Connect 4 Neural Network Project

Name: Martin Butler

Date of Birth:

Student number:

Assessment Identification:

Tutor Name:

Contents

[1. Analysis 3](#_Toc172190852)

[1.1. Problem Identification 3](#_Toc172190853)

[1.1.1. Neural Networks 3](#_Toc172190854)

[1.1.2. Machine Learning 4](#_Toc172190855)

[1.2. Stakeholders 4](#_Toc172190856)

[1.3. Problem Research 4](#_Toc172190857)

[1.4. Proposed Solution 5](#_Toc172190858)

[2. Design of the solution 6](#_Toc172190859)

[2.1. Problem Decomposition 6](#_Toc172190860)

[2.1.1. The Output 6](#_Toc172190861)

[2.1.2. The Input 7](#_Toc172190862)

[2.1.3. Rows and Columns 8](#_Toc172190863)

[2.1.4. Diagonals 8](#_Toc172190864)

[2.1.5. Neural Network 9](#_Toc172190865)

[2.1.6. Weightings 9](#_Toc172190866)

[2.1.7. Optimisation 10](#_Toc172190867)

[2.2. Design 10](#_Toc172190868)

[2.2.1. Play game 10](#_Toc172190869)

[2.2.2. Optimisation 12](#_Toc172190870)

[2.2.3. Command line processing 13](#_Toc172190871)

[2.3. Test Approach 13](#_Toc172190872)

[3. Developing the solution 14](#_Toc172190873)

[3.1. Development 14](#_Toc172190874)

[3.1.1. Overall Game 14](#_Toc172190875)

[3.1.2. Command line 15](#_Toc172190876)

[3.1.3. Optimisation of the Neural Network 17](#_Toc172190877)

[3.2. Post Development Testing 18](#_Toc172190878)

[3.2.1. REQ1 Run on Smart Phone and PC 18](#_Toc172190879)

[3.2.2. REQ2 Robust solution 19](#_Toc172190880)

[3.2.3. REQ3 and REQ4 Multiple games 20](#_Toc172190881)

[3.2.4. REQ5 Play to the rules 21](#_Toc172190882)

[3.2.5. REQ6 Play to a challenging standard 21](#_Toc172190883)

[3.2.6. REQ7 Command-line interface 21](#_Toc172190884)

[4. Evaluation 22](#_Toc172190885)

[4.1. Evaluation Testing 22](#_Toc172190886)

[4.2. Solution Success 22](#_Toc172190887)

[4.3. Final Product 22](#_Toc172190888)

[4.4. Maintenance and Development 23](#_Toc172190889)

# 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; they choose a number between 1 and 7.



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

## Problem Identification

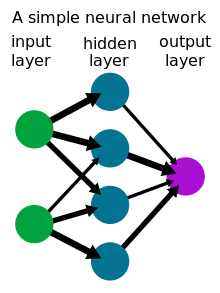
The rules of the Connect 4 game are simple. Each player takes turns to drop one of their coloured pieces into the board. The first player to get 4 in a line is the winner. If the board is full, with no winning player, the game is a draw. Players alternate who starts first.

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 to inform its play. 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 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.

## 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.

## Proposed Solution

To play the game a set of neurons (calculations / sections of software) must be devised, and then connected. The connections must also have 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.

|  |  |  |
| --- | --- | --- |
| **Req. number** | **Requirement** | **Success Criteria** |
| REQ1 | The solution shall run on a smart phone or Personal Computer (PC). | Demonstrate that the program will run on both a mobile phone and laptop. |
| REQ2 | The solution shall be robust. | Demonstrate that the program correctly handles erroneous entries from the player. |
| REQ3 | The solution shall be able to play multiple games, alternating the starting player between the computer and user. | Run multiple games showing that the starting player alternates. |
| REQ4 | The solution shall report the number of games won by each player and any draws. | Run multiple games showing that the program correctly determines the end of a game, which player won, or if it is a draw. |
| REQ5 | The solution shall play the Connect 4 games in accordance with the rules. | Demonstrate that the computer always takes only one valid move. Ensure that the moves drop correctly to the next place in a column. |
| REQ6 | The solution shall be able to play the Connect 4 game to a challenging standard. | Show that the program can beat human players. |
| REQ7 | The solution shall be implemented as a console program that can be called from Python, to support a PyGame graphics program. | Demonstrate that a Python program can correctly execute the Connect 4 program, to calculate computer moves. |

The solution is being deliberately implemented as a console program. The main purpose of the project is to demonstrate the use of a neural network, so does not need graphics. The project is however being developed with expansion in mind. It includes a command-line interface so that a Python program can be written to implement the graphics, calling the Connect 4 program, to calculate the computer move.

As the intention is to investigate the use of a neural network, some niceties of computer games have not been considered. The human player always starts first for the first game, the computer will then be first for the second game. The computer will always play yellow, and the user will always play red.

# 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. There needs to be a neuron 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.



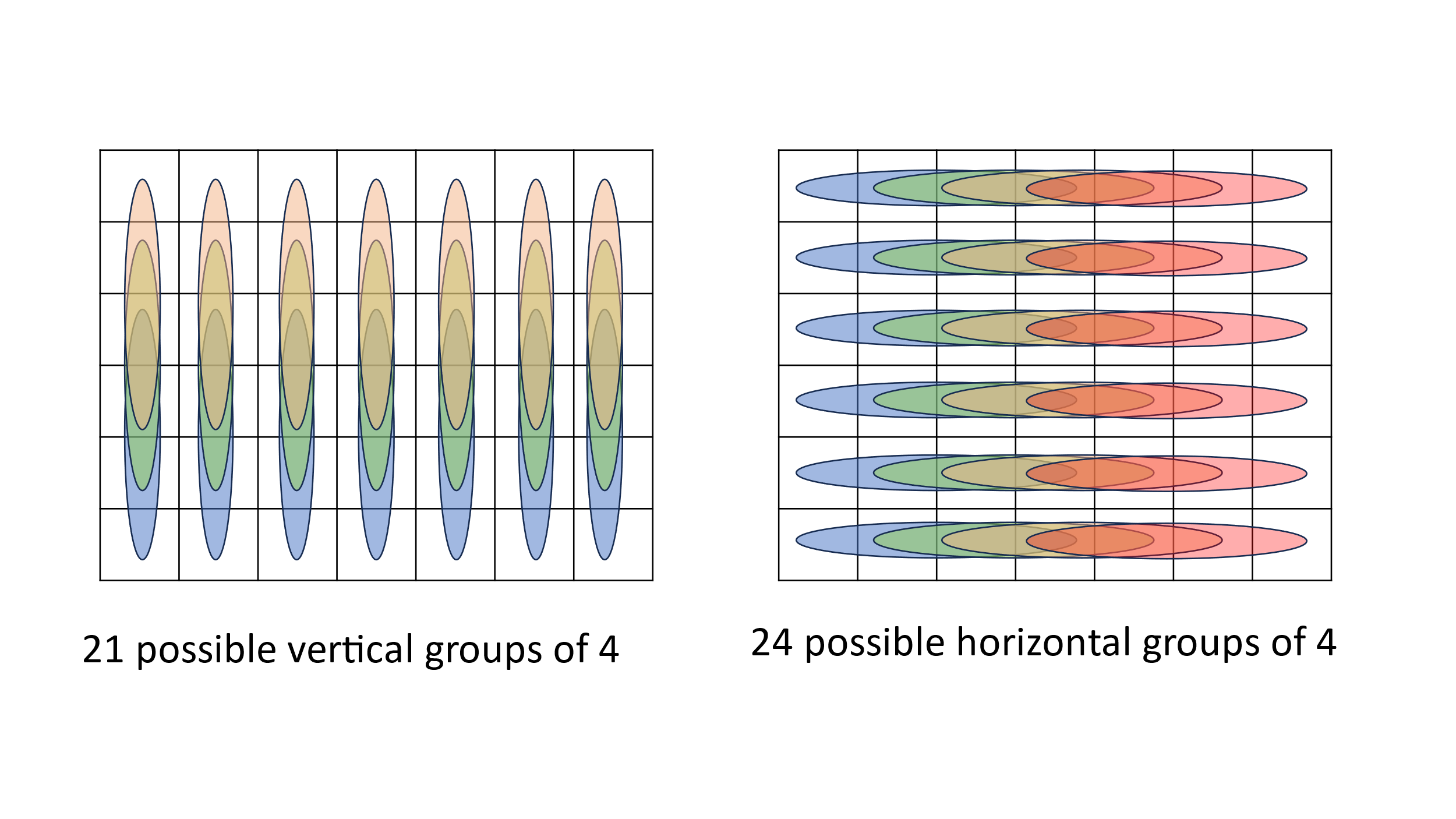
### The Input

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 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:

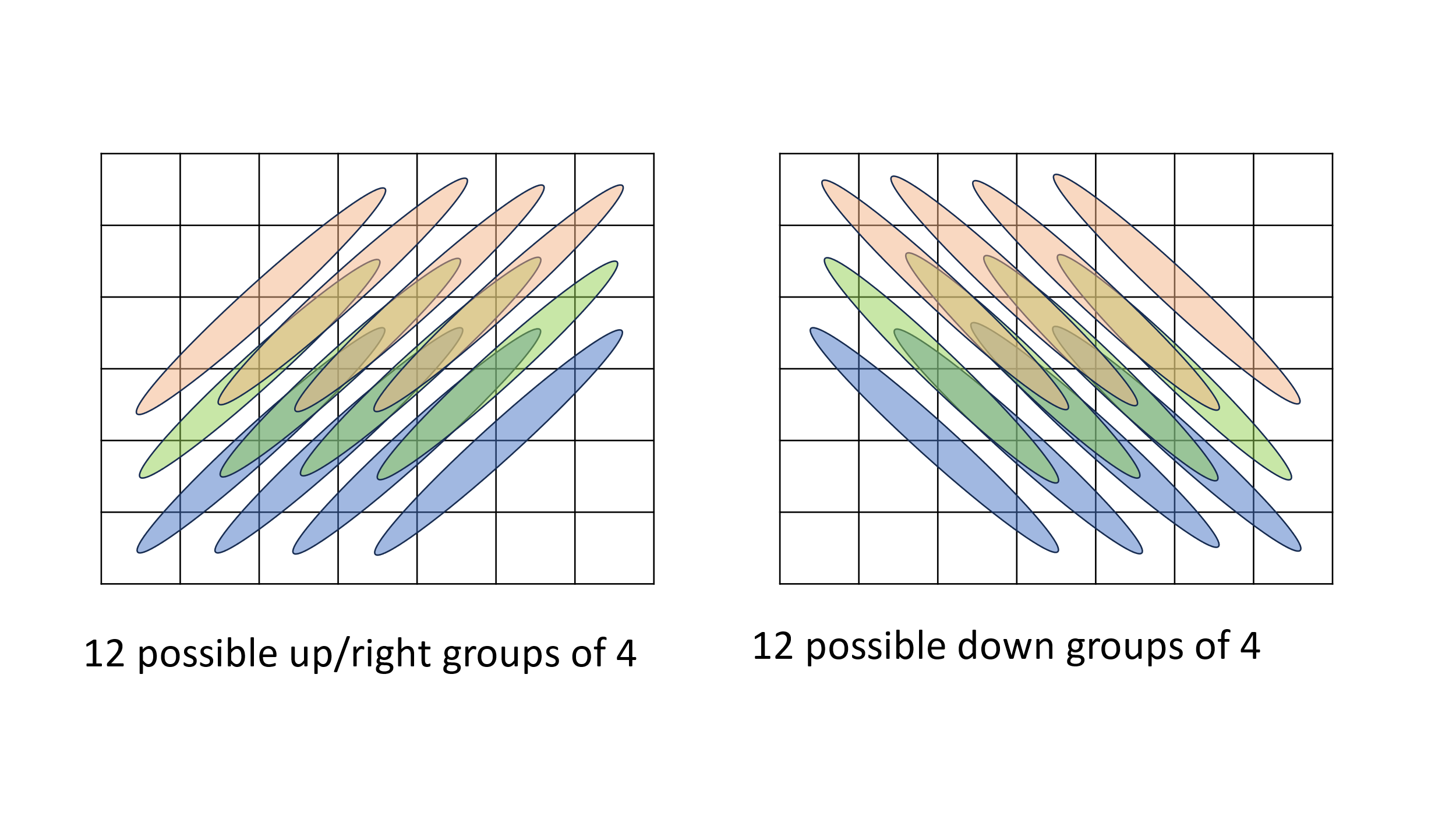


Three in a line is much more important than two in a line or one. The count is therefore used as a power, x^R and x^Y. ‘x’ can then be used as a weighting, the higher the value of x, the bigger the difference for more pieces. For example, for x=4, 4^1 = 4, 4^2 = 16 and 4^3 is 64, giving 3 in a line a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

### Rows and Columns

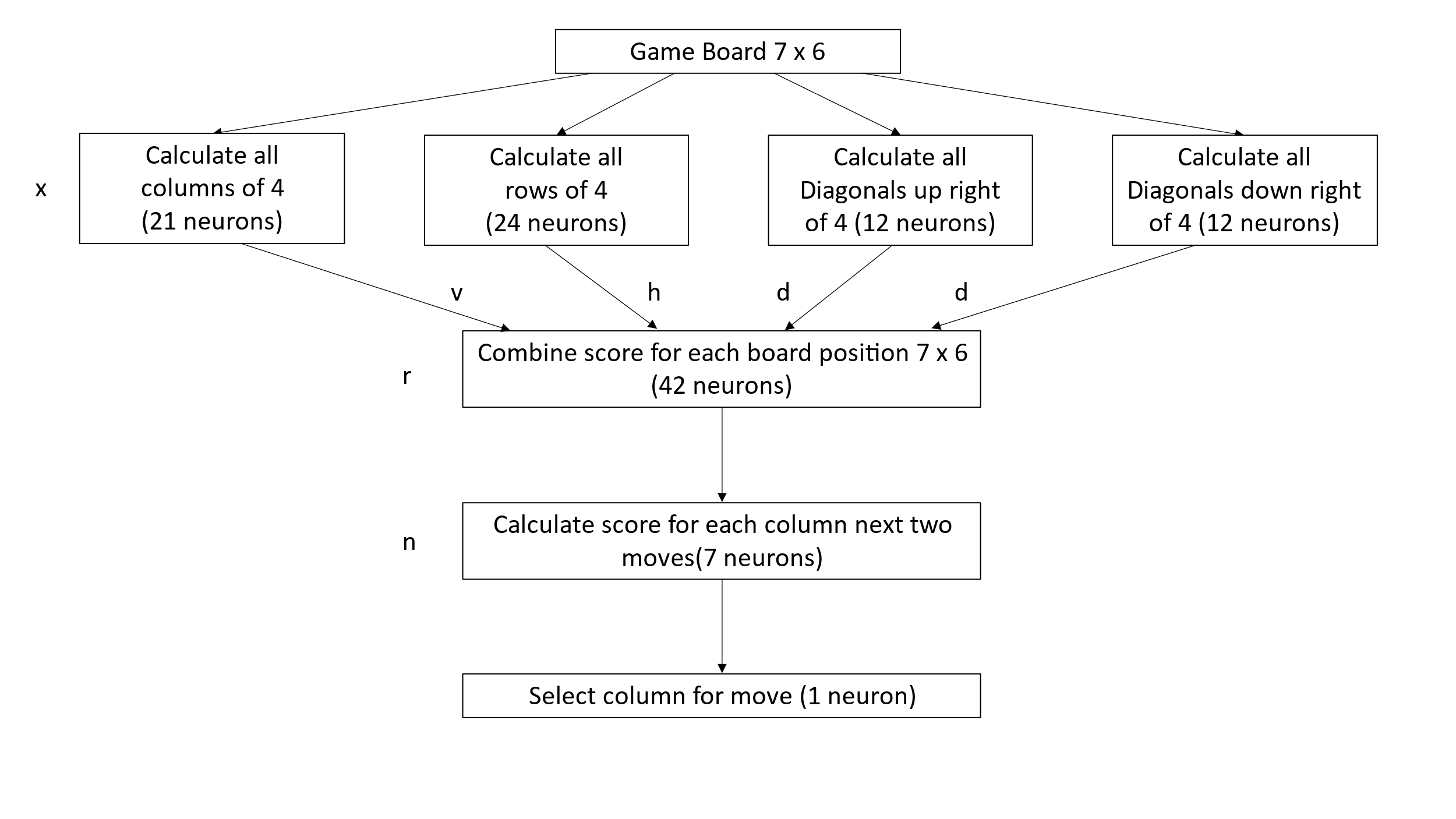
The following shows the possible vertical and horizontal winning lines:

### 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 score for each column is calculated by finding the next free vertical position in the column (next move). The score for both Y (the computer) and R (the opponent) are added together. If the move is good for the opponent, it may be a good idea to block it. The opponents score for the place above is then subtracted; this is to consider if a move gives the opponent an advantage.

Score = Y + R – (next move R x n)

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

### Weightings

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

x – number raised to the power of the Y and R counts

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)

r – random number added

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

A random component is needed, to avoid the computer playing the exact same game every time.

### Optimisation

The neural network can be optimised by the computer playing against itself. A random component is needed to avoid the computer continually playing the same game. A trial set of weightings is played against the default set. If the trial set wins more games, it is then played against the current set. If the trial set again wins more games, the current set is set to match the trial set. The process continues with a new trial set. In this way the weightings can be honed, so that the computer plays well.

## Design

The 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. If a string of characters is passed in representing the state of the board, the neural network processes the board, then returns the move. Otherwise the game loop is run continually.

### 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 2.1.6.

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 2.1.6).

Each vertical R and Y score is multiplied by v (horizonal weighting from section 2.1.6).

Each diagonal R and Y score is multiplied by d (horizonal weighting from section 2.1.6).

The scores for each position (including weightings) are added together along with a random number r (a further weighting from section 2.1.6).

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 – (opponent next move score multiplied by n).

n is the next move weighting (from section 2.1.6)).

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 an advantage.

### Optimisation

The neural network needs to be optimised based on experience. The weightings shown in section 2.1.6 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. Looking at the game, it is clear that 3 pieces in a line is more important than 2 or 1, but a numerical value for this needs to be determined by optimisation.

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 1000 cycles, each with the following steps:

* Select a random set of trial weightings
* Play 100 games of the trial weightings against the default weightings
* If the trial weightings win 20% more games, run 100 games against the current weightings
* If the trial weightings again win 20% more games, set the current weightings to match the trial weightings

### Command line processing

The command-line processing is to support a further development of the program to have a graphical interface. The intention is that the graphical interface can be implemented in Python using the PyGame extensions, the Connect 4 program being called to calculate the move.

If the command line contains at least 42 characters, these are processed to ensure that they correctly contain only 3 different characters: Y for yellow, R for red or space for unused places on the board. The characters are then copied into the gameTable array,

The displayBoard function is called to display the current board to aid diagnostics. calculateMove is then called to calculate the best move for the current board passed in. The move is then output in a text file called Move.txt, so that the Python program can receive the move. Java does not allow for return parameters from the command line.

## Test Approach

The test approach will first concentrate on ensuring that the solution is robust. That all three main options (play game, optimisation and command line processing) can correctly be selected and that these operate in a defined way for input exceptions.

Game play will then be used with manual testing to ensure that the user input is robust, ensuring that invalid moves selected by the user are correctly detected and rejected. Game play will also be used to demonstrate correct counting of winning or drawn games, along with correct alternation of players starting.

The calculateMove function will be initially set to make a random move. This will allow operation of the game ending function and humanMove functions to be manually tested. To ensure that columns cannot be over-filled and that winning lines are correctly detected. This will also allow drawn games to be played, to check that drawn games are also correctly detected.

Once the robust operation of the basic game has been determined, testing will concentrate on optimising the neural network.

# 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 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 then deliberately created to ensure that these were correctly detected.

### Command line

A small Python program was created to demonstrate that the command line processing worked correctly.

A screenshot of a computer program

Description automatically generated

The above Python program makes two calls to the Java program to ensure that the command line is not remembered by the Java program (i.e. it is processed for each move). It also demonstrates the old Move.txt file is overwritten by the new move, each time the Connect 4 program is run. This is demonstrated in the run log, shown over page:

A screenshot of a computer

Description automatically generated

The above shows the display for running the Python program to call the Java program for the Connect 4 game. The board is displayed by the Java program, after it has been received from the Python program, via the command line. This shows that the command line is correctly processed by the Java program. The Python program then receives the move via the Move.txt file. This is shown by the Move in the Java output and the Python output being the same.

The fact that the move is updated to 4 the second time, shows that the Move.txt file is overwritten each time the Java program is called. Which ensures that the new move can be received each time.

Different versions of the Python program were tried to ensure error handling. A string of 41 characters was tried to demonstrate that the Connect 4 program correctly rejects the pass parameter and reports an error. A string with the numeric value of 1 inside the 42-character string was also tried. This showed that the Connect 4 program correctly rejected a pass parameter with the invalid character (not ‘R’, ‘Y’ or space), and reported an error message.

### 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 (1.0 to 8.0)

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)

r – random number added (1 to 10)

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

Initial runs were not promising, as the weightings varied continually without moving towards a common solution. One concern was that the random component was too large, so this was set to 0. This however meant that when weightings were trialled, each player played exactly the same way each time. This therefore did not give a good idea of the weightings.

The random component was therefore fixed for optimisation runs, being a maximum of 2 or 3. Having a small random component ensured that the computer did not always play the same game, but was not that random that it stopped optimisation working. The results of optimisation were still too variable.

The x value is applied before the other weightings in calculations, so has a greater impact on operation. The x parameter was also fixed for optimisation runs, using either 3, 4 or 5. The following combination of r and x settings were each run twice, and the resulting optimised weightings recorded:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| r | x | h | v | d | n |
| 2.0 | 3.0 | 1.8 / 0.9 | 1.7 / 0.6 | 1.7 / 1.3 | 0.8 / 0.7 |
| 2.0 | 4.0 | 0.7 / 0.6 | 1.5 / 1.9 | 1.0 / 1.3 | 0.8 / 0.5 |
| 2.0 | 5.0 | 0.5 / 1.5 | 1.9 / 1.9 | 0.7 / 0.7 | 0.6 / 0.6 |
| 3.0 | 3.0 | 1.9 / 1.8 | 1.6 / 1.6 | 1.1 / 1.9 | 0.6 / 0.7 |
| 3.0 | 4.0 | 1.2 / 1.0 | 1.7 / 1.8 | 1.1 / 1.7 | 0.7 / 0.6 |
| 3.0 | 5.0 | 1.9 / 1.7 | 0.6 / 1.4 | 0.8 / 1.1 | 0.8 / 0.9 |

The weighting for the next move (n) was stable at around 0.7 (average 0.69 from above results), so this was fixed for further optimisation runs. The optimisation runs were also more stable with the lower random component of 2 and the x value of 4. These three weightings were fixed to allow the three remaining weightings to be optimised. This time the optimisation was run 5 times. Each time the previous results were used as the default settings for the next run.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| r | x | h | v | d | n |
| 2.0 | 4.0 | 0.9 | 1.6 | 1.9 | 0.7 |
| 2.0 | 4.0 | 1.3 | 1.9 | 1.7 | 0.7 |
| 2.0 | 4.0 | 0.8 | 1.5 | 1.5 | 0.7 |
| 2.0 | 4.0 | 0.9 | 1.4 | 1.5 | 0.7 |
| 2.0 | 4.0 | 1.2 | 1.8 | 1.9 | 0.7 |
| **2.0** | **4.0** | **1.02** | **1.64** | **1.7** | **0.7** |

Some of the weightings can be easily understood. The weighting for the next move (n) should be less than one, so that it does not stop a column being chosen if it is a winning move. It is also easy to see that a random component is needed so different games are played, but if this is too large, play is inconsistent. The x value of 4.0 seems about right, 3 pieces in a row being 4 times more important than 2, being four times more important than one; this should be enough to ensure a winning move to complete a line of 4 is taken.

The weightings for the horizontal (h), vertical (v) and diagonal (d) did seem to keep changing during optimisation. This suggests that there is no specific good ratio of settings between these three weightings, or that no combination is perfect for all situations. However as the defaults were updated to the settings shown in the above table, changes to the weightings during optimisation runs became less frequent.

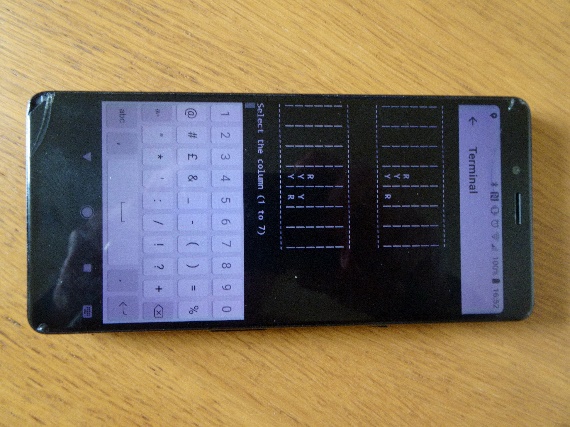
The weightings have therefore been set to the average of the last 5 runs, shown in the above table in bold. Further optimisation seemed to make little change to these weightings.

## Post Development Testing

The post development testing is based around the requirements from section 1.4 as detailed in the following sub-sections:

### REQ1 Run on Smart Phone and PC

The following screen shot show that the program can run an android mobile phone:



Operation on a PC is shown for all other testing which was performed with a laptop.

### REQ2 Robust solution

The following screenshot shows that the program ignores incorrect user input, prompting for a new input:

A screenshot of a computer program

Description automatically generated

If the user tries to add a piece to a column that is already full, they are correctly prompted again to get a valid move.

A screenshot of a computer

Description automatically generated

### REQ3 and REQ4 Multiple games

The following screenshots show that the program can run through multiple games, alternating the starting player and keeping track of games won. After the first game it has changed so that the computer takes the first move.

A screenshot of a computer

Description automatically generated

The following screen shot shows that the computer has won again with a diagonal line of 4, on the left side of the table. It has now won two games (which is correctly shown). It has also changed back to wait for the human player to make the first move.

A screenshot of a computer

Description automatically generated

### REQ5 Play to the rules

To avoid cluttering up the report the results of the testing for this requirement have been summarised, rather than showing a screenshot for each test. Multiple games were played, carefully monitoring the human and computer moves. In all cases the piece occupied the next available place in a column. In all cases moves alternated, neither the human player nor computer player could add more than one piece in their turn. In each case the program correctly detected when there were four pieces in a line. The following screenshot shows a simple example of the computer getting 4 in a line and winning the game:

A screenshot of a computer

Description automatically generated

### REQ6 Play to a challenging standard

So far, I have asked 4 friends to try to beat the game, in all cases they have lost. I have played the game many times during development. With the optimised weightings, I have never beaten the game and have only forced a draw once (which I have been unable to repeat). This demonstrates that the program plays Connect 4 to a high standard.

### REQ7 Command-line interface

The operation of the command line interface was re-tested and gave the same results as shown in section 3.1.2.

# Evaluation

The game is a comparatively simple game, and testing in the previous section has shown that it works as per the requirements and design. The evaluation therefore concentrates on the difference in play between an Artificially Intelligent (AI) player using a neural network, and a human player.

## Evaluation Testing

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 not beaten it). It is therefore no fun to play. The program is therefore a very good example of how a neural network can be used, it however therefore is not a good game.

It is interesting to play from a Computer Science viewpoint. The computer does not play like a human player. As can be seen from the video it tends to play in columns. This is a side effect of calculating the best move from scores. It will therefore tend to place pieces on top of the human players pieces, to block lines. The computer also tends to play the percentages, in that it keeps making moves that give it the maximum options. This means that as the board fills up, the human player is eventually forced into making a move that lets the computer win.

## Solution Success

As stated in the previous section. The solution is a great success. The computer is a very good Connect 4 player. However this had the unexpected side effect of making the game little fun to play. To make the game fun to play further work will be required so that levels of difficulty can be selected, to keep the human player interested.

## Final Product

The following is the final game. The game can be played and meets all of the success criteria. To make the game more pleasing to the eye a graphical interface should be added (see next section).

[MartinButlerAAA/Connect4: Connect 4 game that uses a Neural Network to calculate moves.](https://github.com/MartinButlerAAA/Connect4)

## Maintenance and Development

The program has been deliberately written as a Java console program, with a command-line interface. The command line is as follows:

Java –jar Connect4.jar “ YRYRY YRY R “

The string has spaces for unused positions, Y for the computer and R for the player pieces.

If a string of 42 characters representing the current state of the board is passed in, the program processes the board with the neural network. A move is returned in a file called Move.txt which for example may contain ‘Move 3’. The move is 1 to 7 for columns 1 to 7 starting with the left-hand column. Java cannot return a value to the command line.

The program has been deliberately created with a command line option. The intention is that a Python PyGame program can be written to provide a graphical interface, using the command line program for the AI. The code and operation of the command-line interface is shown in section 3.1.2.

A PyGame graphics interface would make the game look more professional.

The other area of improvement needed, is to add selectable difficulty, or make the game become more difficult as play progresses. This could be achieved by having multiple sets of weightings. It may also be reasonably easily achieved by changing the random component of the weighting. If the game were to start with a much higher random weighting, it would be easier to play (the computer would in effect make mistakes). The random weighting could then be reduced each time the human player won a game. Making the game get more difficult as play progressed.