Preference Learning for Move Prediction and Evaluation Function Approximation in Othello

Thomas Philip Runarsson, Senior Member, IEEE, Simon Lucas, Senior Member, IEEE

Abstract—This paper investigates the use of preference learning as an approach to move prediction and evaluation function approximation, using the game of Othello as a test domain. Using the same sets of features, we compare our approach with least squares temporal difference learning, direct classification, and with the Bradley-Terry model, fitted using minorization-maximization. The results show that the exact way in which preference learning is applied is critical to achieving high performance. Best results were obtained using a combination of board inversion and pair-wise preference learning. This combination significantly outperformed the others under test, both in terms of move prediction accuracy, and in the level of play achieved when using the learned evaluation function as a move selector during game play.

I. INTRODUCTION

Developing machine learning algorithms that can learn to play games to a high standard in a largely unsupervised manner has been a long-standing challenge for AI. More recently Monte Carlo Tree Search (MCTS) has been enormously successful on a number of challenging games, with Go being the pre-eminent example [1]. MCTS is also the leading approach in many other games such as Hex [2] and Havannah [3]. At first glance, the success of MCTS might lead one to expect that learning evaluation functions would become less important, but this is not the case, since nearly all leading MCTS programs rely heavily on knowledge to guide the search. In this case the evaluation function may be used to bias the Monte Carlo rollouts. Furthermore, for games such as chess, minimax search with alpha beta pruning still produces stronger players than MCTS, and the quality of minimax players is heavily dependent on the evaluation function.

As the strength of the evaluation function increases, so the amount of CPU-hungry tree-search to achieve the same standard of play decreases. This is significant for strategy game apps for mobile devices, where the CPU can place heavy demands on the battery when working at peak processing power.

For these reasons, learning good value functions is an important and interesting problem to study. There are three main approaches to learning evaluation functions:

- Use temporal difference learning (TDL) to learn through self-play (e.g. TD-Gammon [4]).
- Use co-evolution to evolve an evaluation function (e.g. Blondie-24 [5]).
- Use some form of supervised learning to learn from a set of game trajectories (e.g. Logistello [6]).

All of these methods have had some famous successes, as indicated in the citations above, and also have some important differences that make for interesting comparisons. TDL uses information available during game play in an attempt to solve a credit assignment problem, whereas co-evolution normally focuses only on the end result. Co-evolution utilises less of the available information [7], but its focus on the end result can lead to greater robustness than TDL.

Co-evolution may learn slowly but eventually achieve higher performance for simple value functions such as weighted piece counters (8) (9). When value functions with thousands of parameters are used, then the more directed search accomplished by TDL seems preferable due to its better use of the available information [7]. A combination of TDL and co-evolution can also work well [10] [11].

In this paper we investigate learning an evaluation function for Othello. Othello is interesting for several reasons, including the way that game states are highly volatile, and the way that piece difference during the middle of the game is very deceptive, with stronger positions often showing poor piece difference.

Value function learning for Othello was held as an IEEE CEC competition [12] for several years. The aim of this competition, and of the work in this paper, is to investigate machine learning in the context of a challenging game. Note that the aim is not to design world-class Othello playing programs, since much of this activity involves the use of opening and end-game databases together with high-performance tree-search algorithms: focus on this would detract from the machine learning aspects.

In the IEEE CEC Othello competitions all evaluation functions were played against each other using exactly the same tree search algorithm: a simple 1-ply minimax, but forced random moves were also introduced to ensure a varied set of outcomes. For the game playing performance evaluation in this paper, we use the same 1-ply search approach, but use a set of 1,000 unique opening positions instead of the forced random moves to ensure a thorough evaluation of playing ability.

Clearly, imitating human-like play is not necessarily optimal, though it may create players that are more interesting to play against. This would be especially true if it proved possible to imitate particular famous players rather than human-like

http://algoval.essex.ac.uk:8080/othello/html/Othello.html
play in general. The techniques discussed in this paper are not limited to game trajectories taken from human games, but could also be computer generated, for example using self-play and policy iteration or Monte Carlo Tree Search.

An initial investigation by the authors illustrated that preference learning significantly outperforms temporal difference learning when imitating game play \cite{12}. The key insight is that making the correct choice is what really matters, which is what preference learning focuses on. This is in contrast to temporal difference learning, which attempts to learn the expected reward (in this case the probability of winning) for each game state.

Results presented in this paper explore the idea in greater depth by analysing the differences observed when using a more sophisticated N-Tuple value function, and also when attempting to learn the policy directly using classification \cite{13}. We also compare performance with the leading technique for move prediction in Go, which involves using minimization maximization (MM) to fit a Bradley-Terry team model to the move selection data, as explained below. As before \cite{12} we use game trajectories taken from human competitions held by the French Othello Federation\footnote{www.fothello.org} as a source of data. In addition to the inclusion of N-Tuple players and a wider range of algorithms for comparison, this paper extends our previous work \cite{12} by evaluating the effects of board inversion as a method of preparing the board for presentation to the evaluation function. Using this technique in conjunction with preference learning we were able to produce one of the strongest 1-ply Othello players known.

The rest of this paper is structured as follows. The next section reviews the different approaches to learning evaluation functions. This is followed by a section discussing the Othello game trajectories used in this study, along with a description of the 1-Tuple and N-Tuple features used. This section (see \textsection III-C) also provides interesting insights into the various ways the move prediction problem can be presented to the learner. This is followed by a description and pseudo-code of the LSTD(\lambda) algorithm in section \textsection IV, direct classification in section \textsection V, and preference learning algorithms employed in section \textsection VI. The MM approach is explained in section \textsection VII, results of the comparison are presented in section \textsection VIII, and section IX concludes.

II. EVALUATION FUNCTION APPROXIMATION

The focus of this paper is on preference learning. Since this is a type of supervised learning, we first describe a number of ways in which supervised approaches have been used to learn evaluation functions, either from game logs or from MCTS players, then motivate our preference learning approach.

A. Regression

In the regression approach a function is learned which outputs a real number indicating the favourability of a board position for the maximising player (and conversely the opposite of this for the minimising player). As mentioned above, this approach was used successfully by Buro to estimate the weights for his Logistello program \cite{6}. When using regression the problem of how to label training examples arises. Buro’s pragmatic approach was to initially label each game position as a win or loss in accordance with the outcome of the game. Game positions were also expanded using game-tree search and the values of the positions were corrected when the terminal states were within the tree.

The supervised learning method has also been explored in the context of move prediction for Go, with much emphasis being placed on the use of Bayesian methods to rank the likelihood of each available move. Recent work on this includes the full Bayesian ranking method of Stern et al \cite{14}, and Coulom’s approach \cite{15} of using minimization-maximization (MM) \cite{16} to fit a Bradley-Terry model to the move selection data.

A recent study by Wistuba et al \cite{17} using the same features and game data for all methods under test, found Coulom’s approach to perform best, slightly outperforming full Bayesian ranking and beating the other methods by a larger margin. Due to the training algorithm used, Coulom’s approach will henceforth be referred to as the MM method and is described and tested in this paper.

TDL can also be used to learn an evaluation function from a set of game logs, where it is usually applied to minimize the Bellman-residuals \cite{18}. The most effective approaches employ least squares temporal difference learning, LSTD(\lambda) \cite{19}, \cite{20}. Apart from \lambda, the decay parameter, LSTD has no control parameters and therefore eliminates the problem of parameter choice leading to poor performance.

When using LSTD the evaluation function will have a linear form in feature space, though the features may be non-linear functions of the board state. An example of such non-linear features are the N-Tuples \cite{21} applied in this paper. Within the constraints of learning a linear function, LSTD aims to approximate the expected future payoff.

B. Classification

Recently it has been argued in \cite{22} that minimizing Bellman-residuals is unnecessarily complex, since predicting precise future payoffs is not necessary for making optimal moves. Furthermore they argue that, for the latter approach, the prediction of single moves neither suggests alternative actions nor offers any means for proper exploration \cite{22}. In \cite{23} it is argued that policies may be easier to represent than value functions.

This has motivated a number of researchers to model the policy directly as a classifier: instead of estimating the value of each game state, states are partitioned into selected and non-selected sets, i.e. they are given different class labels \cite{13}, \cite{24}, \cite{23}. These methods use Monte Carlo rollouts to estimate the value of alternative moves at a given board position. Then, if a move has a statistically greater value than all other moves, it is added to a training set with a positive label, while the rest are added to the training set with a negative label \cite{13}. Labelling moves as selected versus non-selected will also be investigated in this paper, a method we refer to as the direct classification approach. However, we derive the class labels
from the game logs and do not use Monte Carlo roll-outs to filter these.

C. Preference Learning

The classification approach described above has recently been put into a preference-based reinforcement learning framework \[25\]. The classifier is essentially replaced by a label ranker. Each possible move is ranked, where statistically equal moves, according to the rollouts made, have the same rank. Preference pairs can then be created between the different ranks. This scheme also makes better use of the information provided by the rollouts.

The classification and preference learning approaches essentially approximate a utility function, which assigns a utility degree to each move. The one with the highest degree corresponds to the move chosen, or in the case of preference learning, the highest ranked move. This function is different from the value function of TDL, which represents the expected future payoffs received, for example the probability of winning a game.

Preference learning has created much attention recently in the machine learning literature \[26\]. For games, the principles of preference learning have been applied in tuning heuristic evaluation functions in the work on maximizing concordance \[27\]. In \[28\] preference learning was used to model entertainment preferences of children when playing games. Pairwise preference learning has also been used to predict affective states in a 3D prey/predator game \[29\].

In our application of preference learning we are learning from game logs, where the only information supplied is the single move chosen for each state by the human player. Using an Othello game engine we can also generate all possible moves available from that game state, and hence generate the non-selected moves. Our preference learner is therefore limited to the version called Pairwise Approximate Policy Iteration presented in \[25\], though in that work each pair of vectors is given a preference label indicating which one is preferred.

When learning a linear function one approach that is commonly adopted in preference learning, and one we adopt here, is to form feature differentials, where the non-preferred feature vector is subtracted from the preferred feature vector. We call this differential preference learning.

In a two player game, such as Othello, both players may use the same utility function. One player will choose moves that maximize this function while the other chooses moves which minimize it. In this case one can label the feature differential as positive for the maximizing player and negative for the minimizing player. This way of creating training data is quite different from that of the direct classification approach and will result in a different evaluation function, even when the same linear architecture is chosen.

An alternative to minimizing the utility function would be to let each player learn its own separate utility function. A disadvantage of this approach is that it makes limited use of the available information, since it has to learn the same things separately for each player. On the plus side it means that it can also potentially learn subtle nuances, where black and white should genuinely follow different policies given very similar circumstances.

The direct classification methods are unable to use output negation and must therefore use board-inversion (or colour-reversal) in order to learn common policies for each player. Interestingly, the board inversion approach can also be applied in conjunction with differential preference learning and this approach leads to the best results of all methods under test. Details of how each approach is applied are given in Section \[II-C\].

III. Othello Game Trajectories

The game of Othello is played on an \(8 \times 8\) board, with a starting configuration of the middle 4 squares occupied by two white and two black discs. Black plays first and the game continues until the board is full (after 60 non-passing turns), or until neither player is able to move. Note that a player must move if able to, passing only happens when a player has no legal moves available.

Figure 1 shows a game in progress and the seven feasible moves for black. The best move is almost certainly “1a” since once a piece has been placed in a corner it can never be flipped, and in this case would remain black for the duration of the game. Each player’s objective is to maximise the number of disks of their own color at the end of the game. Othello, like many boardgames, fits the model of a two-player, turn taking, zero-sum game, where the utility values for each player at the end of the game are equal in magnitude and opposite in sign.

The strongest Othello program is Logistello.\[3\] The evaluation function used by Logistello also has an essentially linear architecture (but with a sigmoid squashing function at the output) based on 1.5 million pattern-based features using different evaluation functions at 13 different game stages, \(g_s = \max (0, (\#\text{discs} - 13)/4)\). The training data used by Logistello is based on some 80,000 games generated by an

\[3\]To win a single game it is only necessary to have more disks than one’s opponent, but winning margins can be important for player satisfaction and for tournament tie-breaks.

\[\text{http://skatgame.net/mburo/log.html}\]

Figure 1: Othello game in progress with seven possible legal moves for black (dashed circles). Capturing corners is one key strategy in playing Othello, so 1a is probably the best move.
earlier, less tuned version of the program playing against another Othello program (Kitty). Towards the end of the game the positions are labelled perfectly since an endgame negamax search is used. Values of middle and opening game positions are approximations based on the game outcomes that followed those positions.

Logistello then uses a gradient descent algorithm to estimate the model’s parameters. Logistello’s approach [30] corresponds more closely to the supervised learning approach, or LSTD(1), using linear regression to learn the value of positions labelled with the final disc differential estimate. This approach yields significantly better performance than Buro’s previous work [6] where the positions were labelled by the probability of winning. Clearly, labelling on the probability of winning or the outcome of the game is a more general approach, and valid for all board games.

In this work Othello game logs taken from the French Othello League are used to create game trajectories. A linear evaluation function (linear in feature space) is then used to approximate the expected outcome of the game in terms of a win or loss. When choosing a move a one-step-lookahead is performed. The resulting board (after- or post-decision-state) is evaluated and the move with the best corresponding evaluation is chosen. Two different sets of features applied by the linear function will now be described, followed by discussion of data preparation.

A. Weighted Piece Counters as 1-Tuples

The traditional form of weighted piece counter (WPC) for Othello is where an 8 x 8 board is unwound as a 64 element vector \( \phi \). Each element of \( \phi \) is 0 for an empty square, +1 for a black counter and -1 for a white counter. This WPC has a vector \( w \) of 64 weights, one for each square on the board. The evaluation of a board is then calculated as the scalar product \( w \cdot \phi \). Black will then select the move resulting in a board with the highest evaluation, while white would select the one with the lowest evaluation. Alternatively, the white player could reverse the colours on the board and maximize the evaluation function.

As an alternative form of a WPC, we give each square on the board three weights, one each for whether the square is empty, is occupied by black, or is occupied by white. We call the resulting 192-element feature vector \( \phi \), and note that each element is binary valued. Having binary valued features is a requirement for the MM algorithm (see below), so this form of WPC can be used directly by all algorithms without further transformation.

Interestingly, we found that this slightly outperformed the more traditional 64-weight WPC in its ability to predict expert moves and had similar performance in head-to-head matches given the training methods used in this paper. Given that we had some strong 64-weight WPCs readily available from previous research, these will be used to provide additional players for the round-robin evaluation. We will, however, use the 192-weight version for all the learning experiments in this paper. This type of WPC can be implemented as a form of N-Tuple network (see next section), where 64 1-Tuples cover the board.

B. N-Tuples

N-Tuple networks (also called N-Tuple systems) date back to the late 1950s with the optical character recognition work of Bledsoe and Browning [31]. A more detailed treatment of standard N-Tuple systems can be found in [32].

An N-Tuple network operates as an ensemble of simple classifiers. Each individual N-Tuple samples the input space at a set of \( n \) points. The points may be chosen randomly or according to some selected pattern or design. 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. This array indexing approach is of course very efficient, and independent of the size of the array.

N-Tuple networks work in a way similar to the kernel trick used in support vector machines (SVM)s, are related to Kanerva’s sparse distributed memory model, and are also closely related to Random Forests [33]. The low dimensional pattern space (i.e. the Othello board in this case) is projected via a highly non-linear projection into a high dimensional feature space by the N-Tuple indexing process. A linear function is then learned in this high-dimensional space. Hence the training process does not suffer from local minima, and usually converges quickly.

The weights of an N-Tuple system may be set by using either a one-pass training scheme, such as Maximum Likelihood Estimation, or linear regression (as used in this paper), or may alternatively be trained incrementally using back-propagation. The back-propagation rule has a particularly simple form. Each N-Tuple is treated independently. If an N-Tuple index \( i \) occurs \( b \) times for a particular board (game state) \( B \), and the back-propagated value is \( \delta \), then

\[
T[i] = T[i] + b\delta
\]

where \( T \) is the weights table for the N-Tuple. For a given Othello board an index may occur between 0 and 8 times, since each N-Tuple configuration has 8 symmetries. Note that weights are only modified for all the indexes that actually occur. Hence the update speed is independent of the size of the table. If the address space is very large then hashing can be used to store the entries which occur in a more memory-efficient way.

Lucas [21] showed how N-Tuples could be used as a high-performance function approximator for Othello. Note also that N-Tuples are closely related to the pattern-based tables used with great success in Logistello by Buro [30]. There is further evidence that they perform well as function approximators in the on-line Othello Position Evaluation Function League, previously mentioned, that has been run by the second author for several years. All the leading entries are N-Tuple networks, followed by spatial multi-layer perceptrons, then standard multi-layer perceptrons, then weighted piece counters. N-Tuples have also found success in other games, for example learning to play Connect-4 near-optimally at 1-ply [34].

When applying N-Tuple networks as Othello position value functions, it makes sense to model the natural symmetries in the game. Hence each distinct N-Tuple samples the board in
Table I: Sample points for the PRB N-Tuple. Symmetric expansion not included.

[25, 34, 27, 19, 28]
[43, 44, 60, 53, 6, 7]
[56, 48, 40, 9, 1, 0]
[48, 49, 5, 63]
[58, 2, 3, 20, 12]
[46, 38, 22, 45, 29]
[17, 16, 31, 24, 39, 32]
[3, 4, 13, 21]
[5, 60, 12, 3, 2, 57]
[0, 56, 49, 57, 50, 43]
[46, 37, 45, 36, 52, 43]
[14, 6, 5, 12, 4, 3]
[63, 48, 40, 7]
[15, 9, 16, 8, 0]
[51, 50, 43, 42, 34]

8 different ways, accounting for all possible reflections and rotations, but all 8 samples are mapped to the same weights table. When applying an N-Tuple network in this way, the design process consists of choosing the number and the size of N-Tuples to use, then choosing the sample points for each one.

We use a particular N-Tuple network (referred to as ETDL-N-tuple in the results section for Evolved TDL) found by Burrow using evolution [35]. This consists of 15 distinct N-Tuples and was evolved on the basis of its Othello playing ability when trained using TDL, with the weights being updated in accordance with the TD(0) error being applied via equation 1. Evolution was via a (5 + 5) ES run for 150 generations, with a total of 30,000 games being run each generation for the TDL training. Note that any reasonable set of N-Tuples would be fine for the current work, such as those described in [10] [11], and players based on [10] and [11] were used for comparative evaluation in our round-robin league.

The ETDL-N-Tuple network has 6561 weights (features) - approximately 100 times as many as the standard weighted piece counter. Although this is listed in third place in the Othello League previously mentioned, it showed very similar performance to the players above when competing against them in a round robin league, and it requires fewer weights. We use ETDL-N-Tuple to refer to that exact player in the league with those 6561 weights, but we use the same 15 N-Tuple system for all the N-Tuple learning experiments described in this paper. For clarity and repeatability, these are listed on Table 1 and shown with and without the symmetric expansion in Figure 2.

N-Tuple systems for Othello can work well and efficiently with thousands or even hundreds of thousands of weights, but the LSTD algorithm involves inverting a matrix based on the number of features (N-Tuple indices) which occur during the training set. This places a limitation on the size of N-Tuple system that can be used with this algorithm.

C. Posing the Problem

When applying the learning algorithms and feature types to the move preference learning problem there are some interesting choices that arise. These choices can significantly affect the efficiency of the training process and the accuracy of the trained classifier.

The first choice is whether the algorithm will learn a board state value function or a direct move predictor, which we refer to as a move-rater. The second choice is how to pose the classification problem.

1) State Evaluation Function versus Move-Rater: When learning a state evaluation (value) function each possible move is applied to the current board state to generate an afterstate of the board; the feature extraction algorithm is then applied to each board state to create the set of features for each move. These features are then input to the value function.

A move-rater works by directly extracting features associated with each move; this need not consider future game states and therefore may operate with greater efficiency, depending on the nature of the features. Due to its efficiency, this approach is typically used when the aim is to learn a function to guide MCTS rollouts. One way this can operate is by applying a pattern filter to each possible move, with the move square at the centre of the filter. The surrounding board squares are then used to provide contextual features for the move.

In the case of both the move-rater and the state value function, each move is used to produce a set of features, but it is worth making the distinction as to whether the function applies to the entire afterstate of the move, or just to the context of the move in the current board state. Although it would be interesting to investigate using the move-rater approach in Othello (and we are not aware of any previous work where this has been done), for this paper we restrict our attention to learning evaluation functions.

2) Classification: Once a set of features is obtained for each alternative move, we can then learn which ones are associated with the chosen move, as opposed to those for the moves which were not chosen. There are interesting choices to be made regarding the way in which the data is presented to the learner:

1) Label the feature vector of each selected move as class one, and of each non-selected move as class two. Create two separate two-class classifiers: one for when black is to move, one for when white is to move.

2) As for (1), except create a single classifier. If it is white’s...
move, then invert all the colours (so black becomes white and vice versa) In this way, the board is always seen from black’s perspective (i.e. it is always black’s turn to move in the data presented to the learner) and we train a single classifier to learn which moves should be selected.

3) Base classifications on pairwise difference vectors as explained in more detail in section VI on pairwise preference learning. This can be used with or without the board inversion technique mentioned above, and interestingly this turns out to be a critical choice.

IV. LEAST SQUARES TEMPORAL DIFFERENCE LEARNING

The essence of temporal difference learning (TDL) is to learn that states that are close in game trajectories (i.e. tend to occur sequentially) should have similar values. In traditional TDL, a state’s value is updated on-line as a game is played. After the update is calculated it is reduced by a factor $\alpha$ (the learning rate) before being used to update the state. If $\alpha$ is too high then learning can be unstable. If $\alpha$ is too low then learning can be too slow. Tuning $\alpha$ to achieve acceptable performance is a significant problem in TDL. Hence, in recent years, there has been interest in more sophisticated TDL algorithms that do not require a step size to be set. This is the approach taken by Least Squares TDL, LSTD($\lambda$), which eliminates all step size parameters and improves data efficiency. Results for this algorithm on toy problems such as the Boyan Chain are impressive [20].

Our implementation is based on [20] but adapted for a 2-player game, as shown in algorithm 1. The algorithm is complicated by the fact that the two players both update the same value function. Once a move is played the resulting board feature vector for the player $p$ is found and is denoted by $\phi_p$. This feature vector is kept as $\phi_p'$ and applied by the algorithm in the player’s following move. The number of features is $n$ and the purpose of the algorithm is to find the weight vector $w$ that minimises the TD error. There are no intermediate rewards, however, when the game terminates, both players receive a reward of +1 when black wins, -1 when white wins, and 0 for a draw.

The eligibility trace $z_p$ is a convenient way of implementing LSTD($\lambda$), and produces a family of methods spanning a spectrum that has Monte Carlo methods at one end with $\lambda = 1$ and one-step TDL methods at the other with $\lambda = 0$ [16]. The matrix $A$ should not be updated until the second move is made by a player. This is achieved by setting the eligibility trace $z_p$ to zero at the start of a game trajectory. The terminal feature vectors are, by definition, all zeros.

There is a simple case where the matrix $A$ and the vector $b$ have a direct interpretation: this is when $\lambda$ is set to zero (hence TD(0)) and the features implement a direct look-up table such that, in the $k$th state feature, $k$ is one and all other features are zero. In this case the leading diagonal of $A$ counts the number of times the corresponding state was visited, while off-diagonal elements $A_{ij}$ count the number of transitions from state $i$ to state $j$. The vector $b$ records the total reward received in each state. This simple case is not applicable here but nonetheless provides some insight into the nature of $A$ and $b$: for more details refer to Boyan [20].

Note that when N-Tuple features are used, the matrix $A$ may have rows and corresponding columns that are all zeros. This happens when particular N-Tuple features were not encountered during game play. These rows/columns must be removed before the pseudo-inverse of $A$ is found.

### Algorithm 1: LSTD($\lambda$)

<table>
<thead>
<tr>
<th>input : Game trajectories and parameter $\lambda$</th>
<th>output : weights vector $w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Set $A \leftarrow 0$, $b \leftarrow 0$ ;</td>
<td>11. $w \leftarrow A^{-1}b$ ;</td>
</tr>
<tr>
<td>2. for each game trajectory do</td>
<td>12. // use SVD.</td>
</tr>
<tr>
<td>3. $z_p \leftarrow 0$, $\phi_p \leftarrow 0$ ;</td>
<td>13. $b \leftarrow b + (\text{winner})z_p$ ;</td>
</tr>
<tr>
<td>4. $p \leftarrow 1$ ;</td>
<td>14. $A \leftarrow A + z_p(\phi_p')^T$ ;</td>
</tr>
<tr>
<td>5. while $\phi_p$ is not terminal do</td>
<td>15. // terminal feature vectors are all zeros</td>
</tr>
<tr>
<td>6. $\phi_p' \leftarrow \phi_p$ ;</td>
<td>16. end</td>
</tr>
<tr>
<td>7. $\phi_p \leftarrow \text{next board position's features}$ ;</td>
<td></td>
</tr>
<tr>
<td>8. $A \leftarrow A + z_p(\phi_p' - \phi_p)^T$ ;</td>
<td></td>
</tr>
<tr>
<td>9. $z_p \leftarrow \lambda z_p + \phi_p$ ;</td>
<td></td>
</tr>
<tr>
<td>10. $p \leftarrow \text{next player to move}$ ;</td>
<td></td>
</tr>
</tbody>
</table>

V. DIRECT CLASSIFICATION

Lagoudakis and Parr [13] proposed that policies may be approximated directly using binary classification. This is achieved by labelling moves selected as positive and those not selected as negative. Their labelling is based on Monte-Carlo rollouts, where moves are only labelled negative if they are statistically significantly worse than the best move (or moves), and labelled as positive only if a single move is statistically significantly better than all other moves available from the considered position.

We experimented with a similar form of Monte Carlo filtering, but found that it made no significant difference. Therefore we adopted the simpler approach of labelling all selected moves as positive and all non-selected moves as negative. In this case there is a possibility that a move is labelled negative in one game trajectory and positive in another. However, this type of noise is present for all linear classifiers studied.

Here one classifier will be created for both black and white players. To achieve this, board inversion is applied when the white player makes a move, as discussed previously. The aim of the linear classifier is then to satisfy:

$$ w \phi_j > 1 \ \forall j \in S \quad \text{and} \quad w \phi_k < -1 \ \forall k \in N $$

where $S$ and $N$ are the selected and non-selected board states respectively. The training data is unbalanced but may be compensated by weighting the positive labelled data in proportion to the game’s branching factor. We used the
LibLinear\(^7\) machine learning package to find \(w\), using its default settings which are: L2-regularized L2-loss support vector classification, cost parameter \(C = 1\), and no bias term.

VI. DIFFERENTIAL PREFERENCE LEARNING

The aim of preference learning is to make the correct choices rather than minimise some surrogate of this, such as the mean square error. As for the direct classification method, each move is made by considering all the after-states reachable by the set of legal single moves from the current board. The one chosen in the current game log is labelled as the correct move, and all others are labelled as incorrect. This is done in a pairwise manner: the selected move is paired in turn with each non-selected move, hence we arrive at a set of feature vector pairs. In the differential approach these features are subtracted from each other.

In an effort to make the decisions more clear-cut, we formulate the constraint to give the correct decision with a clear margin, arbitrarily chosen to be \(1.0\). In other words, the learner aims to satisfy this constraint for the maximising player:

\[
[w(\phi_j - \phi_k)] > 1 \forall j \in S^+, \ k \in N^+
\]  

(3)

and similarly for the minimising player:

\[
[w(\phi_j - \phi_k)] < -1 \forall j \in S^-, \ k \in N^-
\]  

(4)

where \(S\) and \(N\) are the selected and non-selected board states respectively, with the superscript indicating whether the player is maximising or minimising. Note that taking the pairwise difference between the features of each move tends to make the feature vectors more sparse and section VII-D provides some statistics on this. As with the direct classification approach we use the LibLinear machine learning package to find \(w\), using its default settings.

In general it is impossible to perfectly satisfy these constraints, since the same positions will certainly occur in different games with different choices having been made, otherwise every game would be the same. This is especially true in the very early stages of the game. This presents a similar potential problem for all learners, though the algorithms under test are sufficiently robust to be unaffected by this\(^8\).

With differential preference learning we also have the option of using board inversion (as was used with the direct classification and MM approaches) so that the moves are always seen from the perspective of the maximising player. Using the board inversion approach we now only use equation \(2\) with the aim of continually learning that the weighted feature difference between the chosen move and each non-chosen move should always be positive. This technique of using board inversion together with preference learning leads to the best results of all methods under test and is one of the contributions of this paper.

\(7\)http://www.csie.ntu.edu.tw/~cjlin/liblinear/

\(8\)We investigated the impact of this by estimating confidence bounds for the success of all moves using Monte Carlo methods and removing from the training set all positions where the chosen move was not significantly better than the next best move, but this did not improve test set prediction accuracy.

VII. MINORIZATION MAXIMIZATION

Coulom \(^{37}\) used the Bradley-Terry (BT) model \(^{38}\) for move prediction. The BT model models the strength of player \(i\) with a single value \(\gamma_i\) (where \(\gamma_i > 0 \forall i\)). Then, the probability that player \(i\) beats player \(j\) is given by the formula:

\[
P(i \text{ beats } j) = \frac{\gamma_i}{\gamma_i + \gamma_j}
\]  

(5)

Hence, the log probability that player \(i\) beats player \(j\) is proportional to the difference in their \(\gamma\) values.

This is the basic BT model, and forms the basis of the widely used Elo rating system \(^{39}\). The BT model can also be applied to games of more than two players, and games where each participant is a team. Coulom used a model extended in both these ways to estimate the probability that a particular move would be selected in preference to the alternative moves. He modelled a “game” in the BT model as being a competition between the set of the features associated with each possible move available in the current position (as mentioned in section III-C1), where the winner is the state reached by the winning move and the losers are all other states. The competition is modelled as being between teams of board features, where the \(i\)th feature has a strength \(\gamma_i\) and occurs either zero or once in a given move context.

To give a simple example, suppose there were three possible moves from a given position and that the chosen move (i.e. the winner) leads to a state with features 1, 2 and 3 active, denoted by \(\phi_{123}\), while the other moves (i.e. the losers) lead to states with features \(\phi_{134}\) and \(\phi_{356}\).

The probability of this chosen move being selected according to the model is given by:

\[
P(\phi_{123} \text{ beats } \phi_{34}, \phi_{356}) = \frac{\gamma_{123} \gamma_{34} \gamma_{356}}{\gamma_{123} \gamma_{34} + \gamma_{34} \gamma_{356} + \gamma_{356}}
\]  

(6)

In general if move \(m\) available in board position \(B\) leads to feature set \(\phi_m\), then the strength of this move \(s(m)\) is given by the product of the associated gamma values:

\[
s(m) = \Pi_{i \in \phi_m} \gamma_i
\]  

(7)

Then, according to the Bradley-Terry model, the probability that this move will be selected is given by:

\[
P(m) = \frac{s(m)}{\Sigma_{j \in B} s(j)}
\]  

(8)

Note that, in this model, an individual feature can appear in many teams during a single competition, but may appear only once in each team. It is therefore a natural model for binary-valued features, but is problematic for features which are inherently continuous or multi-valued. For example, the symmetric N-Tuples used below are multi-valued, since each distinct N-Tuple is replicated for all its 8 symmetries, which means the associated feature values may occur between zero and eight times in a single board state.

Coulom also derived a minorization-maximization algorithm for adjusting the \(\gamma\) values to maximise the probability, given the model, that all the expert moves were selected. This is similar to the expectation maximization (EM) algorithm
used to train hidden Markov models (EM is a special case of MM). The algorithm starts with an initial guess of the parameters, then iterates until convergence. Each iteration of the algorithm is guaranteed to either increase the likelihood of the data, or keep it the same. The algorithm stops once the improvement falls below a specified value.

Each \( \gamma \) is updated as follows:

\[
\gamma_{i}^{t+1} = \frac{W_{i}}{\sum_{j} C_{ij}^{t+1} E_{ij}^{t}}
\]

(9)

where \( W_{i} \) is the number of times that \( \phi_{i} \) was a member of a winning team, \( C_{ij}^{t} \) is the sum of the products of the gamma values of its team mates (not including itself) involved in match \( j \) and \( E_{ij}^{t} \) is the sum of the gamma values of all the participants (including itself) in match \( j \). Note that the \( t \) and \( t + 1 \) superscripts indicate that the new value for \( \gamma_{i} \) is based on the previous values of all \( \gamma \) values. Hence, the above formulation suggests that each \( \gamma \) value must be updated serially rather than in batch mode, though Coulom shows how the \( E_{ij} \) calculation can be shared between mutually exclusive \( \gamma \) values that belong to the same feature and never occur together in the same team.

When the model has been trained, the relative value of each move \( m \) is given by its strength \( s(m) \), the product of its gamma values, since the denominator is common for all moves available in a given board position. Hence, moves can be ranked based only on equation (7).

Note then that although the MM training algorithm and the motivation for the MM approach is substantially different from our preference learning model, once either model has been trained it can be applied in a very similar way. There are two differences. One is that the preference approach ranks each move based on the sum of feature weights. If we take the logarithms of the gamma values in the MM approach, then the move ranking algorithm used in each case can be identical. The second difference is that the model used by Coulom only works with binary-valued features, and so some pre-processing of the feature values may be needed before applying MM. We ran some experiments ignoring this detail i.e. using MM to train a BT model with multi-valued features, but the results were significantly worse than when binarising them.

It is worthwhile appreciating the similarities in the classification model, since now the main question that remains is:

Which conceptual framework and hence which training algorithm is most appropriate for training the parameters of the move ranking model?

Given that the true aim is to correctly predict moves, we suggest that the most appropriate approach is the preference learning one where we attempt to directly model which move should be preferred, rather than maximise the likelihood of the training set or minimise the Bellman residuals. However, the question of which approach performs best in practice and by what margin can only be answered though empirical investigation, and a priori it is hard to predict whether preference learning would work better with or without board inversion. Post investigation, it was surprising to us just how poorly the direct classification approach performed.

VIII. Experimental Study

The experimental study examines the difference in performance of all the methods under test in two ways: their ability to predict expert moves, and their ability to play Othello when deployed in a 1-ply minimax search engine.

To recap, and give the abbreviations used in the results tables, the methods under test are:

- Pref – preference learning with negation of outputs to play as black or white.
- iPref – identical to Pref, except using board inversion instead of output negation to play as black or white.
- LSTD(\( \lambda \)) – Least Squares Temporal Difference Learning, applied in the standard way with negation of outputs.
- MM – Minorization Maximization, using board inversion to play as black or white.
- Classify – Two-class classification problem (selected versus non-selected moves), using board inversion to play as black or white.

Each of these is used with both 1-Tuple (the weighted piece counter with 192 weights, 3 weights for each square) and N-Tuple features as described above, leading to a total of 10 players developed for this paper.

Additionally, to provide a wider context, we have also included the following three players from previous studies which are known to have high performance for their type of architecture:

- Heur-WPC – The “standard” Othello heuristic weighted piece counter with 64 weights (though only 10 unique values due to symmetry). The weights for this player were published in (40) and can be found conveniently in [9].
- Coev-WPC – The co-evolved WPC from Samothrakis et al (41), with 64 learned weights (not symmetric). Their method was to use Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (42) in conjunction with an archive to ameliorate the effects of intransitivities when co-evolving players.
- ETDL-N-Tuple – The N-Tuple described in Section III-B with an evolved structure but with weights trained using TDL. Note: all the N-Tuple systems evaluated for matching expert play use exactly the same structure as specified in Table 9 and depicted in Figure 2 only the weights differ. We also used four other N-Tuple players as specified next.
- SJK-CTDL-N-Tuple and SJK-ETDL-N-Tuple from Szubert et al (11) (the SJK comes from the authors’ names). CTDL refers to their co-evolved / TDL trained player while ETDL refers to their evolved / TDL trained player.
- Nash1-N-Tuple and Nash2-N-Tuple were supplied by Manning (10), and trained using a mixture of “Nash Memory”-based evolution and TDL. The Nash2 player was evolved by mutating only the weights while the Nash1 player’s evolution also allowed mutations to the N-Tuple sample points and outperforms the Nash2 player.

Note that the weights for all players are listed in the on-line repository for this paper.

See https://notendur.hi.is/~tpr/pref/
This gives us a total of 17 players to evaluate in the round robin league, though we only include the results from 12 players for matching expert play, since players that have not been trained on the expert play logs do not perform well at this task. To illustrate this point we include Heur-WPC and ETDL-N-Tuple in Table I.

A. Matching Expert Play

A set of 1000 league games is used for training and a further independent 1000 league games are used for testing. The size of the datasets both in terms of the number of patterns and the average number of features per pattern depends on the details of the method, but 1000 games leads to over 400,000 patterns (each game may be up to 60 moves, and each move may have many possible afterstates). The training time depends on the details of each method used and the methods scale differently. Nonetheless it is useful to provide a rough idea of how long each method takes. Where training times are mentioned below they are for an Intel i7 PC using a single core.

When the number of features is large, as in the case of the N-Tuple features, the most costly step in LSTD is finding the pseudo-inverse of the feature matrix. This step depends only on the number of features, not on the number of patterns. In the case of the N-Tuple configuration we use there are potentially 6,561 features, but only 5,355 occur in the training set, which means finding the pseudo-inverse of a 5,355 × 5,355 matrix. This step takes a few minutes for this size of matrix, but scales with the cube of the number of features, so the time taken could be problematic for large N-Tuple systems with tens of thousands of features.

LibLinear (which we use as a black box machine learning tool) works by iteratively solving a quadratic programming problem. The cost of this depends on the number of patterns, the number of features, how easily separable the pattern classes are and the setting of the “slack” variable $C$. For our settings, LibLinear learns the 437,883 N-Tuple patterns in less than a minute. The slowest method under test was MM, which took several hours to train.

The performance of the learned functions on the test set is compared for the different learning methods and feature sets. Test set errors and the percentage of the maximum possible score obtained are reported.

Section VIII-B reports on the results for the learned weights using the 1-TUPLE features. These are then also compared with those of the standard hand-crafted heuristic weights, Heur-WPC.

Section VIII-C reports the equivalent results using the N-Tuple features. Table II includes all the expert move prediction results in a single table for easy comparison.

B. 1-Tuple Features

In previous work [12] we compared the performance of preference learning versus LSTD when learning the weights of a weighted piece counter. These results are still of interest to the current paper since we are now including the MM algorithm in the comparison and direct classification. However, while performing the new experiments, we found that a 1-Tuple system generally outperformed the more standard weighted piece counter, so instead we have used this as our simple form of value function. The 1-Tuple has three weights for each of the 64 squares on the board: a separate weight to be applied for each possible state of the square, empty, white, or black (hence $192 = 3 \times 64$ weights in total). We also experimented with a symmetric 1-Tuple that only stores separate weights for squares that are distinct under symmetry, hence 10 squares (30 weights) in total. However, this did not perform as well as the asymmetric version. The best setting for $\lambda$, for the LSTD, is 0.9 and will be used in all the experimental results reported.

Figure 3 shows how the decision accuracy varies with the stage of the game for each approach. The top set of lines indicates the percentage of correct pair-wise decisions, while the bottom set indicates the percentage of correct move choices. Table II shows the underlying numbers, with the bottom row showing the mean branching factor, the total number of patterns, and the mean accuracy for each approach. Note how differential preference learning and MM substantially outperform LSTD until the final stages of the game, when LSTD performs slightly better. The direct classification approach performs slightly worse but matches MM towards the end of the game.

The 1-Tuples are asymmetric and naturally produce binary-valued features. For the 1-Tuples, MM proved to be the best method, outperforming preference learning by a small but statistically significant margin. However, the preference learner using board inversion performs better at games stages 3 to 5. All methods performed very much better than LSTD.

We analysed the weight vectors learned in each case. One of the most important things to learn in Othello is the value of playing in the corner, and the danger of playing next to the corner (see Figure 1). Only preference learning was able to learn both these things reliably. LSTD learned the value of playing in the corners, but was unable to learn the danger of playing adjacent to a corner. This observation coincides with the way that LSTD learns better in the later stages of the game, at which stage the adjacent cells to a corner may well have been flipped to the colour of the corner occupier, and hence they would no longer show up as being poor moves.

C. N-Tuples

The experiments from the previous section are repeated here, but this time the ETDL-N-Tuple is used as a reference. These results are plotted in Figure 3 and given in full in Table II. The N-Tuples are able to capture the human playing policy with greater accuracy than 1-Tuples. The preference learner performs best when used in board inversion mode (iPref) where it averages 53% move prediction accuracy. This is superior to the next best (Pref) and far superior to MM, and to LSTD by an even greater margin. The direct classification method performs similarly to the MM method, but does not produce a player of the same strength, as illustrated by the round-robin results.

The percentage of correct moves made drops towards the middle of the game, as the branching factor increases and
board states become more variable. The iPref-N-Tuple player is unable to match the human move decisions satisfactorily, however, its performance is better than that of the 1-Tuple. The worst performance of the iPref-N-Tuple is around 40% correct moves made (discs 33 — 36), while the best performing 1-Tuple (MM) drops to just over 20% (discs 21 — 24) (see Table II). The ETDL-N-Tuple makes very different moves to the human experts and so clearly plays a different strategy, albeit a strong one (the second strongest of all evaluation functions under test), as clearly demonstrated by the round-robin results below.

The previous results reported how well the approaches learned to approximate human play. It is also of interest to measure and compare the playing strength of each approach. Playing strength was estimated from a full round robin league where each weight vector was used to play each other one, leads to unbalanced datasets (since at each stage only one move is chosen from the possible ones available). We fed both the unbalanced datasets and ones which were balanced through appropriate replication of the winning patterns into Liblinear, but found in all cases that the difference vector approach was more accurate. The results for the balanced datasets.

For the N-Tuple features (6,561 features) the saving was less pronounced, with the difference method having an average of 64.8 non-zero features versus 82.7 for the classification approach. For the 1-Tuple features (192 features) the difference vectors had an average of 11.8 non-zero ones, while for the classification approach this was 64 (in fact every input has exactly 64 non-zero features, since every square is in one of three possible states).

We measured the average number of features present in the input vectors, given the combination of feature set and classification approach. For the 1-Tuple features (192 features) the difference vectors had an average of 11.8 non-zero ones, while for the classification approach this was 64 (in fact every input has exactly 64 non-zero features, since every square is in one of three possible states).

For the N-Tuple features (6,561 features) the saving was less pronounced, with the difference method having an average of 64.8 non-zero features versus 82.7 for the classification method.

Given the natural efficiency gains, we were interested to observe the effects on performance. The classification approach leads to unbalanced datasets (since at each stage only one move is chosen from the possible ones available). We fed both the unbalanced datasets and ones which were balanced through appropriate replication of the winning patterns into Liblinear, but found in all cases that the difference vector approach was more accurate. The results for the balanced data were slightly better and these are presented in Figures 3 and 4 and Table II.

### E. Round Robin League

The previous results reported how well the approaches learned to approximate human play. It is also of interest to measure and compare the playing strength of each approach. Playing strength was estimated from a full round robin league where each weight vector was used to play each other one, using one-ply minimax search from the same 1,000 positions.

The 1,000 positions were chosen by running a vanilla UCT Othello player (similar to the one described in [43]) using a budget of 5,000 simulations per move. This leads to a reasonable standard of play but with a significant random element. We then harvested 1,000 random unique positions from depth 6 in the game tree (i.e. after three moves each by black and white), and played each player as black and as white from these positions using a 1-ply minimax search with
no noise added. Since there are 17 players in the league, each player played a total of 32,000 games.

We then used BayesElo to rank the players and to assess the likelihood of superiority. Table III shows the rank order of each player together with its Elo rating, with the mean rating set to 1600. The table also shows the percentage of wins and draws attained by each player.

These results are a clear indication of the relative strength of each player when pitted against each other, though it should be kept in mind that significant intransitivities exist when using fixed value functions to dictate Othello playing policy[41].

Clearly the best performing players by a significant margin are SJK-CTDL-N-Tuple and iPref-N-Tuple. When the same training algorithm is used, better performance is always obtained with N-Tuple rather than with 1-Tuple features, though it is interesting to note that the best WPCs (Coev-WPC and Heu-WPC) outperform three of the weaker N-Tuple players.

Table III: Rank order, relative Elo rating and percentage of available points attained with {1.0, 0.5, 0.0} awarded for win, draw and loss respectively. Results are based on a full round-robin league with each player playing 32,000 games from fixed opening positions using 1-ply minimax search. Column b/w indicates the approach taken to distinguish black moves from white moves and column $|w|$ shows the number of weights in each model.

| Rank | Player               | Rating | % Score | b/w | $|w|$ |
|------|----------------------|--------|---------|-----|------|
| 1    | SJK-CTDL-N-Tuple     | 18590  | 78.0%   | neg | 4698 |
| 2    | iPref-N-Tuple        | 1848   | 77.8%   | inv | 6561 |
| 3    | ETDL-N-Tuple         | 1803   | 73.3%   | neg | 6561 |
| 4    | Nash2-N-Tuple        | 1787   | 71.6%   | neg | 8748 |
| 5    | Nash2-N-Tuple        | 1765   | 69.2%   | neg | 8748 |
| 6    | SJK-ETDL-N-Tuple     | 1765   | 69.1%   | neg | 3240 |
| 7    | Pref-N-Tuple         | 1730   | 65.3%   | neg | 6561 |
| 8    | Coev-WPC             | 1597   | 49.5%   | neg | 64  |
| 9    | Heur-WPC             | 1596   | 49.4%   | neg | 64  |
| 10   | MM-N-Tuple           | 1573   | 46.5%   | inv | 6561 |
| 11   | iPref-1-Tuple        | 1499   | 37.9%   | inv | 192 |
| 12   | MM-1-Tuple           | 1479   | 35.5%   | inv | 192 |
| 13   | Pref-1-Tuple         | 1457   | 33.1%   | neg | 192 |
| 14   | Classify-N-Tuple     | 1444   | 31.6%   | inv | 6561 |
| 15   | Classify-1-Tuple     | 1364   | 23.4%   | inv | 192 |
| 16   | LSDT-N-Tuple         | 1348   | 21.9%   | neg | 6561 |
| 17   | LSDT-1-Tuple         | 1293   | 17.1%   | neg | 192 |

While Table III shows the aggregate of a full round robin against all players, it is also interesting to see where each player is strongest. To investigate this we created sub-leagues using the two strongest players together with:

1) All the strongest N-Tuple players (table IV).
2) A selection of the weakest players (table V).

The results when viewed in this way are illuminating. When pitted against the set of strongest players iPref-N-Tuple is clearly the strongest and actually defeats every other player on a head-to-head basis, defeating even SJK-CTDL-N-Tuple 1134.5 to 865.5. When played against the weakest players SJK-CTDL-N-Tuple is the strongest, since it is better at exploiting their weaknesses than iPref-N-Tuple. A possible explanation for this is that during the co-evolutionary training of the CTDL player it will have encountered a great deal of weak play to learn from. Conversely, iPref-N-Tuple has only been trained on the logs of games between strong players, and will have seen fewer of the positions reachable though poor play.

F. Summary and Analysis of Results

MM is the strongest approach when learning weights for the 1-Tuple features, while preference learning with board inversion (iPref) is the strongest by a large margin when learning weights for the N-Tuple features. It is interesting to consider why this difference might arise. Recall that the 1-Tuple and the N-Tuple features differ in the following ways:

1) The N-Tuple features are in a much higher-dimensional space (6561 versus 192) and have a much higher degree of linear separability. Liblinear maximises the soft margin in this high-dimensional space while considering regularisation, whereas MM does not aim for a maximal margin, and its “regularisation” is limited to selecting the value of a small number of priors (three).

2) The N-Tuple features are naturally multi-valued, and the binarization required for MM might be throwing away valuable information. It is possible that cleverer binarization approaches might ameliorate this effect, but exploring this in detail is beyond the scope of this paper.

3) To apply MM we used board inversion to ensure that the move selection problem is always seen from black’s perspective. This might be discarding subtle nuances, a problem more likely to apply to the N-Tuple features. However, the alternative is to learn an MM model separately for black and for white, which doubles the number of weights to learn, and hence is not without its drawbacks. Furthermore, board inversion cannot cause major problems, since the best method under development in this paper (iPref-N-Tuple) uses board inversion.

We undertook further investigation as to which of these possible factors makes the most significant contribution to the strong performance of preference learning compared to MM. Regarding point 1, we compared results on the training set to see if MM was overfitting (due to poorer regularisation) but found that training set performance was very similar to test set performance.

To investigate point 2 we applied the preference learning technique to binarized features, and found that binarization caused a drop in move prediction accuracy from 49.4% to 44.5%. This is a significant drop, but is still 2.0% higher than MM. The types of binarization used by Coulom[15] could be applied to further improve MM, but given that the preference learning approach still outperforms it when given the exact same data, we do not see this as a high priority.

For point 3 we applied MM to predict black moves only in order to remove any effects of the board inversion procedure. This improved training set accuracy by 2.4%, but reduced test set accuracy by 1.7%, presumably due to the reduction in the available training data.

The results suggest that the superior performance of preference learning compared to MM is largely due to its ability to handle multi-valued features, but also in part due to the support

vector machine approach to maximal margin classification in a high-dimensional feature space. Although it is always possible to transform multi-valued features into binary ones, doing this well takes time and effort. The fact that preference learning can handle multi-valued features directly is a distinct advantage compared to MM.

Finally, the effects of board inversion on the preference learning approach when used with N-Tuples are especially interesting, and led to the highest performing system, both for predicting expert moves and when playing in the round-robin league. The board inversion approach focuses on learning strategies which are good for either player, and ignores subtleties where a particular line of play may be good for black but not for white. Doing this effectively doubles the amount of training data within the populated regions of feature space, and hence leads to superior performance.

IX. Conclusion

This paper presented the results of using differential preference learning to imitate human play from game logs. Using logs taken from the French Othello league, we applied preference learning in two ways: using a standard output negation method, and using board inversion. We compared the preference learning approach with LSTD, with MM, and with a more standard classification approach.

For each experiment identical features were used (though binarized for use with MM), and an identical classifier was used to predict the moves; only the learning algorithms differed. Although the MM decision criterion involves a product rather than a sum of weights, taking logarithms allowed us to replicate the classification algorithm across all approaches without altering the decisions made.

In each case, learning was used to estimate the weights of a linear evaluation function in feature space, either using the board vector directly as the set of features, or using the highly non-linear N-Tuple features. The N-Tuple approach provided the best performance and, when used with preference learning in combination with board inversion, outperformed all other methods under test in two ways: it learned to better match expert human decision making and it produced better performing players. To evaluate playing performance we used a round-robin league involving all the players developed in this paper together with a range of players developed by other researchers. Our best player (iPref-N-Tuple) defeated every opponent on a head-to-head basis.

Our previous work showed that differential preference learning outperformed LSTD when using a weighted piece counter value function, so it was not too surprising when this result also held for the N-Tuple weights. The real surprise was the large margin by which preference learning outperformed the MM algorithm when using N-Tuple features, keeping in mind that MM is a leading method for move prediction in Go. A promising avenue for future work is therefore to investigate whether preference learning can be used to improve the performance of leading Go programs.

In addition to the potential for stronger game AI, preference learning has a great deal to offer in better imitating human styles of play, even if this does not lead to play of a higher standard. In many board games, Othello being one example, it is more of a challenge to generate AI behaviour that is fun and interesting to play against, rather than simply being strong.

In this paper our primary goal was to further develop a new approach to move prediction and compare it with state of the art algorithms using two different feature sets. A future goal is to introduce more features (such as mobility) with the aim of achieving even higher accuracy in the imitation of particular expert players. By imitating expert play we were also able to produce one of the strongest known Othello value function based players\[1\]. This is an interesting and non-obvious result, since it would also be possible to have players that closely matched expert play most of the time but made enough disastrous errors to lose most of their games.

We used LibLinear to set the weights of the preference learner, but this has practical limits on the size of dataset it can deal with. An interesting alternative would be to train the system using on-line back-propagation in order to deal with game-log datasets that are orders of magnitude larger than the one used in the current study, the trade-off being the loss of the maximal margin property.

Given the significant margin by which preference learning with board inversion outperforms preference learning with output negation, a promising avenue for future work is using the board inversion technique for temporal difference learning, where currently the output negation approach is standard.

Finally, the approach is not in any way limited to board games, and the preference learning approach may find important application in the development of more human-like non-player characters in video games.

Acknowledgements

This research was partially funded by EPSRC Grant EP/H048588/1.

References


\[1\] Using a standard evaluation method of testing by playing games using 1-ply look-ahead.
Table II: The branching factor (BF) and number of training samples at different game stages. The percentage of correct moves taken on testing data for preference learning with board inversion (iPREF) and without (PREF), MM, LSTD and the direct classification approach (Classify) using both 1-Tuple and N-Tuple features. We also include results of predicting using the standard heuristic weighted piece counter (Heur-WPC), the co-evolved weighted piece counter (Coev-WPC) and the the evolved structure, TDL-trained N-Tuple (ETDL-N-Tuple).

<table>
<thead>
<tr>
<th>#discs</th>
<th>BF</th>
<th>#N</th>
<th>1-Tuple</th>
<th>N-Tuple</th>
<th>Heur-WPC</th>
<th>ETDL-N-Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BF</td>
<td></td>
<td>iPREF</td>
<td>Pref</td>
<td>MM</td>
<td>LSTD</td>
</tr>
<tr>
<td>1–16</td>
<td>7.1</td>
<td>73133</td>
<td>47.4</td>
<td>47.2</td>
<td>35.8</td>
<td>16.4</td>
</tr>
<tr>
<td>17–20</td>
<td>11.0</td>
<td>40045</td>
<td>17.6</td>
<td>31.0</td>
<td>20.8</td>
<td>8.0</td>
</tr>
<tr>
<td>21–24</td>
<td>11.5</td>
<td>42194</td>
<td>25.2</td>
<td>25.7</td>
<td>20.6</td>
<td>9.2</td>
</tr>
<tr>
<td>25–28</td>
<td>11.9</td>
<td>43796</td>
<td>28.1</td>
<td>25.7</td>
<td>23.8</td>
<td>11.0</td>
</tr>
<tr>
<td>29–32</td>
<td>11.7</td>
<td>42818</td>
<td>28.2</td>
<td>24.4</td>
<td>23.8</td>
<td>9.5</td>
</tr>
<tr>
<td>33–36</td>
<td>11.3</td>
<td>41319</td>
<td>28.1</td>
<td>23.7</td>
<td>23.9</td>
<td>9.0</td>
</tr>
<tr>
<td>37–40</td>
<td>10.6</td>
<td>38318</td>
<td>28.9</td>
<td>26.1</td>
<td>25.6</td>
<td>10.9</td>
</tr>
<tr>
<td>41–44</td>
<td>9.6</td>
<td>34308</td>
<td>29.4</td>
<td>28.7</td>
<td>28.3</td>
<td>13.5</td>
</tr>
<tr>
<td>45–48</td>
<td>8.4</td>
<td>29412</td>
<td>29.7</td>
<td>29.3</td>
<td>29.4</td>
<td>17.8</td>
</tr>
<tr>
<td>49–52</td>
<td>7.1</td>
<td>23784</td>
<td>30.5</td>
<td>30.6</td>
<td>31.4</td>
<td>25.9</td>
</tr>
<tr>
<td>53–56</td>
<td>5.5</td>
<td>17385</td>
<td>34.7</td>
<td>32.3</td>
<td>35.2</td>
<td>33.5</td>
</tr>
<tr>
<td>57–60</td>
<td>4.0</td>
<td>10960</td>
<td>40.8</td>
<td>38.5</td>
<td>41.4</td>
<td>42.9</td>
</tr>
<tr>
<td>61–64</td>
<td>2.5</td>
<td>3411</td>
<td>51.9</td>
<td>47.9</td>
<td>51.6</td>
<td>53.0</td>
</tr>
<tr>
<td>∑</td>
<td>8.6</td>
<td>(437883)</td>
<td>33.8</td>
<td>33.0</td>
<td>34.5</td>
<td>18.4</td>
</tr>
</tbody>
</table>

Table IV: League results for the seven strongest N-Tuple players when played against each other. Table shows rank order, round robin results, relative Elo rating and percentage of available points attained with \{1.0, 0.5, 0.0\} awarded for win, draw and loss respectively.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Player</th>
<th>iPREF-N-Tuple</th>
<th>SJK-CTDL-N-Tuple</th>
<th>Nash1-N-Tuple</th>
<th>ETDL-N-Tuple</th>
<th>Nash2-N-Tuple</th>
<th>Pref-N-Tuple</th>
<th>SJK-ETDL-N-Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iPREF-N-Tuple</td>
<td>1134.5</td>
<td>1614.0</td>
<td>1089.0</td>
<td>1575.5</td>
<td>1578.0</td>
<td>1713.5</td>
<td>1746.0</td>
</tr>
<tr>
<td>2</td>
<td>SJK-CTDL-N-Tuple</td>
<td>865.5</td>
<td>1031.0</td>
<td>1111.5</td>
<td>1293.5</td>
<td>1248.0</td>
<td>1306.5</td>
<td>1382.5</td>
</tr>
<tr>
<td>3</td>
<td>Nash1-N-Tuple</td>
<td>713.5</td>
<td>969.0</td>
<td>1044.5</td>
<td>1087.5</td>
<td>995.0</td>
<td>1212.0</td>
<td>1599.0</td>
</tr>
<tr>
<td>4</td>
<td>ETDL-N-Tuple</td>
<td>754.5</td>
<td>781.0</td>
<td>955.5</td>
<td>940.5</td>
<td>1164.0</td>
<td>1184.5</td>
<td>1588.0</td>
</tr>
<tr>
<td>5</td>
<td>Nash2-N-Tuple</td>
<td>662.5</td>
<td>888.5</td>
<td>942.5</td>
<td>1059.5</td>
<td>1011.5</td>
<td>1107.0</td>
<td>1583.0</td>
</tr>
<tr>
<td>6</td>
<td>Pref-N-Tuple</td>
<td>752.0</td>
<td>706.5</td>
<td>1005.0</td>
<td>836.0</td>
<td>893.0</td>
<td>1160.0</td>
<td>1546.0</td>
</tr>
<tr>
<td>7</td>
<td>SJK-ETDL-N-Tuple</td>
<td>628.0</td>
<td>651.0</td>
<td>788.0</td>
<td>815.5</td>
<td>893.0</td>
<td>1160.0</td>
<td>1546.0</td>
</tr>
</tbody>
</table>

Table V: League comprising the two strongest N-Tuple players together with a selection of weak players. Table shows rank order, round robin results, relative Elo rating and percentage of available points attained with \{1.0, 0.5, 0.0\} awarded for win, draw and loss respectively.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Player</th>
<th>SJK-CTDL-N-Tuple</th>
<th>iPREF-N-Tuple</th>
<th>WPC-Coev</th>
<th>WPC-Heu</th>
<th>One-MM</th>
<th>One-Diff</th>
<th>One-Cls</th>
<th>One-LSTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SJK-CTDL-N-Tuple</td>
<td>865.5</td>
<td>1614.0</td>
<td>1089.0</td>
<td>1575.5</td>
<td>1578.0</td>
<td>1713.5</td>
<td>1746.0</td>
<td>1668</td>
</tr>
<tr>
<td>2</td>
<td>iPREF-N-Tuple</td>
<td>1134.5</td>
<td>1031.0</td>
<td>1111.5</td>
<td>1293.5</td>
<td>1248.0</td>
<td>1306.5</td>
<td>1382.5</td>
<td>1644</td>
</tr>
<tr>
<td>3</td>
<td>WPC-Coev</td>
<td>92.5</td>
<td>386.0</td>
<td>911.0</td>
<td>1297.0</td>
<td>1382.5</td>
<td>1296.5</td>
<td>1494.0</td>
<td>1632</td>
</tr>
<tr>
<td>4</td>
<td>WPC-Heu</td>
<td>136.5</td>
<td>200.5</td>
<td>424.5</td>
<td>703.0</td>
<td>1128.0</td>
<td>1306.5</td>
<td>1629.5</td>
<td>1599</td>
</tr>
<tr>
<td>5</td>
<td>One-MM</td>
<td>98.5</td>
<td>158.0</td>
<td>422.0</td>
<td>617.5</td>
<td>1246.0</td>
<td>1595.5</td>
<td>1512</td>
<td>1512</td>
</tr>
<tr>
<td>6</td>
<td>One-Diff</td>
<td>78.5</td>
<td>121.5</td>
<td>286.5</td>
<td>703.5</td>
<td>1306.5</td>
<td>1629.5</td>
<td>1512</td>
<td>1512</td>
</tr>
</tbody>
</table>