Information and Media Technologies 2 (4) : 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22 (5) : 461–471 (2007)

論 文

Technical Papers

A Study of Artificial Neural Network Architectures for Othello Evaluation Functions

Kevin J. Binkley Department of Information and Computer Science, Keio University
kbinkley@soft.ics.keio.ac.jp, http://www.soft.ics.keio.ac.jp/~kbinkley/

Ken Seehart seehart@neuralintegrator.com

Masafumi Hagiwara Department of Information and Computer Science, Keio University
hagiwara@soft.ics.keio.ac.jp, http://www.soft.ics.keio.ac.jp/~hagiwara/intro-hagi.htm

keywords: artificial neural network, temporal difference learning, reinforcement learning, board games, othello

Summary

In this study, we use temporal difference learning (TDL) to investigate the ability of 20 different artificial neural network (ANN) architectures to learn othello game board evaluation functions. The ANN evaluation functions are applied to create a strong othello player using only 1-ply search. In addition to comparing many of the ANN architectures seen in the literature, we introduce several new architectures that consider the game board symmetry. Both embedding the game board symmetry into the network architecture through weight sharing and the outright removal of symmetry through symmetry removal are explored. Experiments varying the number of inputs per game board square from one to three, the number of hidden nodes, and number of hidden layers are also performed. We found it advantageous to consider game board symmetry in the form of symmetry by weight sharing; and that an input encoding of three inputs per square outperformed the one input per square encoding that is commonly seen in the literature. Furthermore, architectures with only one hidden layer were strongly outperformed by architectures with multiple hidden layers. A standard weighted-square board heuristic evaluation function from the literature was used to evaluate the quality of the trained ANN othello players. One of the ANN architectures introduced in this study, an ANN implementing weight sharing and consisting of three hidden layers, using only a 1-ply search, outperformed a weighted-square test heuristic player using a 6-ply minimax search.

1. Introduction

In recent years, many very strong computer board game players have been developed. Perhaps the most famous is IBM's Deep Blue which defeated the world champion Kasparaov in 1997 [Campbell 02]. Key to Deep Blue's success was its ability to search the tree of chess positions very deeply due to its special hardware, its very complex human-tuned game position evaluation function, and its grandmaster game database. Game playing programs of better than human world class level have also been developed for checkers [Schaeffer 00], othello [Buro 99, Buro 02], and backgammon [Tesauro 92, Tesauro 95, Tesauro 02]. Except for backgammon all of these world-class programs relied quite heavily on human expertise to tune their game position evaluation functions, and use a very deep game-tree search.

As described in [Tesauro 95], TD-Gammon learned a backgammon evaluation function using self-play. The evaluation function was an artificial neural network (ANN) using a straightforward board encoding and trained through temporal difference learning (TDL) [Sutton 88, Sutton 98]. The resulting evaluation function, with 1-ply search, was able to play at human expert levels. With a deeper search, and some human touches, TD-Gammon was able to beat the best human players.

Since the success of TDL on backgammon, there have been many studies aimed at using TDL for train-

Information and Media Technologies 2 (4) : 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22 (5) : 461–471 (2007)
© Japanese Society for Artificial Intelligence

ing evaluation functions for other games. KnightCap [Baxter 98] used a TDL method to tune a chess evaluation function. In [Schaeffer 01], TDL was applied to successfully train the weights for the evaluation function of the world-class Chinook checkers program. However, these evaluation functions take advantage of a considerable amount of human-chosen features, rather than using a straightforward board input encoding.

ANNs have been used in many studies of othello evaluation functions. Although most have used straightforward game board encodings, the ANN architectures have varied greatly. In [Yoshioka 99], TDL was combined with self-play, and it was found that a fully-connected ANN with a single hidden-layer did not perform as well as a normalized Gaussian network. A recent study [Chong 05] used an ANN with a spatial processing layer similar to the successful checkers ANN architecture [Chellapilla 01, Fogel 00] to observe the evolution of ANNs learning othello evaluation functions. In [Singer 01], a fully-connected ANN with three inputs per square and a single hidden layer of three hidden nodes was used for co-evolving ANN evaluation functions. Recently, a competition for 1-ply othello ANN evaluation functions has been started at the IEEE 2006 Congress on Evolutionary Computation [Lucas 06]. The competition supported fully connected ANNs and a one input per square encoding.

In this study, we investigate many ANN architectures found in the literature, and introduce several new architectures implementing weight sharing. The final goal being a high-quality othello evaluation function. The resulting ANN evaluation function is applied to create a strong othello player using only 1-ply search. The question comes to mind: if today's computers can search so deep, do we need to improve the quality of the game evaluation function? One application might be an internet game server, trying to serve thousands of clients at once. Another, perhaps more compelling reason is that signs of diminishing returns for deeper search have been reported in the literature, for othello, chess, and suggested for checkers [Junghanns 97]. As diminishing returns are reached, the quality game evaluation function will become the critical component in computer game play.

We only consider architectures that use straightforward board encodings, with the goal of forcing the ANN to learn the game board features. Each board square is encoded into one, two, or three inputs. In previous studies, one input per square is most common [Yoshioka 99, Chong 05, Runarsson 05, Leouski 96], two inputs per square for a go ANN has been used [Mayer 05] (we believe go and othello ANNs should be able to benefit similar architectures), and there is at least one study using three inputs per square for othello [Singer 01].

A common feature of many board games is game board symmetry, there are eight symmetries of the game board for othello (and go). Game board symmetry has been taken into consideration in previous studies for othello [Leouski 96] and go [Schraudolph 01]. We investigate both incorporating board symmetry into the ANN architecture using both weight sharing [LeCun 98] and outright symmetry removal.

In addition to the ANN architectures incorporating symmetry, the standard fully connected ANN, and an ANN with a spatial processing layer [Chellapilla 01, Chong 05] are also studied. Architectures with various numbers of hidden nodes and hidden layers are studied. In total, 20 different ANN architectures are compared.

The rest of this paper is outlined as follows. Section 2 gives a brief introduction to TDL. In Section 3 the details of the ANN architectures investigated are discussed. Section 4 discusses the common settings used across all experiments. In Section 5, the experimental results are given, and in Section 6 concluding comments are given.

2. Temporal Difference Learning

In this study, ANN game player's are created from ANN evaluation functions. Games are played using 1-ply search. When it is the ANN player's turn to move, the player's ANN function is applied to evaluate all possible next positions and the move leading to the most favorable evaluation is chosen. Initially, all ANN players start out with a random set of weights. As games are played, the ANN evaluation function weights are adjusted using temporal difference learning.

Temporal difference learning works as follows. At the end of a board game, we are given a sequence of positions and the final game result. Let us call this sequence of positions $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$ and the result z. The game sequence is broken into two sequences, one sequence where the first player moves and one sequence where the second player moves. Due to the 1-ply search, the positions evaluated by the first player

Information and Media Technologies 2(4): 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22(5): 461–471 (2007)
© Japanese Society for Artificial Intelligence

are the even positions, $\vec{x}_2, \vec{x}_4, \ldots$, and those evaluated by the second player are the odd positions, $\vec{x}_3, \vec{x}_5, \ldots$. Let $\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_{m_1}$ be the positions evaluated by the first player, and $\vec{v}_1, \vec{v}_2, \ldots, \vec{u}_{m_2}$ be the positions evaluated by the second player.

The fact that the game result, z, is not known until the end of the position sequence creates a problem for many supervised learning procedures, where each of the positions is trained with the game result. Suppose the position $\vec{x}_2$, yields a game result, z. Training the evaluation function to yield z for position, $\vec{x}_2$ is not likely to be appropriate, since there can be many mistakes made by either player between position $\vec{x}_2$ and the end of game. The game result may only be very weakly correlated with the earlier positions in the game. This is the familiar temporal credit assignment problem.

A nice solution to this problem is the family of temporal difference methods, TD(λ) [Sutton 88, Sutton 98], known as temporal difference learning (TDL). In this study, TD(0) method is used exclusively. Consider the sequence of m_1 positions evaluated by player one, $\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_{m_1}$, where $f(\vec{w}, \vec{u}_t)$ is the evaluation of the t^{th} position. The t^{th} position results in the following change in the weight vector, $\Delta\vec{w}_t$.

$$\Delta\vec{w}_t = \alpha(f(\vec{w}, \vec{u}_{t+1}) - f(\vec{w}, \vec{u}_t))\nabla_w f(\vec{w}, \vec{u}_t) \quad (1)$$

Where $f(\vec{w}, \vec{u}_{m_1+1}) = z$ and α is a learning rate constant.

The weights are updated after each game played as follows.

(1) Calculate $\vec{a} = \sum_{t=1}^{m_1} \Delta\vec{w}_t$, for player one's moves using $\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_{m_1}$, and z.

(2) Calculate $\vec{b} = \sum_{t=1}^{m_2} \Delta\vec{w}_t$, for player two's moves using $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_{m_2}$, and z.

(3) Update $\vec{w}$: $\vec{w} = \vec{w} + \vec{a} + \vec{b}$

The gradient of the ANN function, $\nabla_w f(\vec{w}, \vec{u}_t)$ is calculated using backpropagation [Rojas 96].

3. Artificial Neural Network Architectures

We will assume the reader is familiar with the fully connected multi-layer ANN [Bishop 96, Rojas 96]. The details of the other architectures studied are discussed in the following sections.

3·1 *Symmetry by Weight Sharing*

Many board games have symmetries. In games like tic-tac-toe, gomoku, othello, and go, there are 8 symmetries which correspond to flipping the board horizontally, vertically, or across a diagonal. The symmetries for a 3 by 3 board are shown in Figure 1. To see the symmetries, pretend you are standing on a square near one of the arrows, and walk straight in the direction of the arrow. When the end is reached, return to where you started; move over to the nearest square not visited yet and walk in the direction of the arrow again; repeat this process until you have visited all squares. The order in which the squares are visited reflect the particular symmetry, we call the order a symmetry permutation. The symmetry labeled 1 in Figure 1 has the permutation 1-4-7-2-5-8-3-6-9, the symmetry labeled 2 has the permutation 3-6-9-2-5-8-1-4-7. In the remainder of this section, symmetry by weight sharing will be illustrated with a 3 by 3 game board, the ideas can be extended to arbitrary sized game boards in a similar manner.

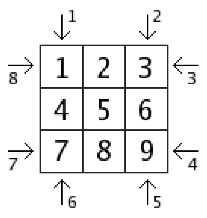


Fig. 1 The arrows illustrate the 8 symmetries of a 3 by 3 board.

Our symmetry by weight sharing method is implemented in the first hidden layer of the ANN. In this layer, each node represents a symmetry permutation. A minimum of 8 hidden nodes are always created, one for each symmetry permutation. These 8 hidden nodes are called a symmetry set. Within a symmetry set the hidden nodes share weights. In the example weight sharing ANN shown in Figure 2, there are 9 input nodes, 16 nodes in the first hidden layer, 3 nodes in the second hidden layer, and 1 output node. Hidden nodes N_{10} to N_{17} all have the same set of weights; however, the weights on the connections from the input nodes are permuted corresponding to the symmetry permutation that the node is representing. The weights on the bias and the output weights to the next layer are shared within the symmetry set, but not permuted.

In the first hidden layer, an arbitrary number of symmetry sets may be created. To do so, another set of weights to be shared is allocated and the weights are permuted as described above. Figure 2 illustrates 2 symmetry sets in the first hidden layer. Additional

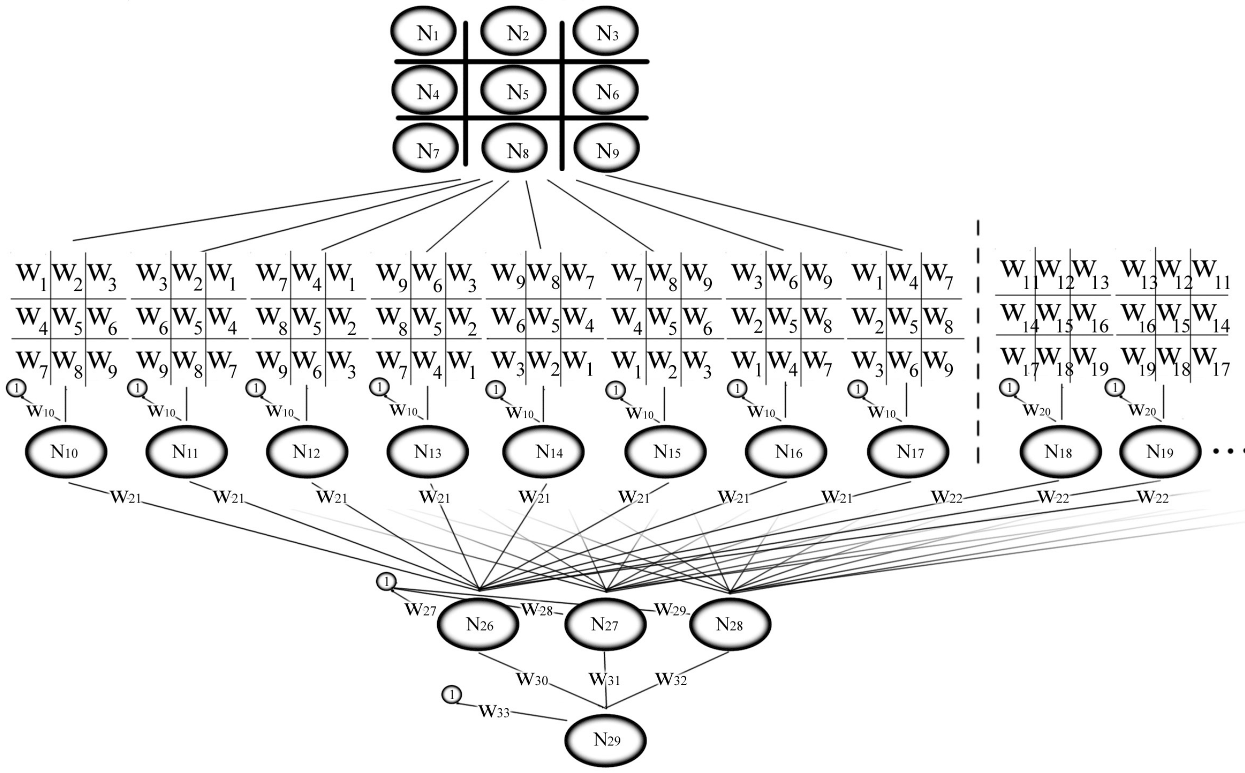


Fig. 2 A simple ANN illustrating symmetry by weight sharing. There are two symmetry sets in the first hidden layer. Unique weights are assigned an id and shared within a symmetry set (see text for a detailed explanation). For each node in the first hidden layer, the 9 weight ids for the connections from the input nodes are shown in a grid above the hidden node in a layout corresponding to the input nodes. N_{10} to N_{17} share the same weights, however the order in which they are connected to the input nodes is permuted. A similar fully connected network would consist of 215 unique weights, this network with weight sharing consists of only 33 unique weights. Biases are indicated with a 1 in a small circle. For clarity, the second symmetry set is not fully shown: N_{20} to N_{25} are analogous to N_{12} to N_{17}. Also, only weights from the first hidden layer to the first hidden node in the second hidden layer are labeled. W_{23} and W_{25} connect the first symmetry set to N_{27} and N_{28} respectively; W_{24} and W_{26} connect the second symmetry set to N_{27} and N_{28} respectively.

hidden layers after the first hidden layer are connected in the normal fully connected manner.

In terms of the number of unique weights, the weight sharing ANN implements a simpler network function than the fully connected ANN. The symmetry by weight sharing ANN in Figure 2 has only 33 ($= (9+1)*2+(2+1)*3+(3+1)*1$) weights (including bias connection weights). A similar fully connected ANN would have 215 ($= (9+1)*16+(16+1)*3+(3+1)*1$) weights.

To get an idea of how the weight sharing ANN evaluation function behaves with respect to symmetries, let us consider the familiar game of tic-tac-toe. Tic-tac-toe positions with just an X in a corner will evaluate to the same value due to weight sharing. In a more complex example, consider the 8 different positions with only an X in a corner and an O next to it. These also evaluate to the same value due to weight sharing. However, the position with an X in a corner and an O next to the opposite corner (in chess, a knight's move away), will evaluate to a different value.

3・2 Input Representation

There are choices as to how the game board is represented. A popular straightforward choice is the one input per square method, a 0 indicating no piece, a 1 indicating player one's piece, a -1 indicating player two's piece.

We compare three input representations: one, two, and three inputs per square. The one input per square encoding is discussed above. In the two inputs per square encoding, each input indicates whether the player has a piece on the square or not: no piece on the square is $(-1,-1)$, player one's piece is indicated by $(1,-1)$, and player two's piece is indicated by $(-1,1)$. For the three inputs per square encoding, the board square is encoded as follows: player one's piece $(1,-1,-1)$, player two's piece $(-1,1,-1)$, and

Information and Media Technologies 2 (4) : 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22 (5) : 461–471 (2007)

no piece $(-1,-1,1)$.

An example of a network architecture using a one input per board square representation and consisting of only 9 squares was shown in Figure 2. For an othello game, using a one input per square encoding, there are 64 inputs. For the two and three input per square cases, each of the 64 squares is imagined to contain and input vector of size two or three respectively, resulting in 128 or 192 inputs respectively.

Many board games have an important symmetry that we can take advantage of in choosing our input representation: an input indicating which player is to move is not necessary. Let $\vec{x}$ be an othello position with white to move. If the black and white stones are swapped to give position $\vec{x}'$, with black to move, then the two evaluations will be the same. More specifically, $f(\vec{w},\vec{x}) = f(\vec{w},\vec{x}')$. We use this symmetry to avoid an input indicating player to move and train only one evaluation function for both players. As positions are evaluated and trained, all positions are converted into a *player to move* position.

3·3 Symmetry Removal

Symmetry can also be removed from the game positions before being fed to the ANN. It is clear that all symmetrical positions should receive the same evaluation from the evaluation function. The evaluation function only needs to be trained on one of the symmetries and give the correct result for this one. Thus, the input space can be reduced by a factor of 8. The trick is to make sure that all of the eight symmetries map to the same symmetry before being passed to the evaluation function.

The algorithm is outlined here. Given a position $\vec{x}$, all eight symmetries, $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_8$, are generated (ignoring degenerate cases). From the 8 symmetries, the lowest symmetry is chosen by sorting $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_8$ by comparing the components of each vector, in a left to right order. After sorting, the lowest symmetry (the the zeroth entry of the sort result), $\vec{x}_l$, is passed to the evaluation function. The symmetry removal step is, necessary during both training and evaluation.

A little thought shows that symmetry removal in addition to the symmetry by weight sharing (Section 3·1), will yield exactly the same result as symmetry by weight sharing alone. This is because an ANN with symmetry by weight sharing will respond to all symmetrical positions exactly the same.

3·4 Spatial Processing Layer

In [Chong 05] an architecture with a spatial processing layer is used for an othello ANN. This architecture was based on the architecture implemented for checkers in [Chellapilla 01]. The first hidden layer of the ANN was a spatial processing layer consisting of all overlapping 3x3 squares, 4x4 squares, 5x5 squares, 6x6 squares, 7x7 squares, and 8x8 squares. On an 8x8 othello board, there is one 8x8 square, four 7x7 squares, nine 6x6 squares, 16 4x4 squares, 25 5x5 squares, and 36 3x3 squares. The first hidden layer has a node for each of these squares, for a total of 91 nodes. The second and third hidden layers were fully connected consisting of 40 and 10 hidden nodes respectively. There was one output node taking inputs from the last hidden layer, and a special input encoding the difference between the number of black and white pieces on the board.

For our study, an ANN with a spatial processing layer exactly as described above was implemented except that the special input encoding the difference between the number of white and black stones on the board was left out. In our opinion this is too much a human-chosen, game-specific feature that should be learned by the ANN. Thus, our spatial processing architecture is identical to the checkers ANN architecture with the exception that we have 64 board input squares instead of the 32 for checkers.

4. Experimental Settings

In this section we discuss the common experimental parameter settings used in this study. Unless otherwise mentioned, these parameters are used in all experiments.

The game studied in this paper is othello. There are many good references discussing the rules of othello, such as http://en.wikipedia.org/wiki/Reversi.

The ANN connection weights are initialized randomly to a value in the range $(-1/\sqrt{f}, 1/\sqrt{f})$, where f is the number of connections coming into the node that the connection terminates at [Bishop 96]. All hidden nodes and the output node use hypertangent activation functions and have biases. For TDL, in equation (1), the learning rate, α, was fixed at 0.001. The TDL weight updates are applied to the ANN using back-propagation [Bishop 96].

The game result value is needed for TDL. It is possible to train an evaluation function to yield 1 for a win, -1 for a loss, and 0 for a draw. However, for some games, such as othello or go, it is relatively easy to give more information to the learning process. For othello, wins by fewer pieces can be given a value closer to 0. In this study, the game result value is determined by $2(1+e^{-5(b-w)/(w+b)})^{-1}-1$, where b and w, respectively represent the number of black and white pieces on the board at the end of the game.

In efforts to further increase the diversity of game positions, the ANNs competed against each other from various game starting positions (similar to exploring starts in [Sutton 98]). In othello when symmetry is considered, after move 1, there is only one unique position. After moves 2, 3, and 4, there are 3, 14, and 60 unique positions respectively. Thus, up to the forth move, there 79 different possible positions. During learning the game start position is picked randomly (without replacement) from the 79 positions. The players always play two games from each position, one as black and one is white. This will alleviate bias in the case that some of the 60 positions are unfavorable to a player.

An experiment consists of two ANNs initialized randomly at the start as described above. The ANNs play games with each other alternating colors. At the end of each game the ANN weights are updated accordingly using TDL as described in Section 2. Preliminary experiments showed that training in pairs performed better than self-play.

In each experiment, training was performed in time periods of 7,680 games, for 40 time periods, a total of 307,200 games. At the end of each time period, 400 games against a standard weighted-square heuristic test player (defined below) were played, with the players alternating who moves first. The ANN player used a search depth of 1-ply, whereas the test player was allowed a 4-ply minimax search depth. Each experiment was run 10 times and an average of the 10 results is presented. In all graphs, a dot represents the average, and error bars (if present) indicate the standard deviation of the 10 results. The solid line represents a moving average of 10 time periods. All graphs show the percent of games lost against a standard heuristic test player.

The standard test player uses the heuristic evaluation function from [Yoshioka 99]. The squares are assigned static weights as shown in Figure 3, and summed according to the following equation (2), where σ_{noise} was set to 5. Note that in [Yoshioka 99], σ_{noise} was set to 10, and a minimax search of 2-ply was used

1	100	-25	10	5	5	10	-25	100	8
	-25	-25	2	2	2	2	-25	-25	
	10	2	5	1	1	5	2	10	
	5	2	1	2	2	1	2	5	
	5	2	1	2	2	1	2	5	
	10	2	5	1	1	5	2	10	
	-25	-25	2	2	2	2	-25	-25	
57	100	-25	10	5	5	10	-25	100	64

Fig. 3 Othello standard weighted-square heuristic evaluation function.

for both players.

$$F(\vec{w},\vec{x}) = \sum_{i=1}^{64} w_i x_i + N(0,\sigma_{noise}) \qquad (2)$$

5. Experimental Results

Table 1 gives an overview of the ANN architectures studied. The two letter codes describe ANN architecture discussed in Section 3: Weight Symmetry (WS), Fully Connected (FC), Symmetry Removal (SR), and Spatial Processing (SP). As an example, consider the WS 128-4×8-1 network. This is a weight sharing network with 128 inputs, 32 hidden nodes in the first hidden layer, and one output node. There are two inputs per board square. The 4×8 notation indicates that there are 4 sets of symmetry nodes, each set containing 8 nodes that share weights. The largest network, in terms of the number of nodes, is the WS 192-16×8-16-8-1 ANN. It is a weight sharing network with 3 inputs per board square, and 3 hidden layers, containing 128 nodes in the first hidden layer, 16 in the second hidden layer, and 8 in the third hidden layer.

The number of connections in each ANN and the number of unique weights are also shown in Table 1, to give an idea of the complexity of each of the ANNs. The CPU time (measured on an AMD Athlon 64 3200+ processor), for one time period, is also given for selected experiments.

For the WS 192-4×8-1 ANN, there are approximately 15.6 games played per second. One complete run of 40 time periods takes about 5 hours and 40 minutes of computer time. To finish an experiment of 10 runs, about 2.4 days of computer time is required.

Information and Media Technologies 2 (4) : 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22 (5) : 461–471 (2007)

Table 1 ANN characteristics and selected CPU times (in seconds) for experiments performed in this study.

ANN	# Conn.	# Wgts	CPU
WS 64-4×8-1	2113	265	177
WS 128-4×8-1	4161	521	338
WS 192-4×8-1	6209	777	508
WS 192-8×8-1	12417	1553	966
WS 192-8×8-4-1	12617	1585	—
WS 192-8×8-8-1	12881	1625	—
WS 192-16×8-1	24833	3105	2040
WS 192-16×8-16-1	26785	3377	2400
WS 192-16×8-16-8-1	26913	3505	2373
FC 192-4-1	777	777	88
FC 192-16-1	3105	3105	286
FC 192-32-1	6209	6209	—
FC 192-32-8-1	6449	6449	—
FC 192-64-1	12417	12417	1123
SR 192-4-1	777	777	88
SR 192-16-1	3105	3105	286
SR 192-32-1	6209	6209	—
SR 192-32-8-1	6449	6449	—
SR 192-64-1	12417	12417	1119
SP 192-91-40-10-1	9316	9316	968

For the largest of the architectures, WS 192-16×8-16-8-1, the experiment took about 11 days of computing time.

Our software is written in C++, and compiled on both Unix and Microsoft Windows systems. The experiments were run as background processes on several computers to speed up the experimental process. Completed experiments were recorded in a database for later analysis. The reader is invited to play against our othello ANN players at http://neuralplay.com/play.

5・1 *Inputs per Square*

Figure 4 gives the experimental results for ANNs architectures with one, two, and three inputs per board square. There is a steady increase in quality of the experimental result as the number of inputs per board square is increased. Even when computation time is taken into consideration, it is advantageous to use the two, or three inputs per board square encodings. Referring to the times shown in Table 1, after 20 time periods, the two input per square ANN has taken a comparable amount of computation time to the one input per square ANN after 40 time periods. Yet, after 20 time periods, the two input per square ANN has clearly achieved a better result. The difference between the two and three input per square ANNs is

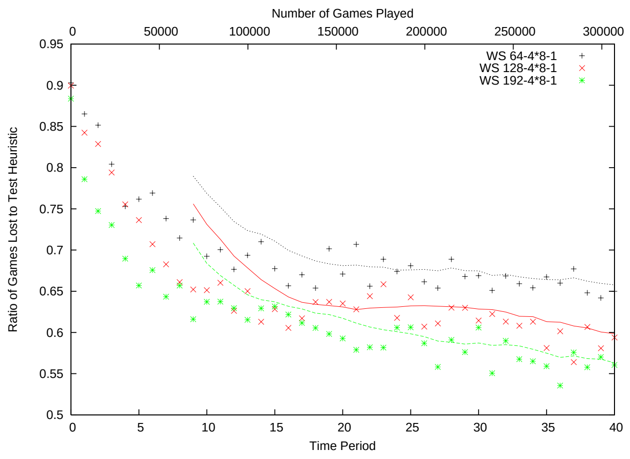


Fig. 4 An experiment varying the number of inputs per square for the board encoding. The ANNs using a two or three inputs per square encoding clearly outperform the ANN using a one input per square encoding. Solid lines represent a 10 period moving average.

not as pronounced, but it appears that three inputs per square is no worse than two inputs per square. Also, although the graph is not conclusive, we believe it is likely that the one, two, and three input per square ANNs are asymptotically approaching different results, with the three input per square ANN producing the best result.

5・2 *Fully Connected Network*

The results for the standard fully connected ANN experiments are given in Figure 5. The fully connected ANNs steadily improve with an increase in the number of hidden nodes in the first hidden layer. Also, as observed for other architectures, an additional layer is a must for improving results. With approximately half as many weights and connections (Table 1), the FC 192-32-8-1 ANN outperforms the FC 192-64-1 ANN.

5・3 *Symmetry Removal*

The results for the symmetry removal experiments are given in Figure 6. The FC 192-32-8-1 ANN is also shown for comparison. One can hope that symmetry removal will perform better than the fully connected architectures. However, when comparing with Figure 5, one finds that the results are mixed. On the architectures with one hidden layer and with 4 or 16 hidden nodes, the results are roughly equal or symmetry removal performs slightly better. However, on the best performing architecture with an additional hidden layer, symmetry removal performs worse.

Information and Media Technologies 2 (4) : 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22 (5) : 461–471 (2007)

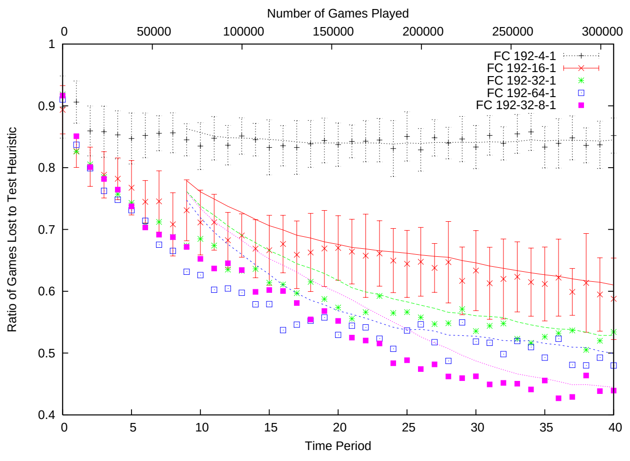


Fig. 5 The results of the fully connected ANNs. Results improve as the number of hidden nodes is increased. An additional hidden layer strongly improves results. Solid lines represent a 10 period moving average.

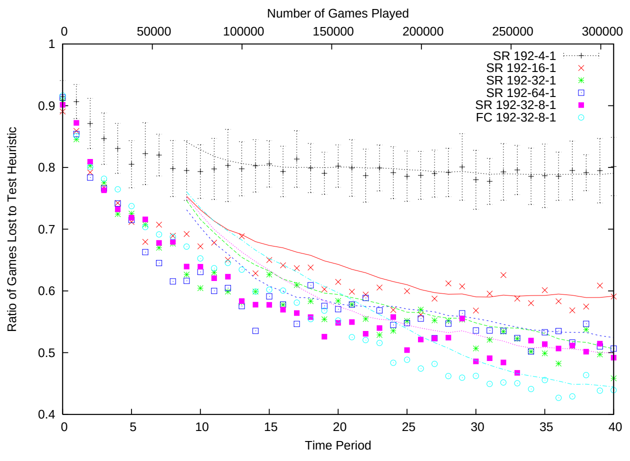


Fig. 6 Results for the symmetry removal ANNs. The FC 192-32-8-1 ANN is shown here for comparison. Solid lines represent a 10 period moving average.

We offer one possible reason as to why symmetry removal may perform worse than a fully connected network. In ANNs with symmetry removal, symmetric positions are mapped to one of the 8 symmetries arbitrarily. Due to this arbitrary mapping, similar othello positions may not get mapped to similar ANN inputs. Two positions might differ from only a single stone, they would be similar, but they might be mapped to completely different symmetries before being presented to the ANN. We note that a SR 2-1 ANN (2 inputs, 1 output) can solve the simple XOR problem, whereas an FC 2-1, clearly can not solve it. SR works very well for small problems because it forces differentiation of the roles of the nodes (in the case of XOR, the two inputs), but we believe this benefit diminishes rapidly as the size of the input domain increases.

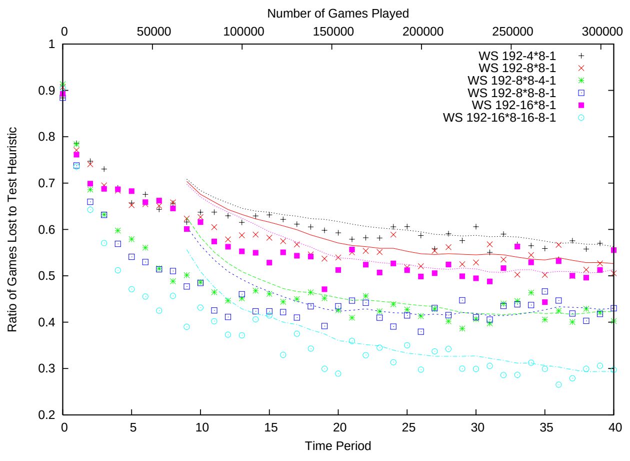


Fig. 7 The results of the weight sharing ANNs. Results improve as the number of symmetry sets is increased from 4 to 16. Additional hidden layers strongly improve results. Solid lines represent a 10 period moving average.

5·4 *Symmetry by Weight Sharing*

In Figure 7, results are given for ANNs using weight sharing with 4, 8, and 16 symmetry sets. The number of hidden layers and hidden units after the symmetry set layer are also varied. First we note that there is an increase in result quality as the number of symmetry sets are increased, from 4, to 8, and finally to 16 symmetry sets.

Second, the results show that with an additional hidden layer after the symmetry sets the results are greatly improved. It is clear the WS 192-8×8-4-1 ANN outperforms WS 192-16×8-1 ANN, despite having roughly half as many weights and connections. Also, it is interesting that no benefits are observed if the number of nodes in the second layer are doubled: the WS 192-8×8-8-1 ANN does not offer further benefit over the WS 192-8×8-4-1 ANN.

Although we did not have enough time in this study to investigate many of the larger, possibly interesting weight sharing ANNs, we picked the three hidden layer, WS 192-16×8-16-8-1 architecture to see if there were nearby limits to what a large weight sharing architecture could do. The large WS 192-16×8-16-8-1 ANN performed the best of all architectures in this study. After the 40 time periods of training, Figure 7 indicates that the result might improve with further training.

5·5 *Comparison of Major ANN Architectures*

Figure 8 gives the results for the spatial processing ANN as described in Section 3·4, along with the best performing fully connected and weight sharing ANNs.

Information and Media Technologies 2 (4) : 1129–1139 (2007)
reprinted from: Transactions of the Japanese Society for Artificial Intelligence 22 (5) : 461–471 (2007)

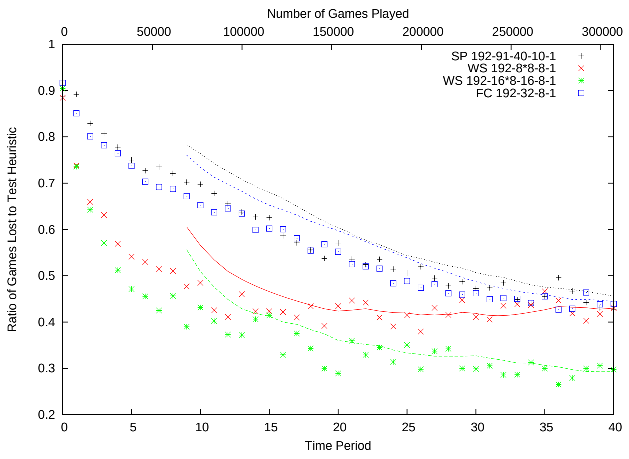


Fig. 8 A comparison of the major ANN architectures studied. The spatial processing ANN is compared to the fully connected ANN and the weight sharing ANN. The fully connected ANN with one additional hidden layer performs as well as the larger spatial processing ANN. The weight sharing ANN performs even better. Solid lines represent a 10 period moving average.

Surprisingly, we find that there is no benefit offered by the spatial processing network compared to smaller fully connected network. The fully connected network, FC 192-32-8-1, has 6,449 connections and weights. The spatial processing network, SP 192-91-40-10-1, has 9316 connections and weights (Table 1). We conclude that rather than using the larger, more complex to implement spatial processing network, one can get faster training and fewer weights, by using a simpler fully connected network for the othello evaluation function task.

The weight sharing network, WS 192-16×8-16-8-1, with only 3,505 weights, performs even better than the FC 32-8-1 ANN. We note that due to the 26,913 connections, compared to only 6,449 for the FC 32-8-1 ANN, training with backpropagation as done in this study takes about 4 times longer. A look at Figure 8 to compare time period 10 to time period 40 gives a rough estimate if extra training time is worthwhile. According to the figure, it appears that even after time period 10, the WS 192-16×8-16-8-1 ANN is yielding a better result than the FC 32-8-1 ANN.

5·6 *Measurement of Quality of Play*

In this section, we measure how well the ANN players are able to play, in a manner that future researchers might be able to use as a benchmark to compare their own othello playing programs. The test heuristic selected for measurement is the weighted-square heuristic from Section 4, with $\sigma_{noise} = 1.0$ in equation (2). The ANNs trained in our study were compared to

Table 2 The percent of games won, lost, and drawn by the average WS 192-16×8-16-8-1 ANN with 1-ply search vs. the test heuristic with increasing search depth. The average 1-ply ANN outperforms a 6-ply test heuristic with $\sigma_{noise} = 1.0$.

	Black			White		
ply	Win	Draw	Loss	Win	Draw	Loss
1	91.4	1.6	7.0	80.2	0.6	19.2
2	81.4	1.2	17.4	80.6	1.4	18.0
3	75.0	1.0	24.0	74.4	1.8	23.8
4	67.6	3.6	28.8	59.0	1.8	39.2
5	66.4	0.4	33.2	73.8	1.6	24.6
6	50.6	1.4	48.0	60.8	1.2	38.0
7	39.8	2.2	58.0	50.2	3.2	46.6

this test heuristic by varying the test heuristic search depth from 1-ply to 7-ply and observing the average wins, draws, and losses. The trained ANNs are restricted to a search depth of 1-ply.

The WS 192-16×8-16-8-1 ANNs from time period 40 were used for this comparison. There were 10 separate runs in each experiment, and two ANNs in each run, giving a total 20 ANNs. Each of the ANNs played 50 games against the test heuristic for search depths from 1-ply to 7-ply. The results were summed for a total of 1,000 games and are presented in Table 2. The table shows that the average WS 192-16×8-16-8-1 ANN using a 1-ply search outperformed the test heuristic using a 6-ply search.

We compared the computation time of the WS 192-16×8-16-8-1 ANN using a 1-ply search to the test heuristic using a 6-ply alpha-beta search. In this comparison, 100 games were played and the computation time used by each player was recorded. The ANN played over 17 times faster and achieved better results. Our software was not optimized for this test, but the results confirm our expectations that a 1-ply well-trained ANN evaluation function can play better and faster than weighted square heuristic using a deep search.

6. Conclusion

We have investigated 20 different artificial neural network (ANN) architectures for othello evaluation functions. In all the architectures, the game specific information was intentionally kept to a minimum to both allow and force the ANN to learn the important characteristics of the game. The following results were obtained.

We introduced a new symmetry by weight sharing ANN architecture taking into consideration 8-fold game board symmetry and applied this architecture to create othello ANN evaluation functions. The largest and best performing weight sharing ANNs in this study consisted of 26,913 connections and 3,505 weights. These ANNs were compared to a weighted square othello heuristic from the literature. When playing at a minimax search depth of 1-ply, the average ANN was found to outperform the test heuristic playing at a minimax depth of 6-ply.

ANN input encodings with one, two, and three inputs per board square were investigated. We found that although the one input per square encoding is applied frequently in the game literature, both two and three input per square encodings performed better, both in terms of quality of result given a fixed amount of CPU time and overall quality of result achieved.

Of the four ANN architectures investigated: the standard fully connected ANN architecture, symmetry by weight sharing, symmetry removal, and spatial processing; symmetry by weight sharing outperformed all the other architectures. Surprisingly, the spatial processing ANN did not perform as well as a fully connected ANN with roughly half as many weights and connections, suggesting that the extra complexity involved in implementing a spatial processing ANN may not be worthwhile for othello ANN evaluation functions.

We have shown that with a straightforward game board encoding and an appropriate ANN architecture it is possible to apply temporal difference learning to train a high-quality ANN othello evaluation function. Together with a simple 1-ply search, a strong othello player can be created.

We believe these results will be applicable to more complex learning tasks. In a future study, we plan to investigate weight sharing ANN architectures for more complex games such as go.

♢ References ♢

[Baxter 98] Baxter, J., Tridgell, A., and Weaver, L.: KnightCap: A Chess Program That Learns by Combining TD(lambda) with Game-Tree Search, in *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 28–36, San Francisco, CA, USA (1998), Morgan Kaufmann Publishers Inc.

[Bishop 96] Bishop, C. M.: *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford, UK (1996)

[Buro 99] Buro, M.: From Simple Features to Sophisticated Evaluation Functions, in *CG '98: Proceedings of the First International Conference on Computers and Games*, pp. 126–145, London, UK (1999), Springer-Verlag

[Buro 02] Buro, M.: Improving heuristic mini-max search by supervised learning, *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 85–99 (2002)

[Campbell 02] Campbell, M., A. Joseph Hoane, J., and Hsu, hsiung F.: Deep Blue, *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 57–83 (2002)

[Chellapilla 01] Chellapilla, K. and Fogel, D. B.: Evolving an expert checkers playing program without using human expertise., *IEEE Transactions on Evolutionary Computation*, Vol. 5, No. 4, pp. 422–428 (2001)

[Chong 05] Chong, S. Y., Tan, M. K., and White, J. D.: Observing the evolution of neural networks learning to play the game of Othello., *IEEE Transactions on Evolutionary Computation*, Vol. 9, No. 3, pp. 240–251 (2005)

[Fogel 00] Fogel, D. B.: Evolving a checkers player without relying on human experience, *Intelligence*, Vol. 11, No. 2, pp. 20–27 (2000)

[Junghanns 97] Junghanns, A., Schaeffer, J., Brockington, M., Bjornsson, Y., and Marsland, T.: Diminishing returns for additional search in chess, *Advances in Computer Chess*, Vol. 8, pp. 53–67 (1997)

[LeCun 98] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P.: Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324 (1998)

[Leouski 96] Leouski, A. V. and Utgoff, P. E.: What a Neural Network Can Learn About Othello, Technical Report UM-CS-1996-010, University of Massachusetts, Amherst, MA, USA (1996)

[Lucas 06] Lucas, S. and Kendall, G.: Evolutionary computation and games, *IEEE Computational Intelligence Magazine*, Vol. 1, No. 1, pp. 10–18 (2006)

[Mayer 05] Mayer, H. and Maier, P.: Coevolution of neural Go players in a cultural environment, in Corne, D., Michalewicz, Z., McKay, B., Eiben, G., Fogel, D., Fonseca, C., Greenwood, G., Raidl, G., Tan, K. C., and Zalzala, A. eds., *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, Vol. 2, pp. 1017–1024, Edinburgh, Scotland, UK (2005), IEEE Press

[Rojas 96] Rojas, R.: *Neural Networks - A Systematic Introduction*, Springer-Verlag, Berlin, Germany (1996)

[Runarsson 05] Runarsson, T. P. and Lucas, S. M.: Coevolution versus Self-play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go, *IEEE Transactions on Evolutionary Computation*, Vol. 9, No. 6, pp. 628–640 (2005)

[Schaeffer 00] Schaeffer, J.: The Games Computers (and People) Play, *Advances in Computers*, Vol. 50, pp. 189–266 (2000)

[Schaeffer 01] Schaeffer, J., Hlynka, M., and Jussila, V.: Temporal Difference Learning Applied to a High-Performance Game-Playing Program., in Nebel, B. ed., *IJCAI*, pp. 529–534, Morgan Kaufmann (2001)

[Schraudolph 01] Schraudolph, N. N., Dayan, P., and Sejnowski, T. J.: Learning to Evaluate Go Positions via Temporal Difference Methods, in Baba, N. and Jain, L. C. eds., *Computational Intelligence in Games*, Vol. 62 of *Studies in Fuzziness and Soft Computing*, chapter 4, pp. 77–98, Springer Verlag, Berlin (2001)

[Singer 01] Singer, J. A.: Co-evolving a Neural-Net Evaluation Function for Othello by Combining Genetic Algorithms and Reinforcement Learning, in *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pp. 377–389, London, UK (2001), Springer-Verlag

[Sutton 88] Sutton, R. S.: Learning to Predict by the Methods of Temporal Differences, *Machine Learning*, Vol. 3, No. 1, pp. 9–44 (1988)

[Sutton 98] Sutton, R. and Barto, A.: *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA (1998)

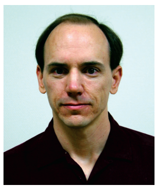
[Tesauro 92] Tesauro, G.: Practical Issues in Temporal Difference Learning, *Machine Learning*, Vol. 8, No. 3-4, pp. 257–277 (1992)

[Tesauro 95] Tesauro, G.: Temporal difference learning and TD-Gammon, *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68 (1995)
[Tesauro 02] Tesauro, G.: Programming backgammon using self-teaching neural nets, *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 181–199 (2002)
[Yoshioka 99] Yoshioka, T., Ishii, S., and Ito, M.: Strategy acquisition for the game "Othello" based on reinforcement learning, *IEICE Transactions on Information and Systems*, Vol. E82-D, No. 12, pp. 1618–1626 (1999)

〔担当委員：Miyazaki Kazuteru〕

Received February 8, 2007.

Author's Profile



Binkley, Kevin J.

Kevin J. Binkley received a B.A. in Applied Math and a B.S. in Chemistry from the University of California Berkeley in 1988 and an M.S. in Computer Science from Stanford University in 1991. He is now pursuing a doctorate at Keio University. His current research interests include artificial neural networks and evolutionary computation. He is a member of IEICE and IEEE.

Seehart, Ken

Ken Seehart received the Bachelor of Arts degree in Computer and Information Sciences from University of California at Santa Cruz in 1989. His research interests include neural networks and computational linguistics.

Hagiwara, Masafumi (Member)

Masafumi Hagiwara received his B.E., M.E. and Ph.D degrees in electrical engineering from Keio University, Yokohama, Japan, in 1982, 1984 and 1987, respectively. Since 1987 he has been with Keio University, where he is now a Professor. From 1991 to 1993, he was a visiting scholar at Stanford University. He received the Niwa Memorial Award, Shinohara Memorial Young Engineer Award, IEEE Consumer Electronics Society Chester Sall Award, Ando Memorial Award, Author Award from the Japan Society of Fuzzy Theory and Systems (SOFT), Technical Award and Paper Award from Japan Society of Kansei Engineering in 1986, 1987, 1990, 1994, 1996, 2003, and 2004, respectively. His research interests include neural networks, fuzzy systems, evolutionary computation and kansei engineering. Dr. Hagiwara is a member of IEICE, IPSJ, SOFT, IEE of Japan, Japan Society of Kansei Engineering, JNNS and IEEE (Senior member).