Connect 4 Neural Network Project

Name: Martin Butler

Date of Birth:

Student number:

Assessment Identification:

Tutor Name:

Contents

[1. Analysis 3](#_Toc202872162)

[1.1. Problem Identification 3](#_Toc202872163)

[1.1.1. Neural Networks 3](#_Toc202872164)

[1.1.2. Machine Learning 4](#_Toc202872165)

[1.2. Stakeholders 4](#_Toc202872166)

[1.3. Problem Research 4](#_Toc202872167)

[1.4. Proposed Solution 5](#_Toc202872168)

[2. Design of the solution 7](#_Toc202872169)

[2.1. Problem Decomposition 7](#_Toc202872170)

[2.1.1. The Output 7](#_Toc202872171)

[2.1.2. The Input 8](#_Toc202872172)

[2.1.3. Rows and Columns 9](#_Toc202872173)

[2.1.4. Diagonals 9](#_Toc202872174)

[2.1.5. Neural Network 10](#_Toc202872175)

[2.1.6. Combine Scores 10](#_Toc202872176)

[2.1.7. Calculate Scores 11](#_Toc202872177)

[2.1.8. Weightings 11](#_Toc202872178)

[2.1.9. Optimisation 11](#_Toc202872179)

[2.2. Design 11](#_Toc202872180)

[2.2.1. Play game 12](#_Toc202872181)

[2.2.2. Optimisation 14](#_Toc202872182)

[2.2.3. Wii U Programming 14](#_Toc202872183)

[2.3. Test Approach 15](#_Toc202872184)

[3. Developing the solution 15](#_Toc202872185)

[3.1. Development 15](#_Toc202872186)

[3.1.1. Overall Game 15](#_Toc202872187)

[3.1.2. Optimisation of the Neural Network 16](#_Toc202872188)

[3.1.3. Porting to Wii U 18](#_Toc202872189)

[3.2. Post Development Testing 20](#_Toc202872190)

[3.2.1. Test case 1: Wii U Game Play 20](#_Toc202872191)

[3.2.2. Test case 2: Robust Game Play 24](#_Toc202872192)

[3.2.3. Test case 3: Difficulty Levels 27](#_Toc202872193)

[3.2.4. Test case 4: Weighting Updates 30](#_Toc202872194)

[4. Evaluation 31](#_Toc202872195)

[4.1. Evaluation Testing 31](#_Toc202872196)

[4.2. Solution Success 31](#_Toc202872197)

[4.3. Final Product 31](#_Toc202872198)

[4.4. Maintenance and Development 32](#_Toc202872199)

# Analysis

The aim of the Connect 4 game is to get four coloured pieces in a horizontal, vertical or diagonal line. Essentially each player chooses between one of seven columns for their move.



The problem is to find a way for the computer to play Connect 4 well.

## Problem Identification

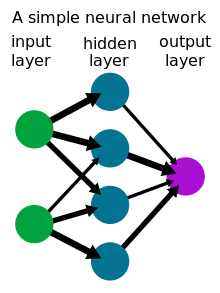
The rules of the Connect 4 game are simple. Players takes turns to drop one of their coloured pieces into the board. The first player to get 4 in a line is the winner. If the board is full, with no winning player, the game is a draw. Players alternate who goes first.

I had been planning to write a program to play Connect 4 for a while, but never got round to it. I wanted to find a way to give the computer the intelligence to play the game well. I had considered a database, where the computer remembered games that it had played. I tried a smaller version of this with Noughts and Crosses, but this did not work well. The computer had no way to tell the difference between good play, and winning due to poor play by the opponent. I therefore dropped the database idea.

I then realised that as a player move is just choosing a column from 1 to 7, Connect 4 would be a great way to try out a Neural Network.

### Neural Networks

A neural network is a group of functions performing calculations (Neurons) that are then linked. The results of neurons are multiplied by weightings, to adjust the importance of individual data points within the processing. The following diagram is taken from Wikipedia:



### Machine Learning

In the case of neural networks they must be run many times, the quality of the output being used to adjust the weightings, until the neural network is doing the job intended. This is one form of machine learning.

In the case of Connect 4, the program can be set so that the computer plays against itself millions of times. Different weightings being played against each other, keeping the weightings that win. In this way the neural network can be optimised.

## Stakeholders

The stakeholders are anyone who needs to know more about neural networks and how they could be applied to games. This can be anyone who is learning about Computer Science and practical uses of neural networks.

During development I found a thriving group of hobbyists writing code for the Wii U games console. There were only a handful of games targeted at the Wii U, and Connect 4 was not available. There are therefore stakeholders amongst like-minded enthusiasts of the Wii U games console (over 1000 downloads of the game to date).

## Problem Research

To understand neural networks, I worked through the Wikipedia page [Neural network - Wikipedia](https://en.wikipedia.org/wiki/Neural_network).

Playing Connect 4 is quite a constrained problem. The only real drawback is the risk that the computer can play too well, making the game less fun to play.

The neural network needs to select from the 7 available columns for the computers move. A method is therefore needed to calculate a score for each of the possible moves, so that the best can be selected.

I also researched program development for the Wii U. I followed online tutorials to get [Homebrew](https://wiiubrew.org/wiki/Main_Page) installed and then set up the development kit ([devkitPro](https://devkitpro.org/wiki/Getting_Started)). I then went through the initial [tutorial](https://github.com/yawut/ProgrammingOnTheU) about using the Application Program Interface (API), which is described in [wut: Wii U Toolchain](https://wut.devkitpro.org/), using some example code found on the [Homebrew App Store](https://hb-app.store/wiiu).

## Proposed Solution

To play the game a set of neurons (calculations / sections of software) must be devised, and then connected. The connections need some form of weighting, so that the operation of the neural network can be optimised.

Having a neuron per column (7 neurons) or a neuron per position on the board (42 neurons (7x6)) were considered. But the operation of each neuron wasn’t obvious. The design of the chosen solution is covered in the next section. The following are the requirements for the Connect 4 game including success criteria.

Originally a console program had been considered, as the main focus was the neural network. However nowadays Graphical User Interfaces (GUIs) are expected. Initial development is therefore targeted at a Java console program to develop the neural network, with the finished game being targeted at the Wii U games console.

| **Req. number** | **Requirement** | **Success Criteria** |
| --- | --- | --- |
| **1** | The solution shall run on the Wii U games console. | Demonstrate program running on Wii U. |
| **2** | The human player shall play Red (R). | Show human plays red. |
| **3** | The computer shall play Yellow (Y). | Show computer plays yellow. |
| **4** | Game play shall be the computer playing the human player. | Demonstrated by playing the game. |
| **5** | The game shall have three difficulty levels easy, medium and hard. | Requirements 5, 6, 18, 19 and 20 need to be tested together. To show that the level of difficulty is changed between easy, medium and hard. To demonstrate that hard is challenging, medium is less challenging than hard and easy is less challenging than medium. Also, that the game is restarted each time the difficulty is changed. |
| **6** | The game shall re-start if the difficulty level is changed. | See requirement 5. |
| **7** | The game shall always start with the human player playing first. | Show that the program waits for the human player to select the first move in the first game. |
| **8** | The starting player shall then alternate between computer and human for subsequent games. | Show that the computer makes the first move in the second game and then the starting player alternates after that. |
| **9** | The human move shall only complete with a valid move (playing a column 1 to 7 that has an available space). | Show that invalid moves are ignored and the game only moves on with a valid move. |
| **10** | The computer move shall only complete with a valid move (playing a column 1 to 7 that has an available space). | Monitor game play to ensure that the computer has always added a yellow piece for a move and only one piece. |
| **11** | There shall be a delay of at least 0.2 seconds between the human move and computer move; so that the human player can see the computer move. | Ensure that the computer move is always displayed as a separate move from the human move. |
| **12** | The human player shall win if they complete a line of four red pieces. | Demonstrate by play that a line of 4 red pieces correctly wins the game for the human player. |
| **13** | The computer shall win if they complete a line of four yellow pieces. | Demonstrate by play that a line of 4 yellow pieces correctly wins the game for the computer. |
| **14** | The game shall be declared a draw if the board is full and neither player has a line of 4 pieces. | Demonstrate that a full board (with no winner) correctly ends the game as a draw. |
| **15** | The number of games won by Red shall be recorded and displayed. | Monitor the red count during games to ensure red wins are correctly recorded and updated. |
| **16** | The number of games won by Yellow shall be recorded and displayed. | Monitor the yellow count during games to ensure yellow wins are correctly recorded and updated. |
| **17** | The number of drawn games shall be recorded and displayed. | Monitor the drawn game count during play to ensure that draws are correctly recorded and updated. |
| **18** | The game shall be challenging when set to the hard level. | See requirement 5. |
| **19** | The medium setting shall be less difficult than the hard level. | See requirement 5. |
| **20** | The easy setting shall be less difficult than the medium level. | See requirement 5. |
| **21** | The game shall support the Wii U ‘Home’ interface so that the game can be correctly stopped. | Press the ‘Home’ button on the Wii U console then the stop button and ensure that the game correctly exits back to the Wii U menu. |
| **22** | The game shall support the Wii U ‘Home’ interface so that the game can be correctly resumed. | Press the home button then the resume button and ensure that the game correctly resumes play at the point the Home button was pressed. |

# Design of the solution

The following section concentrates on the design of the neural network to determine the computer’s move in the Connect 4 game.

## Problem Decomposition

The following sub-sections describe the design of the neural network:

### The Output

It was easier to start at the output. There needs to be one neuron to select from the seven possible columns. A neuron is needed for each column, to work out a score for that column. The select move neuron outputs the column number for the column with the highest score.



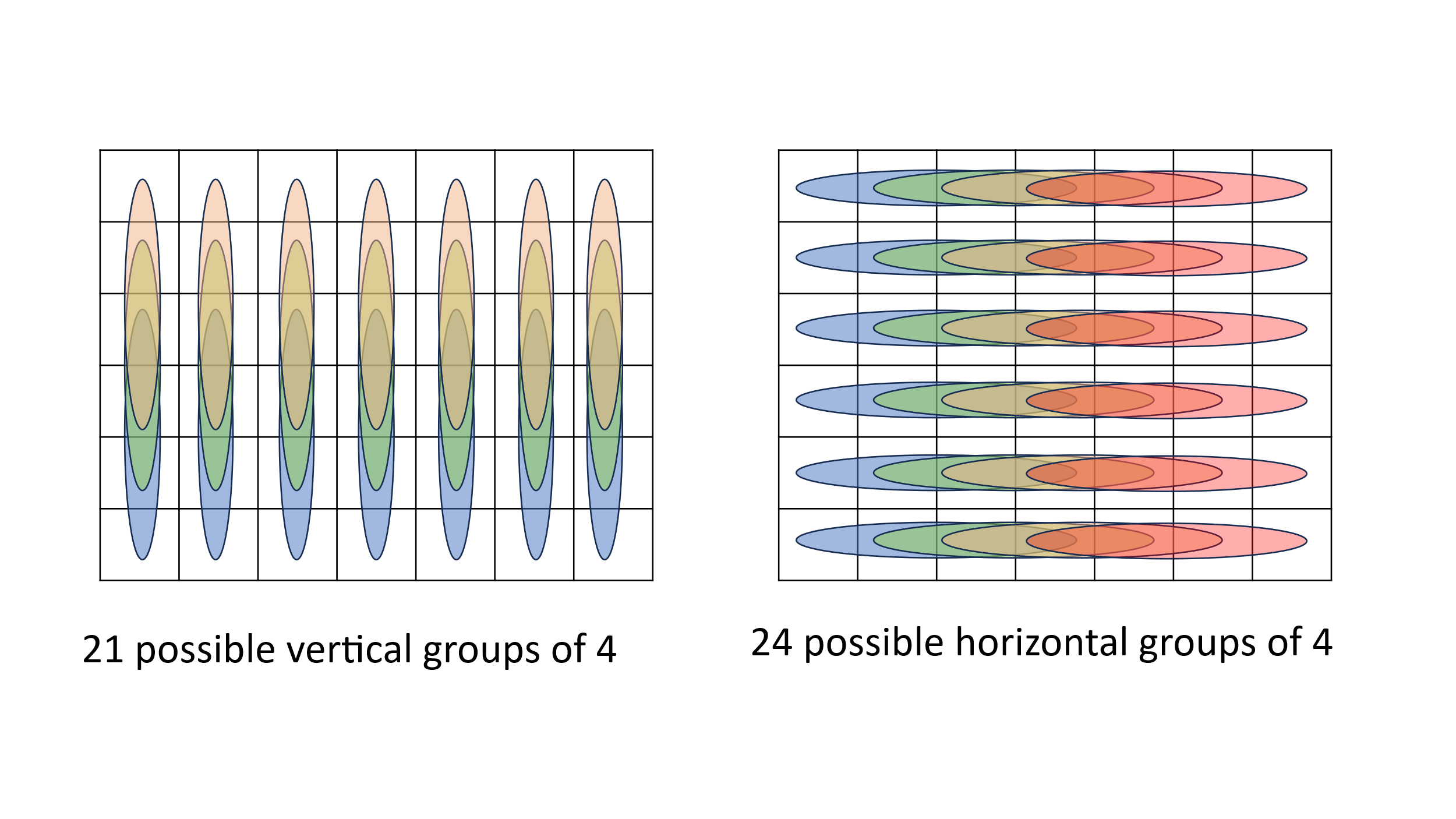
### The Input

The input neurons work out a score for each possible winning line. The number of red (R) and yellow (Y) pieces within a potential line of four are counted. This can be 0, 1, 2 or 3 (4 means the game has already been won). Each neuron counts the number of pieces. If both red and yellow have pieces in the line, the group of 4 cannot be won, so both R and Y are set to 0. The following diagram shows an example:

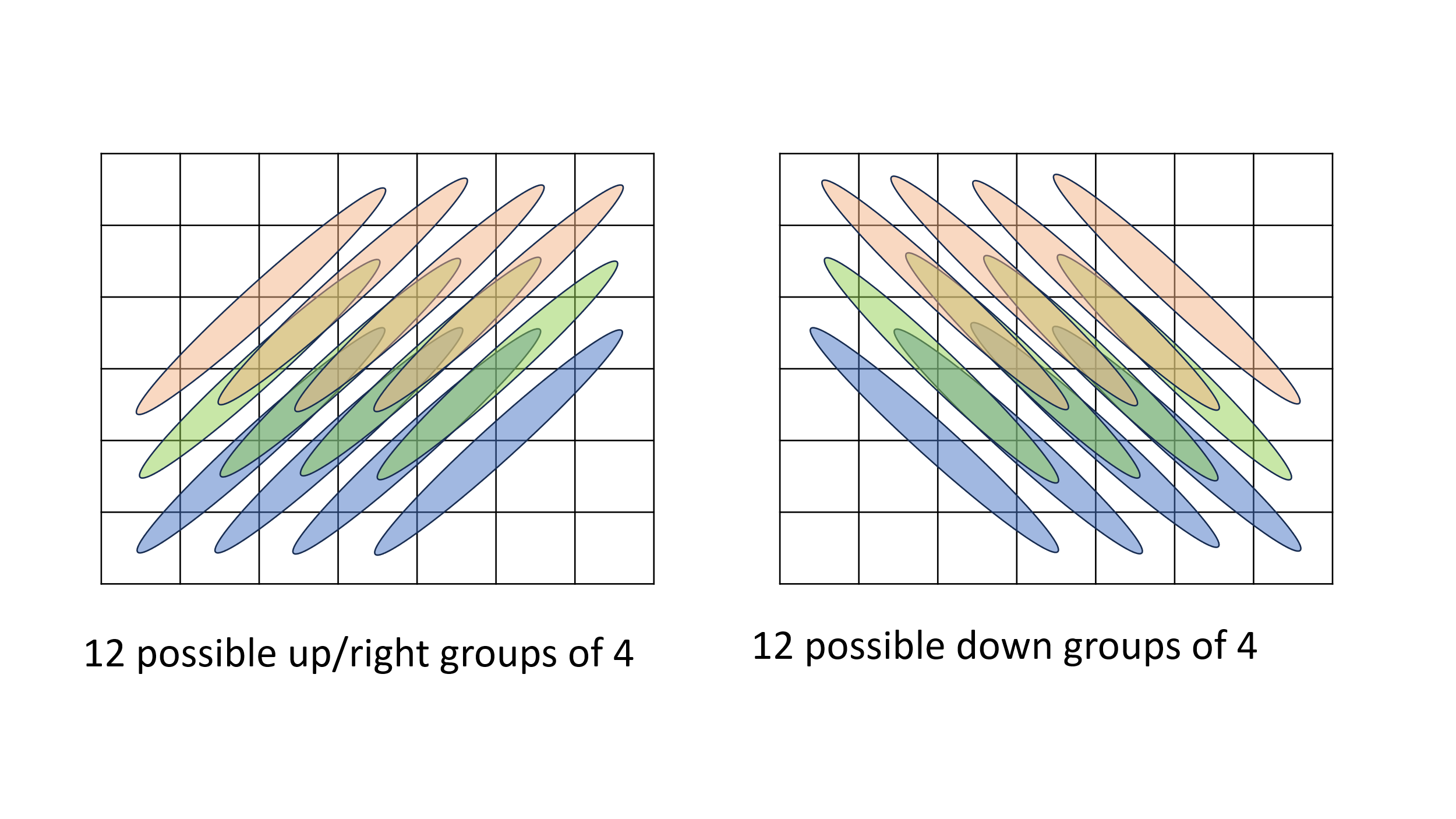


Three in a line is much more important than two in a line or one. The count is therefore used as a power, x^R and x^Y. ‘x’ can then be used as a weighting, the higher the value of x, the bigger the difference for more pieces. For example, for x=4, 4^1 = 4, 4^2 = 16 and 4^3 is 64, giving 3 in a line a much higher score.

### Rows and Columns

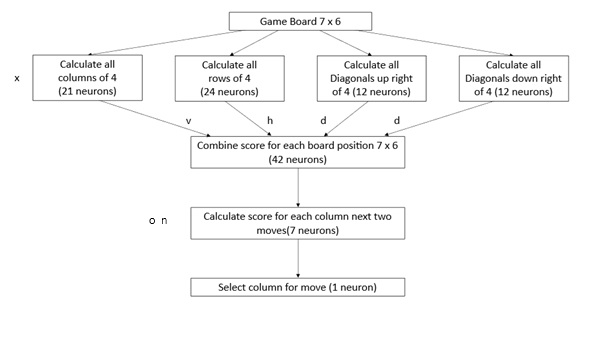
The following shows the possible vertical and horizontal winning lines:

### Diagonals

The following are the possible diagonal winning lines:

### Neural Network

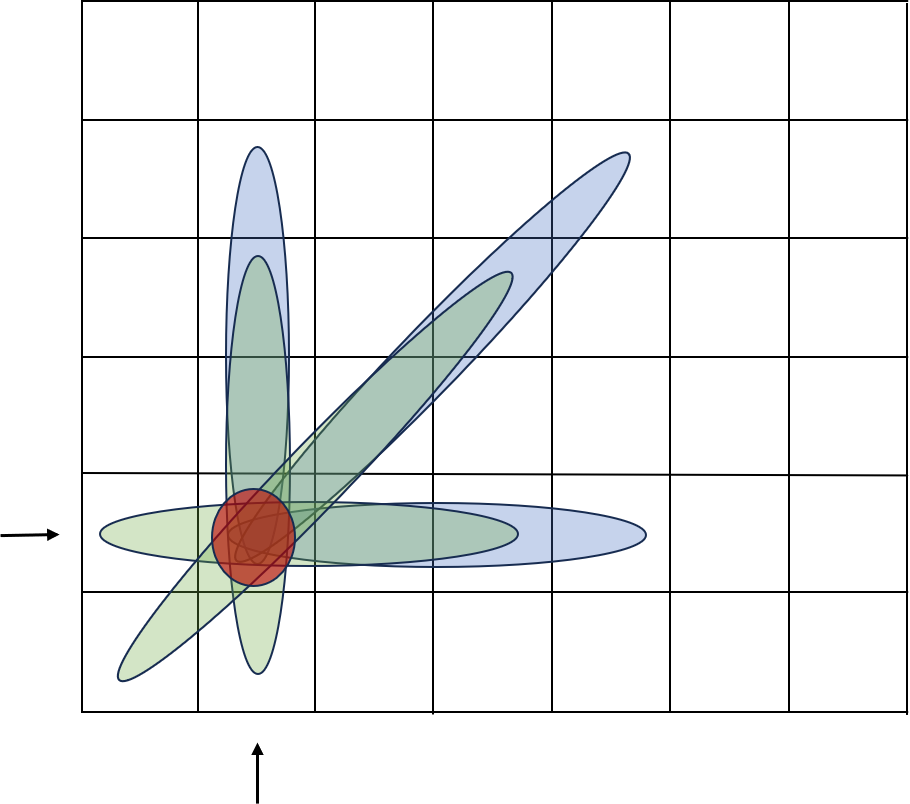
There are a total of 69 possible winning lines of 4 (21 vertical, 24 horizontal and 24 diagonal). This results in 69 neurons to calculate the inputs to the neural network. The following diagram shows the resulting neural network:



The ‘Combine score’ and ‘Calculate score’ neurons are described in the following sub-sections.

### Combine Scores

The total score for each of the 42 positions on the Connect4 game board are calculated. This is the red total and yellow total for all of the possible winning lines that include the board position. The diagram below shows the second column and second row as an example. This position is in 6 possible lines of 4 counters.



### Calculate Scores

The score for each column is calculated by finding the next free vertical position in the column. The score for both Y (the computer) and R (the opponent) are added together. The opponent score for the position above is then subtracted.

Score = Y + (R x o weighting) – (next move R x n weighting)

Weightings are applied to the opponent move scores used in the above calculation. The opponent score is considered as it may be useful for the computer to block a move that is good for the opponent. The next position opponent score is considered to avoid making a move that helps the opponent. Weightings are used so that the importance of these data points can be adjusted during neural network optimisation.

The last neuron then selects the number of the column with the highest score.

### Weightings

The letters on the earlier diagram are the weightings used to optimise the neural network.

x – number raised to the power of the Y and R counts (4 to 7)

v – weighting for vertical groups (0.5 to 2.0)

h – weighting for horizontal groups (0.5 to 2.0)

d – weighting for diagonal groups (0.5 to 2.0)

o – weighting for opponent score (0.5 to 2.0)

n – weighting for opponent next move (0.5 to 2.0)

### Optimisation

The neural network can be optimised by the computer playing against itself. A random component was considered to avoid the same game being continually played. This however was not guaranteed to give different game-play. Optimisation therefore now plays 14 games (starting with position 1 to 7 for each player). A trial set of weightings is played against a default set. If the trial set wins more games, the trial set is played against the current set. If it again wins more games, the current set is updated to the trial set. This process is repeated so that the weighting set can be optimised.

## Design

The initial design is for a console program that can accept a command-line parameter. If the command-line parameter is ‘O’ for Optimise, the optimisation loops are executed to calculate weightings for the neural network.

### Play game

The game is represented by a two-dimensional character array gameTable with 7 columns and 6 rows. Space is an empty position; Y is yellow for the computer pieces and R is red for the player pieces. The operation of the game is split between the main functions which are described in the following sub-sections. Playing the game is implemented as shown with the following pseudo-code:

FOR infinite loop

CALL clearGameTable *to get ready for the next game*

CALL displayBoard *to display the board to the user*

WHILE game not ended *user goes first*

CALL humanMove *get the users move*

CALL displayBoard

CALL gameEnded *check if a player has won or if a draw*

CALL calculateMove *calculate the computer move*

CALL displayBoard

CALL gameEnded

ENDWHILE

CALL clearGameTable

WHILE game not ended *computer goes first*

CALL calculateMove

CALL displayBoard

CALL gameEnded

CALL humanMove

CALL displayBoard

CALL gameEnded

ENDWHILE

END infinite loop

#### clearGameTable

The clearGameTable function sets all places in the gameTable array back to spaces, to signify the table is empty ready to start a new game.

#### displayBoard

The displayBoard function displays the current contents of the gameTable, so that the user can see the current state of the game.

#### humanMove

The humanMove function gets the players move from the keyboard input. It checks that the move is a valid move, in that it is a number from 1 to 7 for one of the seven columns of the game. The function then checks to see if the column is already full, if so, the user is prompted again continually for their move, until it is valid.

#### gameEnded

This function goes through all of the possible horizontal, vertical and diagonal winning lines to check if either Y or R has a winning line. If so, the function exits, returning the winner Y or R. If there is no winner, the function then checks to see if all columns are full, if so, D is returned to indicate a draw.

#### calculateMove

This function runs the neural network to calculate the computer move. This is further broken down into the following sub-functions, which implement the neural network described in section 2.1.5. Calculating the computer move is implemented as shown with the following pseudo-code:

calculateMove

CALL doWinningColumns

CALL doWinningRows

CALL doDiagonalsUp

CALL doDiagonalsDown

CALL doCombinedScores

CALL selectMove

The first four functions in the pseudo-code above count the number or R and Y pieces in the possible winning lines.

The doCombinedScores then adds the possible scores for each of the 42 possible positions on the board. Each position can be part of several lines. For each possible winning line, the following is calculated:

x^R and x^Y where x is the weighting from section 2.1.8.

The weightings are then applied to each possible winning line as follows:

Each horizonal R and Y score is multiplied by h (horizonal weighting from section 2.1.8).

Each vertical R and Y score is multiplied by v (vertical weighting from section 2.1.8).

Each diagonal R and Y score is multiplied by d (diagonal weighting from section 2.1.8).

The scores for each position (including weightings) are added together.

The selectMove function finds the next available position (indicated by a space) in each column and performs the following calculation for that column:

Score = Y combined score + (R combined score multiplied by o) – (opponent next move score multiplied by n).

o is the opponent score weighting and n is the next move weighting (from section 2.1.8).

The scores for both players are considered in the calculation; if it is a good move for the opponent, it may be a good idea to block this move.

The opponents next move score is subtracted, to avoid making a move that gives the opponent too much of an advantage.

### Optimisation

The neural network needs to be optimised based on experience. The weightings shown in section 2.1.8 allow the neural network to be optimised. The importance of the individual calculations in selecting the next move are not known, and can therefore be determined by optimising, by playing different weightings against each other. The weightings can be varied from 0.5 (half) to 2.0 (double) to optimise the data points calculated.

To perform optimisation the computer must be able to play both the red R and Y yellow moves. It must also be able to use different weighting sets to try these out. For optimisation three sets of weightings are used:

Default Weightings Starting nominal values

Current Weightings Best weightings so far selected during optimisation

Trial Weightings Updated weightings to be tried against the Current and Default sets

The optimisation loop is a modification of the game loop, where the humanMove function is replaced by the calculateMove function. The calculateMove(player, weighting) function has two pass parameters added. The player is to select red R or yellow Y; the weighting is default, current or trial.

The optimisation runs a loop of 10000 cycles, each with the following steps:

* Select a random set of trial weightings
* Play 14 games of the trial weightings against the default weightings (each player starting in position 1 to 7 for the first move)
* If the trial weightings win at least 2 more games, run 14 games against the current weightings
* If the trial weightings again win at least 2 more games, set the current weightings to match the trial weightings

### Wii U Programming

Once the neural network has been designed, implemented and optimised, the full Wii U game is developed. The Java code is converted to C code. The two languages are similar, data declarations differ, but much of the actual code is the same. The Java class to run the game being converted to a C file, with some access functions that can be used within the Wii U game.

The game has a GUI developed using the Wii U API (see [wut: Wii U Toolchain](https://wut.devkitpro.org/)); making use of existing sound and graphics code I previously developed (see  [PacMan-ish Wiki](https://github.com/MartinButlerAAA/PacMan-ishU/wiki)). This is not described in detail, as the main focus is the neural network to play the game. The final code is available in GitHub (see section 4.3). The Wii U game essentially provides a more user-friendly version of the displayBoard and humanMove functions described earlier.

## Test Approach

The test approach first ensures that the console solution is robust. Then this is used to optimise the neural network. There is testing to ensure that the neural network as implemented on the Wii U, is performing as per the Java console version. Formal testing is then performed on the Wii U application, testing against the requirements from section 1.4. As this is a program to play Connect 4 there are no data sets or key variables as such.

# Developing the solution

This section has deliberately been limited to describing the notable development points. There were many syntax errors and bugs during initial development, which were identified and corrected. Some examples of the routine development have been included; this section then concentrates on the optimisation of the neural network.

## Development

This section describes the iterative development of the Connect 4 game covering the support functions needed, before concentrating on neural network optimisation.

### Overall Game

The main console program game loop was developed first. This included the clearGameTable, displayBoard, humanMove and gameEnded functions. The calculateMove function was initially written to return a valid random move between 1 and 7. The calculateMove returning a random move made it dumb, which was what was needed to allow the main support functions to be tested.

The different humanMove options could be tried, without the need to consider trying to beat the computer. This allowed each column to be filled to the top, to ensure that these columns could no longer be selected for a move. The correct handling of non-numeric characters and numbers outside of the range 1 to 7 was also tested and shown to be correct.

Format issues with the displayBoard function were immediately identified and fixed while running the software to test out the humanMove and gameEnded functions. Similarly, a coding error that meant the gameTable array was not correctly cleared was immediately obvious during this testing.

The testing of the main game concentrated on the gameEnded function. To ensure full loop coverage, the detection of vertical lines of four in the first and last column were tested and shown to work. The detection of horizontal lines of four in the bottom and top rows were tested and shown to work. The left and right most diagonal lines of 4 were then tested and shown to work. Drawn games were deliberately created to ensure that these were correctly detected.

### Optimisation of the Neural Network

The intention initially was to run multiple optimisation cycles, allowing all 6 optimisation parameters to be varied. The expectation was that multiple optimisation-runs would give broadly similar results, showing that the optimal weightings had been found.

The weightings were initially allowed to be randomly selected in the following ranges:

x – number raised to the power of the Y and R counts (4 to 7)

v – weighting for vertical groups (0.5 to 2.0)

h – weighting for horizontal groups (0.5 to 2.0)

d – weighting for diagonal groups (0.5 to 2.0)

o – weighting for opponent move (0.5 to 2.0)

n – weighting for opponent next move (0.5 to 2.0)

A random component had been considered, but this had a detrimental impact on optimisation, making optimisation runs too variable.

The x value is applied before the other weightings in calculations, so has a greater impact on operation. The x parameter was fixed for optimisation runs, using either 5 or 6. All other weightings were set to a default value of 1.0. The following optimisation results were recorded:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | h | v | d | o | n |
| 5.0 | 1.4 | 1.1 | 0.8 | 0.8 | 0.5 |
| 5.0 | 1.3 | 0.9 | 1.5 | 1.1 | 0.5 |
| 5.0 | 1.0 | 1.9 | 1.1 | 0.7 | 0.6 |
| 6.0 | 1.2 | 1.9 | 0.8 | 0.6 | 0.6 |
| 6.0 | 0.8 | 1.9 | 0.8 | 1.0 | 0.5 |
| 6.0 | 0.6 | 1.3 | 1.8 | 1.8 | 0.8 |

The weighting n for the next move is reasonably consistent across runs, the others varied quite a lot. It is likely that interaction between the weightings causes the volatility. It may be that the difference between playing for horizontal, vertical or diagonal lines does not matter. As h, v and d are applied before o and n, there is likely a lot of interaction between the weightings.

Separate development work had shown that x is better as a higher number. I chanced upon a game where the computer chose the wrong column with x at 5.0, but the correct column for x at 6.0. For subsequent testing x is fixed at 6.0.

To reduce variation a new approach is used. The default set is changed to the optimised set from the previous run:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | h | v | d | o | n |
| 6.0 | 0.6 | 1.3 | 1.8 | 1.8 | 0.8 |
| 6.0 | 1.4 | 1.9 | 1.7 | 0.7 | 0.5 |
| 6.0 | 1.9 | 1.1 | 1.0 | 1.0 | 1.0 |
| 6.0 | 0.9 | 1.4 | 1.4 | 1.7 | 0.5 |
| 6.0 | 0.7 | 2.0 | 0.6 | 1.2 | 0.7 |
| 6.0 | 0.8 | 1.8 | 0.9 | 0.9 | 0.5 |
| 6.0 | 1.3 | 1.9 | 1.0 | 1.0 | 0.9 |
| 6.0 | 0.8 | 0.7 | 1.3 | 0.8 | 0.6 |
| 6.0 | 0.6 | 1.3 | 1.0 | 0.6 | 0.5 |
| 6.0 | 0.5 | 1.8 | 1.2 | 1.6 | 1.5 |

The above approach still resulted in a wide variation in the weightings after each run of 10000. This is likely to do with interaction between the different weightings, but also that no one set of weightings is best for all situations. It is also likely that weightings for horizontal, vertical and diagonal don’t matter. The horizontal, vertical and diagonal weightings were set to one, so that only the opponent move and next move weightings were optimised. The defaults for o and n were set to 1.0.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | h | v | d | o | N |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |

Five runs all resulted in the same values for o and n. So maybe it is better to not optimise all weightings at once. For the next optimisation runs o and n were fixed at 0.5, but h, v and d were optimised (default values of 1.0).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | h | v | d | o | N |
| 6.0 | 1.3 | 1.3 | 0.7 | 0.5 | 0.5 |
| 6.0 | 0.7 | 1.2 | 0.5 | 0.5 | 0.5 |
| 6.0 | 1.5 | 1.7 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.6 | 1.7 | 0.9 | 0.5 | 0.5 |
| 6.0 | 1.6 | 1.9 | 0.8 | 0.5 | 0.5 |

This had less variation, but has not converged on the same solution. The ratios between the three values are probably more important than the actual values. As diagonal has remained closest to 1.0. This has been fixed at 1.0 and the other two optimised.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X | h | v | d | o | N |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| 6.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.5 |
| **6.0** | **1.0** | **1.0** | **1.0** | **0.5** | **0.5** |

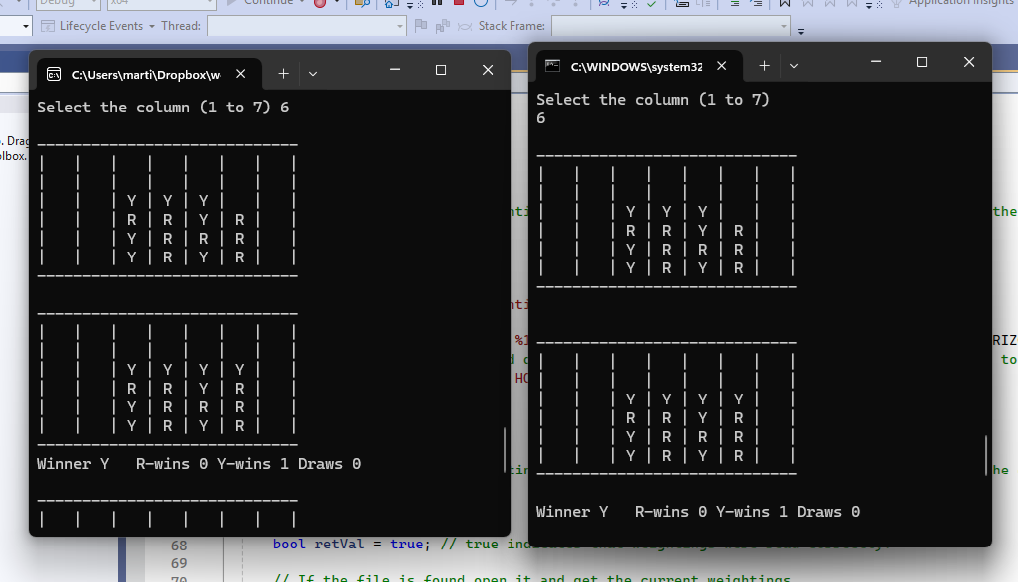
Fixing the diagonal weighting at 1.0 resulted in no changes to the horizontal and vertical weightings. i.e. no better combination of horizontal and vertical weightings could be found when diagonal was fixed at 1.0. The above are the best weightings found, which are simpler than expected. The above optimisation values are now used in the Wii U game.

The optimisation has however demonstrated that no set of weightings is perfect for all possible games. As the operation is deterministic, a set of moves that wins will always win. This has identified that the weightings should have a mechanism to be modified in the game. Tweaking the o and n values up or down by 0.1 has been shown to change the neural network enough to play differently. The game should therefore include logic to adjust the weightings each time the human player wins, to avoid them playing the same moves again to repeatedly win.

### Porting to Wii U

The Java code for the neural network was first converted to C code to run under Microsoft Visual Studio as a console program. C and Java (certainly for console programs) are very similar, so the conversion took a few hours. The Java and C console versions were run side by side with the same human moves being made. The computer moves were compared to ensure that the neural network played exactly the same way in the C version as the Java version.

The following screenshot shows the C Console and Java versions of the program playing exactly the same way:



The C code was then built with the Wii U development kit cross-compiler. Where a further minor update was required. The Wii U does not have the math library for the main processor; I therefore had to write my own version of the math library ‘pow’ (x^y) function. This resulted in probably the most difficult bug I had to deal with. The game played a bit differently on the Wii U to the PC, not always making the same moves. It took me a long time to figure this out. I put my ‘pow’ function into the PC console version, which changed the way this behaved. I eventually tracked the error to my pow function. My code gave a value of 0 for x^1, rather than the correct value of 1. Once this was fixed the Wii U version played exactly the same as the PC version.

The following code snipped shows the correction of ‘return 0.0’ to ‘return 1.0’, which seems trivial, but was enough to significantly change the computer game play.

// Quick approximation of a pow function as the math.h does not seem to be available on Wii U.

double myPow(double num, double power)

{

    int intPower = (int)power;

    double retVal = 0;

    // If the power value passed in is outside a sensible range to process, exit with an error code.

    if (intPower < 0) { return -99.0; }

    if (intPower >= 100) { return -99.0; }

    if (intPower == 0) { **return 1.0;** }  // Handle power of 0.

    if (intPower == 1) { return num; }  // Handle power of 1, i.e. no change.

    // Loop to multiply by the number of times set by power.

    retVal = num;

    for (int a = 1; a < intPower; a++)

    {

        retVal = retVal \* num;

    }

    return retVal;

}

With the above code correction, the Wii U version played exactly the same as the console versions. The following example shows the same game as the console games shown on the previous page (the columns are named by the controller buttons used to play):



As mentioned in section 3.1.2, the C console version was extended to have a text file for the weightings, which is read when the program starts. If the human player wins, the opponent move and/or the next move weighting is changed up or down by 0.1, to ensure that the computer will play differently next time it plays. This avoids the player using the same set of moves to continually win.

The easy, medium and hard game-play levels were added on the Wii U. This involved some trial and error to make the levels playable.

* For the easy level there is a 50% chance that the computer plays a random move, rather than the move chosen by the neural network.
* For the medium level there is a 20% change that the computer plays a random move, rather than using the neural network.
* For the hard level the neural network is always used to calculate the computer move.

As part of this update the Wii U game was extended further so that it adjusts the opponent move (o) and opponent next move (n) weightings up or down by 0.1, to change the way the computer plays (only if the computer loses while playing in hard mode).

## Post Development Testing

Development was mainly done on the Java version of the code running on PC. The Wii U version of the game is the one that is released. The post development testing was performed on the Wii U version of the software. The testing is based around the requirements from section 1.4 as detailed below:

### Test case 1: Wii U Game Play

The following requirements are tested:

1: The solution shall run on the Wii U games console.

2: The human player shall play Red (R).

3: The computer shall play Yellow (Y).

4: Game play shall be the computer playing the human player.

7: The game shall always start with the human player playing first.

11: There shall be a delay of at least 0.2 seconds between the human move and computer move; so that the human player can see the computer move.

21: The game shall support the Wii U ‘Home’ interface so that the game can be correctly stopped.

22: The game shall support the Wii U ‘Home’ interface so that the game can be correctly resumed.

Perform the following test steps:

1. Start the Wii U game console and select Connect 4.
2. Ensure that Connect 4 starts correctly.
3. Select HARD mode.
4. Ensure that there are no pieces on the board, the Wii U is waiting for the human to play.
5. Play column R.
6. Ensure that a red piece appears at the bottom of column R.
7. Ensure that a yellow piece appears after a visible display in column L.
8. Ensure that the game waits for the next human move.
9. Play column R, R, L, Y, X, Y, Y.
10. Ensure that for each move the Wii U plays its own move a small time later.
11. Ensure at the end this sequence Yellow wins and the count is increased.
12. Ensure that after a few seconds delay a new game starts.
13. Ensure that this time the computer has made the first move with a yellow piece.
14. Press the HOME button.
15. Ensure that Connect 4 stops and goes to the Home menu options.
16. Click Resume on the gamepad.
17. Ensure that the Wii U returns to the game.
18. Play column R.
19. Ensure that a red piece drops into column R.
20. Press the ‘HOME’ button, then Close Software.
21. Ensure that the Wii U correctly exits Connect 4 and returns to the main menu.

The test case was executed and all test steps passed. The following show pictures as evidence that the different tests from the above sequence worked correctly:

|  |  |
| --- | --- |
|  | Step 1 selecting Connect 4 from the Wii U menu. |
|  | Step 2 game starts correctly.  Step 4 Wii U is waiting for the human player to make the first move. |
|  | Step 3 The gamepad showing the game after changing from MEDIUM to HARD. |
|  | Step 6 and 7 showing the red piece with the yellow computer piece added shortly afterwards for the computer move. |
|  | Step 9 Playing the moves and step 10 yellow moves following shortly after each red move. |
|  | Step 11 Yellow game count increases. |
|  | Step 12 and 13 a new game starts, this time the Wii U has made the first move. |
|  | Step 15 Home menu when Home button pressed. |
|  | Step 17 the game continued when Resume was pressed.  Step 19 the game continued playing when column R was selected. |
|  | Step 21 The Wii U returned to the main menu when close software was selected. |

### Test case 2: Robust Game Play

The following requirements are tested:

8: The starting player shall then alternate between computer and human for subsequent games.

9: The human move shall only complete with a valid move (playing a column 1 to 7 that has an available space).

10: The computer move shall only complete with a valid move (playing a column 1 to 7 that has an available space).

12: The human player shall win if they complete a line of four red pieces.

13: The computer shall win if they complete a line of four yellow pieces.

14: The game shall be declared a draw if the board is full and neither player has a line of 4 pieces.

15: The number of games won by Red shall be recorded and displayed.

16: The number of games won by Yellow shall be recorded and displayed.

17: The number of drawn games shall be recorded and displayed.

Perform the following test steps:

1. Start the Wii U game console and select Connect 4.
2. Select EASY mode.
3. Ensure that Red Wins, Yellow Wins and Draws are displayed on the gamepad and these are all 0.
4. Play for a red win.
5. Ensure that Red is shown as winning and the red win count increases on the gamepad.
6. Ensure that a new game starts with yellow making the first move.
7. Play for a red loss.
8. Ensure that Yellow is shown as winning and the yellow win count increases on the gamepad.
9. Ensure that a new game starts with an empty board waiting for the player to make the first move.
10. Play for a draw (note this may take several attempts).
11. Ensure that result is shown as a draw and the draw count increases on the gamepad.
12. Press all other gamepad controls.
13. Ensure that they have no impact on the game.
14. Fill a column.
15. Ensure that selecting the full column has no effect. The Wii U keeps waiting until the player selects a valid move.
16. In all cases make sure that the computer plays a yellow piece (and only one yellow piece) after every red move (until the game is won).

The test case was executed and all test steps passed. There were quite a number of games played for the above test sequence (it took several attempts to force a draw). Tests for robustness are to ensure that erroneous inputs are correctly ignored. There is therefore no pictorial evidence as there is no change. The following are some screen captures to show the main points:

|  |  |
| --- | --- |
|  | Step 3 Red wins, Yellow wins and draws are all 0 at the start. |
|  | Step 5 Red Win. |
|  | Step 5 Red win count increases. |
|  | Step 6 new game starts and yellow goes first. |
|  | Step 8 Yellow Win. |
|  | Step 8 Yellow win count increases. |
|  | Step 9 new game starts with empty board wating for Red to make the first move. |
|  | Step 11 Draw. |
|  | Step 11 Draw count increases. |

### Test case 3: Difficulty Levels

The following requirements are tested:

5: The game shall have three difficulty levels easy, medium and hard.

6: The game shall re-start if the difficulty level is changed.

18: The game shall be challenging when set to the hard level.

19: The medium setting shall be less difficult than the hard level.

20: The easy setting shall be less difficult than the medium level.

Perform the following test steps:

1. Start the Wii U game console and select Connect 4.
2. Play column R.
3. Select EASY mode.
4. Ensure that the mode changes to EASY and the board is cleared.
5. Play a game to win.
6. Ensure that it is quite easy to win.
7. Play column R.
8. Select MEDIUM mode.
9. Ensure that the mode changes to MEDIUM and the board is cleared.
10. Play a game to win.
11. Ensure that it is more difficult to win playing MEDIUM.
12. Play column R.
13. Select HARD mode.
14. Ensure that the mode changes to HARD and the board is cleared.
15. Play a game to win.
16. Ensure that it very difficult to win.
17. Select EASY mode.
18. Play a game to win.
19. Ensure that it is back to being easy to win.

The test case was executed and all test steps passed. There were quite a number of games played for the above test. Screenshots have only been captured for the main points:

|  |  |
| --- | --- |
|  | Step 1 Started up in MEDIUM. |
|  | Step 2 Played R column. |
|  | Step 4 changed to EASY. |
|  | Step 4 Board cleared ready for a new game. |
|  | Step 6 it was easy to win in EASY mode. |
|  | Step 9 mode changed to MEDIUM. The screen was also cleared (a picture of this has not been repeated). |
|  | Step 11 in MEDIUM it took more effort to get a win. |
|  | Step 14 mode changed to HARD. The screen also cleared ready for a new game. |
|  | Step 16 it was much harder to get a win. I managed for this test run, but don’t always manage a win in HARD mode. |
|  | Step 19 once the mode was changed back to EASY, it was again easy to win. |

### Test case 4: Weighting Updates

An additional feature has been added during development. The weightings are stored in a text-file so that these can be updated by the game. If the human wins on the hardest mode the weightings o and n can be tweaked up or down by 0.1, which is enough to change computer play. This ensures that the player cannot just keep playing the same set of winning moves.

Perform the following test steps:

1. Take a copy of the weightings.txt file from the SD Card.
2. Start the Wii U game console and select Connect 4.
3. Select HARD mode.
4. Play to get a win.
5. Take a copy of the weightings.txt file from the SD Card.
6. Ensure that the weightings have been trimmed to different values.
7. Lose the game where yellow starts to get to red playing first.
8. Play the same moves again.
9. Ensure that the computer has changed the way it plays.

The test case was executed and all test steps passed. The weightings were changed to a set from earlier development where the moves to win were known.

The following were the weightings used:

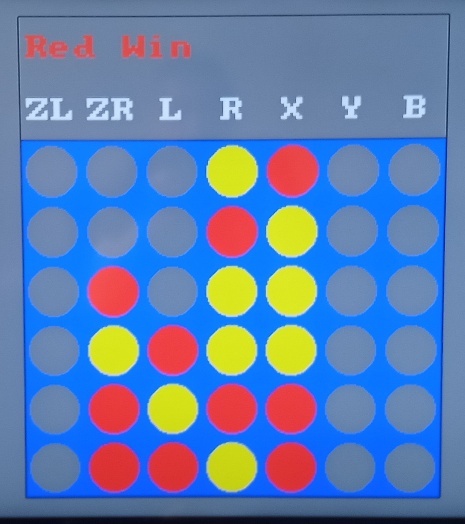
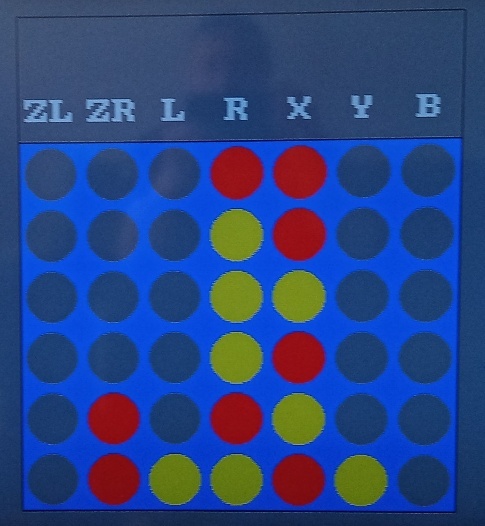
x h v d n o

5.00 1.10 1.30 1.50 0.50 0.70

After the computer lost the weightings were adjusted:

5.00 1.10 1.30 1.50 0.50 0.90

The game play changed. The computer did not play yellow on the ZR column after the weighting changes, which stopped the human playing the same set of moves to win.

# Evaluation

Connect 4 is a comparatively simple game, and testing in the previous section has shown that it works as per the requirements and design.

## Evaluation Testing

The post development testing in section 3.2 includes testing to ensure that the game is robust (see test case 2 in section 3.2.2). For example, test steps 14 and 15 are to ensure that the player cannot select a column that is already full. The game is easy to use on the Wii U version; the player just needs to press the game controller button for the column they want to play (see the following video of game operation). The instructions are displayed on the game pad screen.

The following is a link to a video of two games being played:

*(This would be a link to a YouTube video for the real report)*

When I started the project, I was concerned that the program would not be a very good player and would be easily beaten. In fact, the program has the opposite problem. It is very difficult to play; I have rarely beaten it now that it is optimised. It was no fun to play; so easy medium and hard levels have been added. The program is therefore a very good example of how a neural network can be used.

It is interesting to play against the neural network. It does not play like a human player; each move is calculated to give the best chance of a win i.e. it gives the most chances of a line of four while doing the best to limit the opponent. I found that as the board filled up, I was often forced to play a move that then let the computer win.

## Solution Success

As stated in the previous section. The solution is a great success. The post development testing shows the success of game play. Test case 3.2.4 shows that the game will modify its weightings if it is beaten, so that the player must find a different way to win. The computer is a very good Connect 4 player. To make the game fun to play, easy, medium and hard levels needed to be added.

The game seems to be quite popular; hundreds of people have downloaded it from the [Homebrew App Store - Connect 4](https://hb-app.store/wiiu/Connect4). I have had email feedback from online users that the game has made them ‘think hard’.

As the object is for the computer to play Connect 4 well, there are no unmet criteria. However, the project has concentrated on the neural network to calculate the computer move. The Graphical User Interface is the minimum needed to play the game. It would be more fun to play, and be visually more appealing with a better way to select the column (possibly picking up a piece with the joycon and dropping it into a slot). It would be better if there were animation of pieces dropping into place.

## Final Product

The final product is shown in the video in the previous section. This shows that the game works well and the interface allows the game to be played easily. Assigning a game controller button to each column is an easy way for the player to control the game. However, I found it quite easy to accidentally click the wrong button and make the wrong move. An interface where the player picks up a piece, then presses another button to drop it in a column, would reduce the chance of mistakes.

The following is the final game. The game can be played and meets all of the success criteria.

[MartinButlerAAA/Connect4 WiiU](https://github.com/MartinButlerAAA/Connect4U) (v1.2.0)

The Java development version is also available on GitHub:

[MartinButlerAAA/Connect4: Connect 4 game](https://github.com/MartinButlerAAA/Connect4)

## Maintenance and Development

The program is comparatively small so is easy to maintain. However, as it was developed in Java for the neural-network, then ported to the C language for the Wii U, updating the neural network would be more work. If the neural network were to be updated, it would be better to add optimisation to the C version, rather than going back to the Java version. Either way, this would be more work than if the program had been developed in C in the first place.

The game plays well (although it can be beaten). The current approach of ‘tweaking’ the neural network weightings could be replaced with 4 or 5 good sets of weightings. Each time the computer is beaten it just swaps the set it uses. It could even randomly select the set it used for each game, so the human player could not be sure which way the computer would play.

As stated in the previous two sections, the game is implemented on the Wii U with a Graphical User Interface. It is functional, but is the minimum to make the game work. The game would be more complete with animation. It would be better if the pieces were animated to drop into place, rather than just appearing. It would also be good if the game were enhanced so that it can be played on the hand-held gamepad, as well as on the TV display.