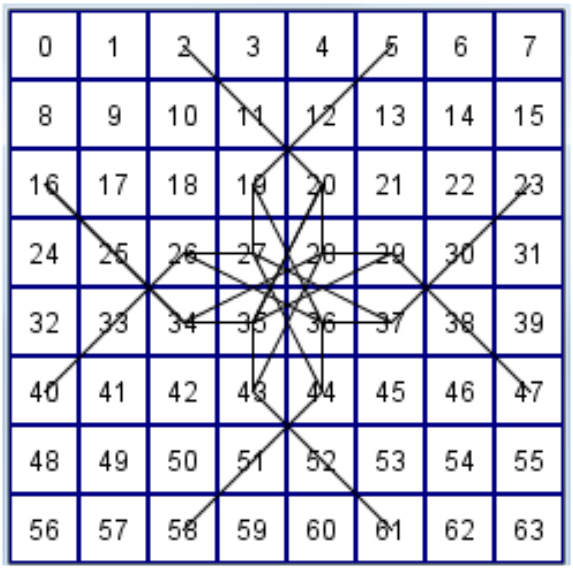

Figure 7: A randomly constructed n-tuple's sample points, together with its expansions.

paper are based on 30 such n-tuples. One would expect some n-tuples to be more useful than others, and there should be scope for evolving the n-tuples sample points while training the look-up table values using TDL. Each randomly constructed n-tuple is automatically expanded to place it's sample points at all symmetrically equivalent positions on the board. This must be done in a way that maintains the same relationship between all the sample points, and is illustrated for the case of a single 3-tuple in Figure 6.

A randomly constructed n-tuple sample is shown in Figure 7, together with all its expansions. The original sample points are: $\{2, 11, 20, 28, 34, 35\}$. Note that the order of the points makes no difference, providing that the same relative ordering is used for all the symmetric expansions also.

4 Learning Value Functions

As explained above, a value function is used to dictate game strategy.

Both Temporal Difference Learning (TDL) and Co-Evolutionary Learning (CEL) are able to acquire game strategies without reference to any expert knowledge of game strategy, and without using any prior available player to train against. Typically, CEL achieves this by generating an initial random population of strategies which are then played against each other, with the parents for each successive generation being chosen on the basis of their playing ability. Standard TDL achieves this through self-play.

The main difference between the two methods (at least in their most typical forms) is that CEL uses only the end information of win/lose/draw aggregated over a set of games, whereas TDL aims to exploit all the information during the course of a game, as well as at the end of each game when the final rewards are known.

Runarsson and Lucas investigated temporal difference learning versus co-evolution for learning small-board Go strategies [21], and for Othello strategies [12]. In both cases they found that TDL learned faster, but that with careful tuning, CEL eventually learned better strategies. In particular, with CEL it was necessary to use parent-offspring weighted averaging in order to cope with the effects of noise. For this paper, only TDL results are reported. Initial experiments with CEL were less successful, though that could be due to an insufficient number of games being played. A thorough comparison of TDL with CEL, and with possible hybrids is an obvious candidate for future work.

In TDL the weights of the evaluation function are updated during game play using a gradient-descent method. Let $\boldsymbol{x}$ be the board observed by a player about to move, and similarly $\boldsymbol{x}'$ the board after the player has moved. Then the evaluation function may be updated during play as follows. This is based on Sutton and Barto [24, p.199], and the formulation of it in Equation 2 is taken directly from Lucas and Runarsson [12].

At each turn of the game, the TDL player either makes an in-game or a terminal (end-game) update. In the case of an in-game update, the value of the previous board position is adjusted to be more similar to the value of the current board position. This is a type of bootstrapping process. For a terminal update, the value of the penultimate board is adjusted to be closer to the final value of that game ($r = +1$ for black win, $r = 0$ for draw, $r = -1$ for white win).

$$
\begin{aligned}
w_i &\leftarrow w_i + \alpha\big[v(\boldsymbol{x}') - v(\boldsymbol{x})\big]\frac{\partial v(\boldsymbol{x})}{\partial w_i} \\
&= w_i + \alpha\big[v(\boldsymbol{x}') - v(\boldsymbol{x})\big]\big(1 - v(\boldsymbol{x})^2\big)x_i
\end{aligned} \tag{2}
$$

where

$$
v(\boldsymbol{x}) = tanh(f(\boldsymbol{x})) = \frac{2}{1 + \exp(-2f(\boldsymbol{x}))} - 1 \tag{3}
$$

is used to force the value function v to be in the range -1 to 1. This method is known as gradient-descent TD(0) [24]. If $\boldsymbol{x}'$ is a terminal state then the game has ended and the following update is used:

$$
w_i \leftarrow w_i + \alpha\big[r - v(\boldsymbol{x})\big]\big(1 - v(\boldsymbol{x})^2\big)x_i
$$

where r corresponds to the final utilities: $+1$ if the winner is Black, -1 when White, and 0 for a draw.

Given the explanation above for how the value function is calculated, the LUT l entries can be seen as the weights of a single layer perceptron. The indexing operation performs a non-linear mapping to high-dimensional feature space, but that mapping is fixed for any particular choice of n-tuples. Since a linear function is being learned, there are no local optima to contend with.

The first is how it is interfaced to the Othello game. The game engine calls a TDL update method for any TDL player after each move has been made: it calls inGameUpdate during a game, or terminalUpdate at the end of a game.

It is instructive to study the Java code that implements this process as shown in Figure 8. The variables are as follows: `op` is the output of the network; `tg` is the target value; `alpha` is the learning rate (set to 0.001); `delta` is the back error term; `prev` is the previous state of the board; `next` is the current state of the board; `net` is an instance variable bound to some neural network type of architecture (an n-tuple system in this case).

The n-tuple system implements the `Net` interface, and an instance of one is bound to the `net` instance variable in the code. The forward method calculates the output of the network given a board as input. The `updateWeight` method propagates an error

```
public void inGameUpdate(double[] prev, double[] next) {
    double op = tanh(net.forward(prev));
    double tg = tanh(net.forward(next));
    double delta = alpha * (tg - op) * (1 - op * op);
    net.updateWeights(prev, delta);
}

public void terminalUpdate(double[] prev, double tg) {
    double op = tanh(net.forward(prev));
    double delta = alpha * (tg - op) * (1 - op * op);
    net.updateWeights(prev, delta);
}
```

Figure 8: The main two methods for TDL learning in Othello.

term, and makes updates based on this in conjunction with the board input. For the n-tuple system the update method is very simple. While the value function was calculated by summing over all LUT entries indexed by the current board state, the update rule simply adds the error term δ to all LUT entries indexed by the current board:

$$l(d) = l(d) + \delta \ \ \forall\, d \in D(b) \tag{4}$$

One of the best features of an n-tuple system is how it scales with size. Due to the constant-time indexing operation, it is independent of the size of the LUT. So, although the LUT size grows exponentially with respect to n, the speed remains almost constant, and linear in the number of n-tuples. Hence, n-tuple value functions with millions of weights can be calculated extremely quickly.

5 Results

Experiments were conducted to test the performance of n-tuple networks trained with TDL. Play performance was tested by playing against the standard heuristic weights.

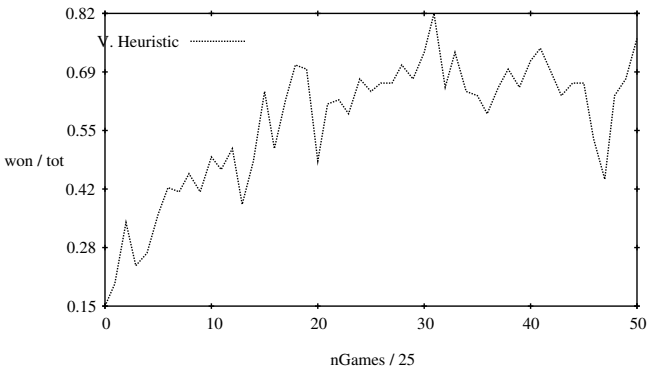

Figure 9: Variation in win ratio against the heuristic player (each sample point based on 100 games, 50 each as black and white).

Figure 9 (from [16]) shows how performance improves with the number of self-play games. After every 25 self play games, performance was measured by playing 100 games against the standard heuristic player (50 each as black and white).

Table 1 (from [16]) shows how performance against the CEC 2006 champion varies with the number of self play games, in this case playing 200 games against the champion (100 each as black and as white). After the first 500 self-play games have been played the Champion is defeated in nearly 70% of games.

Table 1: Performance of TDL N-Tuple Player versus CEC 2006 Champion over 200 games, sampled after varying number of self-play games n_{sp}.

n_{sp}	Won	Drawn	Lost
250	89	5	106
500	135	6	59
750	142	5	53
1000	136	2	62
1250	142	5	53

Not only has the n-tuple based player reached a higher level of performance than any player to date (under this one-ply, 10% forced random move evaluation scheme), it has also done so much more quickly. In order to gain some insight into how the

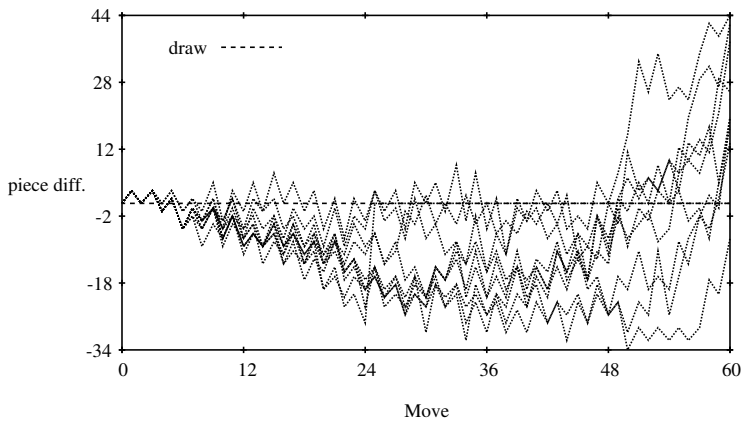

Figure 10: Plot of piece difference for a TD-trained n-tuple system versus the standard heuristic player on over 10 games with the n-tuple system playing as black (positive).

n-tuple system plays, some plots of piece difference versus move number were made, when the n-tuple system played the standard heuristic weights. The results are shown in Figure 10. On this sample, the n-tuple system usually has a worse piece difference during the middle of the game, and only during the final fifth of the game does it begin to dominate. On this test the n-tuple system wins nine games out of ten — this is shown by how many lines finish above the draw line. This particular n-tuple system had 14,772 weights in it.

6 Conclusions

The results show that N-Tuple architectures offer the best method yet for learning position value in the game of Othello. They can be trained very rapidly using temporal difference learning, and reach relatively high playing ability after just 500 games of self-play.

The results for Othello show that the N-Tuple networks very clearly out-perform weighted piece counters and MLPs, both of which have been the staple diet of computational intelligence researchers. It seems most likely that the results will carry over to other board games, and quite possibly to entirely different genres of game.

A likely reason for this is that n-tuple systems factorise well: the values learned in one element of the look-up table are largely independent from the values learned in other parts. However, the size of lookup table (and hence the number of parameters) for a non-trivial game may need to be made very large. This large search space makes for slow progress with evolutionary methods, but temporal difference learning is able to exploit more information, during the course of the game, *and* use features of the input space to directly adjust the weights in the table. In summary, the combination of temporal difference learning with n-tuple systems seems a very promising approach with which to tackle game learning.

References

[1] W. W. Bledsoe and I. Browning, "Pattern recognition and reading by machine," in *Proceedings of the Eastern Joint Computer Conference*, 1959, pp. 225–232.

[2] M. Buro, "ProbCut: An effective selective extension of the Aalpha-Beta algorithm," *ICCA Journal*, vol. 18, pp. 71 – 76, 1995.

[3] ——, "LOGISTELLO – a strong learning othello program," 1997, http://www.cs.ualberta.ca/~mburo/ps/log-overview.ps.gz.

[4] K. Chellapilla and D. Fogel, "Evolving neural networks to play checkers without expert knowledge," *IEEE Transactions on Neural Networks*, vol. 10, no. 6, pp. 1382–1391, 1999.

[5] ——, "Evolving an expert checkers playing program without using human expertise," *IEEE Transactions on Evolutionary Computation*, vol. 5, pp. 422 – 428, 2001.

[6] S. Y. Chong, M. K. Tan, and J. D. White, "Observing the evolution of neural networks learning to play the game of othello," *IEEE Transactions on Evolutionary Computation*, vol. 9, pp. 240 – 251, 2005.

[7] D. Fogel, *Blondie24: playing at the edge of AI*. Morgan Kaufmann Publishers Inc., 2002.

[8] S. Hoque, K. Sirlantzis, and M. C. Fairhurst, "Bit plane decomposition and the scanning n-tuple classifier," *Proceedings of International Workshop on Frontiers in Handwriting Recognition (IWFHR-8)*, pp. 207 – 212, 2002.

[9] P. Kanerva, *Sparse Distributed Memory*. Cambridge, Mass.: MIT Press, 1988.

[10] K.-F. Lee and S. Mahajan, "A pattern classification approach to evaluation function learning," *Artificial Intelligence*, vol. 36, pp. 1 – 25, 1988.

[11] ——, "The development of a world class othello program," *Artificial Intelligence*, vol. 43, pp. 21 – 36, 1990.

[12] S. M. Lucas and T. P. Runarsson, "Temporal difference learning versus co-evolution for acquiring othello position evaluation," in *IEEE Symposium on Computational Intelligence and Games*, 2006.

[13] S. Lucas, "The continuous n-tuple classifier and its application to real-time face recognition," *IEE Proceedings on Vision, Image and Signal Processing*, vol. 145, pp. 343 – 348, (1998).

[14] ——, "Discriminative training of the scanning n-tuple classifier," in *Lecture Notes in Computer Science (2686): Computational Methods in Neural Modelling*. Berlin: Springer-Verlag, (2003), pp. 222 – 229.

[15] ——, "Fast convolutional ocr with the scanning n-tuple grid," in *Proceedings of International Conference on Document Analysis and Recognition (ICDAR)*. IEEE Computer Society, 2005, p. to appear.

[16] ——, "Computational intelligence and games: Challenges and opportunities," *International Journal of Automation and Computing*, p. to appear, 2007.

[17] S. Lucas and A. Amiri, "Statistical syntactic methods for high performance OCR," *IEE Proceedings on Vision, Image and Signal Processing*, vol. 143, pp. 23 – 30, (1996).

[18] D. Michie, "Trial and error," in *In Science Survey, part 2*. Penguin, 1961, pp. 129–145.

[19] J. Pollack and A. Blair, “Co-evolution in the successful learning of backgammon strategy,” *Machine Learning*, vol. 32, pp. 225–240, 1998.

[20] R. Rohwer and M. Morciniec, “A theoretical and experimental account of n-tuple classifier performance,” *Neural Computation*, vol. 8, pp. 629 – 642, (1996).

[21] T. P. Runarsson and S. M. Lucas, “Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go,” *IEEE Transactions on Evolutionary Computation*, vol. 9, pp. 628 – 640, 2005.

[22] A. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of Research and Development*, vol. 3, pp. 211 – 229, 1959.

[23] J. Schaeffer, N. Burch, A. K. Yngvi Bjrnsson, M. Mueller, R. Lake, P. Lu, and S. Sutphen, “Checkers is solved,” *Science*, vol. 317, pp. 1518 – 1522, September 2007.

[24] R. Sutton and A. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

[25] G. Tesauro, “Temporal difference learning and TD-gammon,” *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[26] J. Ullman, “Experiments with the n-tuple method of pattern recognition,” *IEEE Transactions on Computers*, vol. 18, no. 12, pp. 1135–1137, December 1969.

[27] T. Yoshioka, S. Ishii, and M. Ito, “Strategy acquisition for the game ”othello” based on reinforcement learning,” in *IEICE Transactions on Information and Systems E82-D 12*, 1999, pp. 1618–1626.