Teaching a Neural Network to Play Konane

Darby Thompson

Spring 2005

Abstract

A common approach to game playing in Artificial Intelligence involves the use of the Minimax algorithm and a static evaluation function. In this work I investigated using neural networks to replace hand-tuned static evaluation functions. Networks were also trained to evaluate board positions to greater depth levels using Minimax. The networks begin as a random networks and by playing against a random player, the networks are able to match the teacher's performance. This work provides evidence that board representation affects the ability of the network to evaluate Konane board positions. Networks that are taught to output a real-value board evaluation outperform those taught to directly compare two boards after a considerable amount of training. However, the latter networks show more consistent behavior during training, and quickly learn to play at a reasonably good skill level.

1 Introduction

This thesis presents the procedures and results of implementing and evolving artificial neural networks to play Konane, an ancient Hawaiian stone-jumping game. This work includes the development of a several hand-tuned successful board evaluation functions which are used as teachers during the supervised learning process of the networks and then as opponents when evaluating the ability of the neural network player. The back-propagation networks used during this research do not evaluate or predict the final outcome of the game, but rather recommend the best move at each stage of the game. Networks were trained to perform several different tasks, including: simply trying to approximate the teacher's evaluation function, and simulating the results of the Minimax algorithm search (to various depths) using the same evaluation function of the board at each stage.

Games have been a popular vehicle for demonstrating new research in artificial intelligence since the early fifties. This work combines the familiar thought that games are ideal test-beds for exploring the capabilities of neural networks, with a practical implementation of the lesser known game Konane.

1.1 Konane

This ancient Hawaiian version of Checkers is a challenging game of strategy and skill. Originally played using shells and lava rocks as pieces, Konane is a simple game to learn, yet is complicated enough to take a lifetime to master. The objective is simple: be the last player able to make a move.

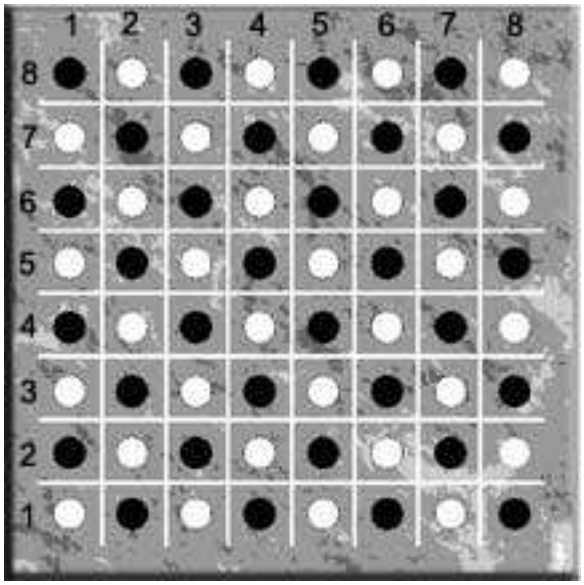


Figure 1: Konane Board Setup[9]
Positions on the board are referenced using (row, column) notation.

Konane is played on a rectangular grid of spaces all of which are initially occupied by alternating black and white pieces (Figure 1). Konane boards do not follow any established pattern in size and range from 6x6 boards to well over 14x14 boards.

To begin the game the first player (black) must remove one of their pieces, either the center piece, one laterally next to it or one at a corner. Using the row and column numbering from Figure 1, black would remove either (1,8), (8,1), (4,5) or (5,4). The second player (white) now removes a piece of their own, adjacent to the space created by black's first move. For example, if black removed (1,8) white may remove (2,8) or (1,7). If black removed (4,5), white may remove (4,6), (4,4), (3,5) or (5,5). Thereafter players take turns making moves on the board.

A move consists of jumping the players piece over an adjacent opponent's piece into an empty space and removing that opponent's piece. Jumping must occur along a row or column (not diagonally) and a player may jump over multiple opponent's pieces provided they are all on the same row/column and are all separated by empty spaces. The arrows in Figure 2 show the possible moves for the black player's piece in position (6,5). The piece may jump up to (6,7), right

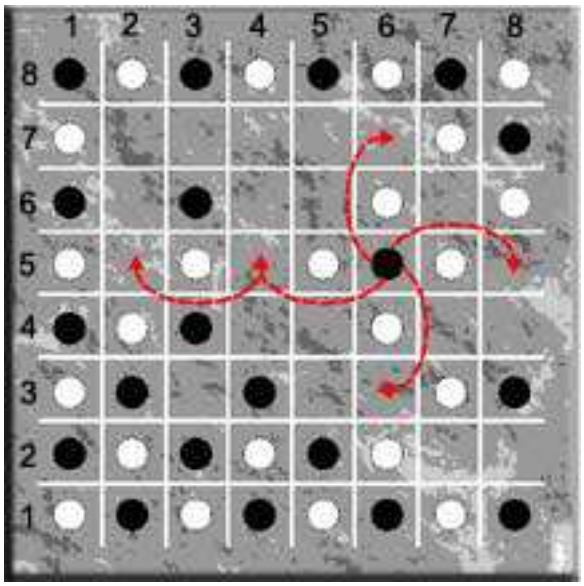


Figure 2: A possible Konane board state during a game[9]

to (8,5), left to (4,5), multiple jump left to (2,5) or down to (6,3). However, these are not the only possible moves for the black player. They may move from (7,8) to (7,6), (7,8) to (7,4), (7,8) to (7,2), (8,7) to (6,7), (8,7) to (8,5), (8,3) to (6,3), (2,3) to (2,5), (5,2) to (7,2) or (6,1) to (6,3).

The game is finished when the current player has no available moves left.

1.2 Neural Networks

Neural Networks have proven effective at solving complex problems which are difficult to solve using traditional boolean rules[13]. An neural network is an interconnected group of artificial neurons that uses a mathematical or computational model for information processing based on a connectionist approach to computation[10]. Using neural networks we can use parallel processing and abstraction to solve real world problems where it is difficult to define a conventional algorithm. Neural networks are devices which consist of interconnected components that vaguely imitate the function of brain neurons. A typical back-propagation network, shown in Figure 3, consists of three layers; the input, hidden and output layers. Each layer consists of a number of nodes which are commonly fully connected to the adjacent layers. Throughout this work networks will always be fully connected. Each connection has a weight associated with it.

The network functions as follows: Values are associated with each input node. These values are then propagated forwards to each node in the hidden layer and are each multiplied by the weight associated with their connection. The weighted inputs at each node in the hidden layer are summed, and passed through a limiting

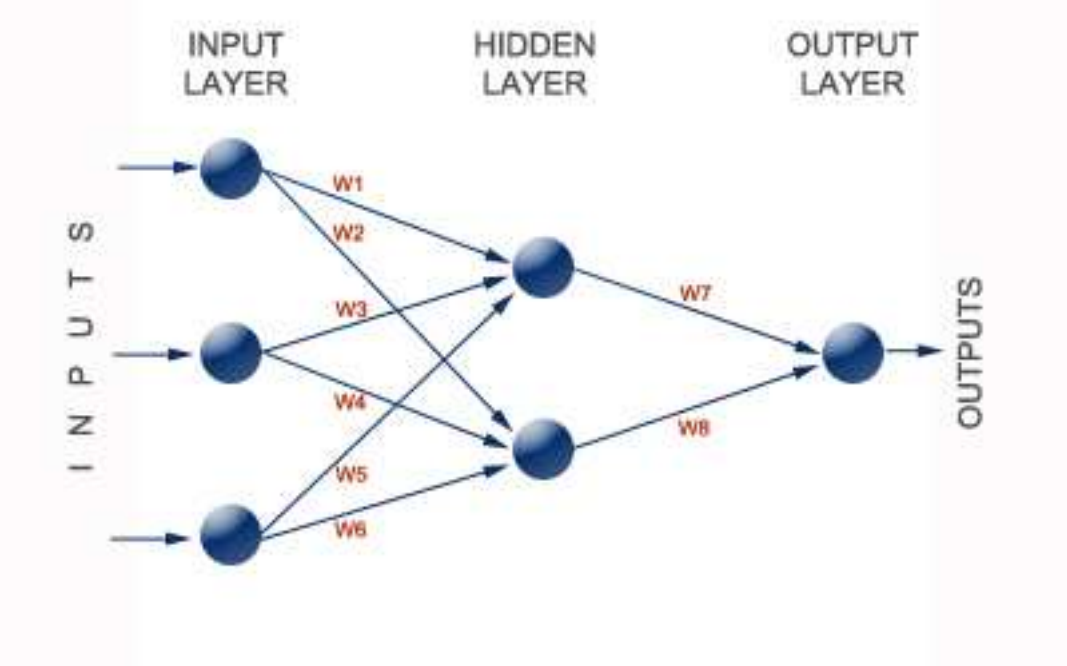


Figure 3: Typical Network Structure

function which scales the output to a fixed range of values. These values are then propagated forwards to each node in the output layer and are again multiplied by the weight associated with their connection. To use the network we simply assign values to each node in the input layer, propagate forwards and then read the output node values.

For the duration of this work I will be using supervised learning. Supervised learning is a machine learning technique for creating a function from training data[10]. The training data consist of pairs of input vectors, and desired outputs. The task of the supervised learner is to predict the value of the function for any valid input object.

When the neural network is learning, values are assigned to the input layer and are propagated forwards as usual. The network then compares the output node values with target values set by a teacher (the supervisor; a function which the network will try to learn). It computes the mean-squared error and then propagates backwards (hence the title 'back-propagation network') across the network, using a learning rule to adjust the weights along the way. The learning rule uses a constant momentum value and epsilon (learning rate) value which can be set to determine how quickly and how much weights change during the learning process. For more details see 'Parallel Distributed Processing' by Rumelhart, Hinton and Williams[13].

Over-training (also known as overfitting) is a potential danger to networks

when training. Typically a network is trained using some set of training examples for which the desired output is known (as described above). The learner is assumed to reach a state where it will also be able to predict the correct output for other examples, thus generalizing to situations not presented during training. However, especially in cases where learning was performed too long or where the training set is small, the learner may adjust to very specific random features of the training data, that have no causal relation to the target function. When this happens, the network may perform poorly when given an input pattern that it has not yet been trained on.

1.3 Game Playing and the Minimax Algorithm

The Minimax algorithm[16] can be applied to any two-player game in which it is possible to enumerate all of the next moves. In theory we could represent the entire game in a search tree, starting with the root node as the original state of the game, and expanding the nodes at each level to represent possible states after the next move has taken place. The levels could be expanded all the way to the end-game state to complete the tree. This, however, becomes very complex when dealing with a game with a high branching factor such as Konane which typically has a branching factor averaging 10 (using an 8x8 board), and impossible with games such as Chess which has an average branching factor of 30. It is therefore not feasible for an artificial player to store the entire tree or search down to the leaves to determine the best move. Instead we use a static evaluation function which estimates the goodness of a board position for one player. The static evaluation function is used in conjunction with the Minimax algorithm and a search is performed through a limited depth of the tree to approximate the best move.

Minimax defines two players; MIN and MAX. A search tree is generated starting with the current game state down to a specified depth. A MAX node is a node that represents the game state directly after a move by the MIN player. Likewise, a MIN node is a node that represents the game state directly after a move by the MAX player. The leaf nodes are evaluated using a static evaluation function defined by the user. Then the values of the inner nodes of the tree are filled in from the leaves upwards. MAX nodes take the maximum value returned by all of their children, while MIN nodes take the minimum value returned by all of their children. The values at each level represent how 'good' a move is for the MAX player. In a game of perfect information such as Konane, the MIN player will always make the move with the lowest value and the MAX player will always make the move with the highest value.

This algorithm is popular and effective in game playing. However it is computationally expensive, so searching deeper than four levels to play Konane is unrealistic during a real-time game. Its expensive nature is partially due to the move/board generation. However, a good static evaluation function for an nxn

Konane board can take up to $O(2n^3)$ time. Therefore search to depth d using the Minimax algorithm can be expected to take $O(2n^3k^d)$ time where k is the average branching factor. The Alpha-Beta pruning algorithm can be used to decrease the potential number of evaluations, however it may prune out potentially excellent moves, and in this work it was not used. It is also important to note that Minimax is dependent on having a decent static evaluation function (although a deeper search will make up for some weakness in the static evaluation function).

1.4 Hypothesis

The research hypotheses are:

1. A back propagation neural network can be taught, using supervised learning, to evaluate Konane boards as effectively as its teacher. When playing against a random player, the performance of the network is comparable to its teacher. When the network plays multiple games against its teacher, each player wins an equivalent number of times.

2. A back propagation neural network can be taught, using supervised learning, to evaluate Konane boards to a depth greater than 1, effectively learning Minimax and a static evaluation function. The skill level of the network is greater than its teacher, and comparable to to its teacher using Minimax to the depth learned by the network. The network can be used alone or in conjunction with Minimax to search deeper than simply using the Minimax algorithm and a static evaluation function when under time constraints.

As well as providing supporting evidence for these hypotheses, I intend to maximize the performance of the artificial neural network players by researching the effects of varying different parameters during training (outlined in section 3).

2 Previous Work

Few studies of Konane have been published, although it has been used in Artificial Intelligence classes[8, 11] when introducing students to heuristic search algorithms. On the other hand, neural networks are used in many different capacities in game playing, creating strong players in games from tic-tac-toe[6] to chess[19]. Genetic algorithms have bred dominant neural network players[4], temporal difference and reinforcement learning strategies have taught networks to play without a teacher[18], and networks have learned to predict the outcome of a game and been taught how to prune a Minimax search tree[12]. These are just a few of the different approaches taken towards integrating neural networks and game playing.

Although I could not find any published work regarding the use of neural networks and Konane, much work has been done to integrate machine learning (an area of AI concerned with the development of techniques which allow computers to "learn") and traditional Checkers. In the 1950s Arthur Samuel wrote a program to play Checkers[14]. Along with a dictionary of moves (a simple form of rote learning), this game relied on the use of a polynomial evaluation function compromising of a subset of features chosen from a larger set of elements. The polynomial was used to evaluate board positions some number of moves into the future using the Minimax algorithm. His program trained itself by playing against a stable copy of itself (self-learning).

Samuel defined 5 characteristics of a good game choice to study machine learning[14]:

1. The activity must not be deterministic in the practical sense. There exists no known algorithm which will guarantee a win or a draw in Konane.

2. A definite goal must exist and at least one criterion or intermediate goal must exist which has a bearing on the achievement of the final goal and for which the sign should be known. In Konane the goal is to deprive the opponent of the possibility of moving, and one of the dominant criteria is the number of pieces of each color on the board, another is the number of movable pieces of each color on the board.

3. The rules of the activity must be definite and they should be known. Konane is a perfect information game and therefore satisfies this requirement.

4. There should be a background of knowledge concerning the activity against which the learning progress can be tested. For the purpose of testing the artificial Konane player, multiple strategic players have been created, as well as a random player.

5. The activity should be one that is familiar to a substantial body of people so that the behavior of the program can be made understandable to them. The ability to have the program play against human opponents (or antagonists) adds spice to the study and, incidentally, provides a convincing demonstration for those who do not believe that machines can learn. Konane has simple rules and is easy to learn and is therefore a perfect candidate for this research.

Research has since been expanded in the area of self-learning. A successful example of this is TD-Gammon[18] which uses a neural network that was trained using temporal difference learning to play Backgammon. A common problem with the self-play approach when used in a deterministic game is that the network tends to explore only some specific portions of the state space. Backgammon is less affected by this due to the random dice rolls during game play. TD-Gammon

is also a very successful example of Temporal difference learning used in game playing. Temporal difference learning is a prediction method[17]. It has been mostly used for solving the reinforcement learning problem. After each move, the network calculates the error between the current output and previous output and back propagates a function of this value. When the network completes a game the value of 1 or 0 is propagated backwards, representing a win or loss.

Another common approach to game playing with neural networks is the process of evolving populations of neural networks to essentially 'breed' a master-level player. This research has been applied successfully to Checkers[4]. In their experiments, neural networks competed for survival in an evolving population. The fully connected feed forward networks were used to evaluation board positions and also utilized the Minimax algorithm. The end result was an artificial network player that was placed in the "Class A" category using the standard rating system. Both this method of learning and reinforcement learning are particularly interesting since they require no expert knowledge to be fed to the network.

Go is a strategic, two-player board game originating in ancient China between 2000 BC and 200 BC. It is a common game used in the study of machine learning due to its complexity. Go is a complete-knowledge, deterministic, strategy game: in the same class as Chess, Checkers and Konane. Its depth arguably exceeds even those games. Its large board (it is played on a 19x19 board) and lack of restrictions allows great scope in strategy. Decisions in one part of the board may be influenced by an apparently unrelated situation, in a distant part of the board, and plays made early in the game can shape the nature of conflict a hundred moves later. In 1994 Schraudolph, Dayan and Sejnowski[15] used temporal difference learning to train a neural network to evaluate Go positions. During this research they trained a network to play using a 9x9 board and verified that weights learned from 9x9 Go offer a suitable basis for further training on the full-size (19x19) board. Rather than using self-play, they trained networks by playing a combination of a random player and two known strategy players and compared the results. This approach is similar to the one I employed in my experiments. They found self-play to be problematic for two reasons: firstly, the search used to evaluate all legal moves is computationally intensive. Secondly, learning from self-play is 'sluggish as the network must bootstrap itself out of ignorance without the benefit of exposure to skilled opponents'[15]. They discovered that playing against a skilled player payed off in the short term, however after 2000 games the network starts to over-train with the skilled player resulting in poor performance against both skilled players.

In 1995 Michael D Ernst documented his combinatorial game theory[1] analysis of Konane[5]. Konane complies with six essential assumptions needed by combinational game theory; it is a two player game, both players have complete information, chance takes no part, players move alternately, the first player unable to move loses, and the game is guaranteed to win. Another property which makes Konane an ideal candidate for combinational game theory is that it can of-

ten can be divided into independent sub-games whose outcomes can be combined to determine the outcome of the entire game[5] Although using game theory to analyze Konane suggests that with enough analysis a winning strategy could be formed (opposing Samuels first criterion), research performed in this area has only focused on 1-dimensional patterns in Konane, and few 2-dimensional sub-game patterns. This form of analysis assigns to each board layout a value indicating which player wins and how many moves ahead the winning player is. This value is actually a sum of the game-theoretic mathematical values assigned to each sub-game. At the same time, Alice Chan and Alice Tsai published a paper on analysis of 1xn Konane games[3]. Both papers concluded that more research needed including the creation of a dictionary of Konane positions and research into the rules for separation of Konane games into sums of smaller games. Along with the requirements for a good game choice as specified by samuel[14], their work supports the use of Konane to study game play and machine learning.

3 Experimental Design

This section presents the basic experimental setup and the methodologies to support the proposed hypotheses.

3.1 Tools Employed

Experiments were conducted on departmental lab computers (2.2GHz Pentium 4 processors) at Bryn Mawr College. The template program -Appendix B- used to run all experiments is written in Python and uses neural network functionalities provided by the Conx modules in Pyro[2].

The template program is primarily based on two classes: Konane and Player. The Konane class implements the rules of Konane for any even square board size. It provides a move generator, validation check for moves, and the ability to play any number of games (verbose, logged or otherwise) between any two Players. The Player class is the base class for the Konane Players and implements the standard structure of a player, including performance tracking and Minimax. Each strategy player is a class which inherits the Konane and Player classes. These individual strategy classes each define their static evaluation function, Minimax max-depth level and name initialization. Currently the template program includes 24 players including a random player and human player.

There are 4 Neural Network players; 2 for neural networks in the process of learning and 2 for testing. The first network player uses one board representation as the input to its network and returns the evaluation value. The second network player uses two concatenated board representations as the input and returns a value representing the comparison between the two boards. Similarly, the first testing network player uses one board representation as the input, and the second

player uses two board representations as the input.

Noise is introduced into the static players that use Minimax with a static evaluation function during draws. When a player encounters two moves with equal value it randomly picks which move to use.

The Conx module, designed to be used by connectionist researchers, is a Python based package that provides an API for neural network scripting. It is based on three classes: Layer (a collection of nodes), Connection (a collection of weights linking two layers) and Network (a collection of layers and connections)[2].

3.2 Training Parameters

When using the template program to run network training experiments the following parameters are revised and changed if necessary:

1. Size of the Konane board: Konane boards vary in size, however, the larger the board size, the more time consuming the experiments are. An assumption in this work is that what can be learned on an 8x8 board should be scaleable up to any size board. Experiments were performed on 4x4 boards in section 5, and 6x6 and 8x8 boards in section 7.

2. Representation of the board: This affects the size of the network and can affect the network's ability to learn a static evaluation function as shown in section 5.1.

3. Structure of the network (either single board evaluations or comparison between 2 boards): Networks trained in section 5.1 as single board evaluators were compared against comparison networks trained in section 5.2.

4. Size of the input, hidden and output layers: Neural networks can be sensitive to their design. In particular, in the way the input is represented, the choice of the number of hidden units and the output interpretation. Hence, I look at input representation in section 5.1, hidden layer size in 5.3 and the output interpretation in section 5.2.

5. Learning Rate [0:1] and Momentum [0:1]: This parameter was not a focus of this work. Based on several simple tests, I set the learning rate and momentum at 0.01 for all the experiments described in this thesis.

6. Teacher function: This function is used as a target for the network outputs. To train a network to perform well against a human player, a strong teacher function must be chosen. This was done in section 3.3.

7. What depth level of Minimax to learn: Initial experiments in section 5 were performed with no look-ahead (i.e. depth level of 1) to support the first hypothesis. Later, the depth level was expanded up to a depth of 4.

8. Opponent to play during training: For the experiments in this thesis, the networks learned by playing against a random player, so that they could learn as much of the search space as possible. If they were to learn while playing a static opponent, the search space would be limited and the network may not learn how to play against other opponents.

9. Number of games to play (i.e. length of training phase): In this work, networks were trained either until their performance plateaued, or training ceased due to time constraints. This was not a focus of this work.

10. How often the network should save its weights: This parameter was not a focus of this work. Networks saved their weights every 5000 games. They also saved their weights when the performance over the past 100 games was at its best.

Neural networks can be trained using online or offline training. Offline training consists of two phases: Data collection and learning. During the first phase, a data set is generated consisting of input and target output values. The network then repeatedly propagates the data set through the network and uses it as training data. Online training removes the need for data collection: the network essentially learns in real time, on-the-fly. Offline training offers the benefit of being able to sample the training data to prevent the network from seeing a particular input pattern much more often than other patterns. However, given an extremely large search space it is often unreasonable to store enough possible input patterns to learn a significant area of the space. As a result, network training is performed online in this work due to the large nature of the training data set in Konane.

3.3 Evaluating Performance

During the training phase, the network player keeps track of its performance over the previous 100 games. After each game played, it records the percentage of games won. When the training is complete, a neural network player is tested using the same template program. When a weights file, representation method and equivalent network structure are given, the Test Network Player will play a number of games against a specified player.

To evaluate the strength of one static strategy player over another, it is insufficient to play two games (each playing both black and white once) due to the noise introduced through static evaluation function ties. As a result, I play 1000 games, rotating who goes first, between the two players to determine their respective skill levels. Overall this measure gives results accurate to approximately $\pm$ 2% in an 8x8 Konane board environment.

3.4 Preliminary Setup

Before teaching a network to play Konane, the teacher function must be chosen. To teach a neural network to learn how to play Konane at an advanced level, I would ideally use the 'best' player's board evaluation function as the teacher. Due to the complexity of the game, it is unreasonable to pick a perfect board evaluation function. I could use Samuels method of evolving an evaluation function[14], however this is not the focus of this research, therefore I use the 'best' evaluation function that I can hand-tune.

To determine which evaluation function to use, 18 different plausible player board evaluation strategies were picked and their skill levels were tested. Each player competed against the random player over 1000 games on an 8x8 Konane board. This experiment was executed 4 times with the players searching to depths of 1 through 4 using Minimax.

The most competent of these players (as determined from the previous experiments) were then set to compete against each other at equal Minimax search depths 1 through 4 (See section 4 for more details). The strongest static evaluation function was then set as the teacher function.

4 Preliminary Experiments

4.1 Finding the 'best' teacher evaluation function

The design of a back-propagation neural network requires a teacher, which, given a set of inputs, will compute the correct (or in this case - a good estimate) output value(s). To teach a neural network to learn how to play Konane at an advanced level, I would ideally use the 'best' player's board evaluation function as the teacher. However, due to the complexity of the game I must settle for a good evaluation function; the 'best' evaluation function that I can formulate myself. To do this, I broke down Konane into six, countable, essential elements (arranged in three player/opponent pairs):

- Number of computer player's pieces left on the board
- Number of opponent's pieces left on the board
- Number of possible moves for the computer player
- Number of possible moves for the opponent
- Number of computer player's pieces that are movable
- Number of opponent's pieces that are movable

I then applied six different functions to each pair of essential elements to develop eighteen different plausible player board evaluation strategies:

4.1.1 Plausible board evaluation functions

a Number of computer player's pieces left on the board

b Number of opponent's pieces left on the board

c Difference between a and b (a-b)

d Weighted Difference between a and b (a-(b*3))

e Ratio of a and b (a/b)

f Weighted Ratio of a and b (a/(b*3))

g Number of possible moves for the computer player

h Number of possible moves for the opponent

i Difference between g and h (g-h)

j Weighted Difference between g and h (g-(h*3))

k Ratio of g and h (g/h)

l Weighted Ratio of g and h (g/(h*3))

m Number of computer player's pieces that are movable

n Number of opponent's pieces that are movable

o Difference between m and n (m-n)

p Weighted Difference between m and n (m-(n*3))

q Ratio of m and n (m/n)

r Weighted Ratio of m and n (m/(n*3))

4.2 Testing the static evaluation functions

To discover which board evaluation function is the most effective at winning Konane, I played 1000 games of each evaluation function using Minimax at depth levels 1-4 against a random player on an 8x8 board (Figure 4).

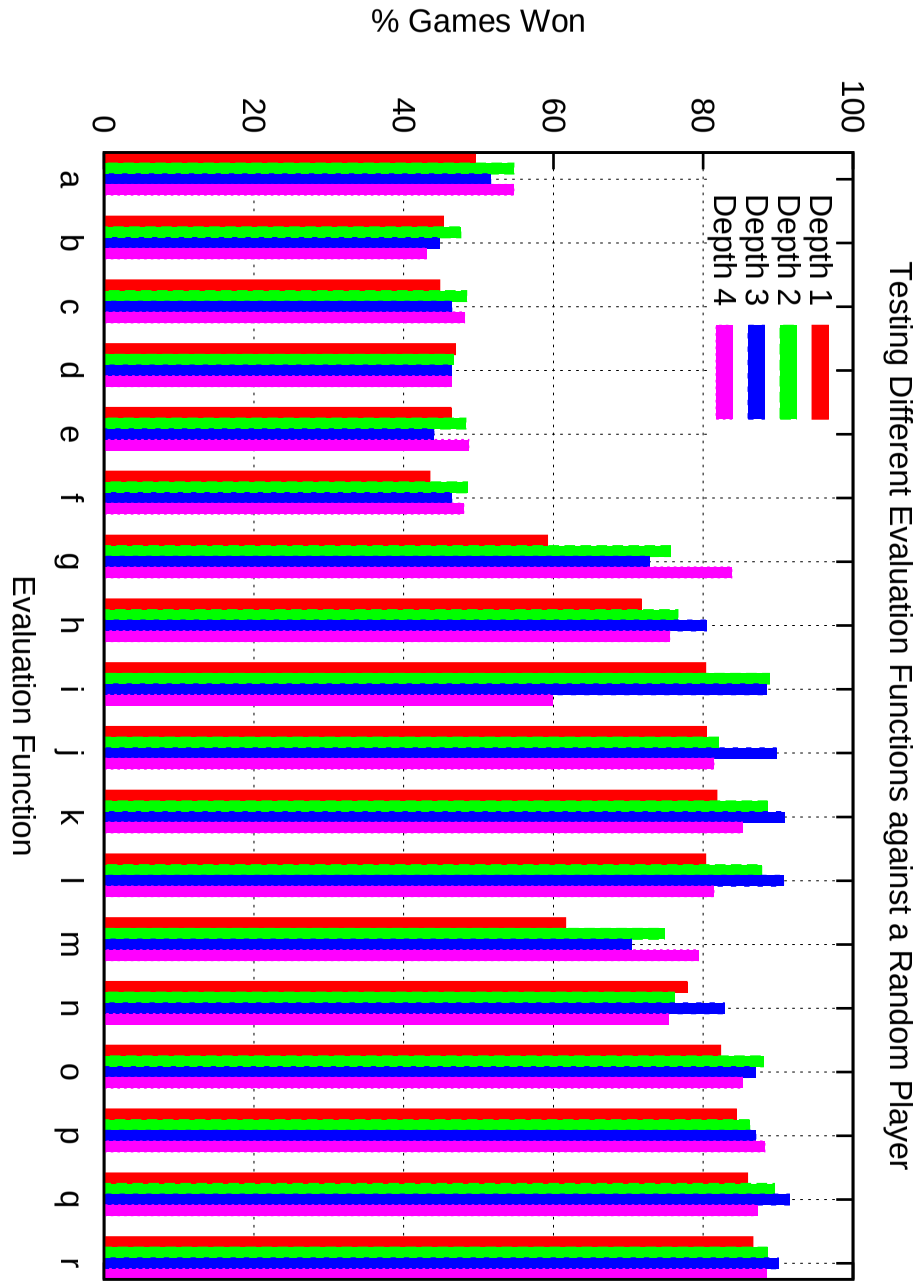


Figure 4: Testing Evaluation Functions against a Random Player
The functions are identified by the letters used in section 4.1.1

Clearly the evaluation functions relying on simply the number of white and black pieces on the board are outperformed by the other evaluation functions. In fact, during a game of Konane, the only time when comparing the number of pieces on the board can be beneficial is after a player moves using a multiple jump and removes more than one of the opponent's pieces. Before that happens, there is always the same ratio of black and white pieces on the board, essentially making these evaluation functions random players. The evaluation functions relying on the number of possible moves or movable pieces for each player perform much better.

The results from Figure 4 show that in some cases, using the Minimax algorithm to a greater depth did not increase a player's strength. This is due to the fact that Minimax assumes that the opponent is an intelligent player that will chose their 'best' move at each stage of the game. However, here the opponent is a random player, therefore using Minimax at a greater depth is not always beneficial to the player.

Playing each strategy against a random player does show some degree of the players strength. However, to be confident that a strategy really is the best of the choices available I played each strategy against the other strategies for 1000 games using Minimax at depth levels 1-4. Since these experiments are extremely time intensive (particularly when the depth level is greater than 2), I chose to only advance the 12 strategies (g-r) that consistently played better than the random player in the previous experiments. Although the player strategies based on these evaluation functions would appear to be static, the choice to play 1000 games was made to take into account the 'noise' created by draws between the evaluation of two or more board positions that occurs during play. When a player is faced with a draw for the best move, it randomly picks one of the moves, as a result it would not be sufficient to simply play each strategy against each other twice (each taking a turn to be the first player).

Table 1 shows the performance of the static evaluation functions g-r when played against each other. The values in each column are the average percentage of games won after playing 1000 games against every other static evaluation function. Evaluation function 'q' has the most consistant strength against other opponents and was therefore chosen to represent the teacher in all future experiments.

5 4x4 Experiments

Since the search space for an 8x8 Konane game is so large, initial experiments were performed on 4x4 boards. Three elements of the network were varied to find the ideal network structure, however the basic structure remained unchanged: A back propagation network was used with one input layer, one hidden layer, and one node in the output layer. The different networks were taught by playing

SEF	Level 1	Level 2	Level 3	Level 4	Average % Games Won
g	37.00	40.09	37.09	40.45	38.66
h	48.55	31.81	31.54	27.09	34.75
i	54.27	51.81	54.90	51.81	53.20
j	45.73	48.36	50.90	45.00	47.50
k	50.27	56.81	58.18	57.18	55.61
l	51.27	59.72	58.18	55.63	56.20
m	33.27	40.18	37.00	40.00	37.61
n	47.45	37.00	39.72	37.36	40.38
o	**61.36**	61.00	54.27	59.54	59.04
p	53.45	41.63	59.09	61.09	53.82
q	**58.73**	**67.00**	**59.64**	**62.45**	**61.96**
r	58.64	**64.54**	**59.45**	**62.36**	**61.25**

Table 1: % Games won against other players at depth levels 1-4.
Bold values indicate top two players at each depth level.
For more detailed tables refer to Appendix A.

200,000 games against a random player, with the teacher evaluation function chosen during the preliminary experiments. The network players keep track of the percentage of games won over the past 100 games during training and, when this percentage is at its highest, they save their weights. To test the effectiveness of the training, the networks then play 1000 games against the random player with this 'best' set of saved weights. Note that using the 'best' set of saved weights in no way is cheating The 'best' is wholly determined by training success. Hence there is no leakage from testing into the selection of the 'best'.

5.1 Different Board Representations

Definition. *The board representation is the function chosen to convert an nxn Konane game board of black, white and vacant spaces into an input vector for a neural network. The length of the input vector (i.e. number of input nodes in the network input layer) is unrestricted, however values in the vector must be in the range [0:1].*

Three neural network players were taught to play Konane using different board representations while playing against a random player. The strongest saved weights from each network's training phase were compared, along with the overall behavior of each player during training. This experiment set illustrates the importance of choosing a good board representation. The goal is to have the network learn which boards are favorable to the player (and to what degree) and which are not, as successfully as possible. As a result, choosing an effective

board representation is essential to provide evidence in support of both proposed hypotheses.

5.1.1 Motivation

The possibilities are endless when it comes to picking a method of representation of a game board; the only restriction is that each input value in a neural network must be between 0 and 1. The representation must obviously reflect changes when a move has taken place, and therefore must have different values associated with vacant and occupied spaces on the board. Even with this additional constraint, the space of representations of the board is large and leaves open for discussion at least the following questions:

- Is it necessary to associate different values with black and white pieces?
- Player and opponent pieces?
- Is it sufficient to have nxn input nodes (when playing on an nxn Konane board)?
- What is more important; associating nodes in the network with spaces on the board, or associating nodes with players?
- Is it more fitting to associate specific nodes as being favorable/unfavorable or to associate larger values as being advantageous and smaller values as harmful?

I did not try to explore the representation space thoroughly. Rather, I developed 3 representations as described below.

5.1.2 Setup

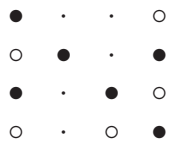
'Representation A' used one node for every space on the board (n^2 inputs). Each node carried a value of 1 if the player's piece was occupying the space, 0.5 if the space was empty and 0 if the opponent's piece was occupying the space. The network had 16 inputs, 10 hidden nodes and 1 output. The thought behind this being that the player's pieces are the most beneficial to the player, opponent's pieces were detrimental and spaces were neither, therefore a high value was placed on the player's pieces vs opponent's pieces which had the lowest value. The success of this representation depends on the network being able to associate large values as being advantageous and smaller values as the opponent. It associates nodes in the network inputs with specific spaces on the board so that the network can abstract connections between locations on the board.

A potential problem with Representation A is that the usefulness of the '0' values in the input vector is limited in a neural network structure; the weights

that '0' is multiplied against become insignificant. Since opponent pieces are essentially destructive to the success of the player, they should at least hold some weight in the network. 'Representation B' used again, one node for every space on the board (n^2 inputs). The first n nodes corresponded to the spaces on the board that were originally occupied by the player, the rest corresponded to the spaces that were originally occupied by the opponent. In this representation 1 indicated that the space was occupied, 0 indicated an empty space. The network structure of this representation was 16 inputs, 10 hidden nodes and 1 output. The success of this representation depends on the ability of the network to associate the first n nodes as being advantageous and the last n nodes as detrimental.

'Representation C' used twice the number of input nodes of the previous representations ($2n^2$ inputs). The first n^2 nodes represent the spaces on the board and have a value of 1 if the player's piece is occupying that space and 0 if the space is empty or occupied by the opponent. The second n^2 nodes are another representation of the spaces on the board and have a value of 1 if occupied by the opponent and 0 if the space is empty or occupied by the player. The network tested had 32 inputs, 20 hidden nodes and 1 output. This representation was tested to see if the network could benefit from having more nodes to abstract patterns from in the input layer. Since each node always corresponds to the same location on the board, there is a potential here for the network to abstract connections between locations on the board. At the same time, the network has the potential to learn that positive values in the first n^2 nodes are advantageous, and positive values in the last n^2 nodes represent the opponent.

An example of the three board representations is shown below:



When the network player is black:
Network Inputs Representation #A: [1, 0.5, 0.5, 0, 0, 1, 0.5, 1, 1, 0.5, 1, 0, 0, 1, 0, 1]
Network Inputs Representation #B: [1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1]
Network Inputs Representation #C: [1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1,
0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0]

When the network player is white:
Network Inputs Representation #A: [0, 0.5, 0.5, 1, 1, 0, 0.5, 0, 0, 0.5, 0, 1, 1, 0, 1, 0]
Network Inputs Representation #B: [0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1]
Network Inputs Representation #C: [0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]

5.1.3 Results

Figure 5 shows the networks performance during the first 100,000 training games. The drop in accuracy from over 75% to 50% as shown by the network using Representation A, is assumed to be an indication of over-training. After playing 100,000 games it would appear that the network using Representation A had been over-trained and learned all that it could from its representation. This network learned much more quickly than the other two representations, however it peaked at a lower percentage.

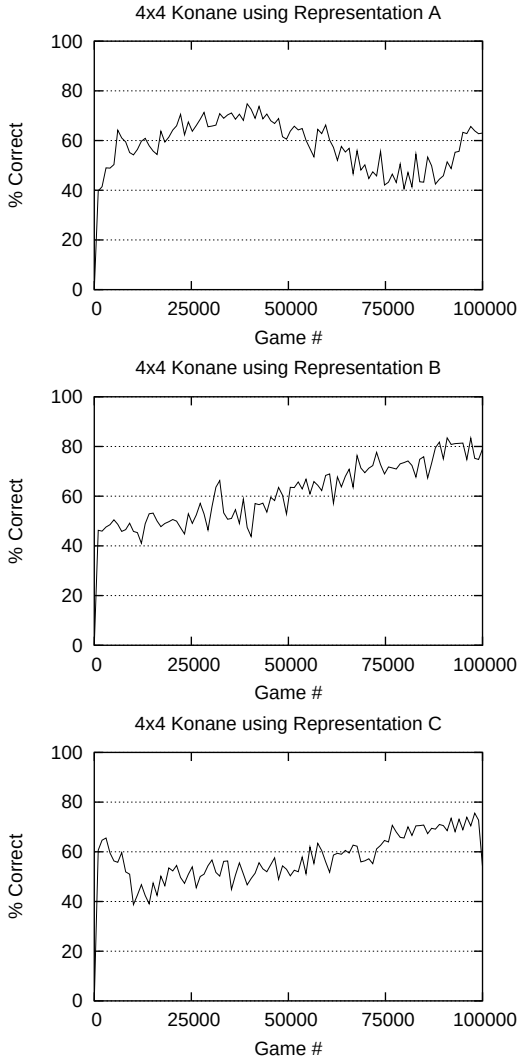


Figure 5: Comparing the behavior during training of Neural Networks to play 4x4 Konane using different board representations against a random player over 100,000 games. The graphs show smoothed representations of the actual data.

Both Representation B and C appeared to still be learning so the experiments were extended to run for 200,000 games. After 175,000 games, Representation B had also peaked. Representation C, on the other, hand never suffered from a long-term decline in accuracy as shown in Figure 6.

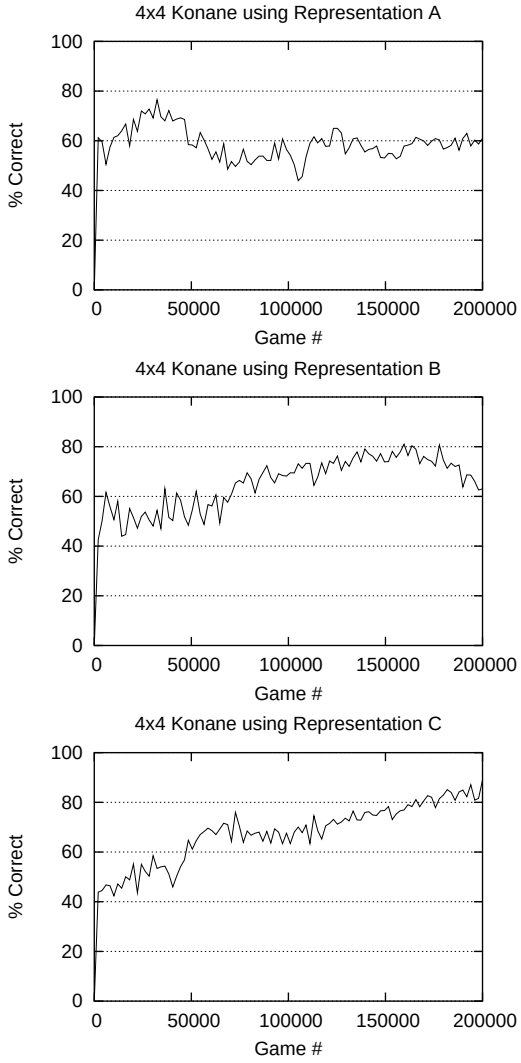


Figure 6: Comparing the behavior during training of Neural Networks to play 4x4 Konane using different board representations against a random player over 200,000 games.

The plots in Figures 5 and 6 show data from a single run each of the networks. To establish that we can draw conclusions from these results, each experiment was run 10 times and the results are shown in Figure 7. Given more time, it would be advisable to repeat experiments and average the results before making conclusions. However, the results show good correlation between each run of the networks, and experiments are time consuming, therefore the rest of the experiments in this work will not be repeated multiple times.

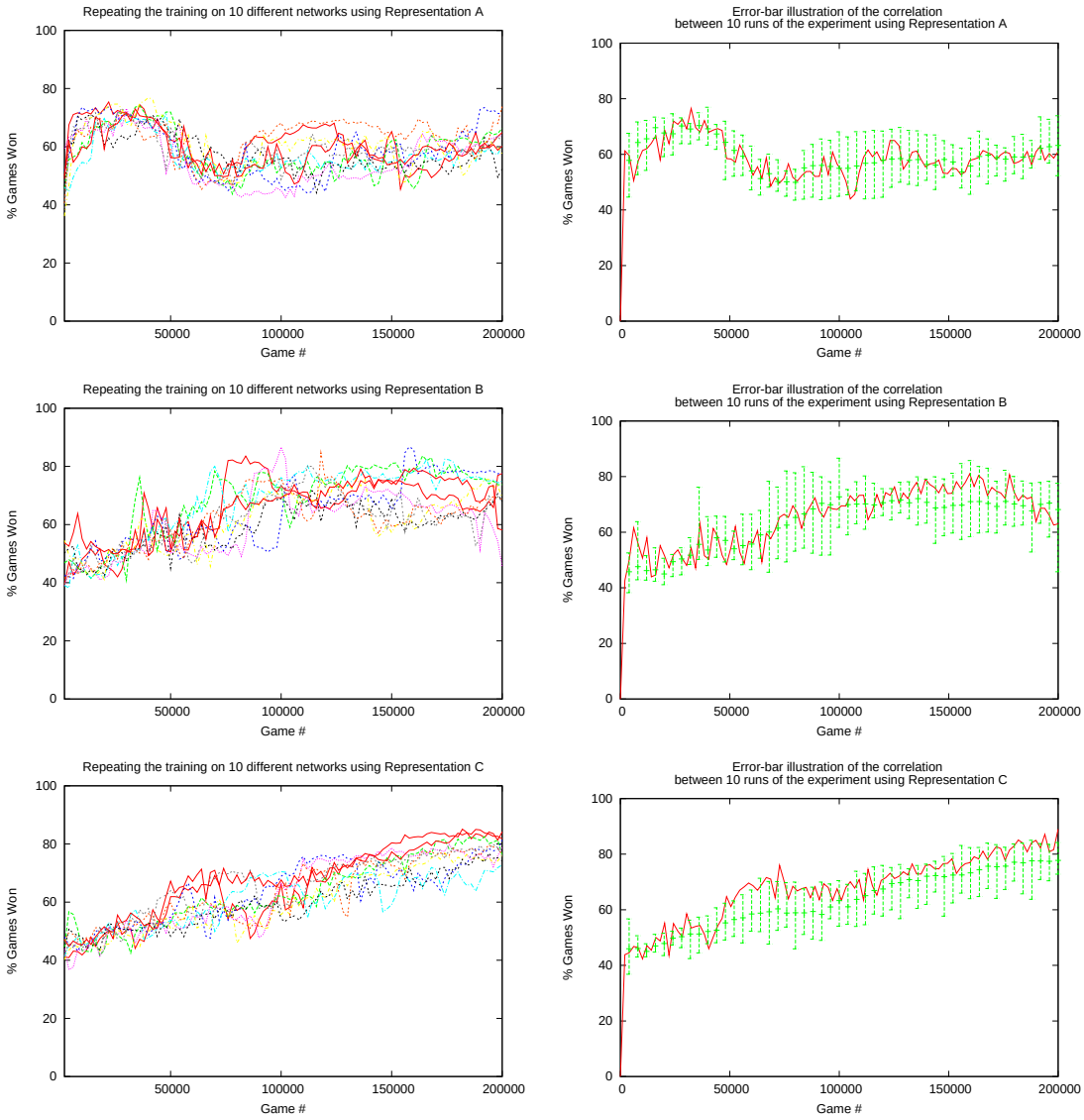


Figure 7:
Graphs to show the correlation between the behavior of multiple networks during repeated experiments.
Graphs in the first column show the behavior of 10 networks using Representations A-C. Graphs in the second column show the behavior of the first network trained and the maximum diversions from this behavior.

Plots of the performance of the network weights saved periodically during training closely follow those in Figure 6. Thus, we expect test set performance to track training performance. This is not surprising since we do not have a "training set" per se. Rather, training is on games played against a random player. Since testing is also done in this way (with the only difference being that learning is turned off) it is unsurprising that training and testing performance are essentially identical.

The weights saved at the highest percentage during training were played against the random player for 1000 games. Representation A (saved at game 35065) won 72% of the games, Representation B (saved at game 165403) won 76% and Representation C (saved at game 190913) won 85% compared to the teacher evaluation function which beats the random player on average 83% of the time. The neural network using Representation C learned a smoothed version of the static evaluation function and this seems to have performed better. It is clear from these figures that both Representation B and Representation C outperformed Representation A, supporting the theory that using '0' to represent an opponent's piece may be inadequate.

Although both networks taught using Representation B and C show comparable (if not better) performance to the Teacher against a random player, it was interesting to analyze their behavior when competing against the Teacher. Figure 8 shows the performance of the weights saved during training when competing against the Teacher.

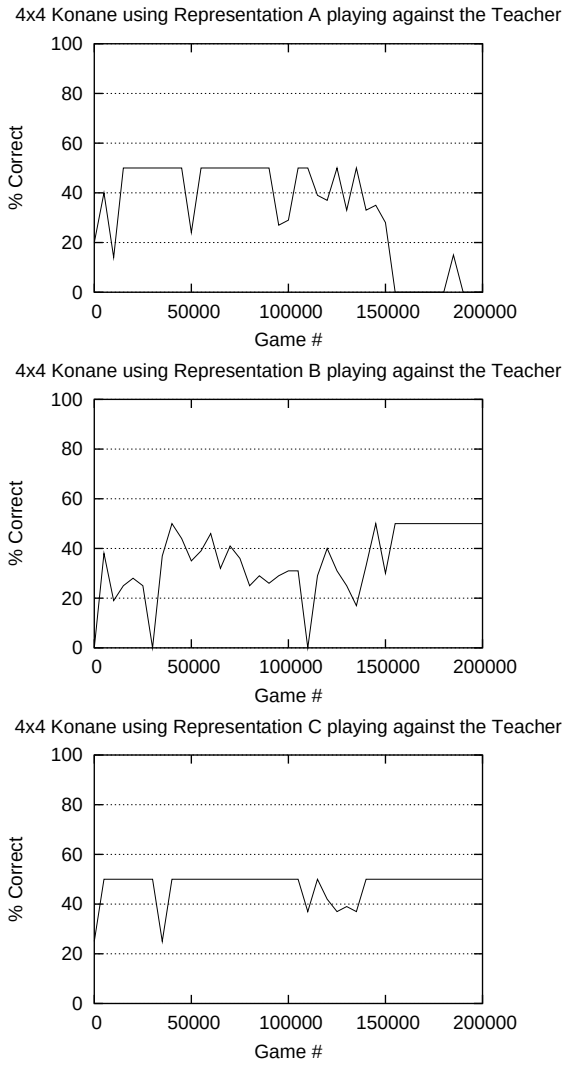


Figure 8: Comparing the behavior of the Neural Network players saved periodically during the learning process (trained to play 4x4 Konane using different board representations) when competing against the Teacher.

These graphs were of particular interest to me since the best performance of any of the saved weights was **exactly** 50% prompting more detailed analysis of 4x4 gameplay in Konane in section 6.

The networks were also tested by playing against the evaluation functions specified in section 4.1.1. Whereas the networks taught using Representation B and Representation C showed increasingly good performance against its opponents, even after 200,000 games, the network taught using Representation A showed a drop in performance after 75,000 games against various opponents with little improvement over the next 125,000 games. The behavior of the networks indicated that when using Representation B and C, they were still learning new board representations towards the end of the training phase. This prompted more analysis as shown in section 7.1.

5.1.4 Conclusions and Analysis

Representation A over-trained early in the learning phase, peaking below 75%. Although it has a nice feature in which the input nodes always represent a specific position on the board (whether the player is black or white), its performance is fundamentally flawed by its use of '0' to represent an element of the board which is so vital to the players evaluation of the board.

Representation B also began to over-train, however this was after a steady progression to a success rate close to that of the Teacher. Considering the fact that it uses an input vector half the size of Representation C, the performance of this network is by no means poor. It obviously benefits from having larger inputs to work with. On the other hand, the structure of the representation prevents the network from abstracting firm connections between the nodes since they represent different board locations depending on whether the player is black or white. For example, consider the following labeling of a 4x4 board:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

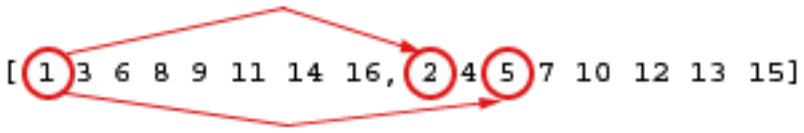
If the neural network player is black, the following positional representation will be used throughout the game (the comma represents the break between player and opponent spaces):

[1 3 6 8 9 11 14 16, 2 4 5 7 10 12 13 15]

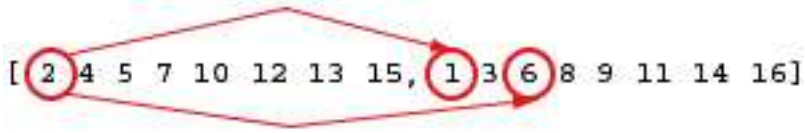
If the neural network player is black, the following positional representation will be used throughout the game (the comma represents the break between player and opponent spaces):

[2 4 5 7 10 12 13 15, 1 3 6 8 9 11 14 16]

Now if we focus on position 1, the most influential spaces affecting this position are 2 and 5. When the player is black, the following nodes are connected:



However, if the player is white and the same connections hold, then the board position 2 is connected to positions 1 6, but would not be connected to position 3 (arguably more influential than position 1):



Although the network managed to learn the evaluation function in this 4x4 game, this could potentially become a problem when training on larger boards. A solution to this problem would be to train two different networks to use the same representation, one to play when the player is black, and one when the player is white. This solution was explored more in section 7.3.

Representation C merged the beneficial properties of both Representation A and B, at the cost of the size of the network, and therefore time. This network learned to perform at the level of its teacher successfully. Each node is statically connected to a specific board position whether the player is black or white. It also provides a distinction between the player spaces and opponent spaces, a component that was so successful in Representation B.

5.2 Comparison Methods

Instead of using one board as an input vector to a neural network whose output node would represent the evaluation value, a different method was analyzed here. This method was to use two boards as input vectors to a neural network which would then compare the two board representations. If the first board is 'better' for the player, the network would return 1. On the other hand, if the second board is 'better' than the first board, the network would return 0. If neither board was more advantageous to a player then the network would return 0.5. This design was motivated by Tesauro's backgammon player[18] which used this sort of representation to become the world champion backgammon player. Tesauro used this representation on the observation that it is often easier to specify pairwise preferences among a set than it is to specify a ranking function for that same set.

5.2.1 Motivation

The structure of the networks used to test different board representations, consisted of 1 input vector representing a board state and 1 output node. The networks were trained to output a value (scaled between 0 and 1) representing the state of the board with regard to the player. 0 represents a complete loss for the player (no hope of recovery; a win for the opponent), 1 represents a winning board for the player and 0.5 represents a board with no advantage to either player. This setup is potentially severely flawed. Figure 9 shows a plot of the raw data collected during the learning phase of the network using Representation B. This figure shows how the network fluctuates dramatically between winning multiple games in a row and scoring high percentage wins, to losing and rapidly dropping in percentage wins.

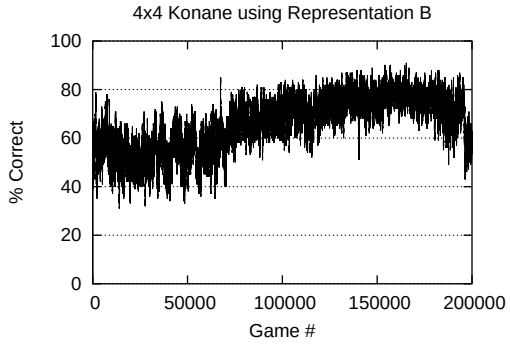


Figure 9: The performance of a network using Representation B during training. This data has not been smoothed.

What may be happening to these networks is that during a 'positive' phase, the majority of input vectors that the network is seeing are advantageous to

the player; i.e. the output training values are between 0.5 and 1. The weights are changed after every game, drawing the output value to be larger in general. Then, when the network sees a completely different game and all its outputs are skewed to be closer to 1, the network looses games. Likewise, during this new losing streak, the network sees input patterns, the majority of which favor the opponent and therefore have target output values between 0 and 0.5. In turn, this skews the weights to produce smaller values as the output. Granted, these two phases seem to cancel each other out in the long run, however this could potentially be avoided by directly comparing boards.

5.2.2 Setup

Since both Representation B and C were successful in their attempts at playing 4x4 Konane, I chose to test the comparison method using both representations. The network for this experiment using Representation B was set up as follows: Input layer of size 32 ($2n^2$), hidden layer of size 20 and an output layer of size 1. The network using Representation C had an input layer of size 64 ($4n^2$), hidden layer of size 30 and an output layer of size 1.

The first half of the input nodes were the relative representation of one board. The second half corresponded to the relative representation of a second board to which the first board would be compared. When choosing a move, the network player inputs the first valid board with the second. The teacher evaluates both boards and sets the output target to be 1 if the first board is better, or 0 if the second board is better. The best board is then compared to the next valid board and so on, until all boards have been examined.

To avoid learning non-symmetrical inputs, when each comparison is done during learning, the two boards are used as training inputs twice, the second time reversing their order in the input vector. The networks trained over 200,000 games against a random player.

5.2.3 Results

The networks both rose to and plateaued at around 80% rapidly as shown in Figure 10. This was an improvement from the original format of network when using Representation B, and was comparable to the original format when using Representation C. It is worth noting that unlike the original experiments on varying the board representation, both representations behaved similarly here, neither over-trained during the 200,000 training phase, and the behavior was consistent once it reached 80%. In fact, these networks display a much steeper learning curve than the original format networks, suggesting that this method of learning may scale up for use on bigger boards better than the original method of learning. However, the behavior of these networks during training does not show any noticeable improvement on the fluctuations during training as shown

in Figure 9. The benefits of using this network structure would be made clearer with a larger search space, therefore these experiments were later repeated using a 6x6 board.

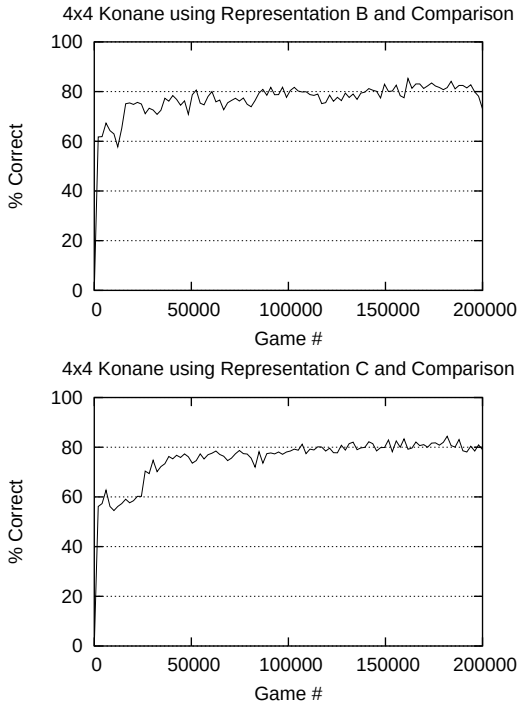


Figure 10: Comparing the behavior during training of Neural Networks to play 4x4 Konane using the comparison method and board representations B and C against a random player for 200,000 games.

5.3 Varying the size of the hidden layer

One board representation was chosen to test whether the standard hidden layer size chosen for the experiments was sufficient. The experiments were also run to determine what effect changing the hidden layer size has on the ability of the network player to recognize patterns in the board and therefore learn a static evaluation function for a Konane board.

5.3.1 Setup

Since Representation C was the most successful board representation when playing 4x4 Konane, I explored the effects of varying the size of the hidden layer using this representation. The network structure was set with 32 input nodes and 1 output node while experiments were run using a hidden layer size of even integers between 12 and 28.

5.3.2 Results

Given 32 inputs and 1 output, varying the hidden layer size had relatively no long term effect on the outcome, as shown in Figure 11. However, in Figure 12 it is apparent that the network using 12 nodes (the minimum number tested) appears to oscillate more violently than the larger hidden layers during the first 200,000 games, indicating that this is a sensible lower bound for a hidden layer size when dealing with networks of this input size.

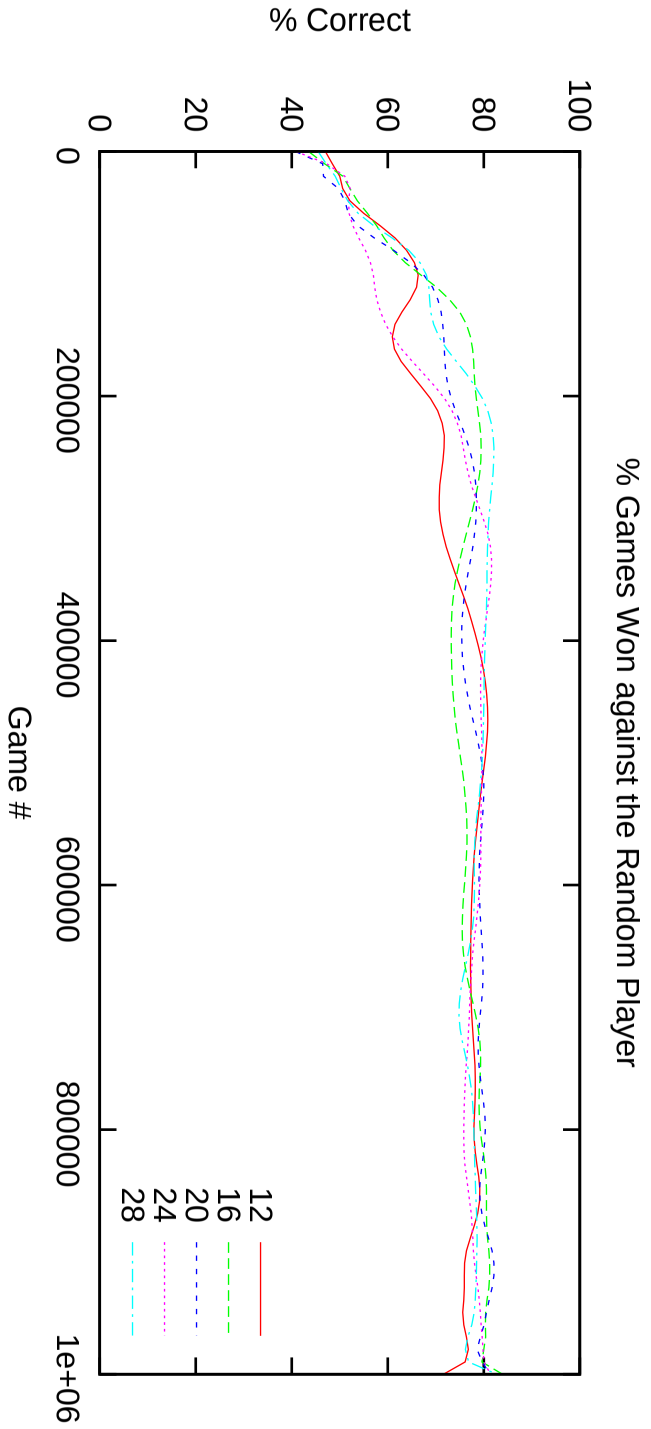


Figure 11: Testing Different Sizes of the Hidden Layer; Long Term Behavior

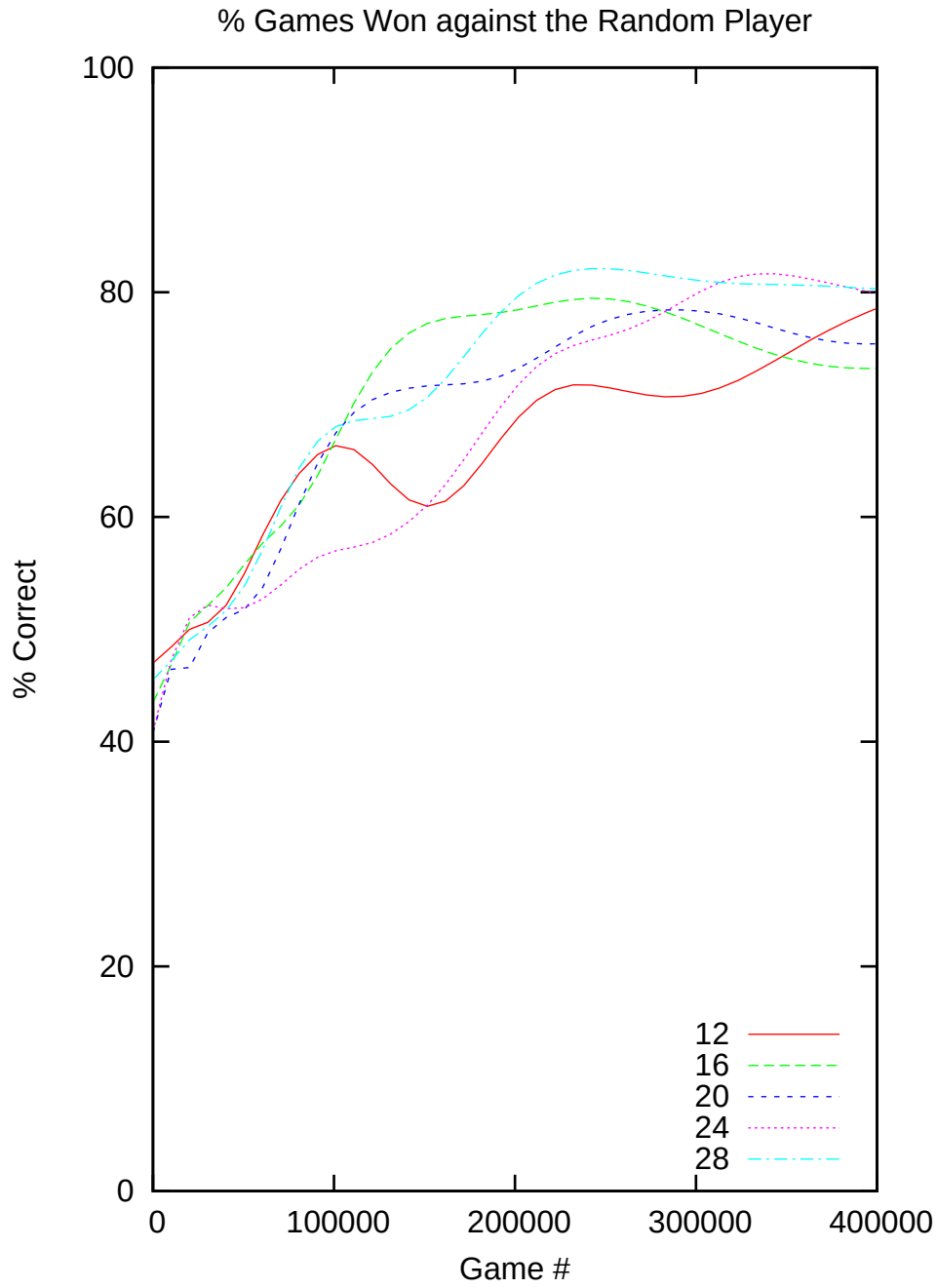


Figure 12: Testing Different Sizes of the Hidden Layer; Short Term Behavior

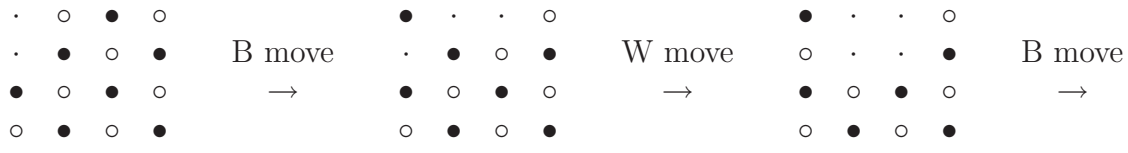
5.3.3 Training Procedures

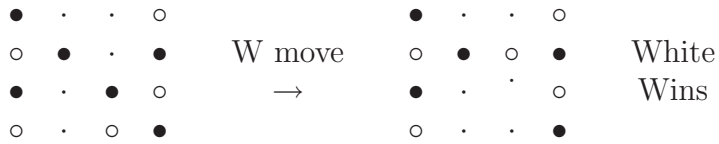
The performance of a neural network is sensitive to the proper setting of the learning rate and momentum. If the learning rate is set too high, the network may oscillate, become unstable and essentially over-train. If the learning rate is too small, the network will take too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and as a result it is often chosen by trial and error. The ideal network would use the highest values for the learning rate and momentum that still converges to the solution. In a 4x4 game of Konane, the search space is small and therefore there is a high chance of over-training when using a large learning rate and momentum. As a result, all experiments on 4x4 Konane were performed with a learning rate 0.01 and momentum 0.01. These values were varied during later training on larger boards.

A low learning rate and momentum were also used since the networks trained using a back propagation sweep on every available move, each move, during every game. As mentioned previously, with a search space as large as Konane, it is infeasible to collect a large enough data set and train offline. Unfortunately, this leaves us with a problem of data sampling. I could restrict the networks to learn only every other move. However, this could lead to a situation where the network has learned no full game play sequences and may therefore perform poorly in the majority of games against a random player, rather than playing well in a number of games, and terribly in a few games. Ideally I would implement a governor which would prevent the network from learning certain board layouts too frequently, however, this is not the focus of this work.

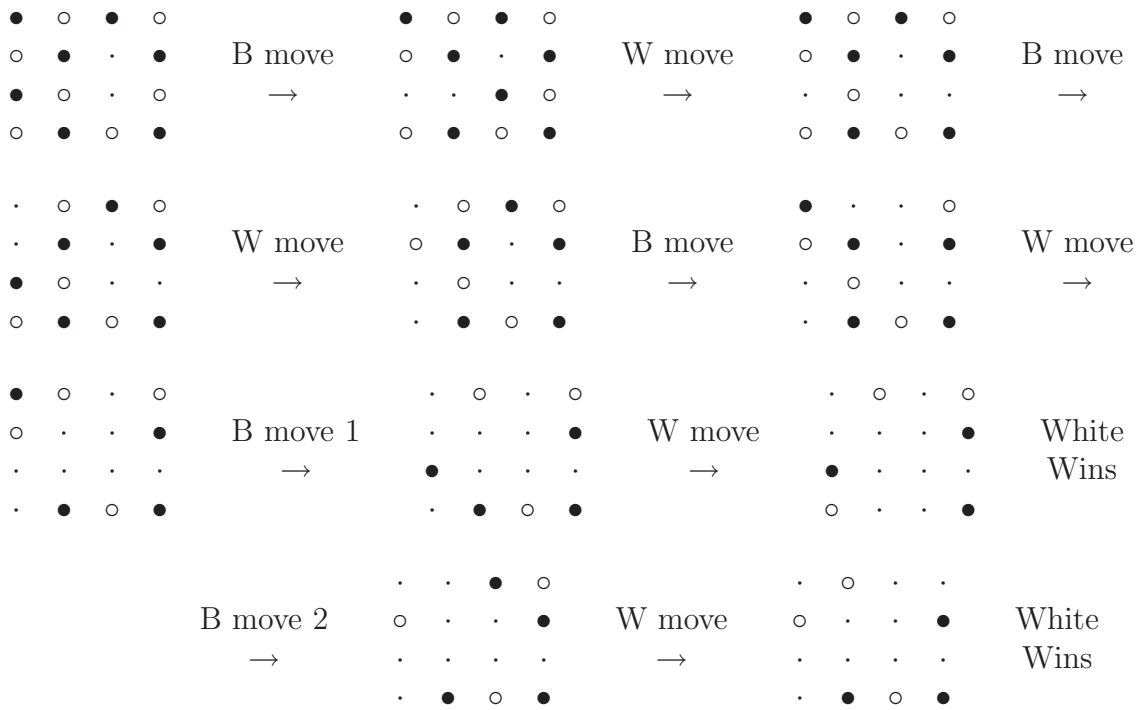
6 Overall Analysis of 4x4 Konane Experiments

Running test experiments on a 4x4 Konane board were primarily useful in terms of providing a relatively small, manageable search space to offer quick debugging and results. During the process of running these experiments, it became clear that the search space was not only smaller than the search space of a 6x6 Konane board, but holds a clear advantage to the second player, reducing the search space even further. In fact, when the search tree is explored, we can see that the white player can force the black player into a single move each time no matter what first move black starts with. In the following game, • starts by choosing to remove a corner piece:





White has a dominant strategy that can force a win. The same is true when • starts by choosing to remove a center piece:



Players using a 4x4 Konane board also lack the ability to multi-jump the opponent, also reducing the number of possible moves at each turn. Although the 4x4 game lacks the complexity of the standard 8x8 game of Konane, it is still considered successful that the networks taught using Representation C and the networks taught using the comparison method learned the dominant strategy as the white player (winning 100% of the time) and also, when playing a random player, learned to take advantage of a poor decision on the opponents behalf when playing as black, to come back and win 60% of the games.

7 6x6 and 8x8 Experiments

Since the 4x4 game of Konane does not include multiple jumps, and is biased towards the second player, some of the experiments performed were repeated on 6x6 and 8x8 boards. The following sections describe the networks trained on the larger boards.

7.1 Different Board Representations

Three networks were trained to play Konane on a 6x6 board using previously defined board representations A, B and C. The networks trained against a random player over a million games. As shown in Figure 13, their performance during training was surprisingly very similar. These results were surprising since the networks learning Konane on a 4x4 board were much more affected by a change in the board representation, but they trained for a similar number of games and, considering the 8x8 game search space is much larger, can we even compare these results?

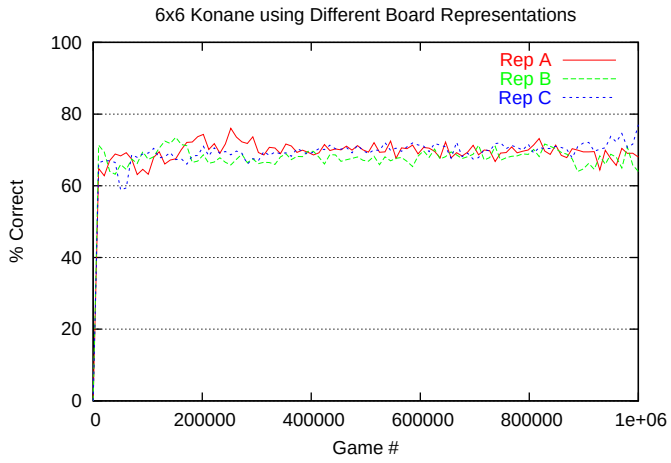


Figure 13: Testing Different Board Representations using a 6x6 Konane Board

The same results were observed when two networks were trained to play Konane on an 8x8 board using the representations B and C (See Figure 14). However, these networks only trained over 400,000 games. Still it is interesting to note that the networks consistently did better on an 8x8 board (where their best result is 83%) than on the 6x6 board (where their best result is 75%). After more analysis of the teacher evaluation function, it became clear that the teacher performs significantly better on the 8x8 Konane board than on the 6x6 board and the networks are obviously reflecting this difference.

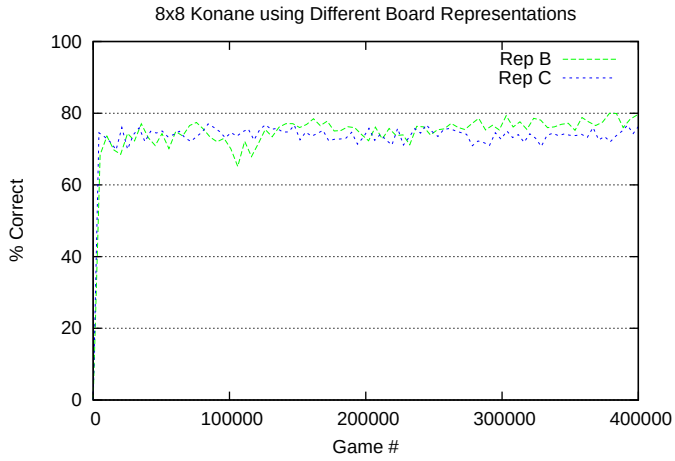


Figure 14: Testing Different Board Representations using a 8x8 Konane Board

Another question is raised; if the networks have reached a plateau and yet are not over-training after a million games, are they learning anything new during

this time? Figure 15 below illustrates the difference between the behavior of the network players saved periodically during training perform against the random player, teacher, and two randomly picked static evaluation function players from section 4.1.1

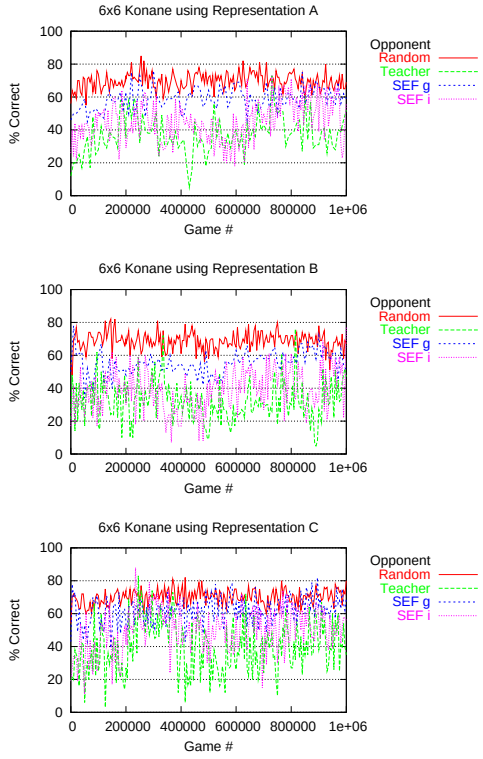


Figure 15: Comparing the behavior during training of Neural Networks to play 6x6 Konane. The networks use different representations and play against a random player, teacher, and static evaluation function players g and i

Although the performance of the networks against the random player barely fluctuates, the performance against the other strategic players is much more volatile. The networks are clearly still learning more board positions even after playing a million games. Given the results from the experiments performed on a 4x4 board, I would propse that we would observe similar differences between the behaviors of the networks using different representations if we were to train the networks over a much larger number of games. We might then observe how the different representations bound the networks capacity to learn more game boards.

7.2 Comparison Method

One network was trained using the comparison method and board representation B and another was trained using the comparison method and board representation C. The performace of these networks is essentially indifferent to that of the networks simply learning the real-valued evaluation of each board (as shown below), implying that nothing is gained from forcing the network to essentially learn a boolean comparison function rather than a continuous function. In fact, the comparison networks are also more computationally intensive since the network input size is twice the size of the traditional networks.

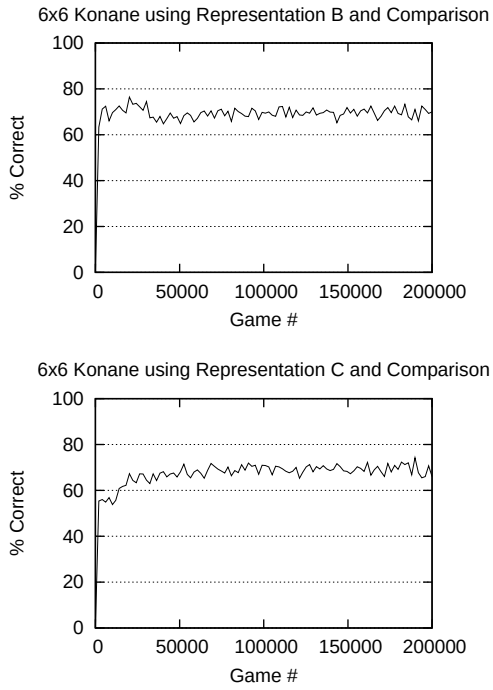


Figure 16: Comparing the behavior during training of Neural Networks to play 6x6 Konane using the Comparison method.

7.3 Black vs White Players

A network was trained using board representation B always playing black, and a second network was trained using the same representation always playing white. This was to determine whether the network would be able to outperform a network learning to play both black and white (resulting in the problem described in section 5.1.4). This experiment gave some unexpected results. The network player learning to play as white, rose quickly to a skill level of 90% against a random player. On the other hand, the network learning to play as the black player only rose to a skill level of 76% against a random player. This was not infact due to the networks themselves, instead this turned out to be a characteristic found in the static evaluation function chosen as the teacher. When 1000 games were played between a random player and the teacher, the teacher won 87% of the games when always going second, and won 78% of the games when playing as the first player (black). In this case, the networks were able to learn the static evaluation function of the teacher and perform at the same skill level. This behavior was learned over 200,000 games.

7.4 Training

To check that the networks were using at least the minimal hidden layer size capable of learning a static evaluation function on a 6x6 board, a network was trained using board representation B with a much larger hidden layer and another with a much smaller hidden layer. The typical network learning 6x6 Konane using board representation B has an input size of 36 and a hidden layer of size 20. Both the networks trained using a hidden layer size of 26 and 14 behaved very similarly during training to the typical network. There was no apparent benefit to enlarging or shrinking the hidden layer by this amount.

Networks were trained using both representation B and C and both the traditional and comparative methods with a 10% increase in the learning rate and momentum to observe what effects this has during the learning phase. These networks were only trained over 200,000 games. During that time they rose to the equivalent performance of the networks tested in sections 7.1 and 7.2 at a million games. The networks plateaued at around 150,000 games and showed no signs of an increase in skill level. This was not a significant rise in performance by any means. The networks trained using a higher learning rate simply plateaued at approximately 3% higher skill level against a random player. These experiments would have to be expanded over a much longer time period to show the long term effects of raising the learning rate and momentum.

On network experienced a surprising result; The network trained using Representation B suffered a dramatic drop from above 70% to approximately 40% perhaps due to a fatal reorganization of its hidden layer. This behavior is shown in Figure 17 below.

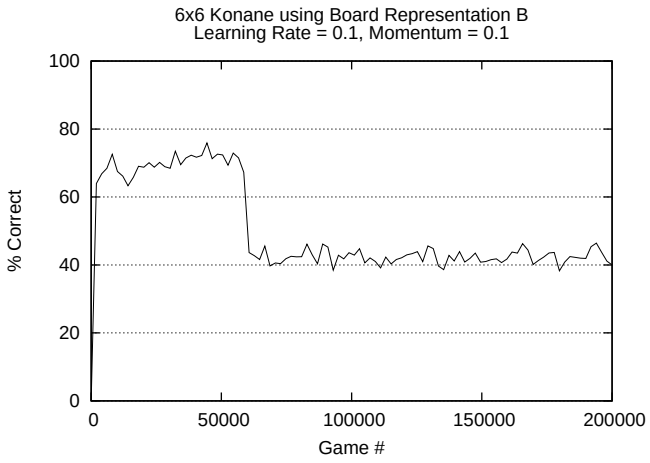


Figure 17: Testing Different Board Representations using a 8x8 Konane Board

7.5 Minimax

Unfortunately, no specific representation, comparison method or learning rate has stood out so far as being the 'optimal' choice for training a network. Therefore, experiments were run to test a networks capability of learning minimax using representations B and C. Networks were trained using both representation B and C to a Minimax depth of 2, 3 and 4. Again, the results were indifferent between the board representations, however, the networks at greater minimax depths plateaued at progressively higher skill levels when trained against a random player.

In the same way that searching to a deeper depth using Minimax helps overcome weaknesses in the static evaluation function, teaching a neural network to learn a static evaluation function to a deeper depth using Minimax results in a stronger network board evaluator. Additionally, the network player is faster than the Minimax algorithm paired with a static evaluation function since it has no need to expand and evaluate the board search tree. Instead of generating and evaluating multiple depths of boards, the network player need only generate and evaluate the first level of boards.

8 Future Work

There is a significant amount of work left to be done in this area. Although my work revolved around simply proving by existence the ability of a network to learn a static evaluation function for Konane using supervised learning, it touched on briefly investigating a few of the many parameters that could potentially be tuned to create the optimal network for this problem. I was able to experiment

with some factors; board representation, network size and comparison methods, however these experiments were completed on the 4x4 board which does not reflect enough of the nature of an 8x8 game of Konane, and the experiments ran on a 6x6 board were restricted.

Given time and enough processing power, it would be interesting to test more board representations on an 8x8 board, for longer periods of time and with varying hidden layer sizes and learning rates. I am curious to know if, given a board and search space as big as the 8x8 game, a network would ever experience catastrophic forgetting[7] or a reorganization of the hidden layer resulting in a jump up to a higher skill level after playing a considerable number of games. Even expanding the training of a network to play 6x6 Konane beyond a million games could potentially result in a jump up from the plateau that the networks in my experiments had been experiencing. I would also have liked to expand my experiments comparing the behavior of networks using a single board as an input and networks using multiple boards as inputs, into the 8x8 board domain. Future work should also include observing what effects a more in depth investigation into the optimal parameters for a network would be on the networks learning the Minimax algorithm to greater depths.

Investigation into using a more complex network structure could also be a potential area of expansion of this work. As an after thought of my experiments, the following network description would be interesting to implement. As usual, the input nodes would correspond to 1 Konane board (nxn inputs). The difference would be the use of two hidden layers. The first is only partially connected and is the same size as the input layer. Each node in the first hidden layer would represent a specific position on the board, and would only be connected to the nodes in the input layer corresponding to the locations directly adjacent to its position on the board. The two hidden layers would then be fully connected and would output 1 value as the evaluation of the board. I believe that the structure change in this network may better equip it to learn pattern matching in the game of Konane.

As mentioned previously, mathematicians have studied Konane, particularly focusing on 1xn games, and have been able to assign potential values to sub-games. This analysis is particularly useful near to the endgame, since pieces are far enough apart that they are considered sub-games. However, there is much more work to be done in the analysis of two dimensional Konane games, and the process of breaking game boards into sub-games.

The purpose of this work was not to evolve a master Konane player; it was limited to using supervised learning. Having said that, it would be interesting to analyze the strategy and performance of a network taught to play Konane using temporal difference learning, and a network evolved using a genetic algorithm.

Finally, there has been some initial research into evolving neural networks to focus Minimax search [Moriart and Miikkulainen]. In their conclusion, Moriart and Miikkulainen noted that their ‘focus networks can overcome not only errors

in the evaluation function but flaws inherent in Minimax itself', very useful features to implement along with supervised learning of a network. This research provided some positive results which, when merged with a network capable of approximating a static evaluation function, could be evolved into a skilled, efficient artificial player.

9 Conclusions

During the 6x6 and 8x8 experiments, weights were saved periodically and when the percentage of games won over the past 100 games was at a maximum. When comparing the performance of the networks during training, it is difficult to distinguish one from another. All of the networks quickly rose to a winning average of 75% and plateaued there. None displayed typical over-training behavior (unlike the experiments run on the 4x4 boards), with the exception of the network trained using a learning rate of 0.1 and board representation B.

One surprising result came from the network using representation A (thought to be fundamentally flawed as a result of its use of '0' to represent the opponent). Out of all the players in all networks playing 6x6 Konane (not performing Minimax) saved during training, the player saved at game 255000 performs the best against both the random player (83%) and its teacher (48%). However, this network supports my first hypothesis that a neural network can be taught, using supervised learning, to evaluate Konane boards as effectively as its teacher. In fact, the smooth approximation to the teacher's evaluation function leaves the network as the stronger player of the two when playing against various strategic players mentioned in section 4.1.1.

When comparing the weights saved at the best percentage game in each network, the skill level only varies between 68% and 85% wins against a random player. The majority of the networks win approximately 35% of games against their teacher, with the exception of the network using board representation A and the network using the comparison method and representation B which beats its teacher 45% of the time.

The networks searching to greater Minimax search depths outperformed the other networks, but only marginally. This is partially due to the time restraint placed on the training phase. However, initial results support the second hypothesis that a neural network can be taught, using supervised learning, to evaluate Konane boards to a depth greater than 1.

The most successful network trained on 8x8 Konane boards was trained using board Representation C at a search depth of 2 (deeper searches were too time consuming and therefore did not train through enough games to reach the skill level of this network). This network consistently beats the random player 87% of the time.

If these experiments were run multiple times over a much longer training

period and were tested against many different strategy players, we would be able to make more solid conclusions about the 'optimal' settings to train a network to learn Konane. However, multiple networks were successfully trained during this work and performed at a skill level comparable to the Teacher.

References

[1] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*, volume 1. A K Peters, Ltd., Natick, Massachusetts, 2nd edition.

[2] D. Blank, K. Konolige, D. Kumar, L. Meeden, and H. Yanco. http://www.pyrorobotics.org.

[3] Alice Chan and Alice Tsai. 1xn konane: A summary of results. 2002.

[4] Kumar Chellapilla and David B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation*, 5(4):422–428, 2001.

[5] Michael D. Ernst. Playing Konane mathematically: A combinatorial game-theoretic analysis. *UMAP Journal*, 16(2):95–121, Spring 1995.

[6] David B. Fogel. Evolutionary computation: Toward a new philosophy of machine intelligence. 1995.

[7] Robert M. French. Catastrophic interference in connectionist networks: Can it be predicted, can it be prevented? In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 1176–1177. Morgan Kaufmann Publishers, Inc., 1994.

[8] Joel H. Gyllenskog. Konane as a vehicle for teaching AI. *SIGART Newsletter*, (56), 1976.

[9] http://www.johnloomis.net.

[10] http://wikipedia.org.

[11] Deepak Kumar, 2004. http://mainline.brynmawr.edu/Courses/cs372/fall2004/.

[12] David E. Moriarty and Risto Miikkulainen. Evolving neural networks to focus minimax search. In *Proceedings of the twelfth national conference on Artificial intelligence*, volume 2, pages 1371–1377, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 2, pages 318–362. MIT Press, 1986.

[14] A. L. Samuel. Some studies in machine learning using the game of checkers. 3(2):210–229, April 1959.

[15] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of Go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing 6*, pages 817–824. Morgan Kaufmann, San Francisco, 1994.

[16] C.E. Shannon. Programming a computer for playing chess. *Phil. Mag.*, 41:256–275, 1950.

[17] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[18] Gerald Tesauro. Temporal difference learning and TD-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[19] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*, Cambridge, MA, 1995. MIT Press.

10 APPENDIX A

The following tables show the detailed results from section 4.2. Values along each row correspond to the row player's percentage of games won against the column player. The last column averages the performance of the row player against all other players. The best three evaluation functions at each level are in bold.

SEF	g	h	i	j	k	l	m	n	o	p	q	r	average
g		47	39	43	42	39	52	36	29	35	20	25	37.00
h	53		38	62	40	40	62	71	38	66	31	33	48.55
i	61	62		57	48	49	62	58	42	47	59	52	54.27
j	57	38	43		61	53	67	29	46	30	32	47	45.73
k	58	60	52	39		49	67	64	34	40	46	44	50.27
l	61	60	51	47	51		73	59	32	45	43	42	51.27
m	48	38	38	33	33	27		38	22	33	29	27	33.27
n	64	29	42	71	36	41	62		37	56	46	38	47.45
o	71	62	58	54	66	68	78	63		62	52	41	**61.36**
p	65	34	53	70	60	55	67	44	38		48	54	53.45
q	80	69	41	68	54	57	71	54	48	52		52	**58.73**
r	75	67	48	53	56	58	73	62	59	46	48		**58.64**

Table 2: Testing Evaluation Functions g-r against each other at depth level 1

SEF	g	h	i	j	k	l	m	n	o	p	q	r	average
g		44	38	42	38	28	51	56	32	45	34	33	40.09
h	56		35	19	21	18	33	51	32	49	20	16	31.81
i	62	65		58	43	36	61	75	40	65	33	32	51.81
j	58	81	42		40	49	59	49	41	61	26	26	48.36
k	62	79	57	60		57	62	73	37	66	41	31	56.81
l	72	82	64	51	43		75	64	56	78	32	40	59.72
m	49	67	39	41	38	25		49	28	38	34	34	40.18
n	44	49	25	51	27	36	51		28	37	36	23	37.00
o	68	68	60	59	63	44	72	72		66	41	58	**61.00**
p	55	51	35	39	34	22	62	63	34		21	42	41.63
q	66	80	67	74	59	68	66	64	59	79		55	**67.00**
r	67	84	68	74	69	60	66	77	42	58	45		**64.54**

Table 3: Testing Evaluation Functions g-r against each other at depth level 2

SEF	g	h	i	j	k	l	m	n	o	p	q	r	average
g		52	31	31	35	24	59	48	35	31	34	28	37.09
h	48		22	28	26	26	44	42	40	26	27	18	31.54
i	69	78		54	38	42	62	66	67	44	40	44	54.90
j	69	72	46		40	38	62	56	58	45	34	40	50.90
k	65	74	62	60		62	61	52	53	29	62	60	58.18
l	76	74	58	62	38		68	58	51	36	59	60	58.18
m	41	56	38	38	39	32		50	29	28	21	35	37.00
n	52	58	34	44	48	42	50		24	33	20	32	39.72
o	65	60	33	42	47	49	71	76		51	54	49	54.27
p	69	74	56	55	71	64	72	67	49		35	38	**59.09**
q	66	73	60	66	38	41	79	80	46	65		42	**59.64**
r	72	82	56	60	40	40	65	68	51	62	58		**59.45**

Table 4: Testing Evaluation Functions g-r against each other at depth level 3

SEF	g	h	i	j	k	l	m	n	o	p	q	r	average
g		58	43	38	35	41	57	54	31	27	29	32	40.45
h	42		30	21	26	28	40	33	22	14	19	23	27.09
i	57	70		51	53	40	69	62	46	40	45	37	51.81
j	62	79	49		40	30	56	52	35	25	33	34	45.00
k	65	74	47	60		53	63	68	45	61	42	51	57.18
l	59	72	60	70	47		66	58	39	44	47	50	55.63
m	43	60	31	44	37	34		48	41	37	34	31	40.00
n	46	67	38	48	32	42	52		17	20	25	24	37.36
o	69	78	54	65	55	61	59	83		51	37	43	59.54
p	73	86	60	75	39	56	63	80	49		51	40	**61.09**
q	71	81	55	67	58	53	66	75	63	49		49	**62.45**
r	68	77	63	66	49	50	69	76	57	60	51		**62.36**

Table 5: Testing Evaluation Functions g-r against each other at depth level 4

11 APPENDIX B

source code will be attached here