Training Neural Networks for Checkers

Daniel Boonzaaier
Supervisor: Adiel Ismail

2017

Thesis presented in fulfilment
of the requirements for the degree of
Bachelor of Science in Honours
at the University of the Western Cape

Declaration

I, Daniel Boonzaaier, declare that this thesis, "Training Neural Networks for Checkers" is my own work, that it has not been submitted before for any degree or assessment at any other university, and that all the sources I have used or quoted have been indicated and acknowledged by means of complete references.

Signature: . . . . . Date: . . . . . .

Printed Name: Daniel Boonzaaier

Abstract

This project will attempt to create a learning program that teaches itself how to play the game of checkers using neural networks and particle swarm optimization. Learning will be done by having the program play checkers numerous times against other neural networks and then evolve based on the outcomes of each game played.

Contents

Chapter 1

Introduction

1.1 Introduction

Machine learning is not a new concept in computer science. Arthur L. Samuels' "Some Studies in Machine Learning Using the Game of Checkers", was originally published in July 1959.

Machine learning is about computer algorithms that allow a computer to learn and is a related to a branch of statistics called computational learning theory. Neural Networks are but one type of Machine Learning methods that use input nodes that are connected to a hidden layer via different weights which are in turn connected to an output layer via more weights.

Checkers is a good game to train as it provides complete information. Which means that the complete status of any game is known to the players at all times throughout the game.

1.2 Proposal

This project will attempt to create a learning program for the game of checkers using a neural network and particle swarm optimization to autonomously learn to play checkers from scratch. In other words, the program will teach itself how to play checkers without any external help.

The main focus of this project will be training of a neural network by having the program play the game of checkers a multitude of times and then using particle swarm optimization to update the weights of the networks. By having the program play against itself the goal is that this program will learn from a predetermined set of rules how to play. Consideration will have to be made in order to make sure that the neural net for this program is not over trained.

The neural networks will be used to evaluate the the checkers boards and provide a score based on said board which represent the value of a move that had been made.

The particle swarm optimization will be used to train the neural networks. This will be done by having the particle swarm optimization fine tune the weights of the neural networks.

1.3 Project Plan

This project will have four main parts that will take place throughout the year of 2017 with each part of the project taking place in each quarter of the year.

The first part of the project which is the analysis of the project is the research into what the project will require and the analysis of said requirements from the stand point of the user and software. Researching past works related to the project as well as technologies and software related to the project will assist in guiding the projects development.

The second part of the project is the projects design and development. This entails the creation of a User Interface Specification and prototype. From this an Object Orientated Analysis and then an Object Orientated Design can be done. This will take a closer look at the setup of the neural network and other related software.

The third part of the project is the projects implementation where the design previously done will be used to create the projects learning program. The implementation will need full documentation.

The final part of the project will be the projects testing, evaluation and presentation. Here the created program will be tested to determine whether it works according to expectations and refined if needed.

Chapter 2

User Requirements

The program shouldn't require much from the user. The user simply needs to determine whether or not the program has developed in its playing abilities.

The program will need to play checkers in some way and learn the best moves to make in order to win.

How the neural net will work, its layout and application will need to be well thought out. The interface be it graphical or otherwise will need to be well planned.

The program simply needs to show that it has learned how to play checkers without outside help from a user. Playing against the program should be possible and could be done further but is not the goal.

Chapter 3

Requirements Analysis

There are previous works done on autonomous game playing systems that involve various games, checkers included. One such work, which was mentioned in the introduction, is Arthur L. Samuels implementation.

Another implementation was done by Nelis Franken and Andries P. Engelbrecht. Particle Swarm Optimisation was used in the implementation of their game playing program.

The paper which will be followed closely in this project will be the paper titled; "Evolving Neural Networks to Play Checkers Without Relying on Expert Knowledge", by Chellapilla and Fogal.

Thus one implementation of the checkers game learning/playing program may be done using the neural net in conjunction with gradient decent in its back propagation or the second option would be to use the particle swarm optimisation technique instead of the back propagation.

All possible moves need to be analysed and the system should determine which move is the best one to make in order to win. The program will need a look ahead to determine possible moves. Only a single look ahead or one ply search depth will be allowed.

Testing of the program should be as simple as seeing whether or not the program follows the rules set out for the game of checkers, if it has learned to play the game properly and that the trained neural networks have improved in their game playing abilities.

Chapter 4

Interface

This program will be interacted with in two different ways. The first is when the training of the neural network is taking place. The second is that of the checkers game.

4.1 Interface for training

There will not be a graphical user interface for the training portion. The interaction that takes place during the training will not be accessed by a user and will only have output to show what is happening during the training process. This output will be visible from within an integrated development environment (IDE) such as PyCharm.

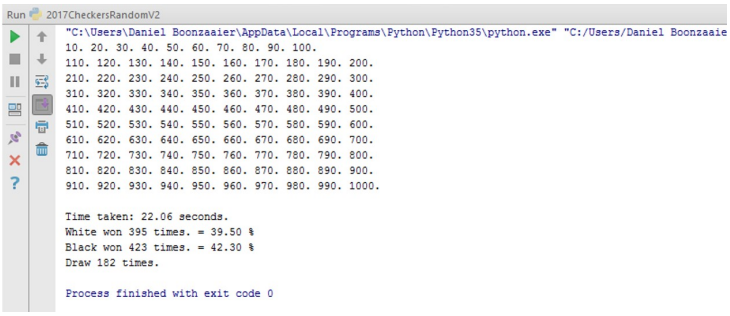

Figure 4.1: Example of output for checkers games played.

4.2 Interface for the game

The game of checkers that a user can play, will have a graphical user interface. It will have a simple start button and quit button on the first screen.

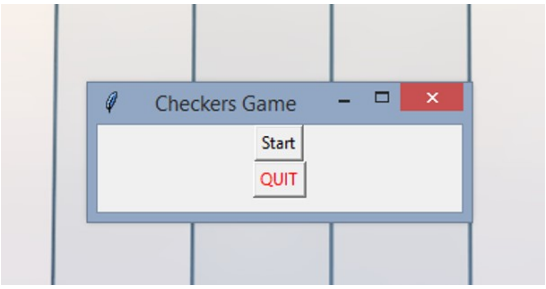

Figure 4.2: Simple Checkers Game GUI.

Once the start button is pressed, a game of checkers will begin where the user will play against the trained neural network. The quit button will end the program.

The game board will be shown and the user can click on a piece to move and then on the tile the player wishes to move said piece to. If the move is valid then the piece will move otherwise an error message will be shown telling the user that it is an invalid move.

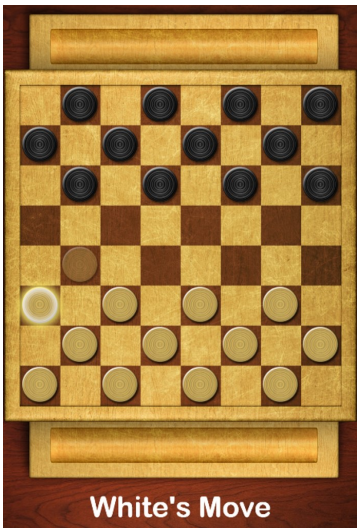

Figure 4.3: Example of Checkers Game Board GUI.

When it is the programs turn to play, the board will be evaluated by the neural network and its selected move will be made. The player may exit the game at any time.

Chapter 5

High Level Design

This high level design will attempt to give an explanation of the components that will make up this project. A brief explanation of each component will be given and how these components will interact.

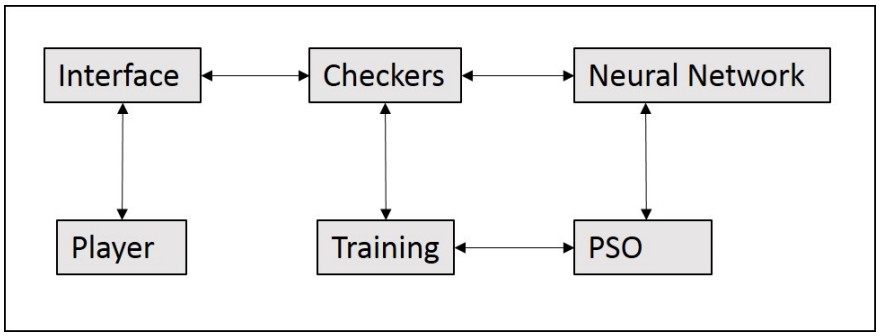

Figure 5.1: Interaction of Components.

5.1 Interface

The interface will be the playable game that a user will interact with. This Interface will interact with the checkers game which controls the game being played and the rules for checkers.

5.2 Player

The Player here refers to any user which will play the game of checkers. The player interacts with the interface which in turn interacts with the checkers game. The player will indicate to the interface what the player wants to do and the interface will interact with Checkers to determine whether whatever the player wants to do is valid for the game.

5.3 Checkers

Checkers refers to the rules and computations behind the interface for the game to work. It will be responsible for the rules of the game and for the neural network that plays against a Player. Checkers will interact with the Neural Network when any game is being played, during training as well as during Training. Checkers also interacts with Training.

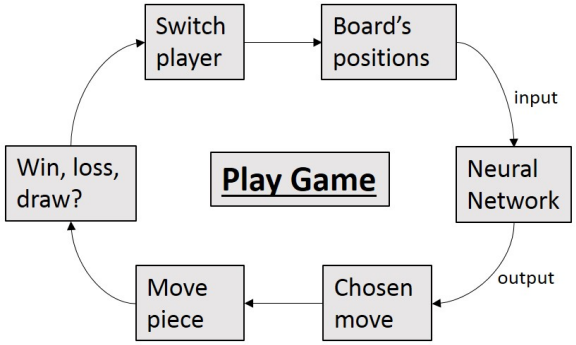

Figure 5.2: Basic Depiction of program playing checkers for Training.

5.4 Training

Training will be done before any user can play a game of checkers. Training involves playing a game of Checkers where the neural network for each agent in the training phase determines which moves should be done. Once a number of games have been played this way a score, or victory score, is calculated from the number of wins, losses and draws. Plus one for a win, minus two for a loss and zero for a draw. Training interacts with the PSO to update the weights of all the agents Neural Networks after each agents score is calculated.

5.5 Neural Network

The Neural Network will work for any board by taking in a thirty two vector input of all board positions. The Neural Network will then output a score based on the positions of the board. This score is obtained for each possible move and the move with the greatest score should be chosen as the best move for that Neural Network.

5.6 PSO (Particle Swarm Optimization)

The Particle Swarm Optimization will work on all the vectors made up of all the weights for each agents Neural Network. The victory scores of the agents will be used to determine the global best. The weights in the vectors will be updated according to a velocity function that takes into account the global best and each agents personal best.

Chapter 6

Low Level Design

The low level design will provide more specific details on the components discussed in the high level design. It will attempt to provide a clear definition of the algorithms used in the creation of the program.

6.1 The Checkers Game

The core of this project is the neural network which will be trained to play checkers. However, to do this, one must first have a checkers game to play. The algorithm with which a game will be played is as such:

6.1.1 Play Checkers Game Algorithm for random moves

1. Run through all current players checkers pieces.
2. If Piece is a normal piece.
   (a) Check positions left forward diagonal and right forward diagonal.
   (b) If position is open then add to list of possible moves.
   (c) If position contains opponents piece.
      i. Check positions diagonally behind opponent piece position.
      ii. If position contains any piece ignore.
      iii. If position is open then check for further jumps until all jumps have been exhausted, steps 2(a) and 2(c), then add to list of possible jump moves.
3. If Piece is a King.
   (a) Check positions left forward diagonal, right forward diagonal, left back diagonal and right back diagonal.
   (b) If position is open then add to list of possible moves.
   (c) If position contains opponents piece.
      i. Check positions diagonally behind/in front of opponent piece position.
      ii. If position contains any piece ignore.
      iii. If position is open then check for further jumps until all jumps have been exhausted, steps 3(a) and 3(c), then add to list of possible jump moves.
4. If there are possibilities to jump and take opponents piece one of these must be chosen at random.
5. Else choose random move from all possible non jump moves.
6. If all opponents pieces have been removed from the board. Current player wins. Stop if game has been completed.
7. Change player. If Player 1 change to Player 2. If Player 2 change to Player 1.
8. Continue above steps until game reaches a conclusion or after 100 moves resulting in a draw.

6.1.2 Finding Checkers Moves

A list of all possible moves will be created for the 32 possible positions on the board where pieces can occur. Then a list of the possible moves for a certain board can be extracted from this list by comparing the positions that are open and which contain pieces to determine which moves are valid. The algorithm to create said list of all possible moves is created as such:

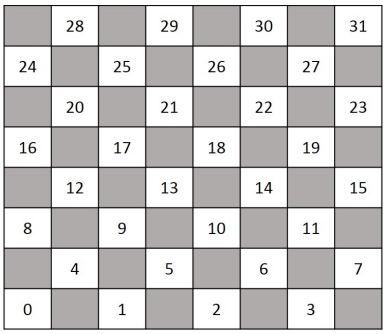

Figure 6.1: Checkers Board with positions 0-31.

List of possible moves:

- P = `[0, 1, 2, ..., 31]`
- moves = `[]`
- for all n in p
  - if `(n % 8) >= 0` and `(n % 8) < 4` then
    - right move = n+4
    - if `n % 8 = 0` then left move = no move
    - else left move = n+3
  - if `(n % 8) >= 4` and `(n % 8) < 8` then
    - left move = n+4
    - if `(n % 8) = 7` then right move = no move
    - else right move = n+5
- append `[n, [left move, right move]]` to list moves

6.2 Neural Network

The neural network that will be used as the brain of the checkers playing program will first be trained. This training will be done to ensure that the neural network used will be able to play the game of checkers and play the game to a decent competency. At minimum the neural network should play better than a game played against that of randomly chosen moves.

In the training of the neural network, the neural network will be used to evaluate board positions and then output a score of the board. These scores will be attributed to each of the available moves and the move with the highest score should be chosen as the best move that should then be carried out. This step will continue throughout the entirety of each played game. This part of the training is referred to as feed forward.

6.2.1 Neural Networks use within a game of checkers

The move that should be played will be obtained by using the neural network to evaluate all the possible valid moves as such:

- maxScore = Set very large negative number
- movesList = Obtain all possible valid moves
- index = will contain the index of selected move
- For each move in MovesList do:
  - boardCopy = create a copy of the games board. The game board consists of a vector of size 32. Each element in the vector represents a place on the board, as can be seen in figure 6.
  - Perform the current move and update the vector boardCopy
  - value = Use boardCopy as input for the neural Network and obtain output. The output being a scalar value.
  - If value is greater than maxScore then
    - Make maxScore equal to value
    - Set index to equal the index of the current move
- Return index as the move that should be played

6.2.2 Details on the Neural Network

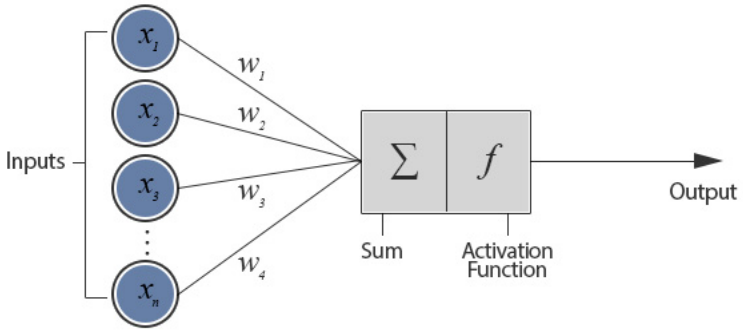

Figure 6.2: Neural Network Example.

The vector of 32 board positions work as the input for our Neural Network. Then at each node in the hidden layer its activation value is calculated via a summation function and sigmoidal evaluation function.

The summation function works, given figure 6.2 above, as such:

$$Sum = X_1W_1 + X_2W_2 + X_3W_3 + ... + X_nW_n,$$

where X stands for the values of the inputs and W stands for the values of the weights.

The sigmoidal evaluation function is as such:

$$f(n) = \frac{1}{1 + e^{-n}}$$

where n stands for the value of the node.

6.3 Particle Swarm Optimization (PSO)

In order to improve the neural networks, that will be played against each other during training, some back propagation needs to be done in order to update the weights of the neural networks. Updating the weights iteration after iteration will lead to the neural networks evolving and eventually produce one or more networks that will be used in the final checkers game.

A PSO algorithm will be used to update the weights of the neural networks. The PSO will work on all the weights of each neural network in a population of a certain size. The weights will make up a vector with each vector corresponding to a particle in the algorithm. Therefore if the swarm is of size n, there will be n particles representing the weights of each neural network, plus a copy of each particle which represent its personal best.

Each particle is updated according to two best values. The first is the best value that the particle has achieved so far, which is the personal best or pbest. The second is the best value is that of the best out of all the particles in the swarm, which is the global best or gbest.

6.3.1 PSO Algorithm

- Initialise particles from weights of neural networks
- For each particle
  - Calculate the fitness value
  - If the fitness value is better than the best fitness value, pbest, then set the current value as the new pbest.
- Choose the particle with the best pbest and set it as the gbest
- For each particle
  - Calculate the particle velocity
  - Update the particles position
- Repeat above steps for certain number of iterations or until minimum error criteria is met.

6.3.2 Particle velocity function

$$V = V + (C1 * rand(0,1) * (pbest - present)) + (C2 * rand(0,1) * (gbest - present)),$$

where c1 and c2 are learning factors, rand(0,1) is a random value between zero and one and present is the particle as it currently is.

6.3.3 Function for particles position

$$Present = Present + ParticleVelocity.$$

6.3.4 Fitness value

The fitness value is calculated for each particle, which is a vector of each neural networks weights. The fitness value is calculated after each neural network has played a number of games against other randomly selected neural networks.

After each game the fitness value is updated by:

- +1 for a win
- -2 for a loss
- 0 for a draw

Chapter 7

Implementation

7.1 Developed on:

- Windows 8.1 / Ubuntu 16.04.1
- Python 3.5
- PyCharm Community Edition 2016.2.2
- Python modules used: math, time, random

7.2 Introduction

The aim of this project is to create an intelligent game playing agent using neural networks and particle swarm optimisation for the game of checkers.

What follows, are the names of the classes that are used for training the neural networks and playing the game of checkers. Only the significant functions have been listed below.

Following that are a number of challenges encountered while coding this project.

Following that is a summary of the results obtained from the training performed.

7.3 Core Code Documentation

7.3.1 Implementation Algorithm

The algorithm used to train the neural networks using particle swarm optimization is as follows:

For 15 iterations:

1. Create population of players which are the neural networks.
2. Instantiate PSO class and associated variables. Associated variables are; initial fitness, index of the best initial fitness and the weight vector of each network in the population.
3. For 500 iterations:
   (a) Update the personal best and global best particles according to new fitness.
   (b) Update each particles velocity according to velocity function.
   (c) Calculate Fitness of all players in population.
4. Obtain network with global best fitness from population.
5. Test global best vs random played moves 10000 times as white player.
6. Test global best vs random played moves 10000 times as black player.

This algorithm produces fifteen trained neural networks that had the best fitness for each of its respective generations. The results from these fifteen neural networks are what is recorded and of these two was chosen to play against a human player. One neural network to act as a white player and another to act as the black player. This also gives the human player the option to watch these two well trained neural networks play each other.

7.3.2 Fitness Algorithm

The fitness of each neural network or "player" in the population is calculated as such:
For 15 iterations:

1. Create varible fitness and set to zero.
2. Play a game of checkers with the nueral network for which the fitness is for as player one and it's opponant neural network as player two. Save the result of the game.
3. If player one is the victor:
   (a) Fitness plus 1.
4. If player two is the victor:
   (a) Fitness minus 2.
5. If the game was a draw:
   (a) Fitness plus 0.

For example, one outcome of using this algorithm to find the fitness for a "player" is, the "player" wins two games, which is a plus two, loses one game, which is a mius two, and draws the others which results in a zero. This would lead to a fitness of zero.

This fitness value represents how well the neural network evaluates the chekers boards, or how well the "player" can play the game of checkers.

7.3.3 Checkers Algorithm - for two Neural Networks

In order to play a game of checkers with two neural networks or "players" from the population playing each other, the following algorithm was implemented:

1. Start Game
2. For 100 iterations:
   (a) For player one's turn:
      i. Obtain all availble moves that can be made and store to an array.
      ii. If there are no jump moves and there are no single moves:
         A. Return that player two wins as there are no moves for player one to make.
      iii. If there are no jump moves:
         A. Calculate the score of each available move using the nueral network feed forward function. Store each score in an array.
         B. Obtain the index of the highest score from the array of all the scores.
         C. Make the move using the index of the highest score.
      iv. If there are jump moves:
         A. Calculate all possible jumps paths from original jump moves and store in new array.
         B. Calculate the score of each available jump path using the nueral network feed forward function. Store each score in an array.
         C. Obtain the index of the highest score from the array of all the scores.
         D. Make the jump using the index of the highest score.
   (b) For player two's turn:
      i. Obtain all availble moves that can be made and store to an array.
      ii. If there are no jump moves and there are no single moves:
         A. Return that player two wins as there are no moves for player one to make.
      iii. If there are no jump moves:
         A. Calculate the score of each available move using the nueral network feed forward function. Store each score in an array.
         B. Obtain the index of the highest score from the array of all the scores.
         C. Make the move using the index of the highest score.
      iv. If there are jump moves:

A. Calculate all possible jumps paths from original jump moves and store in new array.
B. Calculate the score of each available jump path using the nueral network feed forward function. Store each score in an array.
C. Obtain the index of the highest score from the array of all the scores.
D. Make the jump using the index of the highest score.

(c) Check if there is a winner.

(d) If there is a winner:

i. Return the result of the game

3. If after the 100 iterations are complete and there is no victor then return the game as a draw.

In this algorithm you can see the difference from that of a checkers game played with only randomly selected moves, which can be found under the low level design in chapter 6.1 . Now all the moves in the game need to be scored by the neural network or "player" and that score determines which move should then be made. The neural network essentially works as the player's brain, selecting which moves should best be played according to a given board.

7.3.4 Calculate Jump Path Algorithm

Calculating the jump paths is a recursive function created to find all the different branching paths that a piece can take when there are subsequent jumps available after the initial jump has been made. This is needed as all the possible paths represent different moves that can be made and each of these moves needs to be evaluated by the neural networks playing the game of checkers.

This works by using a copy of the checkers board so that jumps can be made and then subsequent jumps found where available. Function name is calculateJumpPaths.

1. Get the type of the piece to move. ie. if the piece is a normal piece or king. This determines which direction to look for more jumps.
2. Make initial jump.
3. Find new available jumps according to the type. Store new jumps into an array.
4. Create an empty array for the output.
5. If there are no new jumps:
   (a) Append jump to output array.
6. Else if there is 1 new jump:
   (a) Append jump to output array.
   (b) Append calculateJumpPaths for the new jump to output array.
7. Else if there are 2 new jumps:
   (a) Append jump to output array.
   (b) Append calculateJumpPaths for the first new jump to output array.
   (c) Append calculateJumpPaths for the second new jump to output array.
8. Else if there are 3 new jumps:
   (a) Append jump to output array.
   (b) Append calculateJumpPaths for the first new jump to output array.
   (c) Append calculateJumpPaths for the second new jump to output array.
   (d) Append calculateJumpPaths for the third new jump to output array.
9. Else if there are 4 new jumps:
   (a) Append jump to output array.
   (b) Append calculateJumpPaths for the first new jump to output array.
   (c) Append calculateJumpPaths for the second new jump to output array.
   (d) Append calculateJumpPaths for the third new jump to output array.
   (e) Append calculateJumpPaths for the fourth new jump to output array.
10. Return the output array.

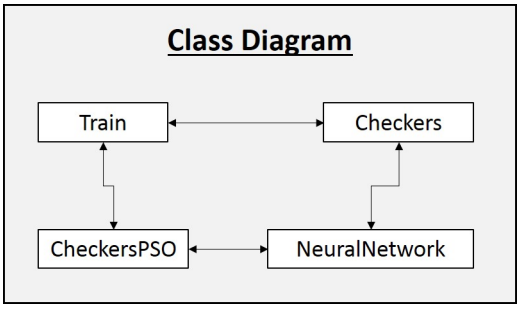

Figure 7.1: Class Diagram depicting how the classes interact with each other for training.

7.3.5 Checkers Class

This class contained all the necessary functions to run the checkers game. This includes the rules of the game, moving pieces, calculating jump paths, showing the board, etc. as well as different types of game functions for different purposes, such as random player versus random player, neural network versus random player, neural network versus neural network and neural network versus a human player.

showBoardV2 () - This function prints out a representation of the checkers board.

gameTestNN (bool, nn1, nn2) - This function is used to play a game of checkers where two neural networks play each other.

gameTestNN_White (bool, nn) - This function is used to play a game of checkers where a neural network plays white against randomly selected moves for black.

gameTestNN_Black (bool, nn) - This function is used to play a game of checkers where a neural network plays black against randomly selected moves for white.

playGameAsWhite (bool, nn) - This function is used to play a game of checkers where a neural network plays black against a human player that plays white.

playGameAsBlack (bool, nn) - This function is used to play a game of checkers where a neural network plays white against a human player that plays black.

7.3.6 NeuralNetwork Class

The main work this class performs is the calculation of the fitness for each board passed to it from the Checkers class. It also interacts with the CheckersPSO class by passing a weight vector back and forth.

updateWeights (weight_vector) - Updates the weights of the neural network with that of the weights passed to it via the weight vector.

calculateHiddenNodes () - Calculates the hidden nodes of the neural network.

calculateOutputNodes () - Calculates the output nodes of the neural network.

feedForward (input_vector) - runs the neural network and returns a fitness value for an input vector passed to it which is a board vector. The ouput for this function is a decimal value between 0 and 1. For example one output of this function might be the value 0.377452, another 0.856331.

7.3.7 CheckersPSO Class

The main work this class performs is the calculation of the fitness for each board passed to it from the Checkers class. It also interacts with the CheckersPSO class by passing a weight vector back and forth.

updateWeights (weight_vector) - Updates the weights of the neural network with that of the weights passed to it via the weight vector.

calculateHiddenNodes () - Calculates the hidden nodes of the neural network.

calculateOutputNodes () - Calculates the output nodes of the neural network.

feedForward (input_vector) - runs the neural network and returns a fitness value for an input vector passed to it which is a board vector.

7.3.8 Train Class

This class brought all the pieces together to train the neural networks, utilising functions from the Checkers, NeuralNetwork and CheckersPSO classes.

calculateFitness () - Calculates the fitness of all neural networks by playing the neural networks against others randomly in the population. Fitness is obtained as a plus one on a win and minus two on a loss. Makes use of the Checkers and NeuralNetwork classes.

obtainBestPlayer () - Trains the neural networks and returns the neural network which is the global best. Makes use of the CheckersPSO class and calculate fitness function.

testPlayerWhite (player) - Tests the neural network player as white versus a player that makes randomly selected moves as black. Makes use of the Checkers class.

testPlayerBlack (player) - Tests the neural network player as black versus a player that makes randomly selected moves as white. Makes use of the Checkers class.

7.4 Graphical User Interface

The following graphical user interface was created using pythons tkinter module. The GUI provides a way for a human player to play against the trained neural network. A player may select to play as either white or red. When the button for either white or red is selected it will be highlighted to provide feedback to the user.

Figure 7.2: Start Windows

The GUI provides feedback for all moves that have been made throughout the game so that the player can keep track of any moves that have been made. The heading of the GUI will change according to whose turn it is, i.e. when it is the white players turn to play it will say White's Turn. The bottom of the GUI provides extra information such as errors, to say when a move can't be made, otherwise it will provide encouragement to the player.

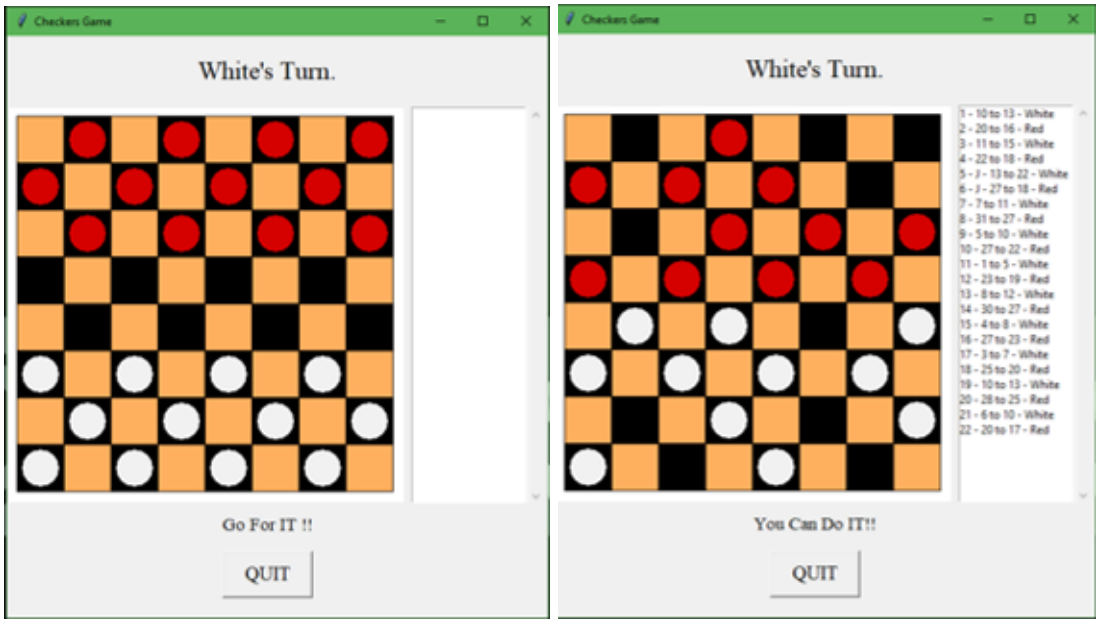

Figure 7.3: Checkers Board Windows

At any time during the game the player may select the quit button to stop playing the game. If the quit button is selected a prompt will be shown to ask the user for confirmation before exiting the game.

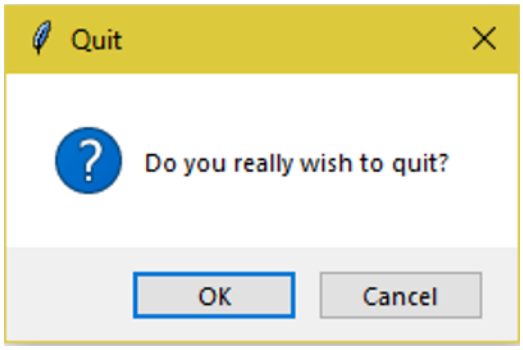

Figure 7.4: Quit Window

When the game is played to a win, loss or a draw, a window will pop up to inform the player of the result and then the game will close once the player has acknowledged the result.

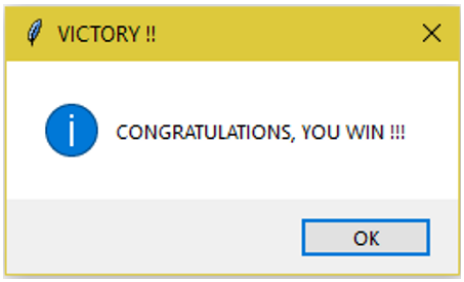

Figure 7.5: Victory Window

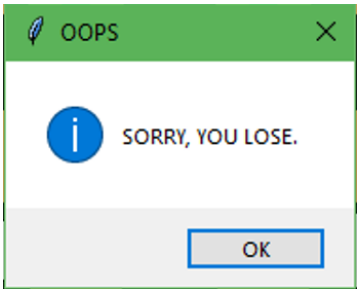

Figure 7.6: Loss Window

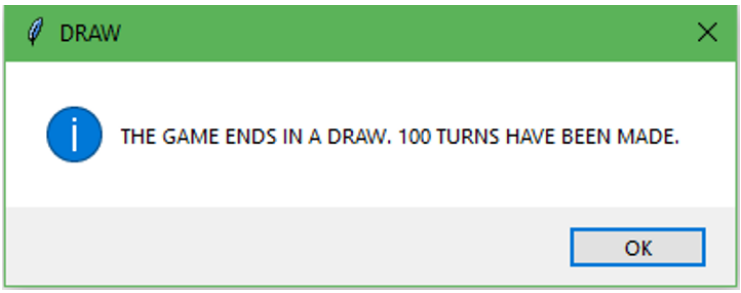

Figure 7.7: Draw Window

7.5 Coding Challenges

7.5.1 Jump Paths

There were various challenges involved with every part of the implementation for this project. However the most challenging part of the coding, I found, came from the checkers game itself, specifically with the rule that allows any piece in the game to jump multiple times so long as there is a piece that it may take.

This rule allows for a jump to take many different paths that all need to be addressed so that the neural network may evaluate each and then be able to choose the best jump to be made according to its calculation. This can be seen in the figure below which shows a board where the white player has an option of three different jumps. These jumps are 24 to 29, 14 to 23 or 14 to 21 to 28.

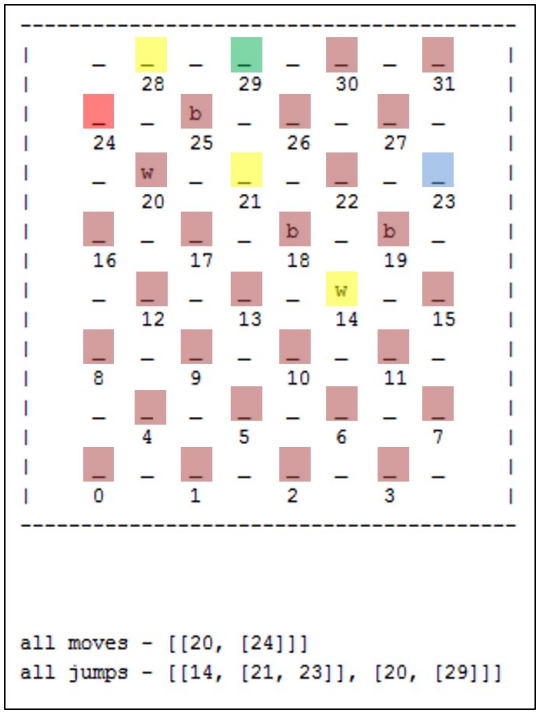

Figure 7.8: Checkers board displaying different possible jumps.

The solution I came up with works using a number of steps. It uses three functions to produce a list of all possible jump paths from a given jump. These functions are; calculateJumpPaths (), createJumpPathsBuilder () and flattenJumpPathArray ().

This solution works by using a recursive search to find each branching jump and appending them to a list. This list is then flattened so that instead of the list of lists that was produced it is instead one long list. This flattened list is then used to extract all the jump paths that are available to the starting jump and returned to be evaluated.

The following figure is an example of the solution.

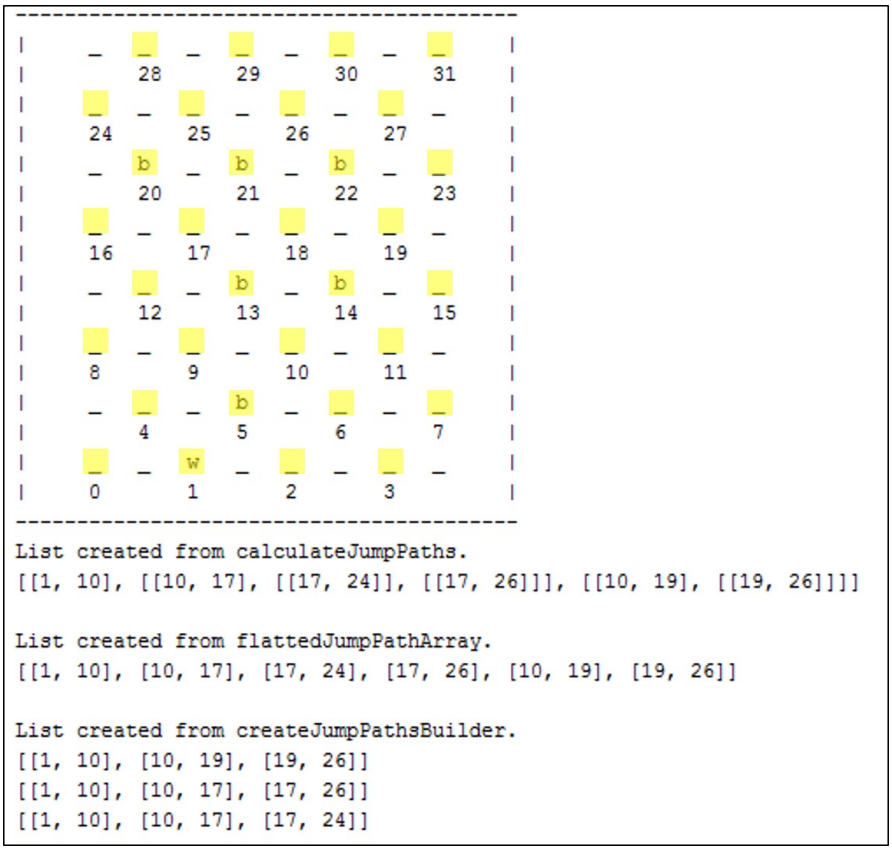

Figure 7.9: Checkers board with a scenario where multiple jump paths are available after the first jump.

Each pair of numbers represents the starting position and the position to move to.

7.5.2 Incorrect Board Setting

Another challenge, which occured early into the implementation, came up when testing of the randomly selected moves games were being played in order to create a baseline with which to compare future results. These games were being played by the hundred thousands. The problem was only noticed when the testing was tried for one million games. After the five hundred thousand iteration mark the results shown started to become noticebly wrong. Games would complete far too quickly and one player would win all the games no matter how many games were played.

What was found later was that there was an error in the way that the boards were resetting every iteration. The boards were not resetting the way that they should have been. The issue was that the remaining peices that were left over after each game were not being removed in certain parts of the board, thus sometimes games would begin with more piece than they should have. This obviously lead to a major bug, however it only required a very minor fix to remove.

The solution was to completely rewrite the entire board each time a new game started thus ensuring that no pieces were left over from previous games.

7.5.3 Incorrect Array Copying

Another issue which came up more than once was incorrect copying of arrays. Python has a nice way in which to copy arrays. You can use a substring command to copy everything from an array and make a new one. The command looks like this:

```
arrayCopy = array[:]
```

However, if you don't do this correctly you may end up with a reference to the array you want a copy of instead of an actual copy. This is very bad when you want to be able to use both arrays for different things, as making a change to one of the arrays will result in the change occuring in both arrays. Thus special care must be taken to ensure that arrays are correctly copied to avoid unnecessary errors from occuring.

7.6 Training Results

The following is the results obtained after the training of the neural networks had been done.

The training was done fifteen times in order to obtain fifteen different trained neural networks. Each of the fifteen training iterations was run 500 times. Within those 500 training iterations, a population of forty neural networks were used.

Each in the population played 15 games randomly versus others in the population in order to establish its fitness value. The fitness was value obtained using the neural networks with the sigmoid function as the activation function.

The maximum velocity for the particle swarm optimization function was set to 0.15, the learning factors and the constrict value were all set to one.

Each iteration ended with testing the global best obtained versus a player making random moves ten thousand times as the white player and then as the black player.

Highest Win Rate		
White NN	Black NN	Average
82,46%	76,68%	71,65%
Iteration 8	Iteration 13	Iteration 8

Figure 7.10: Table showing the highest win rates.

The best result obtained was on the eighth iteration when played as white, which had a win rate of 82.46 percent versus a player making random moves. The best for black, iteration 13, with a win rate of 76.68 percent. These neural networks were selected for use so that a human player may be able to play against them.

Total Time	Total Averages						
108884,81	White Win Percentage	Black Win Percentage	Average Win percentage	White Loss Percentage	Black Loss Percentage	White Draw Percentage	Black Draw Percentage
30h 14m 44s	69,62%	53,83%	61,73%	12,24%	19,77%	18,00%	26,40%

Figure 7.11: Table showing the average win rates.

In total, the training of the fifteen neural networks took thirty hours, fourteen minutes and forty four seconds. The overall average obtained was a win rate of 61.73 percent. This training was run on Ubuntu 16.04.1.

The training was run a final time utilizing the already trained networks in the population in order to determine if a better win rate was possible. The results obtained, was a global best that managed a 75.64 percent win rate as the white player and a 46.96 percent win rate as the black player. The training for this took an additional two hours, fifteen minutes and fourteen seconds.

Chapter 8

Testing and Evalutation

8.1 Introduction

The following section covers testing the checkers system as well as the graphical user interface. Unit and system testing was covered by the developer and the unit testing was done by having other people play the checkers game using the GUI. The results on the effectiveness of the neural network playing against randomly selected moves can be seen in section 7.5: Training Results.

8.2 Unit and System Testing

The unit and system testing involved the developer running a multitude of tests on the GUI in order to ensure that all the components worked together correctly and conformed to the rules of the checkers game. The tests cover both the GUI and the Checkers system components that interact and communicate with the GUI.

The following is a list of tests that were made by the developer:

- Test that only the correct moves can be made.
- Test that all moves are correctly displayed on the GUI.
- Test all possible win, loss and draw conditions and the pop-ups that belong to them.
- Test the possibility of extended jumps and the correctness of jumps.
- Test incorrect moves when there are multiple jumps.
- Test that only the piece that belong to the player can be moved.
- Test the changing of normal pieces into kings.
- Test the error messages.
- Testing all the buttons and make sure they work correctly.
- Test that the neural network is correctly communicating with the GUI.
- Test that the checkers rules are correctly being checked and communicates with the GUI.

Other tests were done on the checkers system to ensure that all move calculations and board tracking are done correctly during the implementation of the Checkers class.

8.3 User Testing

The following table is the feedback obtained from a number of people testing the checkers game by playing a game through the GUI. Feedback is in the form of things the testers liked about the GUI, any bugs found in the GUI and the checkers rules, any advice they could provide about improvements that can be implemented and further comments about the game and the GUI.

Results from User Testing
Likes
• The colour scheme chosen for the GUI is nice. • The moves being tracked is a good idea. • The overall appearance of the GUI is good.
Bugs
• When playing as Red, the white opponents piece didn't change to denote the change to a King piece. • Incorrect spelling of the word congratulations message when the game is won.
Advice
• Add the rules of the checkers game to the start window for the GUI. • Highlighting a piece that is clicked on can be useful. • Add a border to the box displaying the moves that have been made so that it stands out more and know where it ends. • If the player does not select to play White or Red the game should force a selection or randomize the selection for the player.
Comments
• Adding customization options to the GUI and checkers board might be nice, i.e. Being able to change the colours of the board and backgrounds. • Could add more information about the game on completion, i.e. Number of moves that were made, number of kings made. • Create an executable for the game that includes all the pre-requisites and dependencies so that the game can be easily played.

Figure 8.1: Table showing feedback from User Testing.

8.4 User Test Performance

Of all the games the human players played against the trained neural networks every human player was able to beat the neural network. This shows that with the current level of training this system is not simply not capable of standing up to even a novice level Checkers player and is only capable of beating the randomly selected moves that it was tested against.

It is suspected that this outcome is likely due to the fact that the Neural Networks were trained to only have a one move look ahead thus limiting their view of the board and its possibilities. The Neural Networks were only able to perceive the board as they stood after their move and not what the consequences of their moves would make on the board, i.e. what would happen when the opponent played their follow up move.

Chapter 9

User Guide

9.1 Introduction

This section will provide information to any user that wishes to use the software developed in this project. What follows is the requirements needed in order to use the software and also how to run the software. There will also be some basic steps to explain how the GUI should be used.

9.2 Pre-requisites

The following is a list of what is needed in order to run the software:

- Python version 3 or above.
- Python modules: Tkinter, random, time and math need to be installed if not already.
- Any Operating System capable of running Python version 3 or above.
- A Python IDE will be helpful if the Training will be run.

9.3 GUI Usage

When all the pre-requisites are met the checkers game can be run by executing the terminal command python3 GUI.py if on a Linux Operating System. The same can be done on Windows by also adding the file path of both the installed Python files and the file path for GUI.py.

Having a Python IDE is another option to run the game by simply opening GUI.py in the IDE and running it.

When run, the Checkers Game GUI will open and the game can be played. The following are things that must be kept in mid in order to play the game properly:

- In the first window the player must select whether they would like to play as White or Red and then only pressing the start button.
- When the GUI shifts to the Checkers board, the player must click once on the piece they wish to move and then on the square they wish to move to.
- If there are multiple jumps available the player must click on the piece to jump and then the space to jump to for each jump in the path. The player must NOT select the piece to move and then the final position after the full jump path.
- If the player clicks on a piece to move, then changes their mind about the move, the player can simply click again on the same piece to cancel the move.

9.4 Training

If the user wishes to run the training to see the neural networks being trained, they will need some knowledge in coding. The training python script will need to be edited according to their specifications in order to change the number of iterations to train, the number of neural networks to train, the number of test cases they wish to run and finally the number of neural networks for the fitness value calculation.

Alternatively the training can be run with the current settings, however it will take a long time to complete. See section 7.5 for more information.

To run the training file a terminal command can be run similarly to the GUI, with python3 Trainer.py, or it can be run in an IDE of choice.

Chapter 10

Conclusion and Future Work

The purpose of this project was to train neural networks from zero knowledge to play the game of checkers. The neural networks were all trained by allowing them only a single move look ahead.

In order to test whether this was possible, a baseline was obtained by having Checkers games played where all moves made where random and determining the win/loss ratio for both black and white players.

This baseline was then compared to the results obtained after the neural networks had been trained and the global bests obtained. The best neural networks of 15 training iterations were tested against an opponent making only random selected moves.

The results of the comparison showed that the best neural networks did in fact learn and improve, beating the win loss ratio for the randomly selected moves in almost every test case. However the results of the User Test showed that at the current level of training the overall best Neural Networks are not capable of beating a human player.

In order to improve on the training for the Neural Networks it is possible to increase the number of look aheads that the Neural Network is allowed to make thus allowing it to see the consequences of its chosen moves. By increasing the number of look aheads allowed it will also provide the Neural Network the ability to take into account moves that will occur further down the line in the game, just as any human player might do, and plan accordingly.

According to an article titled; "Checkers: TD (λ) Learning Applied for Deterministic Game", which was also aimed at training neural networks for checkers, with increasing levels of the depth search, referred to above as look aheads, so too did the number of won games increase. The results of the paper show a depth search beginning at two and increasing to five. The number of victories for their tests increase from three hundred and thirty four for the depth of two to five hundred and eighty five for a depth of five. The testing was done against a simple heuristics for one thousand games of Checkers.

Applying this idea in this project should similarly increase the performance of the Neural Networks involved.

Chapter 11

Glossary

Neural Network Training Results - 12 September 2017												
		White Victories	Black Victories	Draws	Time Taken (seconds)	White Win Percentage	Black Win Percentage	Average Win percentage	White Loss Percentage	Black Loss Percentage	White Draw Percentage	Black Draw Percentage
Iteration 1	White Neural Network	7837	696	1467	8326,68	78,37%	40,72%	59,55%	6,96%	36,44%	14,67%	22,84%
	Black Neural Network	3644	4072	2284								
Iteration 2	White Neural Network	6611	1128	2261	8180,32	66,11%	52,77%	59,44%	11,28%	14,88%	22,61%	32,35%
	Black Neural Network	1488	5277	3235								
Iteration 3	White Neural Network	7093	912	1995	8505,23	70,93%	63,19%	67,06%	9,12%	7,44%	19,95%	29,37%
	Black Neural Network	744	6319	2937								
Iteration 4	White Neural Network	7723	936	1341	6637	77,23%	52,01%	64,62%	9,36%	21,70%	13,41%	26,29%
	Black Neural Network	2170	5201	2629								
Iteration 5	White Neural Network	6509	1771	1720	6710,35	65,09%	37,66%	51,38%	17,71%	27,70%	17,20%	34,64%
	Black Neural Network	2770	3766	3464								
Iteration 6	White Neural Network	7628	1065	1307	6137,61	76,28%	63,34%	69,81%	10,65%	17,70%	13,07%	18,96%
	Black Neural Network	1770	6334	1896								
Iteration 7	White Neural Network	5862	1239	2899	6363,53	58,62%	51,44%	55,03%	12,39%	17,04%	28,99%	31,52%
	Black Neural Network	1704	5144	3152								
Iteration 8	White Neural Network	8246	768	986	8132,72	82,46%	60,83%	71,65%	7,68%	15,97%	9,86%	23,20%
	Black Neural Network	1597	6083	2320								
Iteration 9	White Neural Network	7163	1101	1736	6351,55	71,63%	48,83%	60,23%	11,01%	16,52%	17,36%	34,65%
	Black Neural Network	1652	4883	3465								
Iteration 10	White Neural Network	7334	904	1762	6077,15	73,34%	48,59%	60,97%	9,04%	28,08%	17,62%	23,33%
	Black Neural Network	2808	4859	2333								
Iteration 11	White Neural Network	7058	1134	1608	6706,56	70,58%	53,10%	61,84%	11,34%	21,40%	16,08%	25,50%
	Black Neural Network	2140	5310	2550								
Iteration 12	White Neural Network	5395	2024	2581	8020,03	53,95%	48,14%	51,05%	20,24%	27,70%	25,81%	24,16%
	Black Neural Network	2770	4814	2416								
Iteration 13	White Neural Network	6224	2096	1680	7793,23	62,24%	76,68%	69,46%	20,96%	7,80%	16,80%	15,52%
	Black Neural Network	780	7668	1552								
Iteration 14	White Neural Network	7089	930	1982	7768,32	70,89%	55,39%	63,14%	9,30%	16,22%	19,82%	28,39%
	Black Neural Network	1622	5539	2839								
Iteration 15	White Neural Network	6665	1663	1672	7174,53	66,65%	54,72%	60,69%	16,63%	20,02%	16,72%	25,26%
	Black Neural Network	2002	5472	2526								

Bibliography

[1] T. O Ayodele *Types of Machine Learning Algorithms*. University of Portsmouth, United Kingdom (2010)

[2] A. L. Samuel *Some Studies in Machine Learning Using the Game of Checkers*. IBM Journal, Vol 3, No. 3, (July 1959)

[3] Cranfill, R., Chang, C. *What is Facebook's architecture?*. Quora.com/What-is-Facebooks-architecture-6 (12/2014)

[4] K. Chellapilla, D. B. Fogel *Evolving Neural Networks to Play Checkers Without Relying on Expert Knowledge*. IEEE Transactions on Neural Networks, Vol. 10, No 6, (Nov 1999)

[5] N. Franken, A. P. Engelbrecht *Evolving intelligent game-playing agents*. Proceedings of SAICSIT, Pages 102-110, (2003)

[6] N. Franken, A. P. Engelbrecht *Comparing PSO structures to learn the game of checkers from zero knowledge*. The 2003 Congress on Evolutionary Computation. (2003)

[7] A. Singh, K. Deep *Use of Evolutionary Algorithms to Play the Game of Checkers: Historical Developments, Challenges and Future Prospects*. Proceedings of the Third International Conference on Soft Computing for Problem Solving, Advances in Intelligent Systems and Computing 259, (2014)

[8] H. Kwasnicka, A. Spirydowicz *Checkers: TD (λ) Learning Applied for Deterministic Game*. Department of Computer Science, Wroclaw University of Technology, Poland. (June 2014)