Preference Learning for Move Prediction and Evaluation Function Approximation in *Othello*

Thomas Philip Runarsson, *Senior Member, IEEE*, and Simon M. 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 (MM). 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.**

***Index Terms*— Computational and artificial intelligence, n-tuple, preference learning, temporal difference learning, *Othello*.**

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 preeminent 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 central processing unit (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 coevolution 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 coevolution normally focuses only on the end result. Coevolution utilizes less of the available information [7], but its focus on the end result can lead to greater robustness than TDL.

Coevolution 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 coevolution 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 Congress on Evolutionary Computation (CEC) competition[1] 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 endgame 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

Manuscript received December 05, 2012; revised May 02, 2013 and July 20, 2013, and September 18, 2013; accepted December 10, 2013. Date of publication March 11, 2014; date of current version September 11, 2014. This work was supported in part by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/H048588/1.

T. P. Runarsson is with the School of Engineering and Natural Sciences, University of Iceland, Reykjavik 101, Iceland (e-mail: tpr@hi.is).

S. Lucas is with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, U.K. (e-mail: sml@essex.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2014.2307272

[1] http://algoval.essex.ac.uk:8080/othello/html/Othello.html

use the same 1-ply search approach, but use a set of 1000 unique opening positions instead of the forced random moves to ensure a thorough evaluation of playing ability.

Clearly, imitating humanlike 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 humanlike 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 MCTS.

An initial investigation by the authors illustrated that preference learning significantly outperforms TDL when imitating game play [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 TDL, 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 analyzing the differences observed when using a more sophisticated N-tuple value function, and also when attempting to learn the policy directly using classification [13]. We also compare performance with the leading technique for move prediction in *Go*, which involves using minorization–maximization (MM) to fit a Bradley–Terry team model to the move selection data, as explained below. As before [12], we use game trajectories taken from human competitions held by the French Othello Federation[2] 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 [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. Section II 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 Section 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 pseudocode of the least squares temporal difference learning [LSTD(λ)] algorithm in Section IV, direct classification in Section V, and preference learning algorithms employed in Section VI. The MM approach is explained in Section VII, results of the comparison are presented in Section 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 favorability of a board position for the maximizing player (and conversely the opposite of this for the minimizing player). As mentioned above, this approach was used successfully by Buro to estimate the weights for his Logistello program [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.* [14], and Coulom's approach [15] of using MM [16] to fit a Bradley–Terry model to the move selection data.

A recent study by Wistuba *et al.* [17] using the same features and game data for all methods under test, found Coulom's method 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 [18]. The most effective approaches employ LSTD(λ) [19], [20]. Apart from λ, 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 nonlinear functions of the board state. An example of such nonlinear features are the N-tuples [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 [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 [22]. In [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 nonselected sets, i.e., they are given different class labels [13], [23], [24]. 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 also added to the training set with a negative label [13]. Labeling moves as selected versus nonselected will also be investigated

[2] www.ffothello.org

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 rollouts 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 3-D 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 nonselected 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 nonpreferred 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.

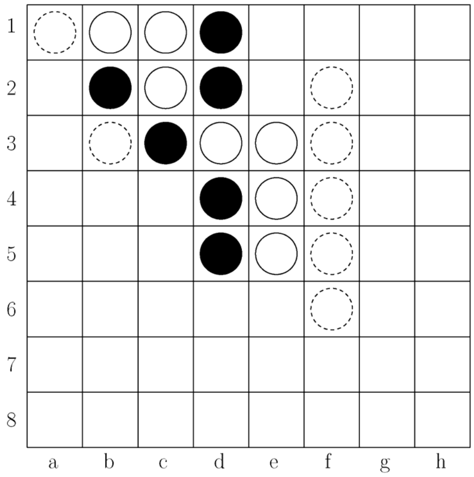

Fig. 1. *Othello* game in progress with seven possible legal moves for black (dashed circles). Capturing corners is one key strategy in playing *Othello*, so “a1” is probably the best move.

The direct classification methods are unable to use output negation and must therefore use board inversion (or color 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 III-C.

III. *Othello* Game Trajectories

The game of *Othello* is played on an 8 × 8 board, with a starting configuration of the middle four squares occupied by two white and two black discs. Black plays first and the game continues until the board is full (after 60 nonpassing 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.

Fig. 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 maximize the number of disks of their own color at the end of the game.[3] *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.[4] 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, $g_s = \max(0, \lfloor \#\text{discs} - 13)/4 \rfloor)$. The training data used by Logistello are based on some 80 000 games generated by

[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.

[4] http://skatgame.net/mburo/log.html

an earlier, less tuned version of the program playing against another *Othello* program (Kitty). Toward the end of the game, the positions are labeled 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 labeled with the final disc differential estimate. This approach yields significantly better performance than Buro's previous work [6] where the positions were labeled by the probability of winning. Clearly, labeling 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 (afterstate or postdecision 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 × 8 board is unwound as a 64 element vector ϕ. Each element of ϕ is 0 for an empty square, $+1$ for a black counter, and -1 for a white counter. This WPC has a vector $\mathbf{w}$ of 64 weights, one for each square on the board. The evaluation of a board is then calculated as the scalar product $\mathbf{w}\phi$. 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 colors 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 ϕ, 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 Section III-B), 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 (SVMs), 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 nonlinear 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 backpropagation. The backpropagation 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 backpropagated value is δ, then

$$T[i] = T[i] + b\delta \qquad (1)$$

where T is the weights table for the N-tuple. For a given *Othello* board, an index may occur between zero and eight times, since each N-tuple configuration has eight 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 online 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 multilayer perceptrons, then standard multilayer 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 eight different ways, accounting for all possible reflections and rotations, but all eight 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.

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]
```

We use a particular N-tuple network [referred to as evolved TDL-Ntuple (ETDL-N-tuple)] 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 (1). Evolution was via a $(5 + 5)$ evolution strategy (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] and [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,[5] 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 in Table I, and shown with and without the symmetric expansion in Fig. 1.

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 center 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 are presented to the learner.

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

b) As for a), except create a single classifier. If it is white's move, then invert all the colors (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.

c) 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 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 online as a game is played. After the update is calculated, it is reduced by a factor α (the learning rate) before being used to update the state. If α is too high, then learning can be unstable. If α is too low, then learning can be too slow. Tuning α 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

[5] It is "prb_nt15_001" in the online league.

approach taken by least squares TDL, LSTD(λ), 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 two-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 $\boldsymbol{\phi}_p$. This feature vector is kept as $\boldsymbol{\phi}'_p$ 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 $\boldsymbol{w}$ that minimizes 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 $\mathbf{z}_p$ is a convenient way of implementing LSTD(λ), and produces a family of methods spanning a spectrum that has Monte Carlo[6] methods at one end with $\lambda = 1$ and one-step TDL methods at the other with $\lambda = 0$ [36]. The matrix $\boldsymbol{A}$ should not be updated until the second move is made by a player. This is achieved by setting the eligibility trace $\mathbf{z}_p$ 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 λ is set to zero [hence TD(0)] and the features implement a direct lookup table such that, in the kth 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 [20].

Note that when N-tuple features are used, the matrix $\boldsymbol{A}$ 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 pseudoinverse of $\boldsymbol{A}$ is found.

V. Direct Classification

Lagoudakis and Parr [13] proposed that policies may be approximated directly using binary classification. This is achieved by labeling moves selected as positive and those not selected as negative. Their labeling is based on Monte Carlo rollouts, where moves are only labeled negative if they are statistically significantly worse than the best move (or moves), and labeled 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 labeling all selected moves as positive and all nonselected moves as negative. In this case, there is a possibility that a move is labeled negative in one game trajectory and positive in another. However, this type of noise is present for all linear classifiers studied.

Algorithm 1: LSTD(λ). Matrix $\boldsymbol{A}$ has dimension $n \times n$ and $\phi_p, \phi'_p, \mathbf{z}_p, \mathbf{b}, \mathbf{w}$ are vectors of dimension $n \times 1$.

```
input  : Game trajectories and parameter $\lambda$
output : weights vector $\mathbf{w}$
Set $\boldsymbol{A} \leftarrow \mathbf{0}, \mathbf{b} \leftarrow \mathbf{0}$ ;                          // initialize
for each game trajectory do
    $\mathbf{z}_p \leftarrow \mathbf{0}, \phi_p \leftarrow \mathbf{0}$ ;                     // for players $p = 1, 2$
    $p \leftarrow 1$ ;                                      // set player to start
    while $\phi_p$ is not terminal do
        $\phi'_p \leftarrow \phi_p$ ;                      // keep player previous state
        $\phi_p \leftarrow$ the next board position's features ;
        $\boldsymbol{A} \leftarrow \boldsymbol{A} + \mathbf{z}_p(\phi'_p - \phi_p)^T$ ; // note initially $\mathbf{z}_p \leftarrow \mathbf{0}$, so $\boldsymbol{A}$ is
        not updated
        $\mathbf{z}_p \leftarrow \lambda \mathbf{z}_p + \phi_p$ ;               // "eligibility vector" update
        $p \leftarrow$ next player to move ;
    end
    for $p = 1, 2$ do
        $\mathbf{b} \leftarrow \mathbf{b} +$ (winner)$\mathbf{z}_p$ ;     // winner is either $+1, 0,$ or $-1$
        $\boldsymbol{A} \leftarrow \boldsymbol{A} + \mathbf{z}_p(\phi'_p)^T$ ; // terminal feature vectors are all zeros
    end
end
$\mathbf{w} \leftarrow \boldsymbol{A}^{-1}\mathbf{b}$ ;                                          // use SVD.
```

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

$$\mathbf{w}\phi_j > 1 \qquad \forall j \in S \quad \text{and} \quad \mathbf{w}\phi_k < -1 \qquad \forall k \in N \tag{2}$$

where S and N are the selected and nonselected board states, respectively. The training data are unbalanced but may be compensated by weighting the positive labeled 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 minimize some surrogate of this, such as the mean square error. As for the direct classification method, each move is made by considering all the afterstates reachable by the set of legal single moves from the current board. The one chosen in the current game log is labeled as the correct move, and all others are labeled as incorrect. This is done in a pairwise manner: the selected move is paired in turn with each nonselected 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 maximizing player

$$[\mathbf{w}(\phi_j - \phi_k)] > 1 \qquad \forall j \in S^+, \qquad k \in N^+ \tag{3}$$

and similarly for the minimizing player

$$[\mathbf{w}(\phi_j - \phi_k)] < -1 \qquad \forall j \in S^-, \qquad k \in N^- \tag{4}$$

where S and N are the selected and nonselected board states, respectively, with the superscript indicating whether the player

[6] In the reinforcement learning sense of the term.

[7] http://www.csie.ntu.edu.tw/~cjlin/liblinear/

is maximizing or minimizing. Note that taking the pairwise difference between the features of each move tends to make the feature vectors more sparse and Section VIII-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 maximizing player. Using the board inversion approach we now only use (3), with the aim of continually learning that the weighted feature difference between the chosen move and each nonchosen 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.

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 γ_i (where $\gamma_i > 0\ \forall i$). Then, the probability that player i beats player j is given by

$$P(i \text{ beats } j) = \frac{\gamma_i}{\gamma_i + \gamma_j}. \quad (5)$$

Hence, the log probability that player i beats player j is proportional to the difference in their γ 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 modeled 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 modeled as being between teams of board features, where the ith feature has a strength γ_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 ϕ_{123}, while the other moves (i.e., the losers) lead to states with features ϕ_{34} and ϕ_{356}.

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

$$P(\phi_{123} \text{ beats } \phi_{34}, \phi_{356}) = \frac{\gamma_1\gamma_2\gamma_3}{\gamma_1\gamma_2\gamma_3 + \gamma_3\gamma_4 + \gamma_3\gamma_5\gamma_6}. \quad (6)$$

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

$$s(m) = \Pi_{i \in \phi_m} \gamma_i. \quad (7)$$

Then, according to the BT model, the probability that this move will be selected is given by

$$P(m) = \frac{s(m)}{\Sigma_{j \in B} s(j)}. \quad (8)$$

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 multivalued. For example, the symmetric N-tuples used below are multivalued, since each distinct N-tuple is replicated for all its eight symmetries, which means the associated feature values may occur between zero and eight times in a single board state.

Coulom also derived an MM algorithm for adjusting the γ values to maximize 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 γ is updated as follows:

$$\gamma_i^{t+1} = \frac{W_i}{\Sigma_j \dfrac{C_{ij}^t}{E_j^t}} \quad (9)$$

where W_i is the number of times that ϕ_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_j^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 γ_i is based on the previous values of all γ values. Hence, the above formulation suggests that each γ value must be updated serially rather than in batch mode, though Coulom shows how the E_j^t calculation can be shared between mutually exclusive γ 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 (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

[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.

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 preprocessing 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 multivalued features, but the results were significantly worse than when binarizing 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 maximize the likelihood of the training set or minimize the Bellman residuals. However, the question of which approach performs best in practice and by what margin can only be answered through empirical investigation, and *a priori* it is hard to predict whether preference learning would work better with or without board inversion. Postinvestigation, 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 as follows:

- 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(λ): least squares TDL, 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 nonselected 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, three weights for each square) and N-tuple features as described above, leading to a total of ten 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 ten unique values due to symmetry). The weights for this player were published in [40] and can be found conveniently in [9].

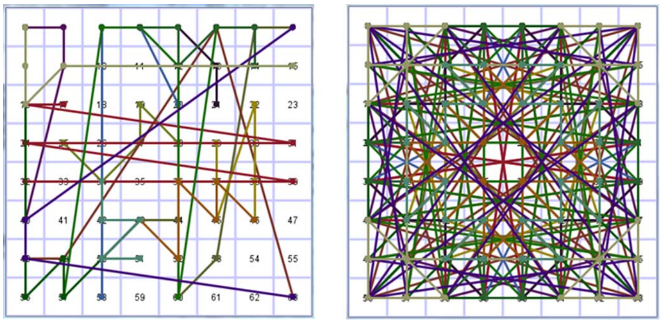

Fig. 2. The N-tuple system used in this study comprising 15 N-tuples, ranging between four and six inputs. Depicted without (left) and with (right) the symmetric expansions.

- Coev-WPC: The coevolved WPC from [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 coevolving players.
- ETDL-N-tuple: The N-tuple described in Section III-B, with an evolved structure but with weights trained using TDL. Note that all the N-tuple systems evaluated for matching expert play use exactly the same structure as specified in Table I and depicted in Fig. 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 [11] (the SJK comes from the authors' names). CTDL refers to their coevolved/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 online repository for this paper.[9]

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 II.

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 data sets 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

[9] See https://notendur.hi.is/~tpr/pref/

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 COEVOLVED WEIGHTED PIECE COUNTER (COEV-WPC) AND THE EVOLVED STRUCTURE, TDL-TRAINED N-TUPLE (ETDL-N-TUPLE)

			1-Tuple					N-Tuple					Heur-WPC	ETDL-N-Tuple
#discs	BF	#N	iPref	Pref	MM	LSTD	Classify	iPref	Pref	MM	LSTD	Classify		
1–16	7.1	73133	47.4	47.2	55.8	16.4	33.8	78.3	75.4	68.8	29.8	68.7	21.9	13.4
17–20	11.0	40045	17.6	31.0	20.8	8.0	20.6	52.4	47.3	43.0	15.2	31.8	2.6	15.5
21–24	11.5	42194	25.2	21.7	20.6	9.2	22.2	49.6	42.9	33.4	21.6	32.2	2.0	20.6
25–28	11.9	43796	28.1	25.7	25.8	11.0	21.8	45.1	42.1	31.1	20.7	34.1	5.2	22.9
29–32	11.7	42818	28.2	24.4	23.8	9.5	20.1	40.5	37.8	26.2	18.4	29.7	4.3	22.1
33–36	11.3	41319	28.2	23.7	23.9	9.0	20.3	40.1	36.6	28.0	17.6	30.2	6.8	24.9
37–40	10.6	38318	28.9	26.1	25.6	10.9	22.4	41.8	37.0	30.6	17.9	32.0	9.4	26.1
41–44	9.6	34308	29.4	28.7	28.3	13.5	25.9	41.5	37.7	31.6	20.4	32.5	14.3	29.6
45–48	8.4	29412	29.7	29.3	29.4	17.8	28.9	43.6	39.5	34.0	21.6	34.7	20.8	31.5
49–52	7.1	23784	30.5	30.6	31.4	25.9	30.6	44.0	41.0	35.3	24.1	36.4	27.2	35.8
53–56	5.5	17385	34.7	32.3	35.2	33.5	34.9	49.0	46.4	40.9	31.3	41.1	33.4	42.2
57–60	4.0	10960	40.8	38.5	41.4	42.9	40.3	53.9	51.8	46.5	39.0	48.3	38.7	49.5
61–64	2.5	3411	51.9	47.9	51.6	53.0	51.0	62.5	59.5	55.9	52.8	57.3	48.0	61.3
$\sum$	8.6	(437883)	33.8	33.0	34.5	18.4	28.6	53.0	49.4	42.5	25.1	42.7	17.6	27.0

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 pseudoinverse 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 6561 features, but only 5355 occur in the training set, which means finding the pseudoinverse of a 5355×5355 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 handcrafted 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 ten squares (30 weights) in total. However, this did not perform as well as the asymmetric version. The best setting for λ, for the LSTD, is 0.9 and will be used in all the experimental results reported.

Fig. 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 pairwise 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 toward 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 analyzed 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 Fig. 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 color of the corner occupier, and hence they would no longer show up as being poor moves.

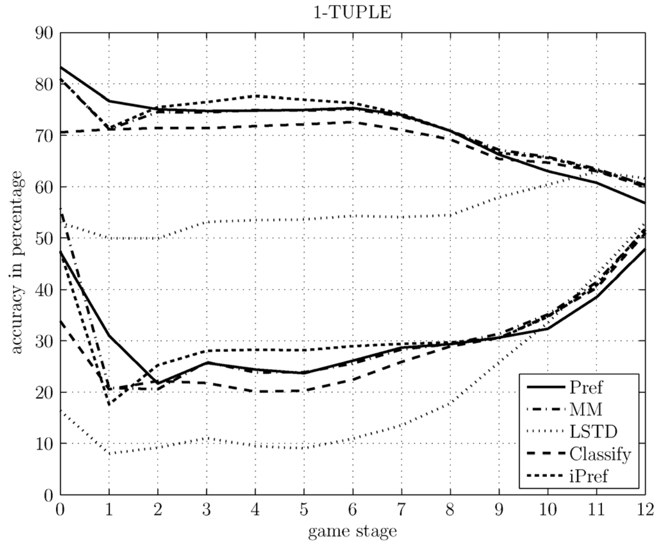

Fig. 3. Graphical representation of the move prediction accuracy given in Table II (bottom five curves) and test-set classification accuracy for all move options when using 1-tuple features.

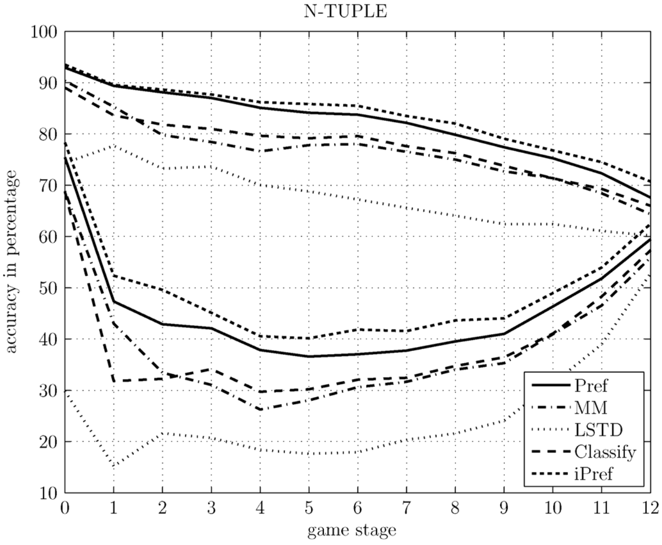

Fig. 4. Graphical representation of the accuracy in actual moves made (bottom five curves) and test-set classification accuracy for all move options using N-tuple features.

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 Fig. 4 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 toward 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.

D. Significance of Using Difference Vectors

The way we formulated the problem for the iPref and Pref methods involves taking the difference of the feature vectors before submitting the vector to the classifier. As previously explained, this is similar to the pairwise approximate policy iteration approach of Furnkrantz *et al.* [25], though they give pairs of vectors different labels rather than explicitly taking the difference of the feature vectors. Taking feature value differences is directly equivalent when using a linear classifier, as we are here, and may also be a good idea when using a nonlinear classifier.

As mentioned previously, an alternative (e.g., [13]) is to simply give the winning and losing moves different class labels. The difference vector approach is more efficient for any classification algorithm that is able to deal with sparse vectors (such as Liblinear for example), since many of the board positions being compared may differ only in a few features. The features which are the same will cancel each other out and hence not involve either any CPU effort at the point of classification, or any storage space when saving the data sets.

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 nonzero ones, while for the classification approach this was 64 (in fact, every input has exactly 64 nonzero features, since every square is in one of three possible states).

For the N-tuple features (6561 features) the saving was less pronounced, with the difference method having an average of 64.8 nonzero 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 data sets (since at each stage only one move is chosen from the possible ones available). We fed both the unbalanced data sets 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 Figs. 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 1000 positions.

The 1000 positions were chosen by running a vanilla Monte Carlo tree search *Othello* player (similar to the one described

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	1850	78.0%	neg	4,698
2	iPref-N-Tuple	1848	77.8%	inv	6,561
3	ETDL-N-Tuple	1803	73.3%	neg	6,561
4	Nash1-N-Tuple	1787	71.6%	neg	8,748
5	Nash2-N-Tuple	1765	69.2%	neg	8,748
6	SJK-ETDL-N-Tuple	1765	69.1%	neg	3,240
7	Pref-N-Tuple	1730	65.3%	neg	6,561
8	Coev-WPC	1597	49.5%	neg	64
9	Heur-WPC	1596	49.4%	neg	64
10	MM-N-Tuple	1573	46.5%	inv	6,561
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	6,561
15	Classify-1-Tuple	1364	23.4%	inv	192
16	LSTD-N-Tuple	1348	21.9%	neg	6,561
17	LSTD-1-Tuple	1293	17.1%	neg	192

in [43]) using a budget of 5000 simulations per move. This leads to a reasonable standard of play but with a significant random element. We then harvested 1000 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[10] 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.

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 subleagues 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 coevolutionary 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 through 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 maximizes the soft margin in this high-dimensional space while considering regularization, whereas MM does not aim for a maximal margin, and its "regularization" is limited to selecting the value of a small number of priors (three).
2) The N-tuple features are naturally multivalued, 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 1), we compared results on the training set to see if MM was overfitting (due to poorer regularization) but found that training set performance was very similar to test set performance.

To investigate 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 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.

[10] http://remi.coulom.free.fr/Bayesian-Elo/

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

Rank	Player	iPref-N-Tuple	SJK-CTDL-N-Tuple	Nash1-N-Tuple	ETDL-N-Tuple	Nash2-N-Tuple	Pref-N-Tuple	SJK-ETDL-N-Tuple	Rating	% Score
1	iPref-N-Tuple	0.0	1134.5	1286.5	1245.5	1337.5	1248.0	1372.0	1683	63.5
2	SJK-CTDL-N-Tuple	865.5	0.0	1031.0	1219.0	1111.5	1293.5	1349.0	1644	57.2
3	Nash1-N-Tuple	713.5	969.0	0.0	1044.5	1057.5	995.0	1212.0	1599	49.9
4	ETDL-N-Tuple	754.5	781.0	955.5	0.0	940.5	1164.0	1184.5	1588	48.2
5	Nash2-N-Tuple	662.5	888.5	942.5	1059.5	0.0	1011.5	1107.0	1583	47.3
6	Pref-N-Tuple	752.0	706.5	1005.0	836.0	988.5	0.0	840.0	1556	42.7
7	SJK-ETDL-N-Tuple	628.0	651.0	788.0	815.5	893.0	1160.0	0.0	1546	41.1

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

Rank	Player	SJK-CTDL-N-Tuple	iPref-N-Tuple	WPC-Coev	WPC-Heu	One-MM	One-Diff	One-Cls	One-LSTD	Rating	% Score
1	SJK-CTDL-N-Tuple	0.0	865.5	1807.5	1547.5	1863.5	1901.5	1921.5	1958.5	1906	84.8
2	iPref-N-Tuple	1134.5	0.0	1614.0	1178.5	1799.5	1842.0	1878.5	1915.5	1866	81.2
3	WPC-Coev	192.5	386.0	0.0	1089.0	1575.5	1578.0	1713.5	1746.0	1668	59.1
4	WPC-Heu	452.5	821.5	911.0	0.0	1297.0	1382.5	1296.5	1494.0	1632	54.7
5	One-MM	136.5	200.5	424.5	703.0	0.0	1128.0	1306.5	1629.5	1512	39.5
6	One-Diff	98.5	158.0	422.0	617.5	872.0	0.0	1246.0	1595.5	1483	35.8
7	One-Cls	78.5	121.5	286.5	703.5	693.5	754.0	0.0	987.0	1399	25.9
8	One-LSTD	41.5	84.5	254.0	506.0	370.5	404.5	1013.0	0.0	1335	19.1

The results suggest that the superior performance of preference learning compared to MM is largely due to its ability to handle multivalued 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 multivalued features into binary ones, doing this well takes time and effort. The fact that preference learning can handle multivalued 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 nonlinear 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 behavior 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.[11] This is an interesting and nonobvious 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 data set it can deal with. An interesting alternative would be to train the system using online backpropagation in order to deal with game-log data sets that are orders of magnitude larger than the one used in the current study, the tradeoff 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 TDL, 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 humanlike nonplayer characters in video games.

[11] Using a standard evaluation method of testing by playing games using 1-ply lookahead.

References

[1] A. Rimmel *et al.*, "Current frontiers in Computer Go," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 229–238, Dec. 2010.

[2] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo tree search in Hex," *IEEE Trans. Comput. Intell. AI Games* vol. 2, no. 4, pp. 251–258, Dec. 2010.

[3] J. A. Stankiewicz, "Knowledge-based Monte-Carlo tree search in Havannah," M.S. thesis, Faculty Humanities Sci., Maastricht Univ., Maastricht, The Netherlands, 2011.

[4] G. Tesauro, "Practical issues in temporal difference learning," *Mach. Learn.*, vol. 8, pp. 257–277, 1992.

[5] K. Chellapilla and D. Fogel, "Evolving neural networks to play checkers without expert knowledge," *IEEE Trans. Neural Netw.*, vol. 10, no. 6, pp. 1382–1391, Nov. 1999.

[6] M. Buro, "Statistical feature combination for the evaluation of game positions," *J. Artif. Intell. Res.* vol. 3, pp. 373–382, 1995 [Online]. Available: http://arxiv.org/abs/cs/9512106

[7] S. M. Lucas, "Investigating learning rates for evolution and temporal difference learning," in *Proc. IEEE Symp. Comput. Intell. Games*, 2008, DOI: 10.1109/CIG.2008.5035614.

[8] T. P. Runarsson and S. M. Lucas, "Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go," *IEEE Trans. Evol. Comput.*, vol. 9, no. 6, pp. 628–640, Dec. 2005.

[9] S. M. Lucas and T. P. Runarsson, "Temporal difference learning versus co-evolution for acquiring Othello position evaluation," in *Proc. IEEE Symp. Comput. Intell. Games*, 2006, pp. 52–59.

[10] E. Manning, "Using resource-limited Nash memory to improve an Othello evaluation function," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 1, pp. 40–53, Mar. 2010.

[11] M. Szubert, W. Jaskowski, and K. Krawiec, "On scalability, generalization, hybridization of coevolutionary learning: A case study for Othello," *IEEE Trans. Comput. Intell. AI Games*, vol. 5, no. 3, pp. 214–226, Sep. 2013, DOI: 10.1109/TCIAIG.2013.2258919.

[12] T. Runarsson and S. Lucas, "Imitating play from game trajectories: Temporal difference learning versus preference learning," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 79–82.

[13] M. Lagoudakis and R. Parr, "Reinforcement learning as classification: Leveraging modern classifiers," in *Proc. 20th Int. Conf. Mach. Learn.*, 2003, vol. 20, pp. 424–431.

[14] D. Stern, R. Herbrich, and T. Graepel, "Bayesian pattern ranking for move prediction in the game of Go," in *Proc. Int. Conf. Mach. Learn.*, 2006, pp. 873–880.

[15] R. Coulom, "Computing Elo ratings of move patterns in the game of Go," *Int. Comput. Games Assoc. J.*, vol. 30, no. 4, pp. 198–208, 2007.

[16] D. R. Hunter, "MM algorithms for generalized Bradley-Terry models," *Ann. Stat.*, vol. 32, pp. 384–406, 2004.

[17] M. Wistuba, L. Schaefers, and M. Platzner, "Comparison of Bayesian move prediction systems for computer Go," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 91–99.

[18] M. Lagoudakis and R. Parr, "Least-squares policy iteration," *J. Mach. Learn. Res.*, vol. 4, pp. 1107–1149, 2003.

[19] S. Bradtke and A. Barto, "Linear least-squares algorithms for temporal difference learning," *Mach. Learn.* vol. 22, no. 1–3, pp. 33–57, 1996.

[20] J. Boyan, "Technical update: Least-squares temporal difference learning," *Mach. Learn.* vol. 49, no. 2, pp. 233–246, 2002.

[21] S. Lucas, "Learning to play Othello with N-tuple systems," *Austral. J. Intell. Inf. Process.*, vol. 4, pp. 1–20, 2008.

[22] W. Cheng, J. Fürnkranz, E. Hüllermeier, and S. Park, "Preference-based policy iteration: Leveraging preference learning for reinforcement learning," in *Machine Learning and Knowledge Discovery in Databases*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2011, vol. 6911, pp. 312–327.

[23] A. Lazaric, M. Ghavamzadeh, and R. Munos, "Analysis of a classification-based policy iteration algorithm," in *Proc. 27th Int. Conf. Mach. Learn.*, 2010 [Online]. Available: http://www.icml2010.org/papers/303.pdf

[24] L. Li, V. Bulitko, and R. Greiner, "Focus of attention in reinforcement learning," *J. Universal Comput. Sci.* vol. 13, no. 9, pp. 1246–1269, 2007.

[25] J. Fürnkranz, E. Hüllermeier, W. Cheng, and S.-H. Park, "Preference-based reinforcement learning: A formal framework and a policy iteration algorithm," *Mach. Learn.*, vol. 89, pp. 123–156, 2012.

[26] J. Fürnkranz and E. E. Hüllermeier, *Preference Learning*. Berlin, Germany: Springer-Verlag, 2010.

[27] D. Gomboc, M. Buro, and T. Marsland, "Tuning evaluation functions by maximizing concordance," *Theor. Comput. Sci.* vol. 349, no. 2, pp. 202–229, Dec. 2005.

[28] G. N. Yannakakis, M. Maragoudakis, and J. Hallam, "Preference learning for cognitive modeling: A case study on entertainment preferences," *IEEE Trans. Syst. Man Cybern. A, Syst. Humans*, vol. 39, no. 6, pp. 1165–1175, Nov. 2009.

[29] H. P. Martinez, Y. Bengio, and G. N. Yannakakis, "Learning deep physiological models of affect," *IEEE Comput. Intell. Mag.*, vol. 8, no. 2, pp. 20–33, May 2013.

[30] M. Buro, "Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello," in *Proc. Games AI Res.*, 1997, pp. 77–96.

[31] W. W. Bledsoe and I. Browning, "Pattern recognition and reading by machine," in *Proc. Eastern Joint Comput. Conf.*, 1959, pp. 225–232.

[32] R. Rohwer and M. Morciniec, "A theoretical and experimental account of N-tuple classifier performance," *Neural Comput.*, vol. 8, pp. 629–642, 1996.

[33] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, pp. 5–32.

[34] M. Thill, P. Koch, and W. Konen, "Reinforcement learning with N-tuples on the game Connect-4," in *Parallel Problem Solving from Nature—PPSN XII*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7491, pp. 184–194.

[35] P. Burrow, "Hybridising evolution and temporal difference learning," Ph.D. dissertation, Dept. Comput. Sci. Electron. Eng., Univ. Essex, Colchester, U.K., 2011.

[36] R. S. Sutton and A. G. Barto*, Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[37] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2007, vol. 4630, pp. 72–83.

[38] R. Bradley and M. Terry, "Rank analysis of incomplete block designs. I. The method of paired comparisons," *Biometrika*, vol. 39, pp. 324–345, 1952.

[39] A. E. Elo*, The rating of chess players: Past and present*. New York: Arco Publishing, 1978.

[40] T. Yoshioka, S. Ishii, and M. Ito, "Strategy acquisition for the game 'Othello' based on reinforcement learning," *IEICE Trans. Inf. Syst.*, vol. 82, pp. 1618–1626, 1999.

[41] S. Samothrakis, S. Lucas, T. Runarsson, and D. Robles, "Coevolving game-playing agents: Measuring performance and intransitivities," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 213–226, Apr. 2013.

[42] N. Hansen, S. Mller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)," *J. Evol. Comput.*, vol. 11, no. 1, 2003, DOI: 10.1162/106365603321828970.

[43] D. Robles, P. Rohlfshagen, and S. M. Lucas, "Learning non-random moves for playing Othello: Improving Monte Carlo tree search," in *Proc. IEEE Conf. Comput. Intell. Games*, 2011, pp. 305–312.

Thomas Philip Runarsson (S'98–M'01) received the M.Sc. degree in mechanical engineering and the Dr. Scient. Ing. degree from the University of Iceland, Reykjavik, Iceland, in 1995 and 2001, respectively.

Since 2001, he has been a Research Professor at the Applied Mathematics and Computer Science Division, Science Institute, University of Iceland and adjunct at the Department of Computer Science, University of Iceland. His present research interests include evolutionary computation, global optimization, and statistical learning.

Simon M. Lucas (M'98–SM'07) received the B.Sc. degree from Kent University, Canterbury, Kent, U.K., in 1986 and the Ph.D. degree from the University of Southampton, Southampton, U.K., in 1991.

He is a Professor of Computer Science at the University of Essex, Colchester, Essex, U.K., where he leads the Game Intelligence Group. His main research interests are games, evolutionary computation, and machine learning, and he has published widely in these fields with over 130 peer-reviewed papers. He is the inventor of the scanning N-tuple classifier.

Prof. Lucas is the founding Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES.